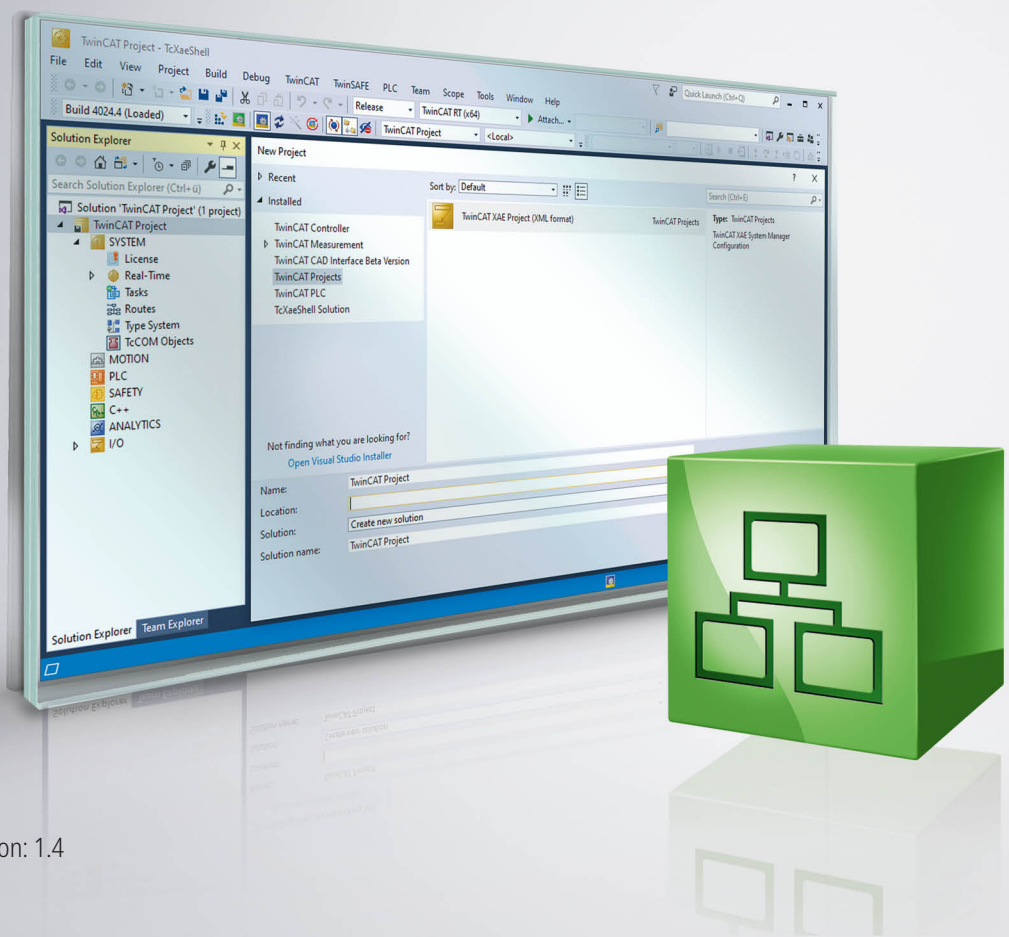
BECKHOFF New Automation Technology

Manual | EN

TF6310

TwinCAT 3 | TCP/IP



2021-05-06 | Version: 1.4

Table of contents

1 Foreword

1.1 Notes on the documentation

This description is only intended for the use of trained specialists in control and automation engineering who are familiar with applicable national standards.
It is essential that the documentation and the following notes and explanations are followed when installing and commissioning the components.
It is the duty of the technical personnel to use the documentation published at the respective time of each installation and commissioning.

The responsible staff must ensure that the application or use of the products described satisfy all the requirements for safety, including all the relevant laws, regulations, guidelines and standards.

Disclaimer

The documentation has been prepared with care. The products described are, however, constantly under development.
We reserve the right to revise and change the documentation at any time and without prior announcement.
No claims for the modification of products that have already been supplied may be made on the basis of the data, diagrams and descriptions in this documentation.

Trademarks

Beckhoff®, TwinCAT®, EtherCAT®, EtherCAT G®, EtherCAT G10®, EtherCAT P®, Safety over EtherCAT®, TwinSAFE®, XFC®, XTS® and XPlanar® are registered trademarks of and licensed by Beckhoff Automation GmbH.
Other designations used in this publication may be trademarks whose use by third parties for their own purposes could violate the rights of the owners.

Patent Pending

The EtherCAT Technology is covered, including but not limited to the following patent applications and patents:
EP1590927, EP1789857, EP1456722, EP2137893, DE102015105702
with corresponding applications or registrations in various other countries.

EtherCAT® is a registered trademark and patented technology, licensed by Beckhoff Automation GmbH, Germany

Copyright

1.2 Safety instructions

Safety regulations

Please note the following safety instructions and explanations!
Product-specific safety instructions can be found on following pages or in the areas mounting, wiring, commissioning etc.

Exclusion of liability

All the components are supplied in particular hardware and software configurations appropriate for the application. Modifications to hardware or software configurations other than those described in the documentation are not permitted, and nullify the liability of Beckhoff Automation GmbH & Co. KG.

Personnel qualification

This description is only intended for trained specialists in control, automation and drive engineering who are familiar with the applicable national standards.

Description of symbols

In this documentation the following symbols are used with an accompanying safety instruction or note. The safety instructions must be read carefully and followed without fail!

⚠ DANGER

Serious risk of injury!

Failure to follow the safety instructions associated with this symbol directly endangers the life and health of persons.

⚠ WARNING

Risk of injury!

Failure to follow the safety instructions associated with this symbol endangers the life and health of persons.

⚠ CAUTION

Personal injuries!

Failure to follow the safety instructions associated with this symbol can lead to injuries to persons.

NOTE

Damage to the environment or devices

Failure to follow the instructions associated with this symbol can lead to damage to the environment or equipment.

Tip or pointer

This symbol indicates information that contributes to better understanding.

2 Overview

The TwinCAT TCP/IP Connection Server enables the implementation/realisation of one or more TCP/IP server/clients in the TwinCAT PLC. With its help, own TCP/IP based protocols (application layer) may be developed directly in a PLC program.

Product components

The product TF6310 TCP/IP consists of the following components, which will be delivered by the setup:

- **PLC library:** Tc2_TcpIp library (implements basic TCP/IP and UDP/IP functionalities).
- **Background program:** TwinCAT TCP/IP Connection Server (process which is used for communication).

2.1 Comparison TF6310 TF6311

The products TF6310 "TCP/IP" and TF6311 "TCP/UDP Realtime" offer similar functionality.

This page provides an overview of similarities and differences of the products:

	TF 6310	**TF 6311**
TwinCAT	TwinCAT 2 / 3	TwinCAT 3
Client/Server	Both	Both
Large / unknown networks	++	+
Determinism	+	++
High-volume data transfer	++	+
Programming languages	PLC	PLC and C++
Operating system	Win32/64, CE5/6/7	Win32/64, CE7
UDP-Mutlicast	Yes	No
Trial license	Yes	Yes
Protocols	TCP, UDP	TCP, UDP, Arp/Ping
Hardware requirements	Variable	TwinCAT-compatible network card
Socket configuration	See operating system (WinSock)	TCP/UDP RT TcCom Parameters

The Windows firewall cannot be used, since the TF6311 is directly integrated in the TwinCAT system. In larger / unknown networks we recommend using the TF6310.

3 Installation

3.1 System requirements

The following system requirements must be met for the function TF6310 TCP/IP to work properly.

Operating systems:

Windows XP Pro SP3

Windows 7 Pro (32-bit and 64-bit)

Windows 10 Pro (32-bit and 64-bit)

Windows XP Embedded

Windows Embedded Standard 2009

Windows Embedded 7

Windows CE6

Windows CE7

TwinCAT:

TwinCAT 3 XAR Build 3098 (or higher)

TwinCAT 3 XAE Build 3098 (or higher)

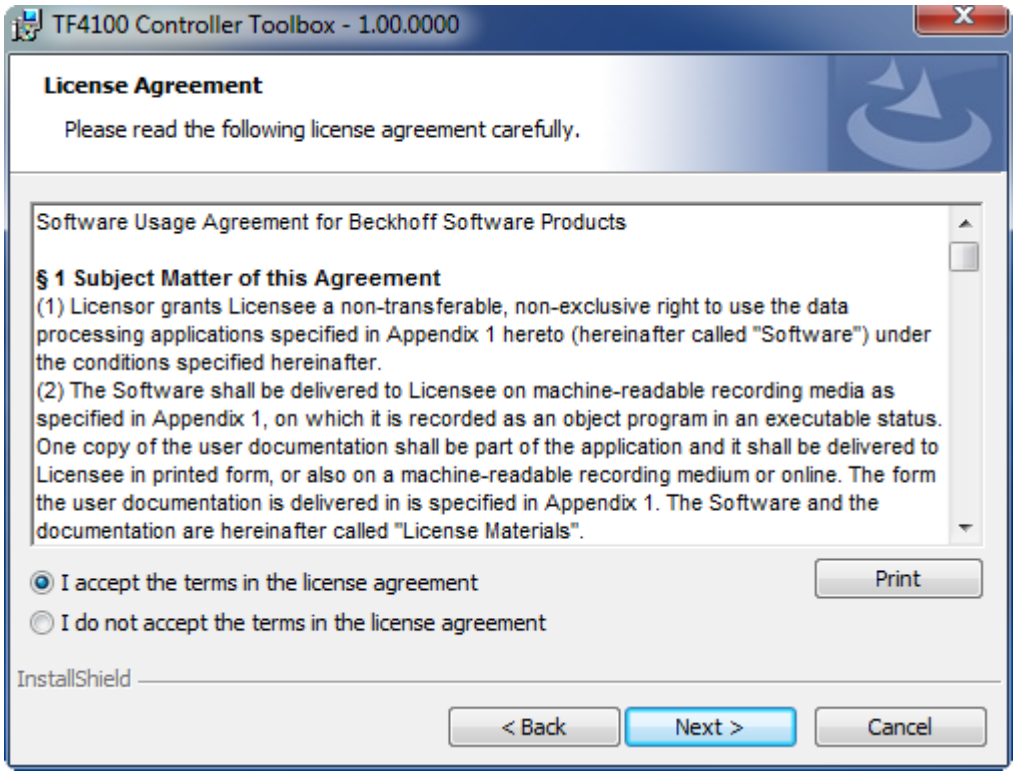
3.2 Installation

The following section describes how to install the TwinCAT 3 Function for Windows-based operating systems.

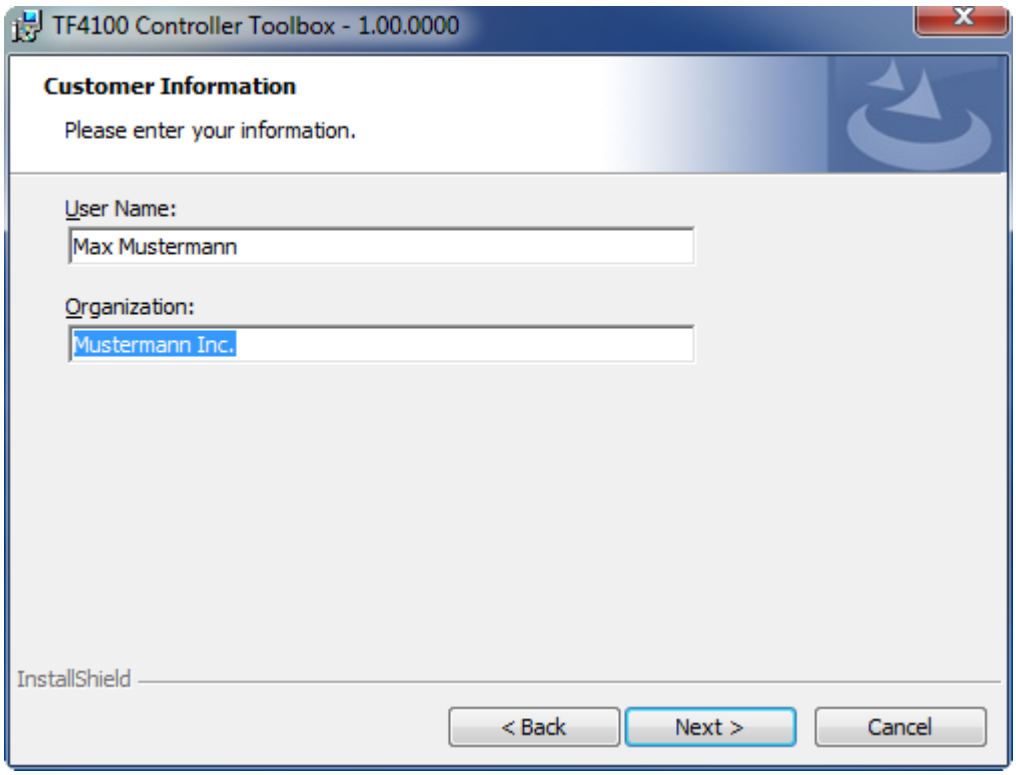
✓ The TwinCAT 3 Function setup file was downloaded from the Beckhoff website.

1. Run the setup file as administrator. To do this, select the command **Run as administrator** in the context menu of the file.
   ⇨ The installation dialog opens.

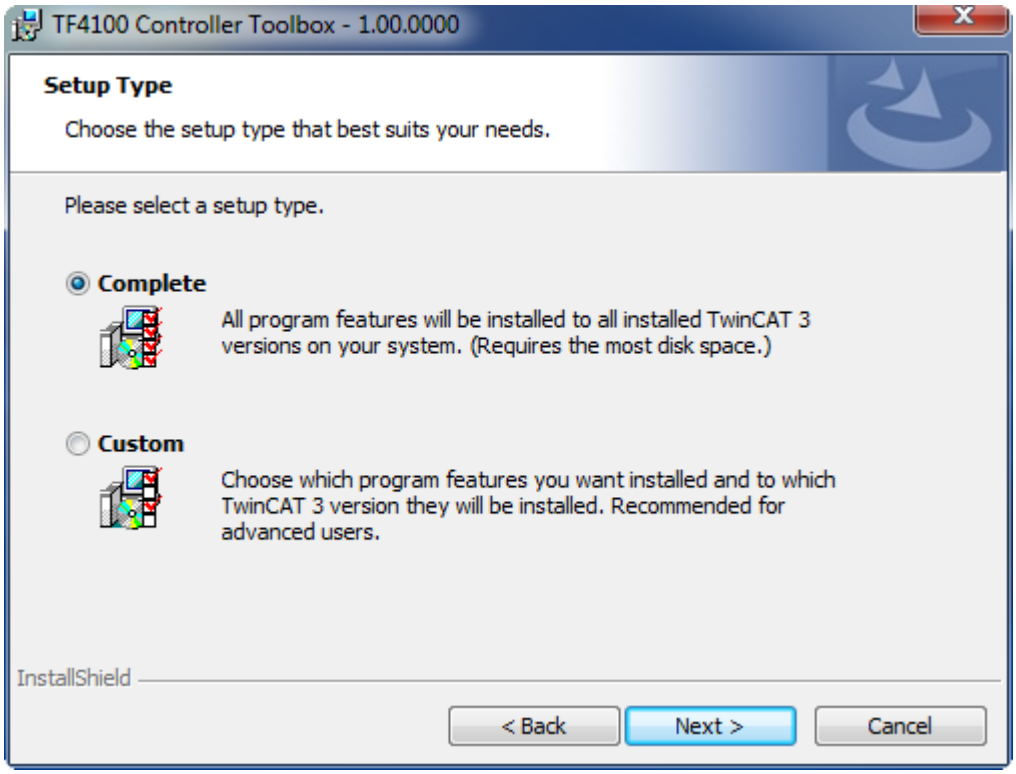
2. Accept the end user licensing agreement and click **Next**.



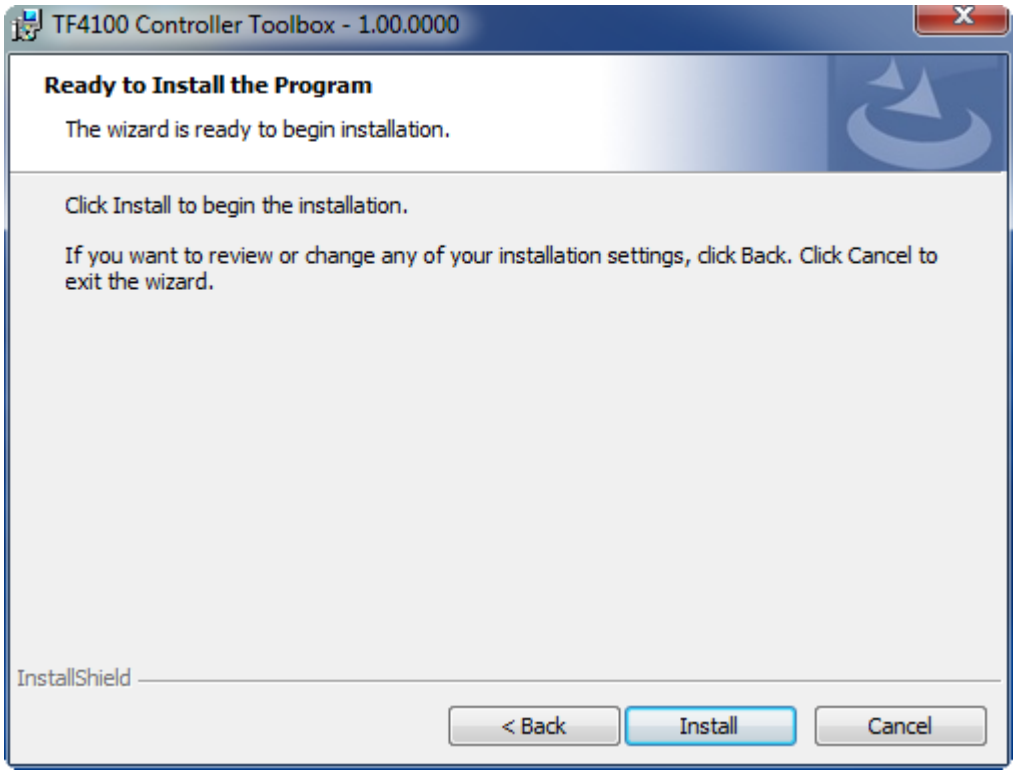
3. Enter your user data.



4. If you want to install the full version of the TwinCAT 3 Function, select **Complete** as installation type. If you want to install the TwinCAT 3 Function components separately, select **Custom**.

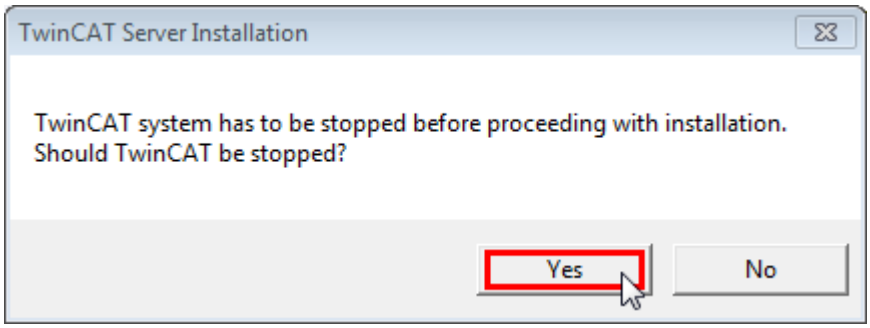


5. Select **Next**, then **Install** to start the installation.

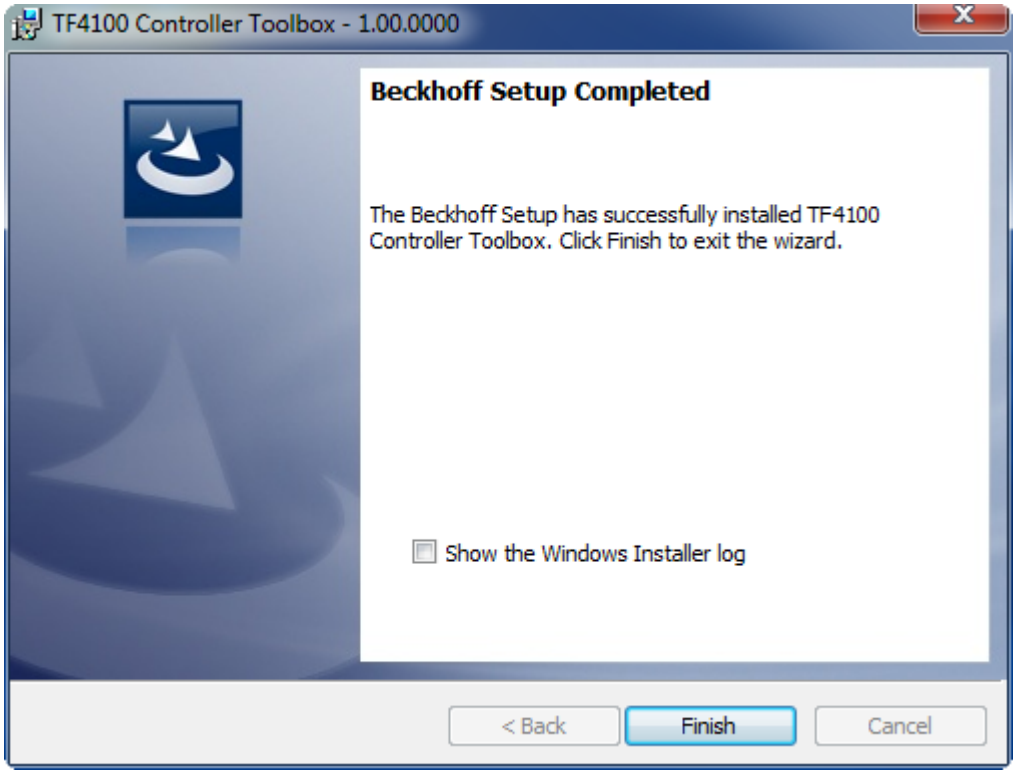


⇨ A dialog box informs you that the TwinCAT system must be stopped to proceed with the installation.

6. Confirm the dialog with **Yes**.



7. Select **Finish** to exit the setup.



⇨ The TwinCAT 3 Function has been successfully installed and can be licensed (see Licensing [▶ 13]).

3.3 Installation Windows CE

This section describes, how you can install the TwinCAT 3 Function TF6310 TCP/IP on a Beckhoff Embedded PC Controller based on Windows CE.

The setup process consists of four steps:

- Download of the setup file [▶ 12]
- Installation on a host computer [▶ 12]
- Transferring the executable to the Windows CE device [▶ 12]
- Software installation [▶ 12]

The last paragraph of this section describes the Software upgrade [▶ 13].

Download of the setup file

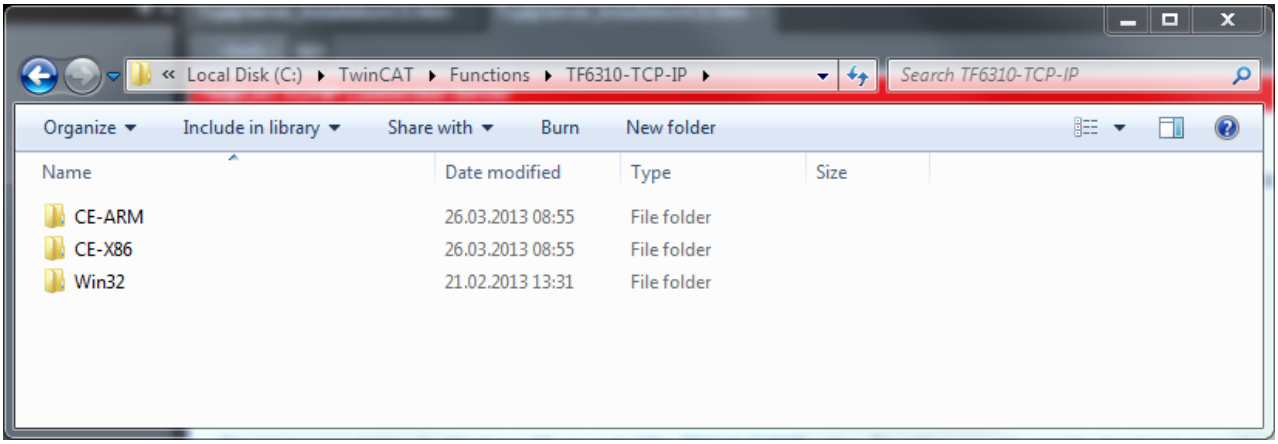
The CAB installation files for Windows CE are part of the TF6310 TCP/IP setup. Therefore you only need to download one setup file from www.beckhoff.com which contains binaries for Windows XP, Windows 7 and Windows CE (x86 and ARM).

The installation procedure of the TF6310 TCP/IP setup is described in the regular installation article (see Installation [▶ 8]).

Installation on a host computer

After installation, the install folder contains three directories - each one for a different hardware platform:

- **CE-ARM:** ARM-based Embedded Controllers running Windows CE, e.g. CX8090, CX9020
- **CE-X86:** X86-based Embedded Controllers running Windows CE, e.g. CX50xx. CX20x0
- **Win32:** Embedded Controllers running Windows XP, Windows 7 or Windows Embedded Standard



The CE-ARM and CE-X86 folders contain the TF6310 CAB files for Windows CE corresponding to the hardware platform of your Windows CE device. This file needs to be transferred to the Windows CE device.

Transferring the executable to the Windows CE device

Transfer the corresponding executable to you Windows CE device. This can be done via one of the following ways:

- via a Shared Folder
- via the integrated FTP-Server
- via ActiveSync
- via a CF card

For more information, please consult the "Windows CE" section in the Beckhoff Information System.

Software installation

After the file has been transferred via one of the above methods, execute the file and acknowledge the following dialog with **Ok**. Restart your Windows CE device after the installation has finished.

After the restart has been completed, the executable files of TF6310 are started automatically in the background.

The software is installed in the following directory on the CE device:

\Hard Disk\TwinCAT\Functions\TF6310-TCP-IP

Upgrade instructions

If you have already a version of TF6310 installed on your Windows CE device, you need to perform the following things on the Windows CE device to upgrade to a newer version:

1. Open the CE Explorer by clicking on **Start > Run** and entering "explorer".
2. Navigate to *\Hard Disk\TwinCAT\Functions\TF6310-TCP-IP\Server.*
3. Rename *TcpIpServer.exe* to *TcpIpServer.old.*
4. Restart the Windows CE device.
5. Transfer the new CAB-File to the CE device.
6. Execute the CAB-File and install the new version.
7. Delete *TcpIpServer.old*.
8. Restart the Windows CE device.

⇨ After the restart is complete, the new version is active.

3.4 Licensing

The TwinCAT 3 Function can be activated as a full version or as a 7-day test version. Both license types can be activated via the TwinCAT 3 development environment (XAE).

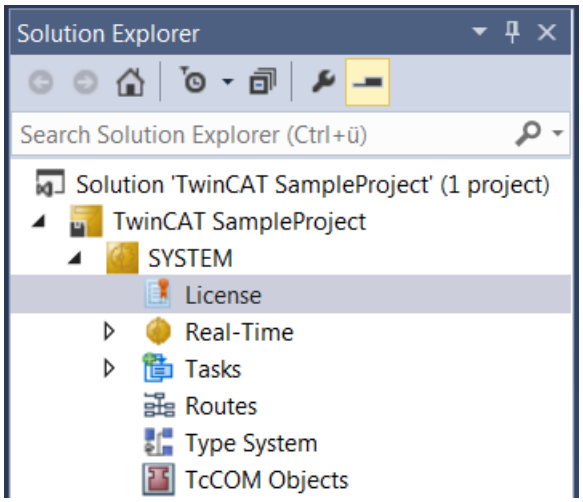
The licensing of a TwinCAT 3 Function is described below. The description is divided into the following sections:

- Licensing a 7-day test version [▶ 13]
- Licensing a full version [▶ 15]

Further information on TwinCAT 3 licensing can be found in the “Licensing” documentation in the Beckhoff Information System (TwinCAT 3 > Licensing).

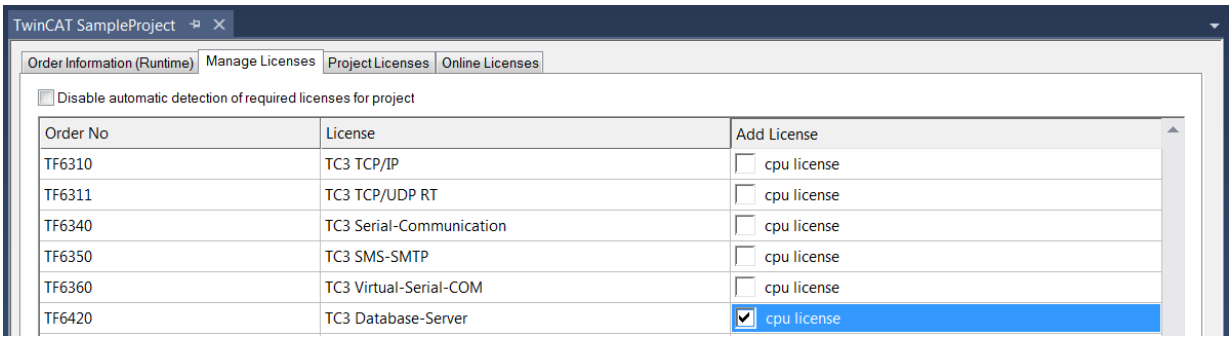
Licensing a 7-day test version

1. Start the TwinCAT 3 development environment (XAE).
2. Open an existing TwinCAT 3 project or create a new project.
3. If you want to activate the license for a remote device, set the desired target system. To do this, select the target system from the **Choose Target System** drop-down list in the toolbar.
   ⇨ The licensing settings always refer to the selected target system. When the project is activated on the target system, the corresponding TwinCAT 3 licenses are automatically copied to this system.
4. In the **Solution Explorer**, double-click **License** in the **SYSTEM** subtree.



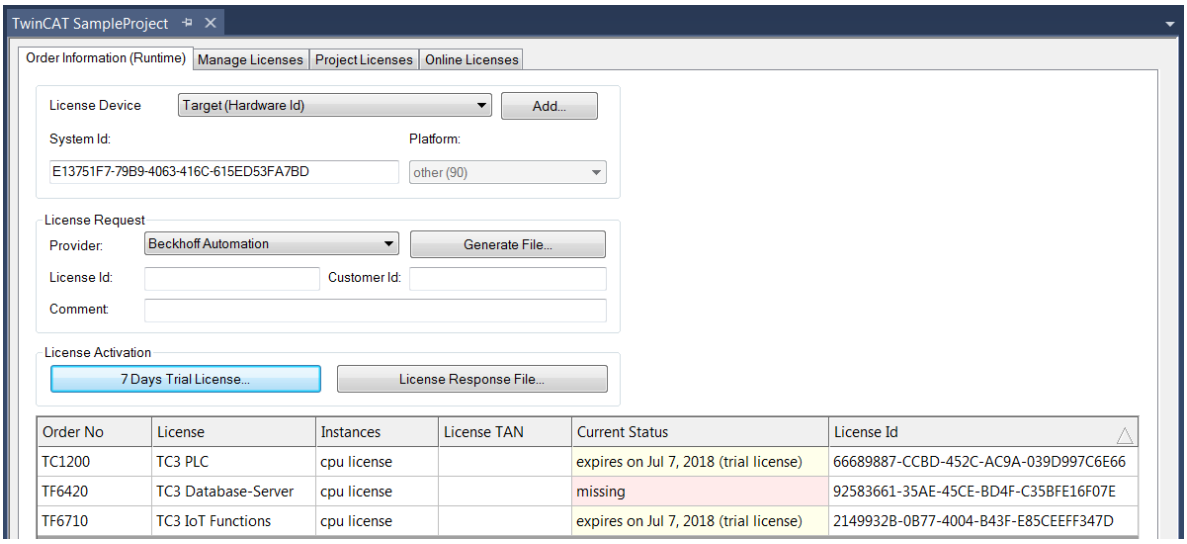
⇨ The TwinCAT 3 license manager opens.

5. Open the **Manage Licenses** tab. In the **Add License** column, check the check box for the license you want to add to your project (e.g. “TF6420: TC3 Database Server“).

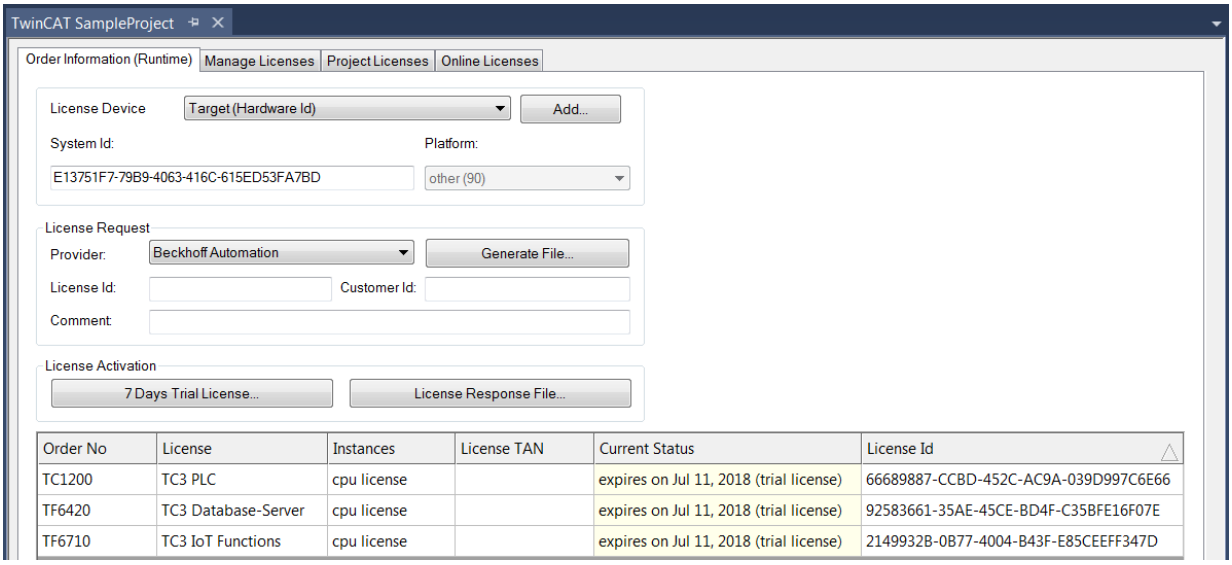


6. Open the **Order Information (Runtime)** tab.

   ⇨ In the tabular overview of licenses, the previously selected license is displayed with the status “missing”**.**



7. Click **7-Day Trial License...** to activate the 7-day trial license.



   ⇨ A dialog box opens, prompting you to enter the security code displayed in the dialog.

8. Enter the code exactly as it appears, confirm it and acknowledge the subsequent dialog indicating successful activation.

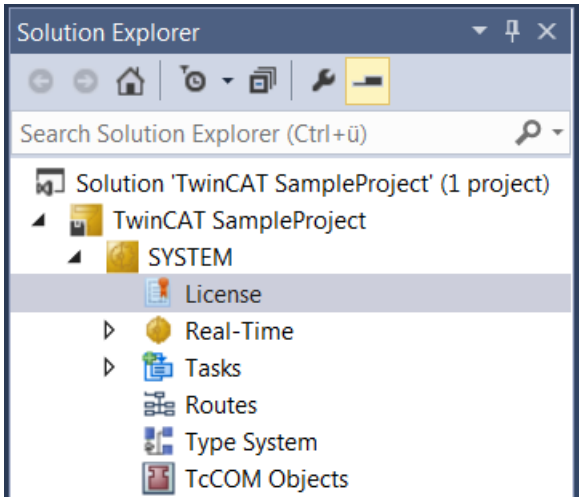
   ⇨ In the tabular overview of licenses, the license status now indicates the expiration date of the license.

9. Restart the TwinCAT system.

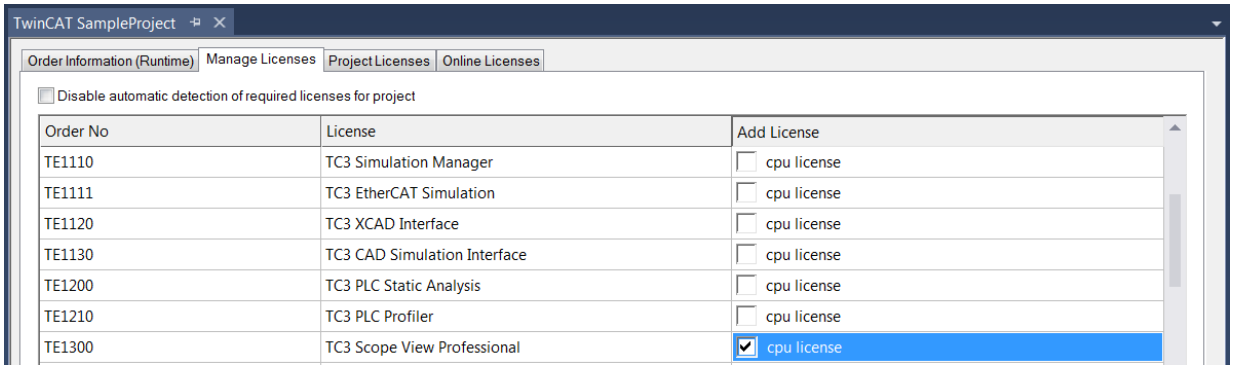
⇨ The 7-day trial version is enabled.

Licensing a full version

1. Start the TwinCAT 3 development environment (XAE).
2. Open an existing TwinCAT 3 project or create a new project.
3. If you want to activate the license for a remote device, set the desired target system. To do this, select the target system from the **Choose Target System** drop-down list in the toolbar.
   ⇨ The licensing settings always refer to the selected target system. When the project is activated on the target system, the corresponding TwinCAT 3 licenses are automatically copied to this system.
4. In the **Solution Explorer**, double-click **License** in the **SYSTEM** subtree.

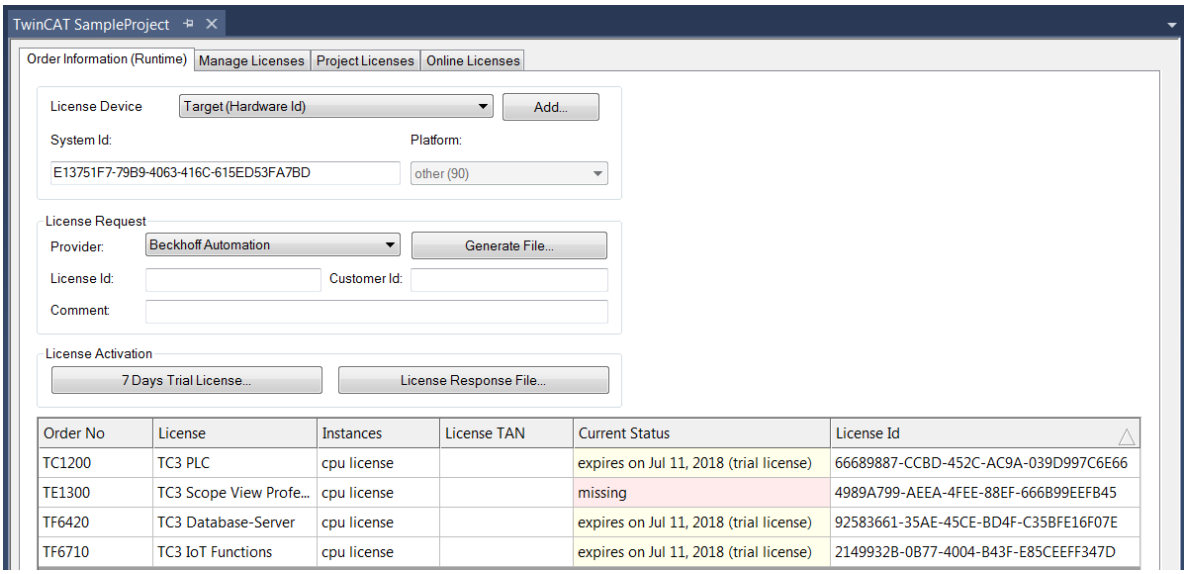


   ⇨ The TwinCAT 3 license manager opens.
5. Open the **Manage Licenses** tab. In the **Add License** column, check the check box for the license you want to add to your project (e.g. “TE1300: TC3 Scope View Professional”).



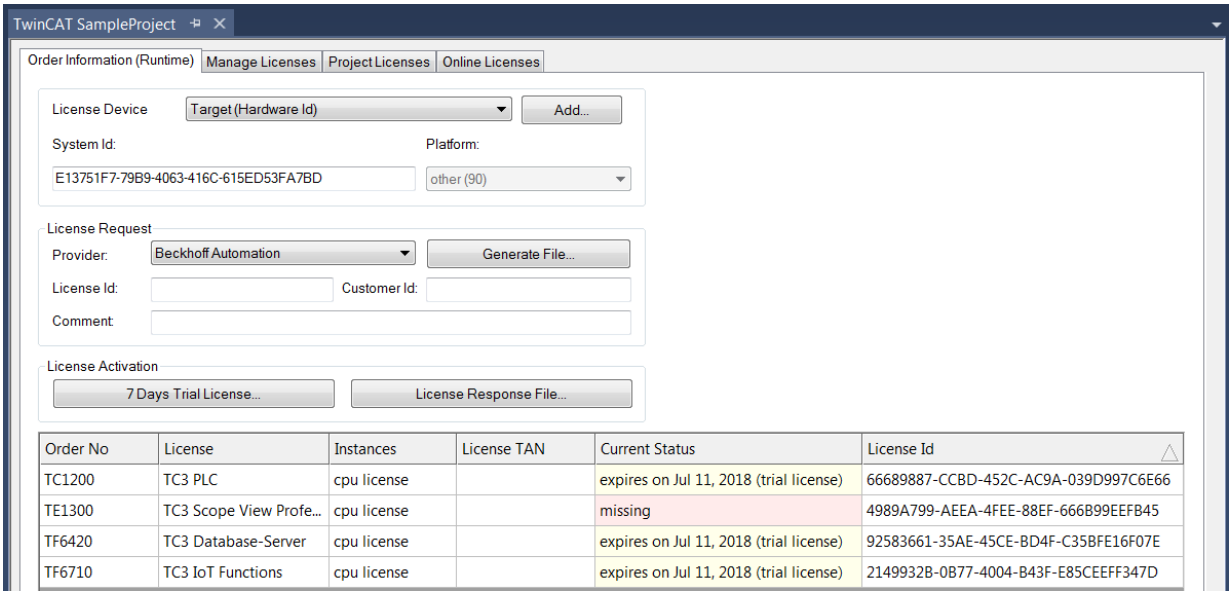
6. Open the **Order Information** tab.

⇨ In the tabular overview of licenses, the previously selected license is displayed with the status “missing”.



A TwinCAT 3 license is generally linked to two indices describing the platform to be licensed:
System ID: Uniquely identifies the device
Platform level: Defines the performance of the device
The corresponding **System Id** and **Platform** fields cannot be changed.

7. Enter the order number (**License Id**) for the license to be activated and optionally a separate order number (**Customer Id**), plus an optional comment for your own purposes (**Comment**). If you do not know your Beckhoff order number, please contact your Beckhoff sales contact.

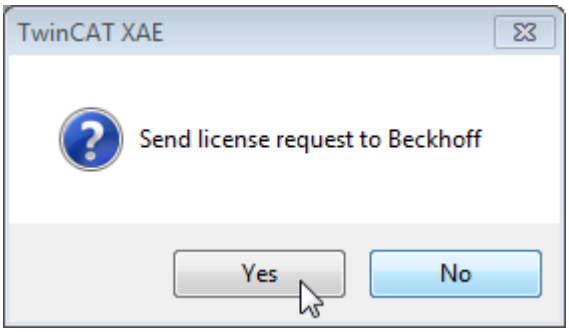


8. Click the **Generate File**... button to create a License Request File for the listed missing license.

   ⇨ A window opens, in which you can specify where the License Request File is to be stored. (We recommend accepting the default settings.)

9. Select a location and click **Save**.

⇨ A prompt appears asking whether you want to send the License Request File to the Beckhoff license server for verification:



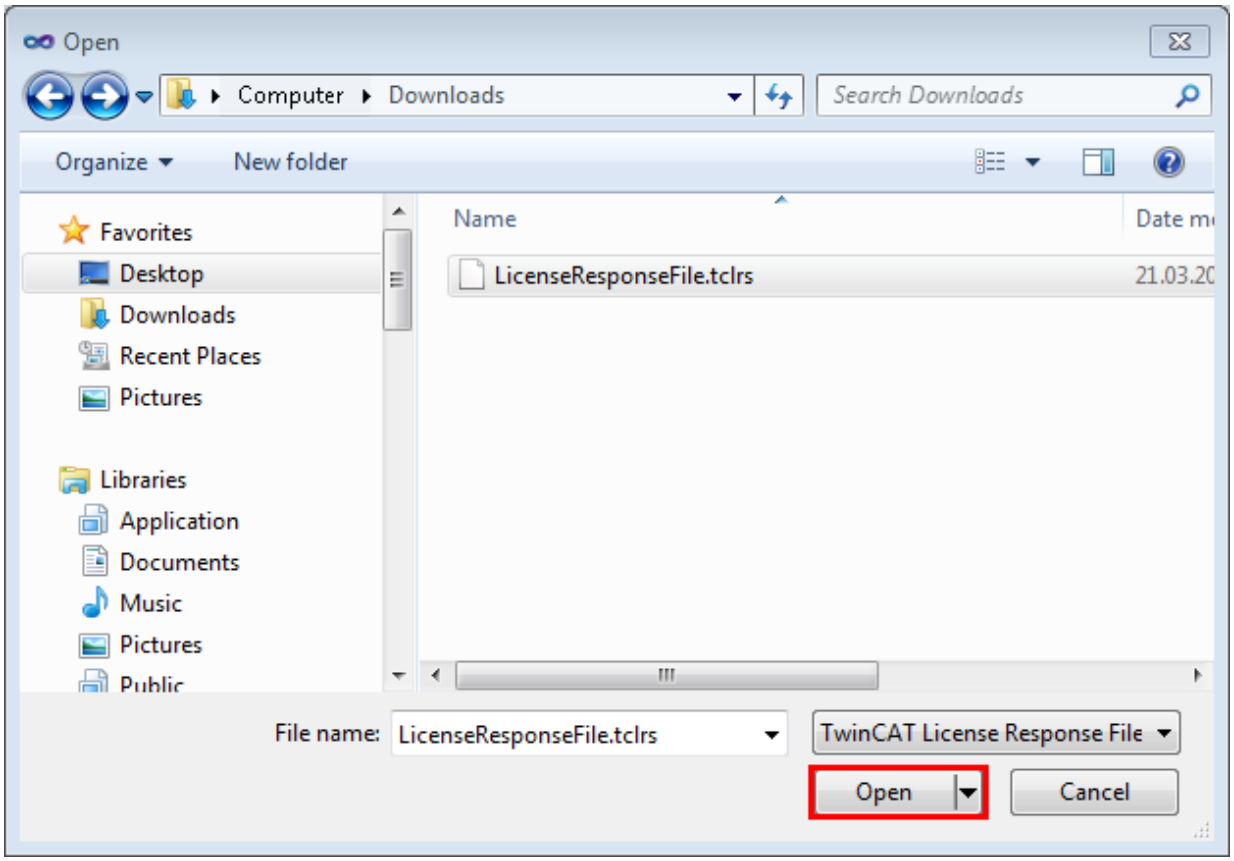
- Click **Yes** to send the License Request File. A prerequisite is that an email program is installed on your computer and that your computer is connected to the internet. When you click **Yes**, the system automatically generates a draft email containing the License Request File with all the necessary information.
- Click **No** if your computer does not have an email program installed on it or is not connected to the internet. Copy the License Request File onto a data storage device (e.g. a USB stick) and send the file from a computer with internet access and an email program to the Beckhoff license server (tclicense@beckhoff.com) by email.

10. Send the License Request File.

    ⇨ The License Request File is sent to the Beckhoff license server. After receiving the email, the server compares your license request with the specified order number and returns a License Response File by email. The Beckhoff license server returns the License Response File to the same email address from which the License Request File was sent. The License Response File differs from the License Request File only by a signature that documents the validity of the license file content. You can view the contents of the License Response File with an editor suitable for XML files (e.g. “XML Notepad”). The contents of the License Response File must not be changed, otherwise the license file becomes invalid.

11. Save the License Response File.
12. To import the license file and activate the license, click **License Response File...** in the **Order Information** tab.

13. Select the License Response File in your file directory and confirm the dialog.



⇨ The License Response File is imported and the license it contains is activated. Existing demo licenses will be removed.

14. Restart the TwinCAT system.

⇨ The license becomes active when TwinCAT is restarted. The product can be used as a full version. During the TwinCAT restart the license file is automatically copied to the directory ... *\TwinCAT\3.1\Target\License* on the respective target system.

3.5 Migration from TwinCAT 2

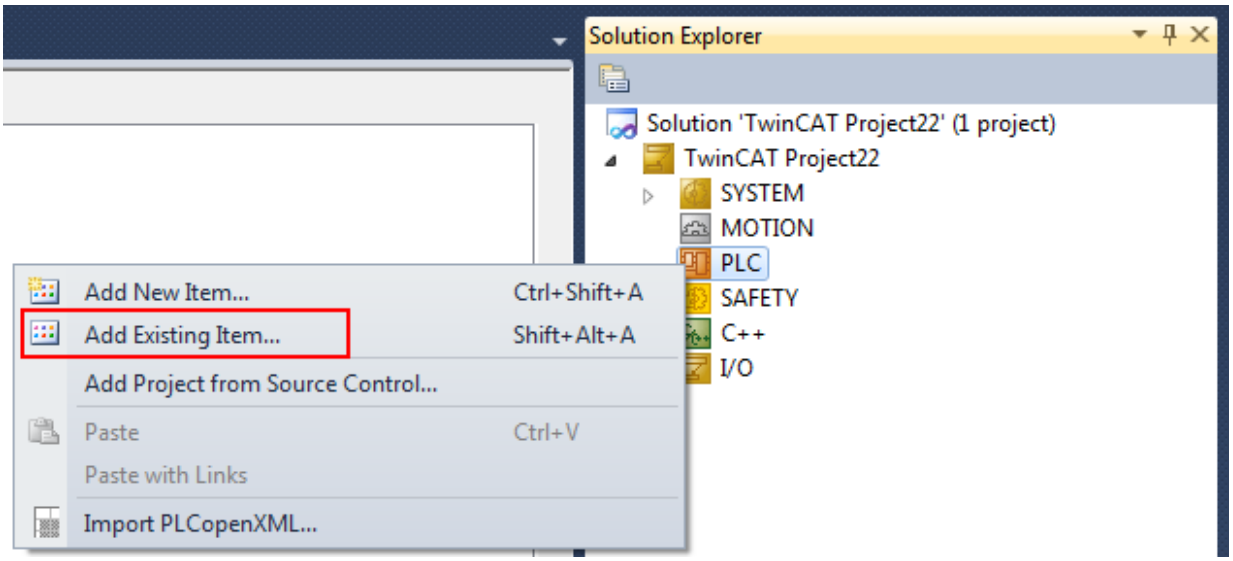
If you would like to migrate an existing TwinCAT 2 PLC project which uses one of the TCP/IP Server's PLC libraries, you need to perform some manual steps to ensure that the TwinCAT 3 PLC converter can process the TwinCAT 2 project file (*.pro). In TwinCAT 2, the Function TCP/IP Server is delivered with three PLC libraries:

- TcpIp.lib
- TcSocketHelper.lib
- TcSnmp.lib

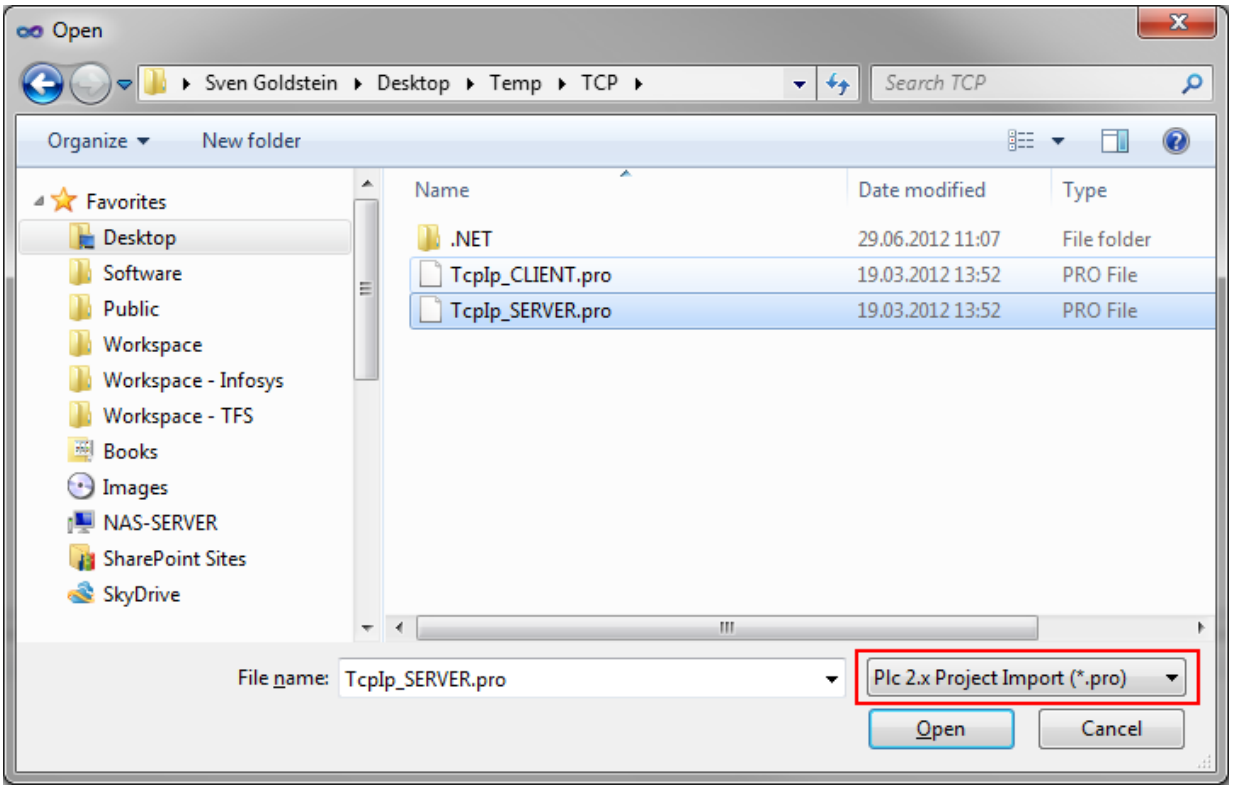
By default, these library files are installed in *C:\TwinCAT\Plc\Lib*. Depending on the library used in your PLC project, you need to copy the corresponding library file to *C:\TwinCAT3\Components\Plc\Converter\Lib* and then perform the following steps:

1. Open the TwinCAT Engineering.
2. Create a new TwinCAT 3 solution.

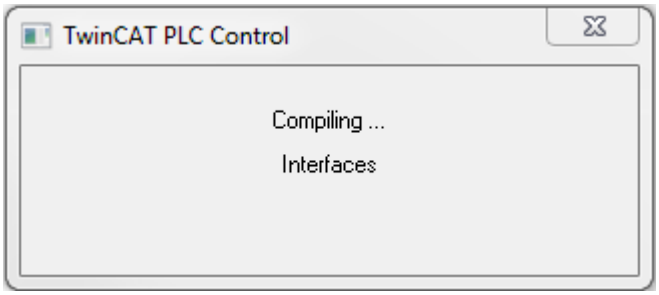
3. Right-click on the "PLC" node and select **Add Existing Item** in the context menu that opens.

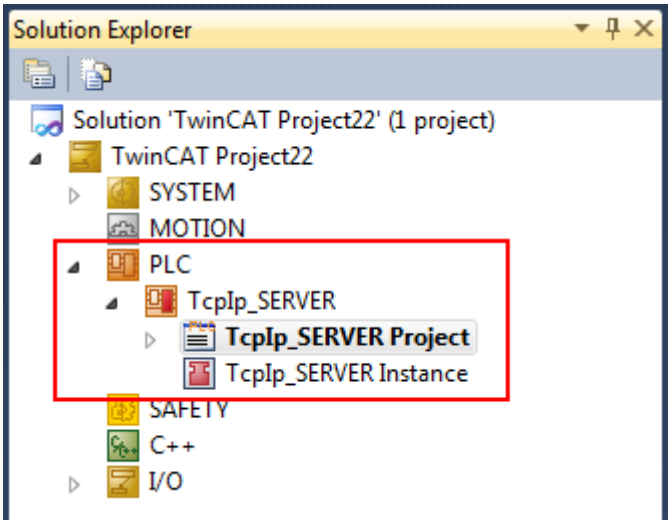


4. In the Open dialog, select the file type "Plc 2.x Project Import (*.pro)", browse to the folder containing your TwinCAT 2 PLC project and select the corresponding.pro file and click **Open**.



⇨ TwinCAT 3 starts the converter process and finally displays the converted PLC project under the “PLC“ node.





4 Technical introduction

This section will give a general overview about the transport protocols TCP and UDP and will also link to the corresponding PLC libraries needed to implement each protocol. Both transport protocols are part of the Internet Protocol suite and therefore an important part of our everyday communication, e.g. the Internet.

Transmission Control Protocol (TCP)

TCP is a connection-oriented transport protocol (OSI layer 4) that can be compared to a telephone connection, where participants have to establish the connection first before data can be transmitted. TCP provides a reliable and ordered delivery of a stream of bytes, therefore it is considered to be a “stream-oriented transport protocol”. The TCP protocol is used for network applications where a receive confirmation is required for the data sent by a client or server. The TCP protocol is well suited for sending larger data quantities and transports a data stream without a defined start and end. For the transmitter this is not a problem since he knows how many data bytes are transmitted. However, the receiver is unable to detect where a message ends within the data stream and where the next data stream starts. A read call on the receiver side only supplies the data currently in the receive buffer (this may be less or more than the data block sent by the other device). Therefore the transmitter has to specify a message structure that is known to the receiver and can be interpreted. In simple cases the message structure may consist of the data and a final control character (e.g. carriage return). The final control character indicates the end of a message. A possible message structure which is indeed often used for transferring binary data with a variable length could be defined as follows: The first data bytes contain a special control character (a so-called start delimiter) and the data length of the subsequent data. This enables the receiver to detect the start and end of the message.

TCP/IP client

A minimum TCP/IP client implementation within the PLC requires the following function blocks:

- An instance of the FB_SocketConnect [▶ 23] and FB_SocketClose [▶ 24] function blocks for establishing and closing the connection to the remote server (Hint: FB_ClientServerConnection [▶ 38] encapsulates the functionality of both function blocks)
- An instance of the FB_SocketSend [▶ 28] and/or FB_SocketReceive [▶ 30] function block for the data exchange with the remote server

TCP/IP server

A minimum TCP/IP server implementation within the PLC requires the following function blocks:

- An instance of the FB_SocketListen [▶ 26] function block for opening the listener socket.
- An instance of the FB_SocketAccept [▶ 27] and FB_SocketClose [▶ 24] function blocks for establishing and closing the connection(s) to the remote clients (Hint: FB_ServerClientConnection [▶ 40] encapsulates the functionality of all three function block)
- An instance of the FB_SocketSend [▶ 28] and/or FB_SocketReceive [▶ 30] function block for the data exchange with the remote clients
- An instance of the FB_SocketCloseAll [▶ 25] function block is required in each PLC runtime system, in which a socket is opened.

The instances of the FB_SocketAccept [▶ 27] and FB_SocketReceive [▶ 30] function blocks are called cyclically (polling), all others are called as required.

User Datagram Protocol (UDP)

UDP is a connection-less protocol, which means that data is sent between network devices without an explicit connection. UDP uses a simple transmission model without implicitly defining workflows for handshaking, reliability, data ordering or congestion control. However, even as this implies that UDP datagrams may arrive out of order, appear duplicated, or congest the wire, UDP is in some cases preffered to TCP, especially in realtime communication because all mentioned features (which are implemented in

TCP) require processing power and therefore time. Because of its connection-less nature, the UDP protocol is well suited for sending small data quantities. UDP is a “packet-oriented/message-oriented transport protocol”, i.e. the sent data block is received on the receiver side as a complete data block.

The following function blocks are required for a minimum UDP client/server implementation:

- An instance of the FB_SocketUdpCreate [▶ 31] and FB_SocketClose [▶ 24] function blocks for opening and closing an UDP socket (Hint: FB_ConnectionlessSocket [▶ 44] encapsulates the functionality of both function)
- An instance of the FB_SocketUdpSendTo [▶ 32] and/or FB_SocketUdpReceiveFrom [▶ 34] function blocks for the data exchange with other devices;
- An instance of the FB_SocketCloseAll [▶ 25] function block in each PLC runtime system, in which a UDP socket is opened

The instances of the FB_SocketUdpReceiveFrom [▶ 34] function block are called cyclically (polling), all others are called as required.

See also: Samples [▶ 55]

5 PLC API

5.1 Function blocks

5.1.1 FB_SocketConnect

Using the function block FB_SocketConnect, a local client can establish a new TCP/IP connection to a remote server via the TwinCAT TCP/IP Connection Server. If successful, a new socket is opened, and the associated connection handle is returned at the hSocket output. The connection handle is required by the function blocks FB_SocketSend [▶ 28] and FB_SocketReceive [▶ 30], for example, in order to exchange data with a remote server. If a connection is no longer required, it can be closed with the function block FB_SocketClose [▶ 24]. Several clients can establish a connection with the remote server at the same time. For each new client, a new socket is opened and a new connection handle is returned. The TwinCAT TCP/IP Connection Server automatically assigns a new IP port number for each client.

VAR_INPUT

```
VAR_INPUT
    sSrvNetId   : T_AmsNetId := '';
    sRemoteHost : T_IPv4Addr := '';
    nRemotePort : UDINT;
    bExecute    : BOOL;
    tTimeout    : TIME := T#45s;(*!!!*)
END_VAR
```

Name	Type	Description
sSrvNetId	T_AmsNetId	String containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.
sRemoteHost	T_IPv4Addr	IP address (Ipv4) of the remote server in the form of a string (e.g., '172.33.5.1'). An empty string can be entered on the local computer for a server.
nRemotePort	UDINT	IP port number of the remote server (e.g., 200).
bExecute	BOOL	The function block is activated by a positive edge at this input.
tTimeout	TIME	Maximum time allowed for the execution of the function block.

Setting the maximum execution time of the function block

Do not set the value "tTimeout" too low, as timeout times of > 30 s can occur in case of a network interruption. If the value is too low, command execution would be interrupted prematurely, and ADS error code 1861 (timeout elapsed) would be returned instead of the Winsocket error WSAETIMED-OUT.

VAR_OUTPUT

```
VAR_OUTPUT
    bBusy     : BOOL;
    bError    : BOOL;
    nErrId    : UDINT;
    hSocket   : T_HSOCKET;
END_VAR
```

Name	Type	Description
bBusy	BOOL	This output is active if the function block is activated. It remains active until acknowledgement.
bError	BOOL	If an error should occur during the transfer of the command, then this output is set once the bBusy output was reset.
nErrId	UDINT	If an bError output is set, this parameter returns the TwinCAT TCP/IP Connection Server error number [▶ 91].
hSocket	T_HSOCKET	TCP/IP connection handle [▶ 52] for the newly opened local client socket.

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.1.2 FB_SocketClose

The function block FB_SocketClose can be used to close an open TCP/IP or UDP socket.

TCP/IP: The listener socket is opened with the function block FB_SocketListen [▶ 26], a local client socket with FB_SocketConnect [▶ 23] and a remote client socket with FB_SocketAccept [▶ 27].

UDP: The UDP socket is opened with the function block FB_SocketUdpCreate [▶ 31].

VAR_INPUT

```
VAR_INPUT
    sSrvNetId   : T_AmsNetId := '';
    hSocket     : T_HSOCKET;
    bExecute    : BOOL;
    tTimeout    : TIME := T#5s;
END_VAR
```

Name	Type	Description
sSrvNetId	T_AmsNetId	String containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.
hSocket	T_HSOCKET	• TCP/IP: Connection handle [▶ 52] of the listener, remote or local client socket to be closed. • UDP: Connection handle of the UDP socket.
bExecute	BOOL	The function block is activated by a positive edge at this input.
tTimeout	TIME	Maximum time allowed for the execution of the function block.

VAR_OUTPUT

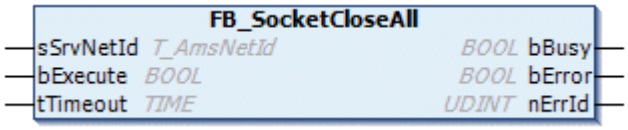
```
VAR_OUTPUT
    bBusy     : BOOL;
    bError    : BOOL;
    nErrId    : UDINT;
END_VAR
```

Name	Type	Description
bBusy	BOOL	This output is active if the function block is activated. It remains active until acknowledgement.
bError	BOOL	If an error should occur during the transfer of the command, then this output is set once the bBusy output was reset.
nErrId	UDINT	If an bError output is set, this parameter returns the TwinCAT TCP/IP Connection Server error number [▶ 91].

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.1.3 FB_SocketCloseAll



If TwinCAT is restarted or stopped, the TwinCAT TCP/IP Connection Server is also stopped. Any open sockets (TCP/IP and UDP connection handles) are closed automatically. The PLC program is reset after a "PLC reset", a "Rebuild all..." or a new "Download", and the information about already opened sockets (connection handles) is no longer available in the PLC. Any open connections can then no longer be closed properly.

The function block FB_SocketCloseAll can be used to close all connection handles (TCP/IP and UDP sockets) that were opened by a PLC runtime system. This means that, if FB_SocketCloseAll is called in one of the tasks of the first runtime systems (port 801), all sockets that were opened in the first runtime system are closed. In each PLC runtime system that uses the socket function blocks, an instance of FB_SocketCloseAll should be called during the PLC start (see below).

VAR_INPUT

```
VAR_INPUT
    sSrvNetId     : T_AmsNetId := '';
    bExecute      : BOOL;
    tTimeout      : TIME := T#5s;
END_VAR
```

Name	Type	Description
sSrvNetId	T_AmsNetId	String containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.
bExecute	BOOL	The function block is activated by a positive edge at this input.
tTimeout	TIME	Maximum time allowed for the execution of the function block.

VAR_OUTPUT

```
VAR_OUTPUT
    bBusy     : BOOL;
    bError    : BOOL;
    nErrId    : UDINT;
END_VAR
```

Name	Type	Description
bBusy	BOOL	This output is active if the function block is activated. It remains active until acknowledgement.
bError	BOOL	If an error should occur during the transfer of the command, then this output is set once the bBusy output was reset.
nErrId	UDINT	If an bError output is set, this parameter returns the TwinCAT TCP/IP connection server error number [▶ 91].

Example of an implementation in ST

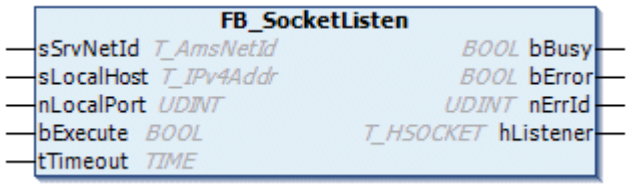
The following program code is used to properly close the connection handles (sockets) that were open before a "PLC reset" or "Download" before a PLC restart.

```
PROGRAM MAIN
VAR
    fbSocketCloseAll : FB_SocketCloseAll;
    bCloseAll        : BOOL := TRUE;
END_VAR
IF bCloseAll THEN(*On PLC reset or program download close all old connections *)
    bCloseAll := FALSE;
    fbSocketCloseAll( sSrvNetId:= '', bExecute:= TRUE, tTimeout:= T#10s );
ELSE
    fbSocketCloseAll( bExecute:= FALSE );
END_IF
IF NOT fbSocketCloseAll.bBusy THEN
(*...
... continue program execution...
...*)
END_IF
```

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.1.4 FB_SocketListen



Using the function block FB_SocketListen, a new listener socket can be opened via the TwinCAT TCP/IP Connection Server. Via a listener socket, the TwinCAT TCP/IP Connection Server can 'listen' for incoming connection requests from remote clients. If successful, the associated connection handle is returned at the hListener output. This handle is required by the function block FB_SocketAccept [▶ 27]. If a listener socket is no longer required, it can be closed with the function block FB_SocketClose [▶ 24]. The listener sockets on an individual computer must have unique IP port numbers.

VAR_INPUT

```
VAR_INPUT
    sSrvNetId   : T_AmsNetId := '';
    sLocalHost  : T_IPv4Addr := '';
    nLocalPort  : UDINT;
    bExecute    : BOOL;
    tTimeout    : TIME := T#5s;
END_VAR
```

Name	Type	Description
sSrvNetId	T_AmsNetId	String containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.
sLocalHost	T_IPv4Addr	Local server IP address (Ipv4) in the form of a string (e.g., '172.13.15.2'). For a server on the local computer (default), an empty string may be entered.
nLocalPort	UDINT	Local server IP port (e.g., 200).
bExecute	BOOL	The function block is activated by a positive edge at this input.
tTimeout	TIME	Maximum time allowed for the execution of the function block.

VAR_OUTPUT

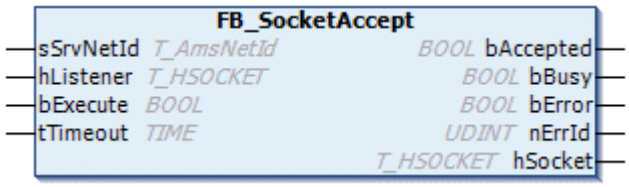
```
VAR_OUTPUT
    bBusy      : BOOL;
    bError     : BOOL;
    nErrId     : UDINT;
    hListener  : T_HSOCKET;
END_VAR
```

Name	Type	Description
bBusy	BOOL	This output is active if the function block is activated. It remains active until acknowledgement.
bError	BOOL	If an error should occur during the transfer of the command, then this output is set once the bBusy output was reset.
nErrId	UDINT	If an bError output is set, this parameter returns the TwinCAT TCP/IP connection server error number [▶ 91].
hListener	T_HSOCKET	Connection handle [▶ 52] for the new listener socket.

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.1.5 FB_SocketAccept



The remote client connection requests arriving at the TwinCAT TCP/IP Connection Server have to be acknowledged (accepted). The function block FB_SocketAccept accepts the incoming remote client connection requests, opens a new remote client socket and returns the associated connection handle. The connection handle is required by the function blocks FB_SocketSend [▶ 28] and FB_SocketReceive [▶ 30] in order to exchange data with the remote client, for example. All incoming connection requests first have to be accepted. If a connection is no longer required or undesirable, it can be closed with the function block FB_SocketClose [▶ 24].

A server implementation requires at least one instance of this function block. This instance has to be called cyclically (polling) from a PLC task. The function block can be activated via a positive edge at the bExecute input (e.g., every 5 seconds).

If successful, the bAccepted output is set, and the connection handle to the new remote client is returned at the hSocket output. No error is returned if there are no new remote client connection requests. Several remote clients can establish a connection with the server at the same time. The connection handles of

several remote clients can be retrieved sequentially via several function block calls. Each connection handle for a remote client can only be retrieved once. It is recommended to keep the connection handles in a list (array). New connections are added to the list, and closed connections must be removed from the list.

VAR_INPUT

```
VAR_INPUT
    sSrvNetId     : T_AmsNetId := '';
    hListener     : T_HSOCKET;
    bExecute      : BOOL;
    tTimeout      : TIME := T#5s;
END_VAR
```

Name	Type	Description
sSrvNetId	T_AmsNetId	String containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.
hListener	T_HSOCKET	Connection handle [▶ 52] of the listener sockets. This handle must first be requested via the function block FB_SocketListen [▶ 26].
bExecute	BOOL	The function block is activated by a positive edge at this input.
tTimeout	TIME	Maximum time allowed for the execution of the function block.

VAR_OUTPUT

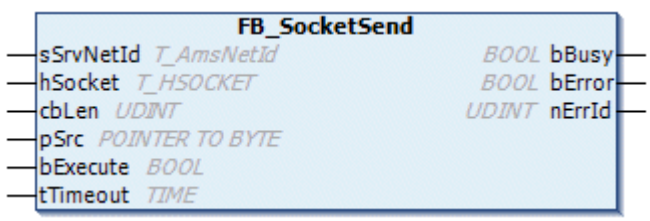
```
VAR_OUTPUT
    bAccepted : BOOL;
    bBusy     : BOOL;
    bError    : BOOL;
    nErrId    : UDINT;
    hSocket   : T_HSOCKET;
END_VAR
```

Name	Type	Description
bAccepted	BOOL	This output is set if a new connection to a remote client was established.
bBusy	BOOL	This output is active if the function block is activated. It remains active until acknowledgement.
bError	BOOL	If an error should occur during the transfer of the command, then this output is set once the bBusy output was reset.
nErrId	UDINT	If an bError output is set, this parameter returns the TwinCAT TCP/IP Connection Server [▶ 91]error number.
hSocket	T_HSOCKET	Connection handle [▶ 52] of a new remote client.

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.1.6 FB_SocketSend



Using the function block FB_SocketSend, data can be sent to a remote client or remote server via the TwinCAT TCP/IP Connection Server. A remote client connection will first have to be established via the function block FB_SocketAccept [▶ 27], or a remote server connection via the function block FB_SocketConnect [▶ 23].

VAR_INPUT

```
VAR_INPUT
    sSrvNetId  : T_AmsNetId := '';
    hSocket    : T_HSOCKET;
    cbLen      : UDINT;
    pSrc       : POINTER TO BYTE;
    bExecute   : BOOL;
    tTimeout   : TIME := T#5s;
END_VAR
```

Name	Type	Description
sSrvNetId	T_AmsNetId	String containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.
hSocket	T_HSOCKET	Connection handle [▶ 52] of the communication partner to which data are to be sent.
cbLen	UDINT	Number of date to be sent in bytes.
pSrc	POINTER TO BYTE	Address (pointer) of the send buffer.
bExecute	BOOL	The function block is activated by a positive edge at this input.
tTimeout	TIME	Maximum time allowed for the execution of the function block.

Setting the execution time of the function block

If the transmit buffer of the socket is full, for example because the remote communication partner receives the transmitted data not quickly enough or large quantities of data are transmitted, the FB_SocketSend function block will return ADS timeout error 1861 after the tTimeout time. In this case, the value of the tTimeout input variable has to be increased accordingly.

VAR_OUTPUT

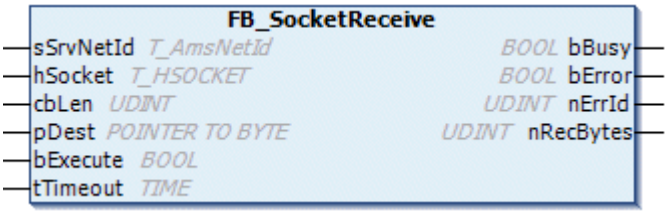
```
VAR_OUTPUT
    bBusy      : BOOL;
    bError     : BOOL;
    nErrId     : UDINT;
END_VAR
```

Name	Type	Description
bBusy	BOOL	This output is active if the function block is activated. It remains active until acknowledgement.
bError	BOOL	If an error should occur during the transfer of the command, then this output is set once the bBusy output was reset.
nErrId	UDINT	If an bError output is set, this parameter returns the TwinCAT TCP/IP Connection Server error number [▶ 91].

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.1.7 FB_SocketReceive



Using the function block FB_SocketReceive, data from a remote client or remote server can be received via the TwinCAT TCP/IP Connection Server. A remote client connection will first have to be established via the function block FB_SocketAccept [▶ 27], and a remote server connection via the function block FB_SocketConnect [▶ 23]. The data can be received or sent in fragmented form (i.e. in several packets) within a TCP/IP network. It is therefore possible that not all data may be received with a single call of the FB_SocketReceive instance. For this reason, the instance has to be called cyclically (polling) within the PLC task, until all required data have been received. During this process, an rising edge is generated at the bExecute input, e.g. every 100 ms. If successful, the data received last are copied into the receive buffer. The nRecBytes output returns the number of the last successfully received data bytes. If no new data could be read during the last call, the function block returns no error and nRecBytes == zero.

In a simple protocol for receiving, for example, a zero-terminated string on a remote server, the function block FB_SocketReceive, for example, will have to be called repeatedly until the zero termination was detected in the data received.

Timeout value setup

If the remote device was disconnected from the TCP/IP network (on the remote side only) while the local device is still connected to the TCP/IP network, the FB_SocketReceive function block returns no error and no data. The socket is still open, but no data are received. In this case, the application may possibly wait for remaining data bytes indefinitely. It is recommended to implement timeout monitoring in the PLC application. If not all data were received after a certain period, e.g. 10 seconds, the connection has to be closed and reinitialised.

VAR_INPUT

```
VAR_INPUT
    sSrvNetId  : T_AmsNetId := '';
    hSocket    : T_HSOCKET;
    cbLen      : UDINT;
    pDest      : POINTER TO BYTE;
    bExecute   : BOOL;
    tTimeout   : TIME := T#5s;
END_VAR
```

Name	Type	Description
sSrvNetId	T_AmsNetId	String containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.
hSocket	T_HSOCKET	Connection handle [▶ 52] of the communication partner from which data are to be received.
cbLen	UDINT	Maximum available buffer size (in bytes) for the data to be read.
pDest	POINTER TO BYTE	Address (pointer) of the receive buffer.
bExecute	BOOL	The function block is activated by a positive edge at this input.
tTimeout	TIME	Maximum time allowed for the execution of the function block.

VAR_OUTPUT

```
VAR_OUTPUT
    bBusy      : BOOL;
    bError     : BOOL;
```

```
    nErrId    : UDINT;
    nRecBytes : UDINT;
END_VAR
```

Name	Type	Description
bBusy	BOOL	This output is active if the function block is activated. It remains active until acknowledgement.
bError	BOOL	If an error should occur during the transfer of the command, then this output is set once the bBusy output was reset.
nErrId	UDINT	If an bError output is set, this parameter returns the TwinCAT TCP/IP Connection Server error number.
nRecBytes	UDINT	Number of the last successfully received data bytes.

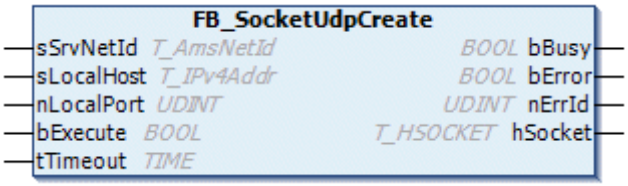
Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

Also see about this

- Overview of the error codes [▶ 91]

5.1.8 FB_SocketUdpCreate



The function block FB_SocketUdpCreate can be used to open a client/server socket for the User Datagram Protocol (UDP). If successful, a new socket is opened, and the associated socket handle is returned at the hSocket output. The handle is required by the function blocks FB_SocketUdpSendTo [▶ 32] and FB_SocketUdpReceiveFrom [▶ 34] in order to exchange data with a remote server, for example. If a UDP socket is no longer required, it can be closed with the function block FB_SocketClose [▶ 24]. The port address nLocalHost is internally reserved by the TCP/IP Connection Server for the UDP protocol (a "bind" is carried out). Several network adapters may exist in a PC. The input parameter sLocalHost determines the network adapter to be used. If the sLocalHost input variable is ignored (empty string), the TCP/IP Connection Server uses the default network adapter. This is usually the first network adapter from the list of the network adapters in the Control Panel.

Automatically created network connections

If an empty string was specified for sLocalHost when FB_SocketUdpCreate was called and the PC was disconnected from the network, the system will open a new socket under the software loopback IP address: '127.0.0.1'.

Automatically created network connections with several network adapters

If two or more network adapters are installed in the PC and an empty string was specified as sLocalHost, and the default network adapter was then disconnected from the network, the new socket will be opened under the IP address of the second network adapter.

Setting a network address

In order to prevent the sockets from being opened under a different IP address, you can specify the sLocalHost address explicitly or check the returned address in the handle variable (hSocket), close the socket and re-open it.

VAR_INPUT

```
VAR_INPUT
    sSrvNetId  : T_AmsNetId := '';
    sLocalHost : T_IPv4Addr := '';
    nLocalPort : UDINT;
    bExecute   : BOOL;
    tTimeout   : TIME:= T#5s;
END_VAR
```

Name	Type	Description
sSrvNetId	T_AmsNetId	String containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.
sLocalHost	T_IPv4Addr	Local IP address (Ipv4) of the UDP client/server socket as a string (e.g., '172.33.5.1'). An empty string may be specified for the default network adapter.
nLocalPort	UDINT	Local IP port number of the UDP client/server socket (e.g., 200).
bExecute	BOOL	The function block is activated by a positive edge at this input.
tTimeout	TIME	Maximum time allowed for the execution of the function block.

VAR_OUTPUT

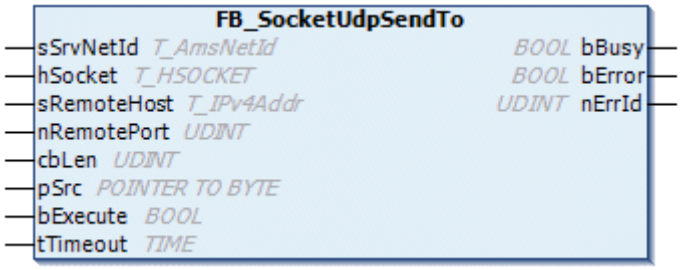
```
VAR_OUTPUT
    bBusy     : BOOL;
    bError    : BOOL;
    nErrId    : UDINT;
    hSocket   : T_HSOCKET;
END_VAR
```

Name	Type	Description
bBusy	BOOL	This output is active if the function block is activated. It remains active until acknowledgement.
bError	BOOL	If an error should occur during the transfer of the command, then this output is set once the bBusy output was reset.
nErrId	UDINT	If an bError output is set, this parameter returns the TwinCAT TCP/IP Connection Server error number [▶ 91].
hSocket	T_HSOCKET	Handle of the newly opened UDP client/server socket [▶ 52].

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.1.9 FB_SocketUdpSendTo



The function block FB_SocketUdpSendTo can be used to send UDP data to a remote device via the TwinCAT TCP/IP Connection Server. The UDP socket must first be opened with the function block FB_SocketUdpCreate [▶ 31].

VAR_INPUT

```
VAR_INPUT
    sSrvNetId   : T_AmsNetId := '';
    hSocket     : T_HSOCKET;
    sRemoteHost : T_IPv4Addr;
    nRemotePort : UDINT;
    cbLen       : UDINT;
    pSrc        : POINTER TO BYTE;
    bExecute    : BOOL;
    tTimeout    : TIME := T#5s;
END_VAR
```

Name	Type	Description
sSrvNetId	T_AmsNetId	String containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.
hSocket	T_HSOCKET	Handle of an opened UDP socket [▶ 52].
sRemoteHost	T_IPv4Addr	IP address (Ipv4) in string form (e.g., '172.33.5.1') of the remote device to which data is to be sent. An empty string can be entered on the local computer for a device.
nRemotePort	UDINT	IP port number (e.g., 200) of the remote device to which data is to be sent.
cbLen	UDINT	Number of date to be sent in bytes. The maximum number of data bytes to be sent is limited to 8192 bytes (constant TCPADS_MAXUDP_BUFFSIZE in the library in order to save memory space).
pSrc	POINTER TO BYTE	Address (pointer) of the send buffer.
bExecute	BOOL	The function block is activated by a positive edge at this input.
tTimeout	TIME	Maximum time allowed for the execution of the function block.

Setting the size of the received data bytes

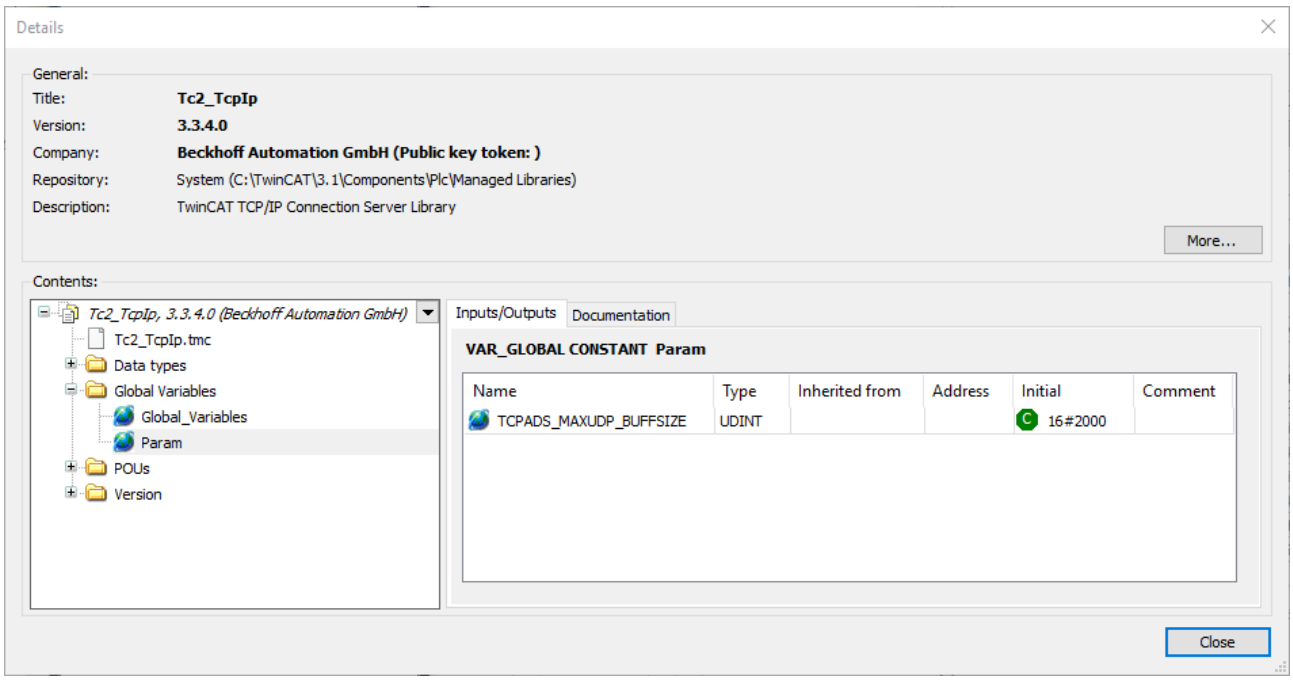
Available in product version: TwinCAT TCP/IP Connection Server v1.0.50 or higher: The maximum number of data bytes to be received can be increased (only when absolutely unavoidable).

TwinCAT 2

1. Redefine global constants in the PLC project (in the example, the maximum number of data bytes to be received is to be increased to 32000):
   ```
   VAR_GLOBAL CONSTANT
       TCPADS_MAXUDP_BUFFSIZE : UDINT := 32000;
   END_VAR
   ```
2. Activate the option **Replace constants** in the dialog of the TwinCAT PLC controller (Project > Options ... > Build).
3. Rebuild Project.

TwinCAT 3

In TwinCAT 3, this value can be edited via a parameter list of the PLC library (from version 3.3.4.0).



VAR_OUTPUT

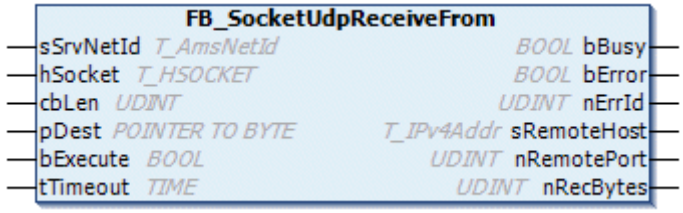
```
VAR_OUTPUT
    bBusy     : BOOL;
    bError    : BOOL;
    nErrId    : UDINT;
END_VAR
```

Name	Type	Description
bBusy	BOOL	This output is active if the function block is activated. It remains active until acknowledgement.
bError	BOOL	If an error should occur during the transfer of the command, then this output is set once the bBusy output was reset.
nErrId	UDINT	If an bError output is set, this parameter returns the TwinCAT TCP/IP Connection Server error number [▶ 91].

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.1.10 FB_SocketUdpReceiveFrom



Using the function block FB_SocketUdpReceiveFrom, data from an open UDP socket can be received via the TwinCAT TCP/IP Connection Server. The UDP socket must first be opened with the function block FB_SocketUdpCreate [▶ 31]. The instance of the FB_SocketUdpReceive function block has to be called cyclically (polling) within the PLC task. During this process, an rising edge is generated at the bExecute

input, e.g. every 100ms. If successful, the data received last are copied into the receive buffer. The nRecBytes output returns the number of the last successfully received data bytes. If no new data could be read during the last call, the function block returns no error and nRecBytes == zero.

VAR_INPUT

```
VAR_INPUT
    sSrvNetId : T_AmsNetId := '';
    hSocket   : T_HSOCKET;
    cbLen     : UDINT;
    pDest     : POINTER TO BYTE;
    bExecute  : BOOL;
    tTimeout  : TIME := T#5s;
END_VAR
```

Name	Type	Description
sSrvNetId	T_AmsNetId	String containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.
hSocket	T_HSOCKET	Handle of an opened UDP socket [▶ 52], whose data are to be received.
cbLen	UDINT	Maximum available buffer size (in bytes) for the data to be read. The maximum number of data bytes to be received is limited to 8192 bytes (constant TCPADS_MAXUDP_BUFFSIZE in the library in order to save memory space).
pDest	POINTER TO BYTE	Address (pointer) of the receive buffer.
bExecute	BOOL	The function block is activated by a positive edge at this input.
tTimeout	TIME	Maximum time allowed for the execution of the function block.

Setting the size of the received data bytes

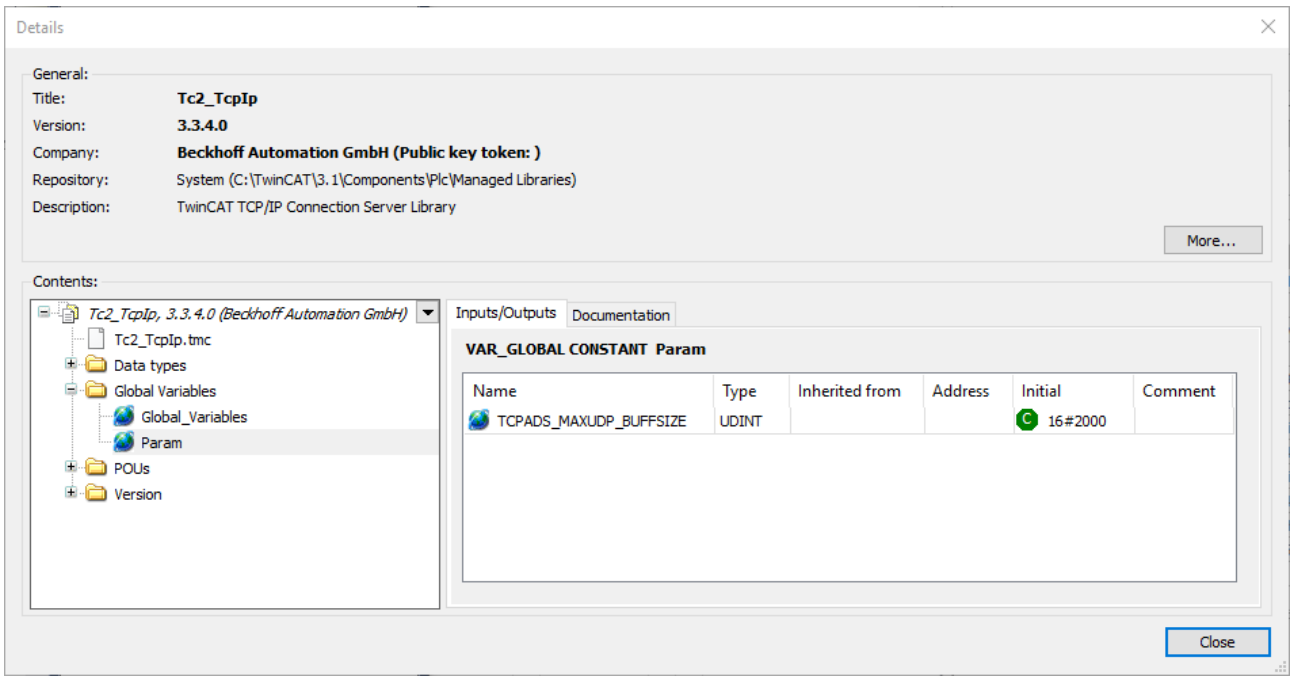
Available in product version: TwinCAT TCP/IP Connection Server v1.0.50 or higher: The maximum number of data bytes to be received can be increased (only when absolutely unavoidable).

TwinCAT 2

1. Redefine global constants in the PLC project (in the example, the maximum number of data bytes to be received is to be increased to 32000):
   ```
   VAR_GLOBAL CONSTANT
       TCPADS_MAXUDP_BUFFSIZE : UDINT := 32000;
   END_VAR
   ```
2. Activate the option **Replace constants** in the dialog of the TwinCAT PLC controller (Project > Options ... > Build).
3. Rebuild Project.

TwinCAT 3

In TwinCAT 3, this value can be edited via a parameter list of the PLC library (from version 3.3.4.0).



VAR_OUTPUT

```
VAR_OUTPUT
    bBusy       : BOOL;
    bError      : BOOL;
    nErrId      : UDINT;
    sRemoteHost : T_IPv4Addr := '';
    nRemotePort : UDINT;
    nRecBytes   : UDINT;
END_VAR
```

Name	Type	Description
bBusy	BOOL	This output is active if the function block is activated. It remains active until acknowledgement.
bError	BOOL	If an error should occur during the transfer of the command, then this output is set once the bBusy output was reset.
nErrId	UDINT	If an bError output is set, this parameter returns the TwinCAT TCP/IP Connection Server error number [▶ 91].
sRemoteHost	T_IPv4Addr	If successful, IP address (Ipv4) of the remote device whose data were received.
nRemotePort	UDINT	If successful, IP port number of the remote device whose data were received (e.g., 200).
nRecBytes	UDINT	Number of data bytes last successfully received.

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.1.11 FB_SocketUdpAddMulticastAddress

Binds the Server to a Multicast IP address so that Multicast UDP packets can be received. This function blocks requires a previously established UDP socket handle, which can be requested using the function block FB_SocketUdpCreate [▶ 31].

VAR_INPUT

```
VAR_INPUT
    sSrvNetId      : T_AmsNetId := '';
    hSocket        : T_HSOCKET;
    sMulticastAddr : STRING(15);
    bExecute       : BOOL;
    tTimeout       : TIME := T#5s;
END_VAR
```

Name	Type	Description
sSrvNetId	T_AmsNetId	String containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.
hSocket	T_HSOCKET	Connection handle [▶ 52] of the listener socket. This handle must first be requested with the function block FB_SocketUdpCreate [▶ 31].
sMulticastAddr	T_IPv4Addr	Multicast IP address to which the binding should take place.
bExecute	BOOL	The function block is activated by a positive edge at this input.
tTimeout	TIME	Maximum time allowed for the execution of the function block.

VAR_OUTPUT

```
VAR_OUTPUT
    bBusy     : BOOL;
    bError    : BOOL;
    nErrId    : UDINT;
END_VAR
```

Name	Type	Description
bBusy	BOOL	This output is active if the function block is activated. It remains active until acknowledgement.
bError	BOOL	If an error should occur during the transfer of the command, then this output is set once the bBusy output was reset.
nErrId	UDINT	If an bError output is set, this parameter returns the TwinCAT TCP/IP connection server error number [▶ 91].

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.1.12 FB_SocketUdpDropMulticastAddress

Removes the binding to a Multicast IP address, which has been added previously via the function block FB_SocketUdpAddMulticastAddress [▶ 36].

VAR_INPUT

```
VAR_INPUT
    sSrvNetId      : T_AmsNetId := '';
    hSocket        : T_HSOCKET;
    sMulticastAddr : STRING(15);
    bExecute       : BOOL;
    tTimeout       : TIME := T#5s;
END_VAR
```

Name	Type	Description
sSrvNetId	T_AmsNetId	String containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.
hSocket	T_HSOCKET	Connection handle [▶ 52] of the listener socket. This handle must first be requested with the function block FB_SocketUdpCreate [▶ 31].
sMulticastAddr	T_IPv4Addr	Multicast IP address to which the binding should take place.
bExecute	BOOL	The function block is activated by a positive edge at this input.
tTimeout	TIME	Maximum time allowed for the execution of the function block.

VAR_OUTPUT

```
VAR_OUTPUT
    bBusy     : BOOL;
    bError    : BOOL;
    nErrId    : UDINT;
END_VAR
```

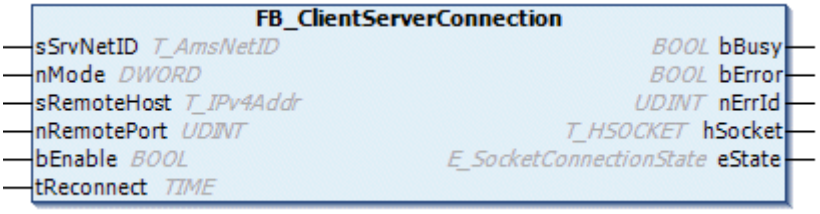
Name	Type	Description
bBusy	BOOL	This output is active if the function block is activated. It remains active until acknowledgement.
bError	BOOL	If an error should occur during the transfer of the command, then this output is set once the bBusy output was reset.
nErrId	UDINT	If an bError output is set, this parameter returns the TwinCAT TCP/IP Connection Server error number [▶ 91].

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.1.13 Helper

5.1.13.1 FB_ClientServerConnection



The function block FB_ClientServerConnection can be used to manage (establish or remove) a client connection. FB_ClientServerConnection simplifies the implementation of a client application by encapsulating the functionality of the two function blocks FB_SocketConnect [▶ 23] and FB_SocketClose [▶ 24]

internally. The integrated debugging output of the connection status facilitates troubleshooting in the event of configuration or communication errors. In addition, a minimum client application only requires an instance of the FB_SocketSend [▶ 28] function block and/or an instance of the FB_SocketReceive [▶ 30] function block.

In the first step, a typical client application establishes the connection with the server via the FB_ClientServerConnection function block. In the next step instances of FB_SocketSend and/or FB_SocketReceive can be used to exchange data with the server. When a connection is closed depends on the requirements of the application.

VAR_INPUT

```
VAR_INPUT
    sSrvNetID   : T_AmsNetID := '';
    nMode       : DWORD := 0;
    sRemoteHost : T_IPv4Addr := '';
    nRemotePort : UDINT;
    bEnable     : BOOL;
    tReconnect  : TIME := T#45s;(*!!!*)
END_VAR
```

Name	Type	Description
sSrvNetID	T_AmsNetID	String containing the AMS network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.
nMode	DWORD	parameter flags (modes). The permissible parameters are listed here and can be combined by ORing: CONNECT_MODE_ENABLEDBG: Activates logging of debugging messages in the application log. In order to view the debugging messages open the TwinCAT System Manager and activate log view.
sRemoteHost	T_IPv4Addr	IP address (Ipv4) of the remote server in the form of a string (e.g., '172.33.5.1'). An empty string can be entered on the local computer for a server.
nRemotePort	UDINT	IP port number of the remote server (e.g., 200).
bEnable	BOOL	As long as this input is TRUE, the system attempts to establish a new connection at regular intervals until a connection was established successfully. Once established, a connection can be closed again with FALSE.
tReconnect	TIME	Cycle time used by the function block to try and establish the connection.

Setting the cycle time for the connection

The tReconnect value should not be set too low, since timeout periods of > 30 s may occur in the event of a network interruption. If the value is too low, command execution would be interrupted prematurely, and ADS error code 1861 (timeout elapsed) would be returned instead of the Winsocket error WSAETIMEDOUT.

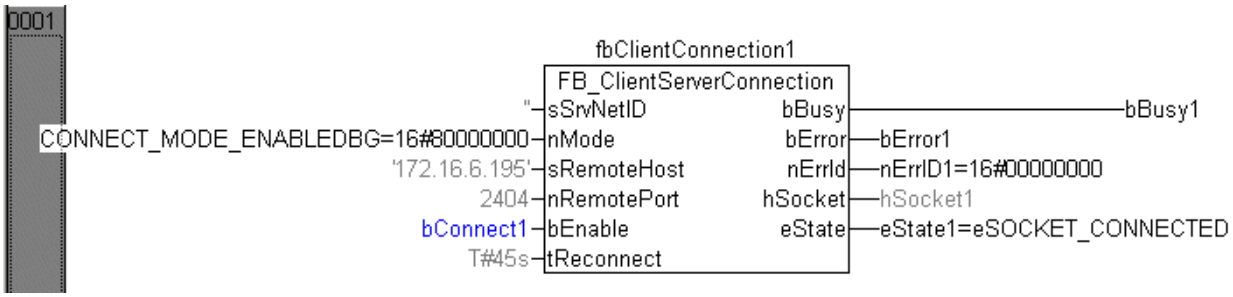
VAR_OUTPUT

```
VAR_OUTPUT
    bBusy      : BOOL;
    bError     : BOOL;
    nErrId     : UDINT;
    hSocket    : T_HSOCKET;
    eState     : E_SocketConnectionState := eSOCKET_DISCONNECTED;
END_VAR
```

Name	Type	Description
bBusy	BOOL	TRUE, as long as the function block is active.
bError	BOOL	Becomes TRUE if an error code occurs.
nErrID	UDINT	If an bError output is set, this parameter returns the TwinCAT TCP/IP Connection Server error number [▶ 91].
hSocket	T_HSOCKET	Connection handle [▶ 52] for the newly opened local client socket. If successful, this variable is transferred to the instances of the function blocks FB_SocketSend [▶ 28] and/or FB_SocketReceive [▶ 30].
eState	E_SocketConnectionState	Returns the current connection status [▶ 49].

Example of a call in FBD

```
PROGRAM MAIN
VAR
    fbClientConnection1 : FB_ClientServerConnection;
    bConnect1           : BOOL;
    bBusy1              : BOOL;
    bError1             : BOOL;
    nErrID1             : UDINT;
    hSocket1            : T_HSOCKET;
    eState1             : E_SocketConnectionState;
END_VAR
```

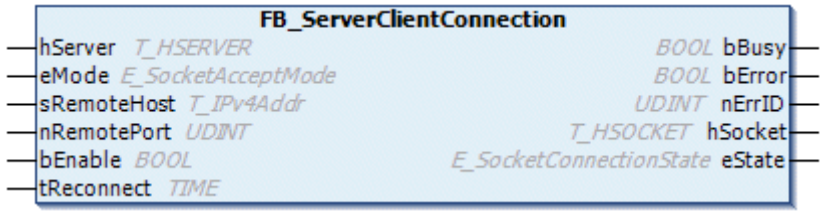


You can find further applications examples (and source code) here: Examples [▶ 55]

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.1.13.2 FB_ServerClientConnection



The function block FB_ServerClientConnection can be used to manage (establish or remove) a server connection. FB_ServerClientConnection simplifies the implementation of a server application by encapsulating the functionality of the three function blocks FB_SocketListen [▶ 26], FB_SocketAccept [▶ 27] and FB_SocketClose [▶ 24] internally. The integrated debugging output of the connection status facilitates troubleshooting in the event of configuration or communication errors. In addition, a minimum server application only requires an instance of the FB_SocketSend [▶ 28] function block and/or an instance of the FB_SocketReceive [▶ 30] function block.

In the first step, a typical server application establishes the connection with the client via the FB_ServerClientConnection function block (more precisely, the server application accepts the incoming connection request). In the next step, instances of FB_SocketSend and/or FB_SocketReceive can be used to exchange data with the server. When a connection is closed depends on the requirements of the application.

VAR_IN_OUT

```
VAR_IN_OUT
    hServer     : T_HSERVER;
END_VAR
```

Name	Type	Description
hServer	hServer	Server handle [▶ 52]. This input variable has to be initialized via the F_CreateServerHnd [▶ 45] function.

VAR_INPUT

```
VAR_INPUT
    eMode       : E_SocketAcceptMode := eACCEPT_ALL;
    sRemoteHost : T_IPv4Addr := '';
    nRemotePort : UDINT := 0;
    bEnable     : BOOL;
    tReconnect  : TIME := T#1s;
END_VAR
```

Name	Type	Description
eMode	E_SocketAcceptMode	Defines whether all or only certain connections are to be accepted [▶ 49].
sRemoteHost	T_IPv4Addr	IP address (Ipv4) in string form (e.g., '172.33.5.1') of the remote client whose connection is to be accepted. For a client on the local computer an empty string may be specified.
nRemotePort	UDINT	IP port number (e.g., 200) of the remote client whose connection is to be accepted.
bEnable	BOOL	As long as this input is TRUE, the system attempts to establish a new connection at regular intervals until a connection was established successfully. Once established, a connection can be closed again with FALSE.
tReconnect	TIME	Cycle time used by the function block to try and establish a connection.

VAR_OUTPUT

```
VAR_OUTPUT
    bBusy     : BOOL;
    bError    : BOOL;
    nErrID    : UDINT;
    hSocket   : T_HSOCKET;
    eState    : E_SocketConnectionState := eSOCKET_DISCONNECTED;
END_VAR
```

Name	Type	Description
bBusy	BOOL	TRUE, as long as the function block is active.
bError	BOOL	Becomes TRUE if an error code occurs.
nErrId	UDINT	If an bError output is set, this parameter returns the TwinCAT TCP/IP Connection Server error number [▶ 91].
hSocket	T_HSOCKET	Connection handle [▶ 52] for the newly opened remote client socket. If successful, this variable is transferred to the instances of the function blocks FB_SocketSend [▶ 28] and/or FB_SocketReceive [▶ 30].
eState	E_SocketConnectionState	Returns the current connection status [▶ 49].

Example in FBD

The following example illustrates initialization of a server handle variable. The server handle is then transferred to three instances of the FB_ServerClientConnection function block.

```
PROGRAM MAIN
VAR
    hServer             : T_HSERVER;
    bListen             : BOOL;

    fbServerConnection1 : FB_ServerClientConnection;
    bConnect1           : BOOL;
    bBusy1              : BOOL;
    bError1             : BOOL;
    nErrID1             : UDINT;
    hSocket1            : T_HSOCKET;
    eState1             : E_SocketConnectionState;

    fbServerConnection2 : FB_ServerClientConnection;
    bConnect2           : BOOL;
    bBusy2              : BOOL;
    bError2             : BOOL;
    nErrID2             : UDINT;
    hSocket2            : T_HSOCKET;
    eState2             : E_SocketConnectionState;

    fbServerConnection3 : FB_ServerClientConnection;
    bConnect3           : BOOL;
    bBusy3              : BOOL;
    bError3             : BOOL;
    nErrID3             : UDINT;
    hSocket3            : T_HSOCKET;
    eState3             : E_SocketConnectionState;
END_VAR
```

Online View:

The first connection is activated (`bConnect1 = TRUE`), but the connection has not yet been established (passive open).

The second connection has not yet been activated (`bConnect2 = FALSE`) (closed).

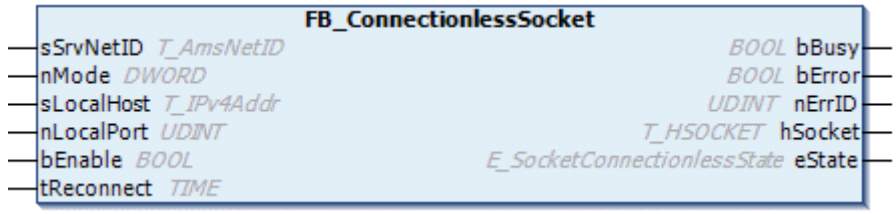
The third connection has been activated (`bConnect3 = TRUE`) and a connection to the remote client has been established.

You can find further applications examples (and source code) here: Examples [▶ 55]

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.1.13.3 FB_ConnectionlessSocket



A UDP socket can be managed (opened/generated and closed) with the function block FB_ConnectionlessSocket. FB_ConnectionlessSocket simplifies the implementation of a UDP application by encapsulating the functionality of the two function blocks FB_SocketUdpCreate [▶ 31] and FB_SocketClose [▶ 24] internally. The integrated debugging output of the socket status facilitates troubleshooting in the event of configuration or communication errors. In addition, a minimum UDP application only requires an instance of the FB_SocketUdpSendTo [▶ 32] function block and/or an instance of the FB_SocketUdpReceiveFrom [▶ 34] function block.

In the first step, a typical UDP application opens a connection-less UDP socket with the FB_ConnectionlessSocket function block. In the next step, instances of FB_SocketUdpSendTo and/or FB_SocketUdpReceiveFrom can be used for exchanging data with another communication device. When a UDP socket is closed depends on the requirements of the application (e.g. in the event of a communication error).

VAR_INPUT

```
VAR_INPUT
    sSrvNetID     : T_AmsNetID := '';
    nMode         : DWORD := 0;
    sLocalHost    : T_Ipv4Addr := '';
    nLocalPort    : UDINT;
    bEnable       : BOOL;
    tReconnect    : TIME := T#45s;(*!!!*)
END_VAR
```

Name	Type	Description
sSrvNetID	T_AmsNetID	String containing the AMS network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.
nMode	DWORD	parameter flags (modes). The permissible parameters are listed here and can be combined by ORing. CONNECT_MODE_ENABLEDBG: Activates logging of debugging messages in the application log. In order to view the debugging messages open the TwinCAT System Manager and activate log view.
sLocalHost	T_Ipv4Addr	IP address (Ipv4) in string form (e.g., '172.33.5.1') of the local network adapter. An empty string may be specified for the default network adapter.
nLocalPort	UDINT	IP port number (e.g., 200) on the local computer.
bEnable	BOOL	As long as this input is TRUE, attempts are made cyclically to open a UDP socket until a connection has been established. An open UDP socket can be closed again with FALSE.
tReconnect	TIME	Cycle time with which the function block tries to open the UDP socket.

Setting the cycle time for the connection

The tReconnect value should not be set too low, since timeout periods of > 30 s may occur in the event of a network interruption. If the value is too low, command execution would be interrupted prematurely, and ADS error code 1861 (timeout elapsed) would be returned instead of the Winsocket error WSAETIMEDOUT.

VAR_OUTPUT

```
VAR_OUTPUT
    bBusy   : BOOL;
    bError  : BOOL;
    nErrId  : UDINT;
    hSocket : T_HSOCKET;
    eState  : E_SocketConnectionlessState := eSOCKET_CLOSED;
END_VAR
```

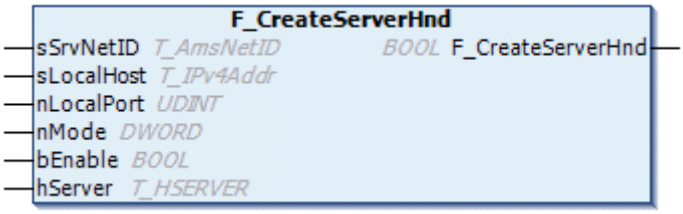
Name	Type	Description
bBusy	BOOL	TRUE, as long as the function block is active.
bError	BOOL	Becomes TRUE if an error code occurs.
nErrID	UDINT	If an bError output is set, this parameter returns the TwinCAT TCP/IP Connection Server error number [▶ 91].
hSocket	T_HSOCKET	Connection handle [▶ 52] for the newly opened UDP socket. If successful, this variable is transferred to the instances of the function blocks FB_SocketUdpSendTo [▶ 32] and/or FB_SocketUdpReceiveFrom [▶ 34].
eState	E_SocketConnectionlessState	Returns the current connection status [▶ 49].

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.2 Functions

5.2.1 F_CreateServerHnd



The function F_CreateServerHnd is used to initialise/set the internal parameters of a server handle variable hServer. The server handle is then transferred to the instances of the FB_ServerClientConnection [▶ 40] function block via VAR_IN_OUT. An instance of the FB_ServerClientConnection function block can be used to manage (establish or remove) a sever connection in a straightforward manner. The same server handle can be transferred to several instances of the FB_ServerClientConnection function block, in order to enable the server to establish several concurrent connections.

FUNCTION F_CreateServerHnd : BOOL

```
VAR_IN_OUT
    hServer         : T_HSERVER;
END_VAR
VAR_INPUT
    sSrvNetID       : T_AmsNetID := '';
    sLocalHost      : STRING(15) := '';
    nLocalPort      : UDINT := 0;
    nMode           : DWORD := LISTEN_MODE_CLOSEALL (* OR CONNECT_MODE_ENABLEDBG*);
 bEnable            : BOOL := TRUE;
END_VAR
```

Name	Type	Description
hServer	T_HSERVER	Server handle [▶ 52] variable whose internal parameters are to be initialized.
sSrvNetID	T_AmsNetID	String containing the AMS network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.
sLocalHost	T_IPv4Addr	Local server IP address (Ipv4) in the form of a string (e.g. '172.13.15.2'). For a server on the local computer (default), an empty string may be entered.
nLocalPort	UDINT	Local server IP port (e.g. 200).
nMode	DWORD	parameter flags (modes). The permissible parameters are listed here and can be combined by ORing. LISTEN_MODE_CLOSEALL: All previously opened socket connections are closed (default). CONNECT_MODE_ENABLEDBG: Activates logging of debugging messages in the application log. In order to view the debugging messages open the TwinCAT System Manager and activate log view.
bEnable	BOOL	This input determines the behavior of the listener socket. A listener socket opened beforehand remains open as long as this input is TRUE. If this input is FALSE, the listener socket is closed automatically, but only once the last (previously) accepted connection was also closed.

Return value

TRUE	No error
FALSE	Error – incorrect parameter value

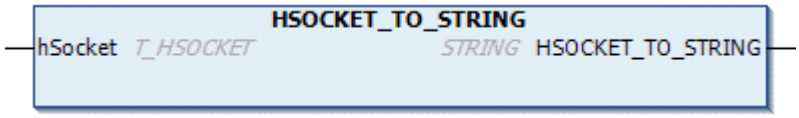
Example:

See FB_ServerClientConnection [▶ 40].

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.2.2 HSOCKET_TO_STRING



The function converts the connection handle of type T_HSOCKET to a string (e.g. for debug outputs).

The returned string has the following format: "Handle:0xA[BCD] Local:a[aa].b[bb].c[cc].d[dd]:port Remote:a[aa].b[bb].c[cc].d[dd]:port".

Example: "Handle:0x4001 Local:172.16.6.195:28459 Remote:172.16.6.180:2404"

FUNCTION HSOCKET_TO_STRING : STRING

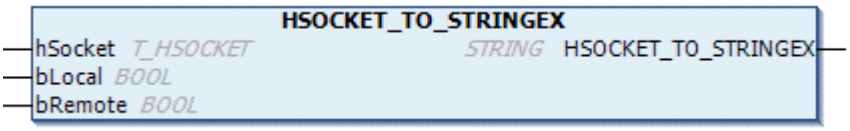
```
VAR_INPUT
    hSocket : T_HSOCKET;
END_VAR
```

Name	Type	Description
hSocket	T_HSOCKET	The connection handle [▶ 52] to be converted.

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.2.3 HSOCKET_TO_STRINGEX



The function converts the connection handle of type T_HSOCKET to a string (e.g. for debug outputs).

The returned string has the following format: "Handle:0xA[BCD] Local:a[aa].b[bb].c[cc].d[dd]:port Remote:a[aa].b[bb].c[cc].d[dd]:port".

Example: "Handle:0x4001 Local:172.16.6.195:28459 Remote:172.16.6.180:2404"

The parameters bLocal and bRemote determine whether the local and/or remote address information should be included in the returned string.

FUNCTION HSOCKET_TO_STRINGEX : STRING

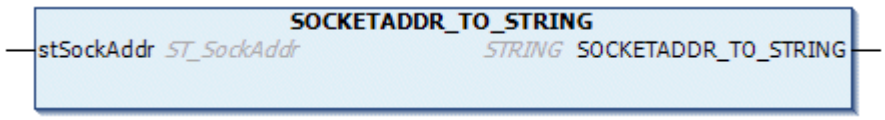
```
VAR_INPUT
    hSocket : T_HSOCKET;
    bLocal  : BOOL;
    bRemote : BOOL;
END_VAR
```

Name	Type	Description
hSocket	T_HSOCKET	The connection handle [▶ 52] to be converted.
bLocal	BOOL	TRUE: include the local address, FALSE: exclude the local address.
bRemote	BOOL	TRUE: include the remote address, FALSE: exclude the remote address.

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.2.4 SOCKETADDR_TO_STRING



The function converts a variable of type ST_SockAddr to a string (e.g. for debug outputs).

The returned string has the following format: "a[aa].b[bb].c[cc].d[dd]:port"

Example: "172.16.6.195:80"

FUNCTION SOCKETADDR_TO_STRING : STRING

```
VAR_INPUT
    stSockAddr : ST_SockAddr;
END_VAR
```

Name	Type	Description
stSockeAddr	ST_SockAddr	The variable to be converted.

See ST_SockAddr [▶ 51]

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.2.5 [Obsolete]

5.2.5.1 F_GetVersionTcpIp

This function can be used to read PLC library version information.

FUNCTION F_GetVersionTcpIp : UINT

```
VAR_INPUT
    nVersionElement : INT;
END_VAR
```

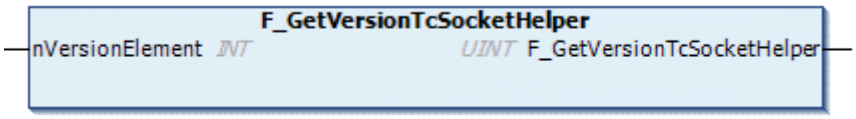
nVersionElement : Version element to be read. Possible parameters:

- 1 : major number;
- 2 : minor number;
- 3 : revision number;

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.2.5.2 F_GetVersionTcSocketHelper



This function reads version information from the PLC library.

FUNCTION F_GetVersionTcSocketHelper : UINT

```
VAR_INPUT
    nVersionElement : INT;
END_VAR
```

nVersionElement : Version element, that is to be read. Possible parameters:

- 1 : major number;
- 2 : minor number;

- 3 : revision number;

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.3 Data types

5.3.1 E_SocketAcceptMode

```
TYPE E_SocketAcceptMode:
(* Connection accept modes *)
(
    eACCEPT_ALL, (* Accept connection to all remote clients *)
    eACCEPT_SEL_HOST, (* Accept connection to selected host address *)
    eACCEPT_SEL_PORT, (* Accept connection to selected port address *)
    eACCEPT_SEL_HOST_PORT (* Accept connection to selected host and port address *)
);
END_TYPE
```

The variable E_SocketAcceptMode defines which connections are to be accepted by a server.

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.3.2 E_SocketConnectionState

```
TYPE E_SocketConnectionState:
(
    eSOCKET_DISCONNECTED,
    eSOCKET_CONNECTED,
    eSOCKET_SUSPENDED
);
END_TYPE
```

TCP/IP Socket Connection Status (eSOCKET_SUSPENDED == the status changes e.g. from eSOCKET_CONNECTED => eSOCKET_DISCONNECTED).

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.3.3 E_SocketConnectionlessState

```
TYPE E_SocketConnectionlessState:
(
    eSOCKET_CLOSED,
    eSOCKET_CREATED,
    eSOCKET_TRANSIENT
);
END_TYPE
```

Status information of a connection-less UDP socket (eSOCKET_TRANSIENT == the status changes from eSOCKET_CREATED => eSOCKET_CLOSED, for example).

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.3.4 E_WinsockError

```
TYPE E_WinsockError :
 (
    WSOK,
    WSAEINTR        := 10004 ,
(* A blocking operation was interrupted by a call to WSACancelBlockingCall. *)
    WSAEBADF        := 10009 ,(* The file handle supplied is not valid. *)
    WSAEACCES     := 10013 ,
(* An attempt was made to access a socket in a way forbidden by its access permissions. *)
    WSAEFAULT     := 10014 ,
(* The system detected an invalid pointer address in attempting to use a pointer argument in a call.
 *)
    WSAEINVAL     := 10022 ,(* An invalid argument was supplied. *)
    WSAEMFILE     := 10024 ,(* Too many open sockets. *)
    WSAEWOULDBLOCK     := 10035 ,(* A non-
blocking socket operation could not be completed immediately. *)
    WSAEINPROGRESS     := 10036 ,(* A blocking operation is currently executing. *)
    WSAEALREADY     := 10037 ,(* An operation was attempted on a non-
blocking socket that already had an operation in progress. *)
    WSAENOTSOCK     := 10038 ,(* An operation was attempted on something that is not a socket. *)
    WSAEDESTADDRREQ     := 10039 ,
(* A required address was omitted from an operation on a socket. *)
    WSAEMSGSIZE     := 10040 ,
(* A message sent on a datagram socket was larger than the internal message buffer or some other net
work limit, or the buffer used to receive a datagram into was smaller than the datagram itself. *)
    WSAEPROTOTYPE     := 10041 ,
(* A protocol was specified in the socket function call that does not support the semantics of the s
ocket type requested. *)
    WSAENOPROTOOPT     := 10042 ,
(* An unknown, invalid, or unsupported option or level was specified in a getsockopt or setsockopt c
all. *)
    WSAEPROTONOSUPPORT := 10043 ,
(* The requested protocol has not been configured into the system, or no implementation for it exist
s. *)
    WSAESOCKTNOSUPPORT := 10044 ,
(* The support for the specified socket type does not exist in this address family. *)
    WSAEOPNOTSUPP     := 10045 ,
(* The attempted operation is not supported for the type of object referenced. *)
    WSAEPFNOSUPPORT     := 10046 ,
(* The protocol family has not been configured into the system or no implementation for it exists. *
)
    WSAEAFNOSUPPORT     := 10047 ,
(* An address incompatible with the requested protocol was used. *)
    WSAEADDRINUSE     := 10048 ,(* Only one usage of each socket address (protocol/network address/
port) is normally permitted. *)
    WSAEADDRNOTAVAIL     := 10049 ,(* The requested address is not valid in its context. *)
    WSAENETDOWN     := 10050 ,(* A socket operation encountered a dead network. *)
    WSAENETUNREACH     := 10051 ,(* A socket operation was attempted to an unreachable network. *)
    WSAENETRESET     := 10052 ,(* The connection has been broken due to keep-
alive activity detecting a failure while the operation was in progress. *)
    WSAECONNABORTED     := 10053 ,
(* An established connection was aborted by the software in your host machine. *)
    WSAECONNRESET     := 10054 ,(* An existing connection was forcibly closed by the remote host. *)
    WSAENOBUFS     := 10055 ,
(* An operation on a socket could not be performed because the system lacked sufficient buffer space
 or because a queue was full. *)
    WSAEISCONN     := 10056 ,(* A connect request was made on an already connected socket. *)
    WSAENOTCONN     := 10057 ,
(* A request to send or receive data was disallowed because the socket is not connected and (when se
nding on a datagram socket using a sendto call) no address was supplied. *)
    WSAESHUTDOWN     := 10058 ,
(* A request to send or receive data was disallowed because the socket had already been shut down in
 that direction with a previous shutdown call. *)
    WSAETOOMANYREFS     := 10059 ,(* Too many references to some kernel object. *)
      WSAETIMEDOUT     := 10060 ,
(* A connection attempt failed because the connected party did not properly respond after a period o
f time, or established connection failed because connected host has failed to respond. *)
    WSAECONNREFUSED     := 10061 ,
(* No connection could be made because the target machine actively refused it. *)
```

```
    WSAELOOP        := 10062 ,(* Cannot translate name. *)
    WSAENAMETOOLONG     := 10063 ,(* Name component or name was too long. *)
    WSAEHOSTDOWN     := 10064 ,
(* A socket operation failed because the destination host was down. *)
    WSAEHOSTUNREACH     := 10065 ,(* A socket operation was attempted to an unreachable host. *)
    WSAENOTEMPTY     := 10066 ,(* Cannot remove a directory that is not empty. *)
    WSAEPROCLIM     := 10067 ,
(* A Windows Sockets implementation may have a limit on the number of applications that may use it s
imultaneously. *)
    WSAEUSERS     := 10068 ,(* Ran out of quota. *)
    WSAEDQUOT     := 10069 ,(* Ran out of disk quota. *)
    WSAESTALE     := 10070 ,(* File handle reference is no longer available. *)
    WSAEREMOTE     := 10071 ,(* Item is not available locally. *)
    WSASYSNOTREADY     := 10091 ,
(* WSAStartup cannot function at this time because the underlying system it uses to provide network
services is currently unavailable. *)
    WSAVERNOTSUPPORTED := 10092 ,(* The Windows Sockets version requested is not supported. *)
    WSANOTINITIALISED := 10093 ,
(* Either the application has not called WSAStartup, or WSAStartup failed. *)
    WSAEDISCON     := 10101 ,
(* Returned by WSARecv or WSARecvFrom to indicate the remote party has initiated a graceful shutdown
 sequence. *)
    WSAENOMORE     := 10102 ,(* No more results can be returned by WSALookupServiceNext. *)
    WSAECANCELLED     := 10103 ,
(* A call to WSALookupServiceEnd was made while this call was still processing. The call has been ca
nceled. *)
    WSAEINVALIDPROCTABLE := 10104 ,(* The procedure call table is invalid. *)
    WSAEINVALIDPROVIDER := 10105 ,(* The requested service provider is invalid. *)
    WSAEPROVIDERFAILEDINIT := 10106 ,
(* The requested service provider could not be loaded or initialized. *)
    WSASYSCALLFAILURE := 10107 ,(* A system call that should never fail has failed. *)
    WSASERVICE_NOT_FOUND := 10108 ,
(* No such service is known. The service cannot be found in the specified name space. *)
    WSATYPE_NOT_FOUND := 10109 ,(* The specified class was not found. *)
    WSA_E_NO_MORE     := 10110 ,(* No more results can be returned by WSALookupServiceNext. *)
    WSA_E_CANCELLED     := 10111 ,
(* A call to WSALookupServiceEnd was made while this call was still processing. The call has been ca
nceled. *)
    WSAEREFUSED     := 10112 ,(* A database query failed because it was actively refused. *)
    WSAHOST_NOT_FOUND := 11001 ,(* No such host is known. *)
    WSATRY_AGAIN     := 11002 ,
(* This is usually a temporary error during hostname resolution and means that the local server did
not receive a response from an authoritative server. *)
    WSANO_RECOVERY     := 11003 ,(* A non-recoverable error occurred during a database lookup. *)
    WSANO_DATA     := 11004 (* The requested name is valid and was found in the database, but it doe
s not have the correct associated data being resolved for. *)
 );
END_TYPE
```

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.3.5 ST_SockAddr

Structure with address information for an open socket.

```
TYPE ST_SockAddr : (* Local or remote endpoint address *)
STRUCT
    nPort : UDINT; (* Internet Protocol (IP) port. *)
    sAddr : STRING(15); (* String containing an (Ipv4) Internet Protocol dotted address. *)
END_STRUCT
END_TYPE
```

Name	Type	Description
nPort	UDINT	Internet Protocol (IP) port
sAddr	STRING(15)	Internet Protocol address separated by periods (Ipv4) in the form of a string e.g.: "172.34.12.3"

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.3.6 T_HSERVER

The variable of this type represents a TCP/IP Server Handle. The Handle has to be initialized with F_CreateServerHnd [▶ 45] bevor it can be used. In doing so the internal parameters of variables T_HSERVER are set.

Preserve the default structure elements

The structure elements are not to be written or changed.

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.3.7 T_HSOCKET

Variables of this type represent a connection handle or a handle of an open socket. Via this handle, data can be sent to or received from a socket. The handle can be used to close an open socket.

```
TYPE T_HSOCKET
STRUCT
    handle     : UDINT;
    localAddr  : ST_SockAddr; (* Local address *)
    remoteAddr : ST_SockAddr; (* Remote endpoint address *)
END_STRUCT
END_TYPE
```

Name	Type	Description
handle	UDINT	Internal TwinCAT TCP/IP Connection Server socket handle
localAddr	ST_SockAddr	Local socket address [▶ 51]
remoteAddr	ST_SockAddr	Remote socket address [▶ 51]

The following sockets can be opened and closed via the TwinCAT TCP/IP Connection Server: listener socket, remote client socket, or local client socket. Depending on which of these sockets was opened by the TwinCAT TCP/IP Connection Server, suitable address information is entered into the localAddr and remoteAddr variables.

Connection handle on the server side

- The function block FB_SocketListen [▶ 26] opens a listener socket and returns the connection handle of the listener socket.
- The connection handle of the listener sockets is transferred to the function block FB_SocketAccept [▶ 27]. FB_SocketAccept will then return the connection handles of the remote clients.
- The function block FB_SocketAccept returns a new connection handle for each connected remote client.
- The connection handle is then transferred to the function blocks FB_SocketSend [▶ 28] and/or FB_SocketReceive [▶ 30], in order to be able to exchange data with the remote clients.
- A connection handle of a remote client that is not desirable or no longer required is transferred to the function block FB_SocketClose [▶ 24], which closes the remote client socket.

- A listener socket connection handle that is no longer required is also transferred to the function block FB_SocketClose, which closes the listener socket.

Connection handle on the client side

- The function block FB_SocketConnect [▶ 23] returns the connection handle of a local client socket.
- The connection handle is then transferred to the function blocks FB_SocketSend [▶ 28] and FB_SocketReceive [▶ 30], in order to be able to exchange data with a remote server.
- The same connection handle is then transferred to the function block FB_SocketClose [▶ 24], in order to close a connection that is no longer required.

The function block FB_SocketCloseAll [▶ 25] can be used to close all connection handles (sockets) that were opened by a PLC runtime system. This means that, if FB_SocketCloseAll is called in one of the tasks of the first runtime systems (port 801), all sockets that were opened in the first runtime system are closed.

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.4 Global constants

5.4.1 Global Variables

```
VAR_GLOBAL CONSTANT
    AMSPORT_TCPIPSRV              : UINT := 10201;

    TCPADS_IGR_CONLIST            : UDINT := 16#80000001;
    TCPADS_IGR_CLOSEBYHDL         : UDINT := 16#80000002;
    TCPADS_IGR_SENDBYHDL          : UDINT := 16#80000003;
    TCPADS_IGR_PEERBYHDL          : UDINT := 16#80000004;
    TCPADS_IGR_RECVBYHDL          : UDINT := 16#80000005;
    TCPADS_IGR_RECVFROMBYHDL      : UDINT := 16#80000006;
    TCPADS_IGR_SENDTOBYHDL        : UDINT := 16#80000007;
    TCPADS_IGR_MULTICAST_ADDBYHDL : UDINT := 16#80000008;
    TCPADS_IGR_MULTICAST_DROPBYHDL: UDINT := 16#80000009;

    TCPADSCONLST_IOF_CONNECT  : UDINT := 1;
    TCPADSCONLST_IOF_LISTEN   : UDINT := 2;
    TCPADSCONLST_IOF_CLOSEALL : UDINT := 3;
    TCPADSCONLST_IOF_ACCEPT   : UDINT := 4;
    TCPADSCONLST_IOF_UDPBIND  : UDINT := 5;

    TCPADS_MAXUDP_BUFFSIZE    : UDINT := 16#2000; (8192 bytes)

    TCPADS_NULL_HSOCKET : T_HSOCKET := ( handle := 0, remoteAddr := ( nPort := 0, sAddr := '' ), loc
alAddr :=  ( nPort := 0, sAddr := '' ) );(* Empty (not initialized) socket *)

    LISTEN_MODE_CLOSEALL   : DWORD := 16#00000001 (* FORCED close of all previous opened sockets *)
    LISTEN_MODE_USEOPENED  : DWORD := 16#00000002 (* Try to use allready opened listener socket *)
    CONNECT_MODE_ENABLEDBG : DWORD := 16#80000000 (* Enables/Disables debugging messages *)
END_VAR
```

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

5.4.2 Library version

All libraries have a specific version. This version is shown in the PLC library repository too.
A global constant contains the library version information:

Global_Version

```
VAR_GLOBAL CONSTANT
    stLibVersion_Tc2_TcpIp : ST_LibVersion;
END_VAR
```

To compare the existing version to a required version the function F_CmpLibVersion (defined in Tc2_System library) is offered.

TwinCAT 2 compatibility

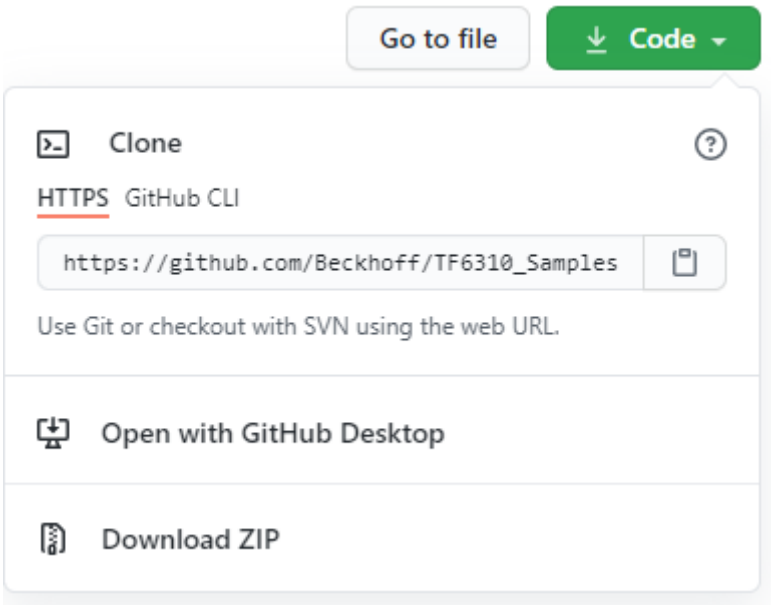
All other possibilities known from TwinCAT2 libraries to query a library version are obsolete!

Requirements

Development environment	Target system type	PLC libraries to include (category group)
TwinCAT v3.1.0	PC, or CX (x86, X64, ARM)	Tc2_TcpIp (communication)

6 Samples

Sample code and configurations for this product can be obtained from the corresponding repository on GitHub: https://github.com/Beckhoff/TF6310_Samples . There you have the option to clone the repository or download a ZIP file containing the sample.



6.1 TCP

6.1.1 Sample01: "Echo" client/server (base blocks)

6.1.1.1 Overview

The following example shows an implementation of an "echo" client/server. The client sends a test string to the server at certain intervals (e.g. every second). The remote server then immediately resends the same string to the client.

In this sample, the client is implemented in the PLC and as a .NET application written in C#. The PLC client can create several instances of the communication, simulating several TCP connections at once. The .NET sample client only establishes one concurrent connection. The server is able to communicate with several clients.

In addition, several instances of the server may be created. Each server instance is then addressed via a different port number which can be used by the client to connect to a specific server instance. The server implementation is more difficult if the server has to communicate with more than one client.

Feel free to use and customize this sample to your needs.

System requirements

- TwinCAT 3 Build 3093 or higher
- TwinCAT 3 Function TF6310 TCP/IP
- If two computers are used to execute the sample (one client and one server), the Function TF6310 needs to be installed on both computers
- If one computer is used to execute the sample, e.g. client and server running in two separate PLC runtimes, both PLC runtimes need to run in separate tasks

- To run the .NET sample client, only .NET Framework 4.0 is needed

Project downloads

https://github.com/Beckhoff/TF6310_Samples/tree/master/PLC/TCP/Sample01

https://github.com/Beckhoff/TF6310_Samples/tree/master/C%23/SampleClient

Project description

The following links provide documentation for the three components. Additionally, an own article explains how to start the PLC samples with step-by-step instructions.

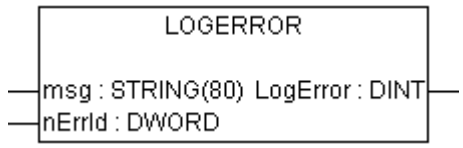
- Integration in TwinCAT and Test [▶ 57] (Starting the PLC samples)
- PLC Client [▶ 60] (PLC client documentation: FB_LocalClient function block [▶ 60])
- PLC Server [▶ 64] (PLC serve documentation: FB_LocalServer function block [▶ 64])
- .NET client [▶ 70] (.NET client documentation: .NET sample client [▶ 70])

Auxiliary functions in the PLC sample projects

In the example projects, several functions, constants and function blocks are used, which are briefly described below:

LogError function

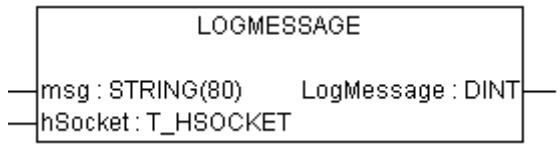
```
FUNCTION LogError : DINT
```



The function writes a message with the error code into the log book of the operating system (Event Viewer). The global variable bLogDebugMessages must first be set to TRUE.

LogMessage function

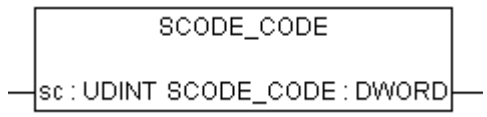
```
FUNCTION LogMessage : DINT
```



The function writes a message into the log book of the operating system (Event Viewer) if a new socket was opened or closed. The global variable bLogDebugMessages must first be set to TRUE.

SCODE_CODE function

```
FUNCTION SCODE_CODE : DWORD
```



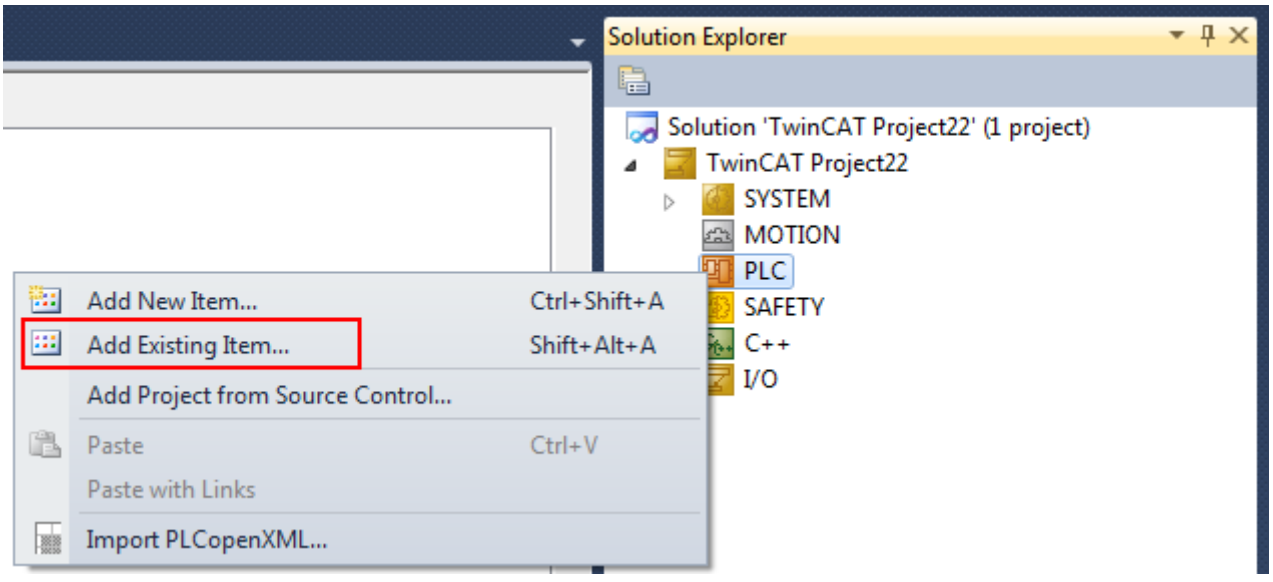
The function masks the lower 16 bits of a Win32 error code returns them.

Global variables

Name	Default value	Description
bLogDebugMessages	TRUE	Activates/deactivates writing of messages into the log book of the operating system
MAX_CLIENT_CONNECTIONS	5	Max. number of remote clients, that can connect to the server at the same time.
MAX_PLCPRJ_RXBUFFER_SIZE	1000	Max. length of the internal receive buffer
PLCPRJ_RECONNECT_TIME	T#3s	Once this time has elapsed, the local client will attempt to re-establish the connection with the remote server
PLCPRJ_SEND_CYCLE_TIME	T#1s	The test string is sent cyclically at these intervals from the local client to the remote server
PLCPRJ_RECEIVE_POLLING_TIME	T#1s	The client reads (polls) data from the server using this cycle
PLCPRJ_RECEIVE_TIMEOUT	T#10s	After this time has elapsed, the local client aborts the reception if no data bytes could be received during this time
PLCPRJ_ERROR_RECEIVE_BUFFER_OVERFLOW	16#8101	Sample project error code: Too many characters without zero termination were received
PLCPRJ_ERROR_RECEIVE_TIMEOUT	16#8102	Sample project error code: No new data could be received within the timeout time (PLCPRJ_RECEIVE_TIMEOUT)

6.1.1.2 Integration in TwinCAT and Test

The following section describes how to prepare and start the PLC server and client. The PLC samples are delivered as TwinCAT 3 PLC project files. To import a PLC project into TwinCAT XAE, first create a new TwinCAT 3 Solution. Then select the command **Add Existing Item** in the context menu of the PLC node and select the downloaded sample file (*Plc 3.x Project archive (*.tpzip)* as file type) in the dialog that opens. After confirming the dialog, the PLC project is added to the solution.



PLC server sample

Create a new TwinCAT 3 solution in TwinCAT XAE and import the TCP/IP server project. Select a target system. Make sure that you have created licenses for TF6310 and that the Function is also installed on the selected target system. Leave the TwinCAT 3 solution open.

```
PROGRAM MAIN
VAR
    fbServer         : FB_LocalServer := ( sLocalHost := '127.0.0.1' (*own IP address!
*), nLocalPort := 200 );
    bEnableServer    : BOOL := TRUE;
    fbSocketCloseAll : FB_SocketCloseAll := ( sSrvNetID := '', tTimeout := DEFAULT_ADS_TIMEOUT );
    bCloseAll        : BOOL := TRUE;
END_VAR

IF bCloseAll THEN (*On PLC reset or program download close all old connections *)
    bCloseAll := FALSE;
    fbSocketCloseAll( bExecute:= TRUE );
ELSE
    fbSocketCloseAll( bExecute:= FALSE );
END_IF

IF NOT fbSocketCloseAll.bBusy THEN
    fbServer( bEnable := bEnableServer );
END_IF
```

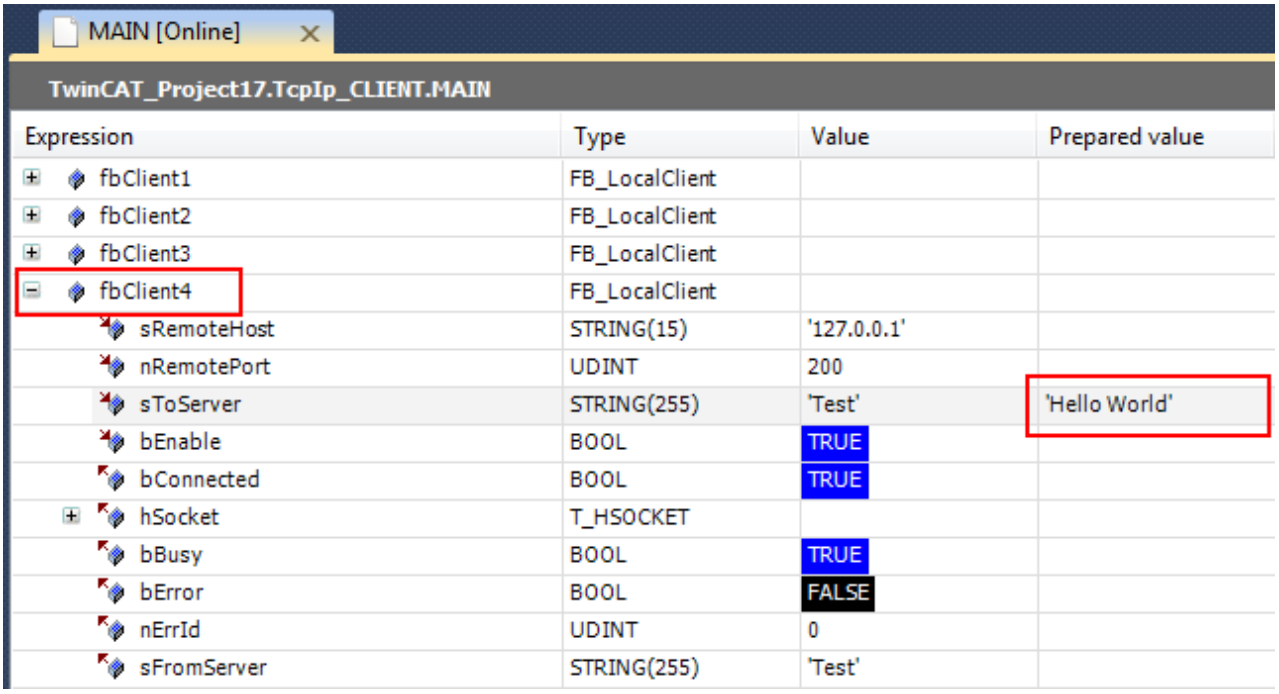
PLC client sample

In the same TwinCAT 3 solution, import the TCP/IP client project as a second PLC project. Link this PLC project to another task than the server sample. The server's IP address has to be adapted to your remote system (initialization values of the sRemoteHost variables). In this case, the server is located on the same machine, therefore enter 127.0.0.1. Activate the configuration, then login and start both PLC projects, beginning with the server.

```
PROGRAM MAIN
VAR
    fbClient1     : FB_LocalClient := ( sRemoteHost:= '127.0.0.1' (* IP address of remote server! *)
, nRemotePort:= 200 );
    fbClient2     : FB_LocalClient := ( sRemoteHost:= '127.0.0.1', nRemotePort:= 200 );
    fbClient3     : FB_LocalClient := ( sRemoteHost:= '127.0.0.1', nRemotePort:= 200 );
    fbClient4     : FB_LocalClient := ( sRemoteHost:= '127.0.0.1', nRemotePort:= 200 );
    fbClient5     : FB_LocalClient := ( sRemoteHost:= '127.0.0.1', nRemotePort:= 200 );

    bEnableClient1 : BOOL := TRUE;
    bEnableClient2 : BOOL := FALSE;
    bEnableClient3 : BOOL := FALSE;
    bEnableClient4 : BOOL := FALSE;
    bEnableClient5 : BOOL := FALSE;

    fbSocketCloseAll    : FB_SocketCloseAll := ( sSrvNetID := '', tTimeout := DEFAULT_ADS_TIMEOUT );
    bCloseAll     : BOOL := TRUE;

    nCount     : UDINT;
END_VAR
```

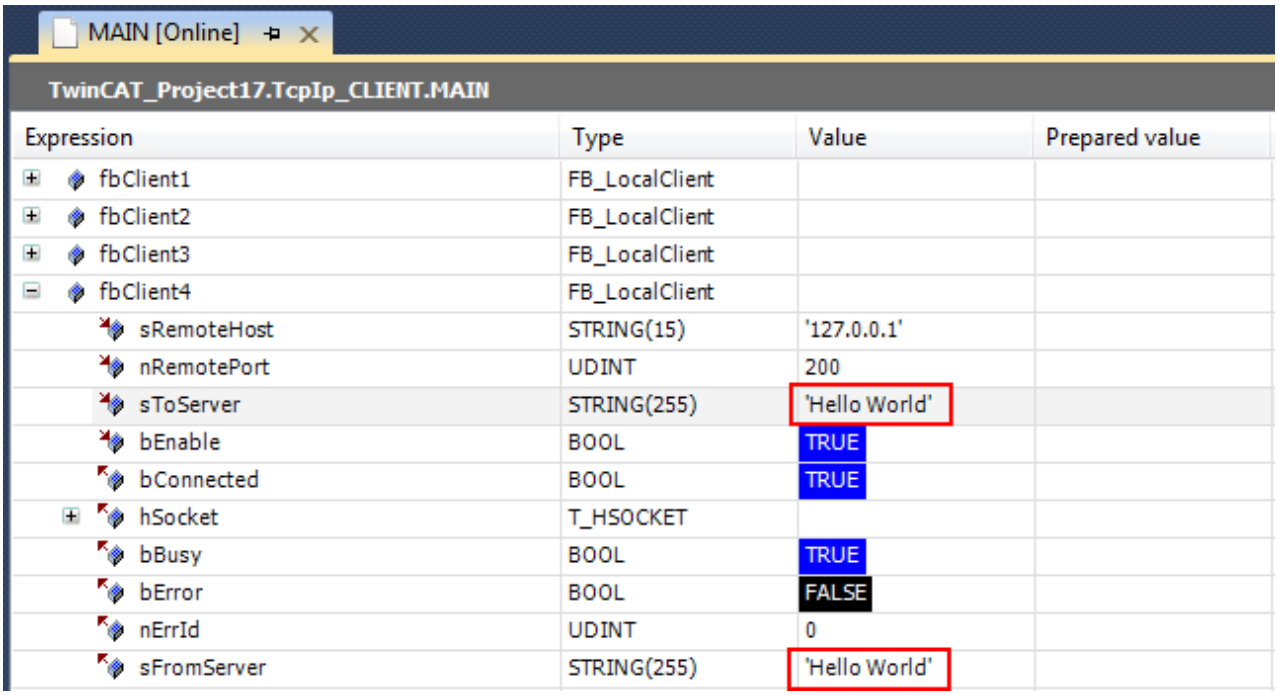
```
IF bCloseAll THEN (*On PLC reset or program download close all old connections *)
    bCloseAll := FALSE;
    fbSocketCloseAll( bExecute:= TRUE );
ELSE
    fbSocketCloseAll( bExecute:= FALSE );
END_IF

IF NOT fbSocketCloseAll.bBusy THEN
    nCount := nCount + 1;
    fbClient1( bEnable := bEnableClient1, sToServer := CONCAT( 'CLIENT1-', UDINT_TO_STRING( nCount )
 ) );
    fbClient2( bEnable := bEnableClient2, sToServer := CONCAT( 'CLIENT2-', UDINT_TO_STRING( nCount )
 ) );
    fbClient3( bEnable := bEnableClient3, sToServer := CONCAT( 'CLIENT3-', UDINT_TO_STRING( nCount )
 ) );
    fbClient4( bEnable := bEnableClient4 );
    fbClient5( bEnable := bEnableClient5 );
END_IF
```

Up to five client instances can be activated by setting the bEnableClientX variable. Each client sends a string (default: 'TEST') to the server approximately every second. The server returns the same string to the client (echo). For the test, a string with a counter value is generated automatically for the first three instances. The first client is activated automatically when the program is started. Set the bEnableClient4 variable in the client project to TRUE. The new client instance will then attempt to establish a connection with the server. If successful, the 'TEST' string is sent cyclically. Now open the fbClient4 instance of the FB_LocalClient function block. Double-click to open the dialog for writing the sToString variable. Change the value of the string variable, for example to 'Hello'.



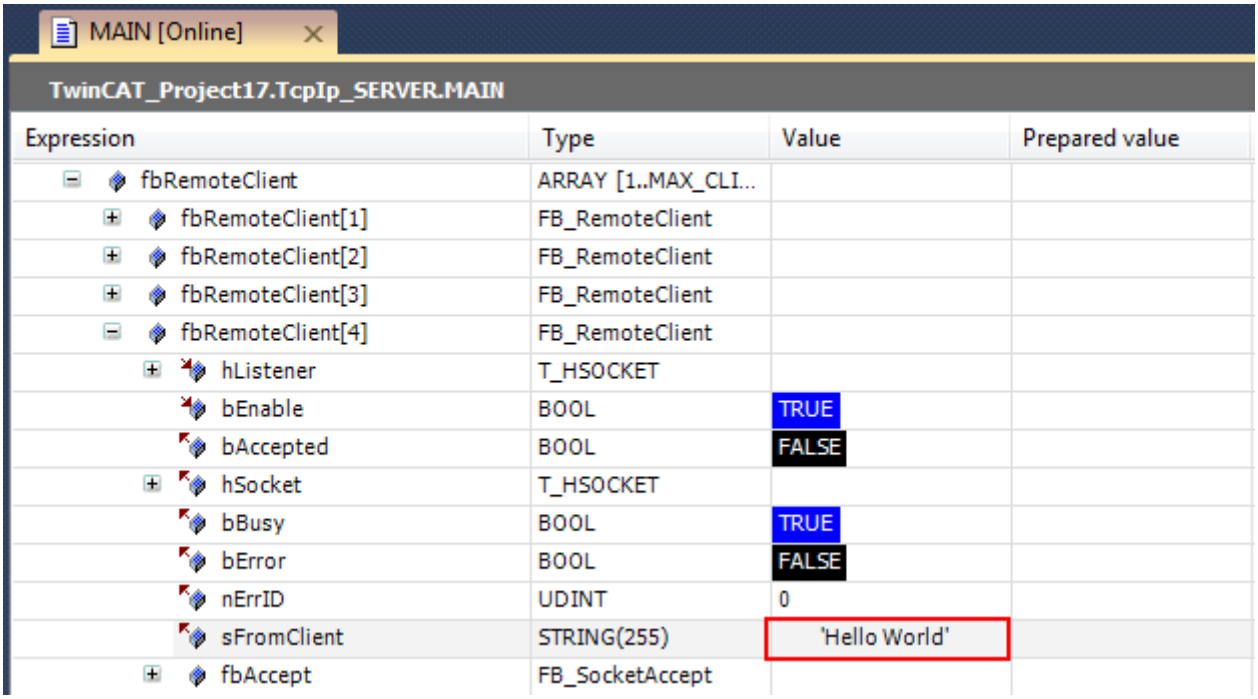
Expression	Type	Value	Prepared value
fbClient1	FB_LocalClient		
fbClient2	FB_LocalClient		
fbClient3	FB_LocalClient		
fbClient4	FB_LocalClient		
sRemoteHost	STRING(15)	'127.0.0.1'	
nRemotePort	UDINT	200	
sToServer	STRING(255)	'Test'	'Hello World'
bEnable	BOOL	TRUE	
bConnected	BOOL	TRUE	
hSocket	T_HSOCKET		
bBusy	BOOL	TRUE	
bError	BOOL	FALSE	
nErrId	UDINT	0	
sFromServer	STRING(255)	'Test'	

Close the dialog with **OK**. Write the new value into the PLC. Shortly afterwards, the value is send back by the server can also be seen online.



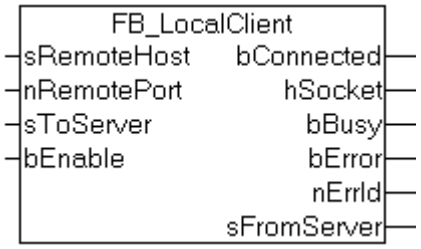
Expression	Type	Value	Prepared value
fbClient1	FB_LocalClient		
fbClient2	FB_LocalClient		
fbClient3	FB_LocalClient		
fbClient4	FB_LocalClient		
sRemoteHost	STRING(15)	'127.0.0.1'	
nRemotePort	UDINT	200	
sToServer	STRING(255)	'Hello World'	
bEnable	BOOL	TRUE	
bConnected	BOOL	TRUE	
hSocket	T_HSOCKET		
bBusy	BOOL	TRUE	
bError	BOOL	FALSE	
nErrId	UDINT	0	
sFromServer	STRING(255)	'Hello World'	

Now open the fbServer instance of the FB_LocalServer function block in the server project. Our string: 'Hello' can be seen in the online data of the server.



6.1.1.3 PLC Client

6.1.1.3.1 FB_LocalClient



If the bEnable input is set, the system will keep trying to establish the connection to the remote server once the PLCPRJ_RECONNECT_TIME has elapsed. The remote server is identified via the sRemoteHost IP address and the nRemotePort IP port address. The data exchange with the server was encapsulated in a separate function block (FB_ClientDataExcha [▶ 62]). Data exchange is always cyclic once PLCPRJ_SEND_CYCLE_TIME has elapsed. The sToServer string variable is sent to the server, and the string sent back by the server is returned at output sFormServer. Another implementation, in which the remote server is addressed as required is also possible. In the event of an error, the existing connection is closed, and a new connection is established.

Interface

```
FUNCTION_BLOCK FB_LocalClient
VAR_INPUT
    sRemoteHost     : STRING(15) := '127.0.0.1';(* IP adress of remote server *)
    nRemotePort     : UDINT := 0;
    sToServer       : T_MaxString:= 'TEST';
    bEnable         : BOOL;
END_VAR
VAR_OUTPUT
    bConnected      : BOOL;
    hSocket         : T_HSOCKET;
    bBusy           : BOOL;
    bError          : BOOL;
    nErrId          : UDINT;
    sFromServer     : T_MaxString;
END_VAR
```

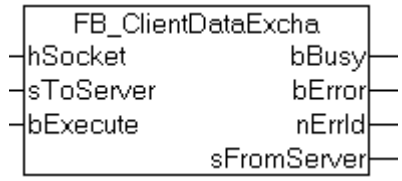
```
VAR
    fbConnect       : FB_SocketConnect := ( sSrvNetId := '' );
    fbClose         : FB_SocketClose := ( sSrvNetId := '', tTimeout := DEFAULT_ADS_TIMEOUT );
    fbClientDataExcha : FB_ClientDataExcha;

    fbConnectTON     : TON := ( PT := PLCPRJ_RECONNECT_TIME );
    fbDataExchaTON   : TON := ( PT := PLCPRJ_SEND_CYCLE_TIME );
    eStep            : E_ClientSteps;
END_VAR
```

Implementation

```
CASE eStep OF
    CLIENT_STATE_IDLE:
        IF bEnable XOR bConnected THEN
            bBusy := TRUE;
            bError := FALSE;
            nErrid := 0;
            sFromServer := '';
            IF bEnable THEN
                fbConnectTON( IN := FALSE );
                eStep := CLIENT_STATE_CONNECT_START;
            ELSE
                eStep := CLIENT_STATE_CLOSE_START;
            END_IF
        ELSIF bConnected THEN
            fbDataExchaTON( IN := FALSE );
            eStep := CLIENT_STATE_DATAEXCHA_START;
        ELSE
            bBusy := FALSE;
        END_IF

    CLIENT_STATE_CONNECT_START:
        fbConnectTON( IN := TRUE, PT := PLCPRJ_RECONNECT_TIME );
        IF fbConnectTON.Q THEN
            fbConnectTON( IN := FALSE );
            fbConnect( bExecute := FALSE );
                fbConnect(sRemoteHost := sRemoteHost,
                nRemotePort := nRemotePort,
                bExecute        := TRUE );
            eStep := CLIENT_STATE_CONNECT_WAIT;
        END_IF

    CLIENT_STATE_CONNECT_WAIT:
        fbConnect( bExecute := FALSE );
        IF NOT fbConnect.bBusy THEN
            IF NOT fbConnect.bError THEN
                bConnected := TRUE;
                hSocket         := fbConnect.hSocket;
                eStep     := CLIENT_STATE_IDLE;
                LogMessage( 'LOCAL client CONNECTED!', hSocket );
            ELSE
                LogError( 'FB_SocketConnect', fbConnect.nErrId );
                nErrId := fbConnect.nErrId;
                eStep := CLIENT_STATE_ERROR;
            END_IF
           END_IF

    CLIENT_STATE_DATAEXCHA_START:
        fbDataExchaTON( IN := TRUE, PT := PLCPRJ_SEND_CYCLE_TIME );
        IF fbDataExchaTON.Q THEN
            fbDataExchaTON( IN := FALSE );
            fbClientDataExcha( bExecute := FALSE );
            fbClientDataExcha( hSocket := hSocket,
                    sToServer := sToServer,
                    bExecute    := TRUE );
            eStep := CLIENT_STATE_DATAEXCHA_WAIT;
        END_IF

    CLIENT_STATE_DATAEXCHA_WAIT:
        fbClientDataExcha( bExecute := FALSE );
        IF NOT fbClientDataExcha.bBusy THEN
            IF NOT fbClientDataExcha.bError THEN
                sFromServer := fbClientDataExcha.sFromServer;
                eStep     := CLIENT_STATE_IDLE;
            ELSE
                (* possible errors are logged inside of fbClientDataExcha function block *)
                nErrId := fbClientDataExcha.nErrId;
                     eStep :=CLIENT_STATE_ERROR;
```

```
            END_IF
        END_IF

    CLIENT_STATE_CLOSE_START:
        fbClose( bExecute := FALSE );
        fbClose(    hSocket:= hSocket,
            bExecute:= TRUE );
        eStep := CLIENT_STATE_CLOSE_WAIT;

    CLIENT_STATE_CLOSE_WAIT:
        fbClose( bExecute := FALSE );
        IF NOT fbClose.bBusy THEN
            LogMessage( 'LOCAL client CLOSED!', hSocket );
            bConnected := FALSE;
            MEMSET( ADR(hSocket), 0, SIZEOF(hSocket));
            IF fbClose.bError THEN
                LogError( 'FB_SocketClose (local client)', fbClose.nErrId );
                nErrId := fbClose.nErrId;
                eStep := CLIENT_STATE_ERROR;
            ELSE
                bBusy := FALSE;
                bError := FALSE;
                     nErrId := 0;
                eStep := CLIENT_STATE_IDLE;
            END_IF
        END_IF

    CLIENT_STATE_ERROR: (* Error step *)
        bError := TRUE;
        IF bConnected THEN
            eStep := CLIENT_STATE_CLOSE_START;
        ELSE
            bBusy := FALSE;
            eStep := CLIENT_STATE_IDLE;
        END_IF
END_CASE
```

6.1.1.3.2 FB_ClientDataExcha



In the event of an rising edge at the *bExecute* input, a zero-terminated string is sent to the remote server, and a string returned by the remote server is read. The function block will try reading the data until zero termination was detected in the string received. Reception is aborted in the event of an error, and if no new data were received within the PLCPRJ_RECEIVE_TIMEOUT timeout time. Data are attempted to be read again after a certain delay time, if no new data could be read during the last read attempt. This reduces the system load.

Interface

```
FUNCTION_BLOCK FB_ClientDataExcha
VAR_INPUT
    hSocket     : T_HSOCKET;
    sToServer   : T_MaxString;
    bExecute    : BOOL;
END_VAR
VAR_OUTPUT
    bBusy       : BOOL;
    bError      : BOOL;
    nErrId      : UDINT;
    sFromServer : T_MaxString;
END_VAR
VAR
    fbSocketSend : FB_SocketSend := ( sSrvNetID := '', tTimeout := DEFAULT_ADS_TIMEOUT );
    fbSocketReceive : FB_SocketReceive := ( sSrvNetID := '', tTimeout := DEFAULT_ADS_TIMEOUT );
    fbReceiveTON : TON;
    fbDisconnectTON : TON;
    RisingEdge : R_TRIG;
```

```
    eStep     : E_DataExchaSteps;
    cbReceived, startPos, endPos, idx : UDINT;
    cbFrame     : UDINT;
    rxBuffer    : ARRAY[0..MAX_PLCPRJ_RXBUFFER_SIZE] OF BYTE;
END_VAR
```

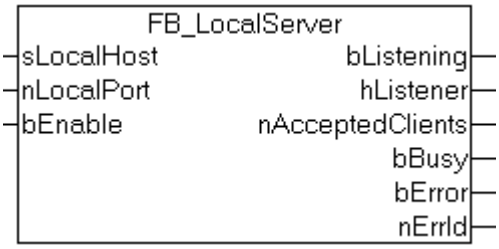
Implementation

```
RisingEdge( CLK := bExecute );
CASE eStep OF
    DATAEXCHA_STATE_IDLE:
        IF RisingEdge.Q THEN
            bBusy := TRUE;
            bError := FALSE;
            nErrid := 0;
            cbReceived := 0;
            fbReceiveTON( IN := FALSE, PT := T#0s ); (* don't wait, read the first answer data immed
iately *)
            fbDisconnectTON( IN := FALSE, PT := T#0s );(* disable timeout check first *)
            eStep := DATAEXCHA_STATE_SEND_START;
        END_IF

    DATAEXCHA_STATE_SEND_START:
        fbSocketSend( bExecute := FALSE );
        fbSocketSend( hSocket := hSocket,
                pSrc := ADR( sToServer ),
                cbLen := LEN( sToServer ) + 1,(* string length inclusive zero delimiter *)
                bExecute:= TRUE );
        eStep := DATAEXCHA_STATE_SEND_WAIT;

    DATAEXCHA_STATE_SEND_WAIT:
        fbSocketSend( bExecute := FALSE );
        IF NOT fbSocketSend.bBusy THEN
            IF NOT fbSocketSend.bError THEN
                eStep := DATAEXCHA_STATE_RECEIVE_START;
            ELSE
                     LogError( 'FB_SocketSend (local client)', fbSocketSend.nErrId );
                nErrId := fbSocketSend.nErrId;
                eStep := DATAEXCHA_STATE_ERROR;
            END_IF
        END_IF

    DATAEXCHA_STATE_RECEIVE_START:
        fbDisconnectTON( );
        fbReceiveTON( IN := TRUE );
        IF fbReceiveTON.Q THEN
            fbReceiveTON( IN := FALSE );
            fbSocketReceive( bExecute := FALSE );
            fbSocketReceive( hSocket:= hSocket,
                    pDest:= ADR( rxBuffer ) + cbReceived,
                    cbLen:= SIZEOF( rxBuffer ) - cbReceived,
                    bExecute:= TRUE );
            eStep := DATAEXCHA_STATE_RECEIVE_WAIT;
        END_IF

    DATAEXCHA_STATE_RECEIVE_WAIT:
        fbSocketReceive( bExecute := FALSE );
        IF NOT fbSocketReceive.bBusy THEN
            IF NOT fbSocketReceive.bError THEN
                     IF (fbSocketReceive.nRecBytes > 0) THEN(* bytes received *)
                    startPos        := cbReceived;(* rxBuffer array index of first data byte *)
                    endPos          := cbReceived + fbSocketReceive.nRecBytes - 1;
(* rxBuffer array index of last data byte *)
                    cbReceived := cbReceived + fbSocketReceive.nRecBytes;
(* calculate the number of received data bytes *)
                    cbFrame     := 0;(* reset frame length *)
                    IF cbReceived < SIZEOF( sFromServer ) THEN(* no overflow *)
                        fbReceiveTON( PT := T#0s ); (* bytes received => increase the read (polling)
 speed *)
                        fbDisconnectTON( IN := FALSE );(* bytes received => disable timeout check *)
                        (* search for string end delimiter *)
                        FOR idx := startPos TO endPos BY 1 DO
                                    IF rxBuffer[idx] = 0 THEN(* string end delimiter found *)
                                cbFrame := idx + 1;
(* calculate the length of the received string (inclusive the end delimiter) *)
                                MEMCPY( ADR( sFromServer ), ADR( rxBuffer ), cbFrame );
(* copy the received string to the output variable (inclusive the end delimiter) *)
                                MEMMOVE( ADR( rxBuffer ), ADR( rxBuffer[cbFrame] ), cbReceived -
 cbFrame );(* move the reamaining data bytes *)
```

```
                                cbReceived := cbReceived - cbFrame;
(* recalculate the remaining data byte length *)
                                bBusy := FALSE;
                                eStep := DATAEXCHA_STATE_IDLE;
                                EXIT;
                                    END_IF
                            END_FOR
                        ELSE(* there is no more free read buffer space => the answer string should be te
rminated *)
                            LogError( 'FB_SocketReceive (local client)', PLCPRJ_ERROR_RECEIVE_BUFFER_OVE
RFLOW );
                            nErrId := PLCPRJ_ERROR_RECEIVE_BUFFER_OVERFLOW;(* buffer overflow !*)
                            eStep := DATAEXCHA_STATE_ERROR;
                        END_IF
                    ELSE(* no bytes received *)
                        fbReceiveTON( PT := PLCPRJ_RECEIVE_POLLING_TIME );
(* no bytes received => decrease the read (polling) speed *)
                        fbDisconnectTON( IN := TRUE, PT := PLCPRJ_RECEIVE_TIMEOUT );
(* no bytes received => enable timeout check*)
                        IF fbDisconnectTON.Q THEN (* timeout error*)
                                fbDisconnectTON( IN := FALSE );
                            LogError( 'FB_SocketReceive (local client)', PLCPRJ_ERROR_RECEIVE_TIMEOUT );
                            nErrID := PLCPRJ_ERROR_RECEIVE_TIMEOUT;
                            eStep := DATAEXCHA_STATE_ERROR;
                        ELSE(* repeat reading *)
                            eStep := DATAEXCHA_STATE_RECEIVE_START; (* repeat reading *)
                        END_IF
                    END_IF
                ELSE(* receive error *)
                    LogError( 'FB_SocketReceive (local client)', fbSocketReceive.nErrId );
                    nErrId := fbSocketReceive.nErrId;
                    eStep := DATAEXCHA_STATE_ERROR;
                END_IF
               END_IF

    DATAEXCHA_STATE_ERROR:(* error step *)
        bBusy := FALSE;
        bError := TRUE;
        cbReceived := 0;
        eStep := DATAEXCHA_STATE_IDLE;
END_CASE
```

6.1.1.4 PLC Server

6.1.1.4.1 FB_LocalServer



The server must first be allocated a unique sLocalHost IP address and an nLocaPort IP port number. If the bEnable input is set, the local server will repeatedly try to open the listener socket once the PLCPRJ_RECONNECT_TIME has elapsed. The listener socket can usually be opened at the first attempt, if the TwinCAT TCP/IP Connection Server resides on the local PC. The functionality of a remote client was encapsulated in the function block FB_RemoteClient [▶ 66]. The remote client instances are activated once the listener socket was opened successfully. Each instance of the FB_RemoteClient corresponds to a remote client, with which the local server can communicate simultaneously. The maximum number of remote clients communicating with the server can be modified via the value of the MAX_CLIENT_CONNECTIONS constant. In the event of an error, first all remote client connections are closed, followed by the listener sockets. The nAcceptedClients output provides information about the current number of connected clients.

Interface

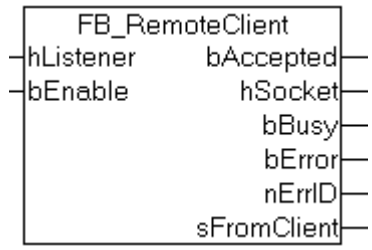
```
FUNCTION_BLOCK FB_LocalServer
VAR_INPUT
    sLocalHost       : STRING(15) := '127.0.0.1';(* own IP address! *)
    nLocalPort       : UDINT := 0;
    bEnable          : BOOL;
END_VAR
VAR_OUTPUT
    bListening       : BOOL;
    hListener        : T_HSOCKET;
    nAcceptedClients : UDINT;
    bBusy            : BOOL;
    bError           : BOOL;
    nErrId           : UDINT;
END_VAR
VAR
    fbListen        : FB_SocketListen := ( sSrvNetID := '', tTimeout := DEFAULT_ADS_TIMEOUT );
    fbClose         : FB_SocketClose := ( sSrvNetID := '', tTimeout := DEFAULT_ADS_TIMEOUT );
    fbConnectTON    : TON := ( PT := PLCPRJ_RECONNECT_TIME );
    eStep           : E_ServerSteps;
    fbRemoteClient  : ARRAY[1..MAX_CLIENT_CONNECTIONS ] OF FB_RemoteClient;
    i               : UDINT;
END_VAR
```

Implementation

```
CASE eStep OF

    SERVER_STATE_IDLE:
        IF bEnable XOR bListening THEN
            bBusy := TRUE;
            bError := FALSE;
            nErrId := 0;
            IF bEnable THEN
                fbConnectTON( IN := FALSE );
                eStep := SERVER_STATE_LISTENER_OPEN_START;
            ELSE
                eStep := SERVER_STATE_REMOTE_CLIENTS_CLOSE;
            END_IF
        ELSIF bListening THEN
            eStep := SERVER_STATE_REMOTE_CLIENTS_COMM;
        END_IF

    SERVER_STATE_LISTENER_OPEN_START:
        fbConnectTON( IN := TRUE, PT := PLCPRJ_RECONNECT_TIME );
        IF fbConnectTON.Q THEN
            fbConnectTON( IN := FALSE );
            fbListen( bExecute := FALSE );
            fbListen( sLocalHost:= sLocalHost,
                nLocalPort:= nLocalPort,
                bExecute    := TRUE );
                eStep := SERVER_STATE_LISTENER_OPEN_WAIT;
        END_IF

    SERVER_STATE_LISTENER_OPEN_WAIT:
        fbListen( bExecute := FALSE );
        IF NOT fbListen.bBusy THEN
            IF NOT fbListen.bError THEN
                bListening := TRUE;
                hListener := fbListen.hListener;
                eStep     := SERVER_STATE_IDLE;
                LogMessage( 'LISTENER socket OPENED!', hListener );
            ELSE
                LogError( 'FB_SocketListen', fbListen.nErrId );
                nErrId := fbListen.nErrId;
                eStep := SERVER_STATE_ERROR;
            END_IF
        END_IF

    SERVER_STATE_REMOTE_CLIENTS_COMM:
        eStep := SERVER_STATE_IDLE;
        nAcceptedClients := 0;
        FOR i:= 1 TO MAX_CLIENT_CONNECTIONS DO
            fbRemoteClient[ i ]( hListener := hListener, bEnable := TRUE );
            IF NOT fbRemoteClient[ i ].bBusy AND fbRemoteClient[ i ].bError THEN (*FB_SocketAccept r
eturned error!*)
                    eStep := SERVER_STATE_REMOTE_CLIENTS_CLOSE;
                EXIT;
```

```
            END_IF
            (* count the number of connected remote clients *)
            IF fbRemoteClient[ i ].bAccepted THEN
                nAcceptedClients := nAcceptedClients + 1;
            END_IF
        END_FOR

    SERVER_STATE_REMOTE_CLIENTS_CLOSE:
        nAcceptedClients := 0;
        eStep := SERVER_STATE_LISTENER_CLOSE_START; (* close listener socket too *)
        FOR i:= 1 TO MAX_CLIENT_CONNECTIONS DO
            fbRemoteClient[ i ]( bEnable := FALSE );(* close all remote client (accepted) sockets *)
            (* check if all remote client sockets are closed *)
            IF fbRemoteClient[ i ].bAccepted THEN
                eStep := SERVER_STATE_REMOTE_CLIENTS_CLOSE; (* stay here and close all remote client
s first *)
                nAcceptedClients := nAcceptedClients + 1;
            END_IF
        END_FOR

    SERVER_STATE_LISTENER_CLOSE_START:
        fbClose( bExecute := FALSE );
        fbClose(    hSocket := hListener,
                    bExecute:= TRUE );
        eStep := SERVER_STATE_LISTENER_CLOSE_WAIT;

    SERVER_STATE_LISTENER_CLOSE_WAIT:
        fbClose( bExecute := FALSE );
        IF NOT fbClose.bBusy THEN
            LogMessage( 'LISTENER socket CLOSED!', hListener );
            bListening := FALSE;
            MEMSET( ADR(hListener), 0, SIZEOF(hListener));
            IF fbClose.bError THEN
                LogError( 'FB_SocketClose (listener)', fbClose.nErrId );
                nErrId := fbClose.nErrId;
                eStep := SERVER_STATE_ERROR;
            ELSE
                bBusy := FALSE;
                bError := FALSE;
                nErrId := 0;
                eStep := SERVER_STATE_IDLE;
            END_IF
        END_IF

    SERVER_STATE_ERROR:
        bError := TRUE;
           IF bListening THEN
            eStep := SERVER_STATE_REMOTE_CLIENTS_CLOSE;
        ELSE
            bBusy := FALSE;
            eStep := SERVER_STATE_IDLE;
        END_IF

END_CASE
```

6.1.1.4.2 FB_RemoteClient



If the bEnable input is set, an attempt is made to accept the connection request of a remote client, once the PLCPRJ_ACCEPT_POOLING_TIME has elapsed. The data exchange with the remote client was encapsulated in a separate function block (FB_ServerDataExcha [▶ 68]). Once the connection was established successfully, the instance is activated via the FB_ServerDataExcha function block. In the event of an error, the accepted connection is closed, and a new connection is established.

Interface

```
FUNCTION_BLOCK FB_RemoteClient
VAR_INPUT
    hListener     : T_HSOCKET;
    bEnable       : BOOL;
END_VAR
VAR_OUTPUT
    bAccepted     : BOOL;
    hSocket       : T_HSOCKET;
    bBusy         : BOOL;
    bError        : BOOL;
    nErrID        : UDINT;
    sFromClient   : T_MaxString;
END_VAR
VAR
    fbAccept            : FB_SocketAccept := ( sSrvNetID := '', tTimeout := DEFAULT_ADS_TIMEOUT );
    fbClose             : FB_SocketClose := ( sSrvNetID := '', tTimeout := DEFAULT_ADS_TIMEOUT );
    fbServerDataExcha     : FB_ServerDataExcha;
    fbAcceptTON     : TON := ( PT := PLCPRJ_ACCEPT_POLLING_TIME );
    eStep         : E_ClientSteps;
END_VAR
```

Implementation

```
CASE eStep OF

    CLIENT_STATE_IDLE:
        IF bEnable XOR bAccepted THEN
            bBusy := TRUE;
            bError := FALSE;
            nErrId := 0;
            sFromClient := '';
            IF bEnable THEN
                fbAcceptTON( IN := FALSE );
                eStep := CLIENT_STATE_CONNECT_START;
            ELSE
                eStep := CLIENT_STATE_CLOSE_START;
            END_IF
        ELSIF bAccepted THEN
            eStep := CLIENT_STATE_DATAEXCHA_START;
        ELSE
            bBusy := FALSE;
        END_IF

    CLIENT_STATE_CONNECT_START:
        fbAcceptTON( IN := TRUE, PT := PLCPRJ_ACCEPT_POLLING_TIME );
        IF fbAcceptTON.Q THEN
            fbAcceptTON( IN := FALSE );
            fbAccept( bExecute := FALSE );
            fbAccept( hListener := hListener,
                              bExecute:= TRUE );
            eStep := CLIENT_STATE_CONNECT_WAIT;
        END_IF

    CLIENT_STATE_CONNECT_WAIT:
        fbAccept( bExecute := FALSE );
        IF NOT fbAccept.bBusy THEN
            IF NOT fbAccept.bError THEN
                IF fbAccept.bAccepted THEN
                    bAccepted := TRUE;
                    hSocket := fbAccept.hSocket;
                    LogMessage( 'REMOTE client ACCEPTED!', hSocket );
                END_IF
                eStep := CLIENT_STATE_IDLE;
            ELSE
                LogError( 'FB_SocketAccept', fbAccept.nErrId );
                nErrId := fbAccept.nErrId;
                eStep := CLIENT_STATE_ERROR;
                END_IF
        END_IF

    CLIENT_STATE_DATAEXCHA_START:
        fbServerDataExcha( bExecute := FALSE );
        fbServerDataExcha( hSocket := hSocket,
                bExecute := TRUE );
        eStep := CLIENT_STATE_DATAEXCHA_WAIT;

    CLIENT_STATE_DATAEXCHA_WAIT:
```

```
        fbServerDataExcha( bExecute := FALSE, sFromClient=>sFromClient );
        IF NOT fbServerDataExcha.bBusy THEN
            IF NOT fbServerDataExcha.bError THEN
                eStep := CLIENT_STATE_IDLE;
            ELSE
                (* possible errors are logged inside of fbServerDataExcha function block *)
                nErrId := fbServerDataExcha.nErrID;
                eStep := CLIENT_STATE_ERROR;
            END_IF
        END_IF

    CLIENT_STATE_CLOSE_START:
        fbClose( bExecute := FALSE );
        fbClose(    hSocket:= hSocket,
                bExecute:= TRUE );
        eStep := CLIENT_STATE_CLOSE_WAIT;

    CLIENT_STATE_CLOSE_WAIT:
        fbClose( bExecute := FALSE );
           IF NOT fbClose.bBusy THEN
            LogMessage( 'REMOTE client CLOSED!', hSocket );
            bAccepted := FALSE;
            MEMSET( ADR( hSocket ), 0, SIZEOF( hSocket ) );
            IF fbClose.bError THEN
                LogError( 'FB_SocketClose (remote client)', fbClose.nErrId );
                nErrId := fbClose.nErrId;
                eStep := CLIENT_STATE_ERROR;
            ELSE
                bBusy := FALSE;
                bError := FALSE;
                nErrId := 0;
                eStep := CLIENT_STATE_IDLE;
            END_IF
        END_IF

    CLIENT_STATE_ERROR:
        bError := TRUE;
        IF bAccepted THEN
            eStep := CLIENT_STATE_CLOSE_START;
        ELSE
            eStep := CLIENT_STATE_IDLE;
                bBusy := FALSE;
        END_IF

END_CASE
```

6.1.1.4.3 FB_ServerDataExcha

In the event of an rising edge at the bExecute input, a zero-terminated string is read by the remote client and returned to the remote client, if zero termination was detected. The function block will try reading the data until zero termination was detected in the string received. Reception is aborted in the event of an error, and if no new data were received within the PLCPRJ_RECEIVE_TIMEOUT timeout time. Data are attempted to be read again after a certain delay time, if no new data could be read during the last read attempt. This reduces the system load.

Interface

```
FUNCTION_BLOCK FB_ServerDataExcha
VAR_INPUT
    hSocket     : T_HSOCKET;
    bExecute    : BOOL;
END_VAR
VAR_OUTPUT
    bBusy       : BOOL;
    bError      : BOOL;
    nErrID      : UDINT;
    sFromClient : T_MaxString;
```

```
END_VAR
VAR
    fbSocketReceive : FB_SocketReceive := ( sSrvNetId := '', tTimeout := DEFAULT_ADS_TIMEOUT );
    fbSocketSend : FB_SocketSend := ( sSrvNetId := '', tTimeout := DEFAULT_ADS_TIMEOUT );
    eStep     : E_DataExchaSteps;
    RisingEdge : R_TRIG;
    fbReceiveTON : TON;
    fbDisconnectTON     : TON;
    cbReceived, startPos, endPos, idx : UDINT;
    cbFrame     : UDINT;
    rxBuffer        : ARRAY[0..MAX_PLCPRJ_RXBUFFER_SIZE] OF BYTE;
END_VAR
```

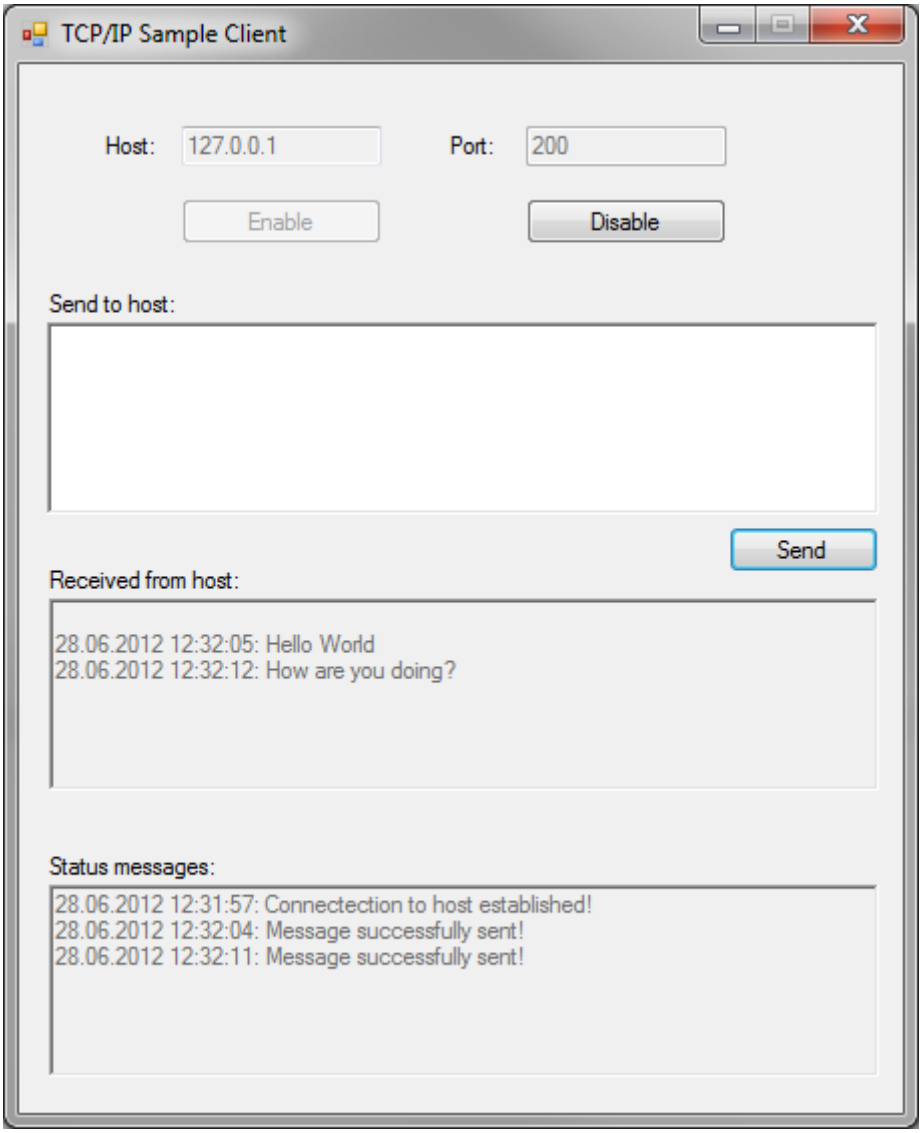
Implementation

```
RisingEdge( CLK := bExecute );
CASE eStep OF

    DATAEXCHA_STATE_IDLE:
        IF RisingEdge.Q THEN
            bBusy := TRUE;
            bError := FALSE;
            nErrId := 0;
            fbDisconnectTON( IN := FALSE, PT := T#0s );(* disable timeout check first *)
            fbReceiveTON( IN := FALSE, PT := T#0s ); (* receive first request immediately *)
            eStep := DATAEXCHA_STATE_RECEIVE_START;
        END_IF

    DATAEXCHA_STATE_RECEIVE_START: (* Receive remote client data *)
        fbReceiveTON( IN := TRUE );
        IF fbReceiveTON.Q THEN
            fbReceiveTON( IN := FALSE );
            fbSocketReceive( bExecute := FALSE );
            fbSocketReceive(    hSocket := hSocket,
                    pDest := ADR( rxBuffer ) + cbReceived,
                    cbLen := SIZEOF( rxBuffer ) - cbReceived,
                    bExecute    := TRUE );
            eStep := DATAEXCHA_STATE_RECEIVE_WAIT;
        END_IF

    DATAEXCHA_STATE_RECEIVE_WAIT:
        fbSocketReceive( bExecute := FALSE );
           IF NOT fbSocketReceive.bBusy THEN
            IF NOT fbSocketReceive.bError THEN

                IF (fbSocketReceive.nRecBytes > 0) THEN(* bytes received *)

                    startPos        := cbReceived;(* rxBuffer array index of first data byte *)
                    endPos         := cbReceived + fbSocketReceive.nRecBytes - 1;
(* rxBuffer array index of last data byte *)
                    cbReceived := cbReceived + fbSocketReceive.nRecBytes;
(* calculate the number of received data bytes *)
                    cbFrame     := 0;(* reset frame length *)

                    IF cbReceived < SIZEOF( sFromClient ) THEN(* no overflow *)

                        fbReceiveTON( IN := FALSE, PT := T#0s ); (* bytes received => increase the r
ead (polling) speed *)
                        fbDisconnectTON( IN := FALSE, PT := PLCPRJ_RECEIVE_TIMEOUT );
(* bytes received => disable timeout check *)

                        (* search for string end delimiter *)
                        FOR idx := startPos TO endPos BY 1 DO
                                    IF rxBuffer[idx] = 0 THEN(* string end delimiter found *)
                                cbFrame := idx + 1;
(* calculate the length of the received string (inclusive the end delimiter) *)
                                MEMCPY( ADR( sFromClient ), ADR( rxBuffer ), cbFrame );
(* copy the received string to the output variable (inclusive the end delimiter) *)
                                MEMMOVE( ADR( rxBuffer ), ADR( rxBuffer[cbFrame] ), cbReceived -
 cbFrame );(* move the reamaining data bytes *)
                                cbReceived := cbReceived - cbFrame;
(* recalculate the reamaining data byte length *)
                                eStep := DATAEXCHA_STATE_SEND_START;
                                EXIT;
                            END_IF
                               END_FOR

                    ELSE(* there is no more free read buffer space => the answer string should be te
rminated *)
```

```
                        LogError( 'FB_SocketReceive (remote client)', PLCPRJ_ERROR_RECEIVE_BUFFER_OV
ERFLOW );
                        nErrId := PLCPRJ_ERROR_RECEIVE_BUFFER_OVERFLOW;(* buffer overflow !*)
                        eStep := DATAEXCHA_STATE_ERROR;
                    END_IF

                ELSE(* no bytes received *)
                    fbReceiveTON( IN := FALSE, PT := PLCPRJ_RECEIVE_POLLING_TIME );
(* no bytes received => decrease the read (polling) speed *)
                    fbDisconnectTON( IN := TRUE, PT := PLCPRJ_RECEIVE_TIMEOUT );
(* no bytes received => enable timeout check*)
                    IF fbDisconnectTON.Q THEN (* timeout error*)
                        fbDisconnectTON( IN := FALSE );
                               LogError( 'FB_SocketReceive (remote client)', PLCPRJ_ERROR_RECEIVE_TI
MEOUT );
                        nErrID := PLCPRJ_ERROR_RECEIVE_TIMEOUT;
                        eStep := DATAEXCHA_STATE_ERROR;
                    ELSE(* repeat reading *)
                        eStep := DATAEXCHA_STATE_RECEIVE_START; (* repeat reading *)
                    END_IF
                END_IF
            ELSE(* receive error *)
                LogError( 'FB_SocketReceive (remote client)', fbSocketReceive.nErrId );
                nErrId := fbSocketReceive.nErrId;
                eStep := DATAEXCHA_STATE_ERROR;
            END_IF
        END_IF

    DATAEXCHA_STATE_SEND_START:
        fbSocketSend( bExecute := FALSE );
        fbSocketSend( hSocket := hSocket,
                               pSrc := ADR( sFromClient ),
                        cbLen := LEN( sFromClient ) + 1,
(* string length inclusive the zero delimiter *)
                        bExecute:= TRUE );
        eStep := DATAEXCHA_STATE_SEND_WAIT;

    DATAEXCHA_STATE_SEND_WAIT:
        fbSocketSend( bExecute := FALSE );
        IF NOT fbSocketSend.bBusy THEN
            IF NOT fbSocketSend.bError THEN
                bBusy := FALSE;
                eStep := DATAEXCHA_STATE_IDLE;
            ELSE
                LogError( 'fbSocketSend (remote client)', fbSocketSend.nErrId );
                nErrId := fbSocketSend.nErrId;
                eStep := DATAEXCHA_STATE_ERROR;
            END_IF
        END_IF

    DATAEXCHA_STATE_ERROR:
        bBusy := FALSE;
        bError := TRUE;
        cbReceived := 0;(* reset old received data bytes *)
           eStep := DATAEXCHA_STATE_IDLE;
END_CASE
```

6.1.1.5 .NET client

This project example shows how a client for the PLC TCP/IP server can be realized by writing a .NET4.0 application using C#.



This sample client makes use of the .NET libraries System.Net and System.Net.Sockets which enable a programmer easy access to socket functionalities. By pressing the button **Enable**, the application attempts to cyclically (depending on the value of TIMERTICK in [ms]) establish a connection with the server. If successful, a string with a maximum length of 255 characters can be sent to the server via the "Send" button. The server will then take this string and send it back to the client. On the server side, the connection is closed automatically if the server was unable to receive new data from the client within a defined period, as specified by PLCPRJ_RECEIVE_TIMEOUT in the server sample - by default 50 seconds.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Net;
using System.Net.Sockets;

/* ######################################################################################
 * This sample TCP/IP client connects to a TCP/IP-Server, sends a message and waits for the
 * response. It is being delivered together with our TCP-Sample, which implements an echo server
 * in PLC.
 * ###################################################################################### */
namespace TcpIpServer_SampleClient
{
 publicpartialclassForm1 : Form
 {
    /
* ######################################################################################
    * Constants
```

```
    * ########################################################################################## */
privateconstint RCVBUFFERSIZE = 256; // buffer size for receive bufferprivateconststring DEFAULTIP =
 "127.0.0.1";
    privateconststring DEFAULTPORT = "200";
    privateconstint TIMERTICK = 100;

    /
* ##########################################################################################
    * Global variables
    * ########################################################################################## */
privatestaticbool _isConnected; // signals whether socket connection is active or notprivatestaticSo
cket _socket; // object used for socket connection to TCP/IP-
ServerprivatestaticIPEndPoint _ipAddress; // contains IP address as entered in text fieldprivatestat
icbyte[] _rcvBuffer; // receive buffer used for receiving response from TCP/IP-Serverpublic Form1()
    {
     InitializeComponent();
    }

    privatevoid Form1_Load(object sender, EventArgs e)
    {
     _rcvBuffer = newbyte[RCVBUFFERSIZE];

     /
* ##########################################################################################
     * Prepare GUI
     * ########################################################################################## */
     cmd_send.Enabled = false;
     cmd_enable.Enabled = true;
     cmd_disable.Enabled = false;
     rtb_rcvMsg.Enabled = false;
     rtb_sendMsg.Enabled = false;
     rtb_statMsg.Enabled = false;
     txt_host.Text = DEFAULTIP;
     txt_port.Text = DEFAULTPORT;

     timer1.Enabled = false;
     timer1.Interval = TIMERTICK;
     _isConnected = false;
    }

    privatevoid cmd_enable_Click(object sender, EventArgs e)
    {
     /
* ##########################################################################################
     * Parse IP address in text field, start background timer and prepare GUI
     * ########################################################################################## */
try
     {
        _ipAddress = newIPEndPoint(IPAddress.Parse(txt_host.Text), Convert.ToInt32(txt_port.Text));
        timer1.Enabled = true;
        cmd_enable.Enabled = false;
        cmd_disable.Enabled = true;
        rtb_sendMsg.Enabled = true;
        cmd_send.Enabled = true;
        txt_host.Enabled = false;
        txt_port.Enabled = false;
        rtb_sendMsg.Focus();
     }
     catch (Exception ex)
       {
        MessageBox.Show("Could not parse entered IP address. Please check spelling and retry. " + ex
);
     }
    }

    /
* ##########################################################################################
    * Timer periodically checks for connection to TCP/IP-
Server and reestablishes if not connected
    * ########################################################################################## */
privatevoid timer1_Tick(object sender, EventArgs e)
    {
     if (!_isConnected)
        connect();
    }

    privatevoid connect()
    {
     /
* ##########################################################################################
```

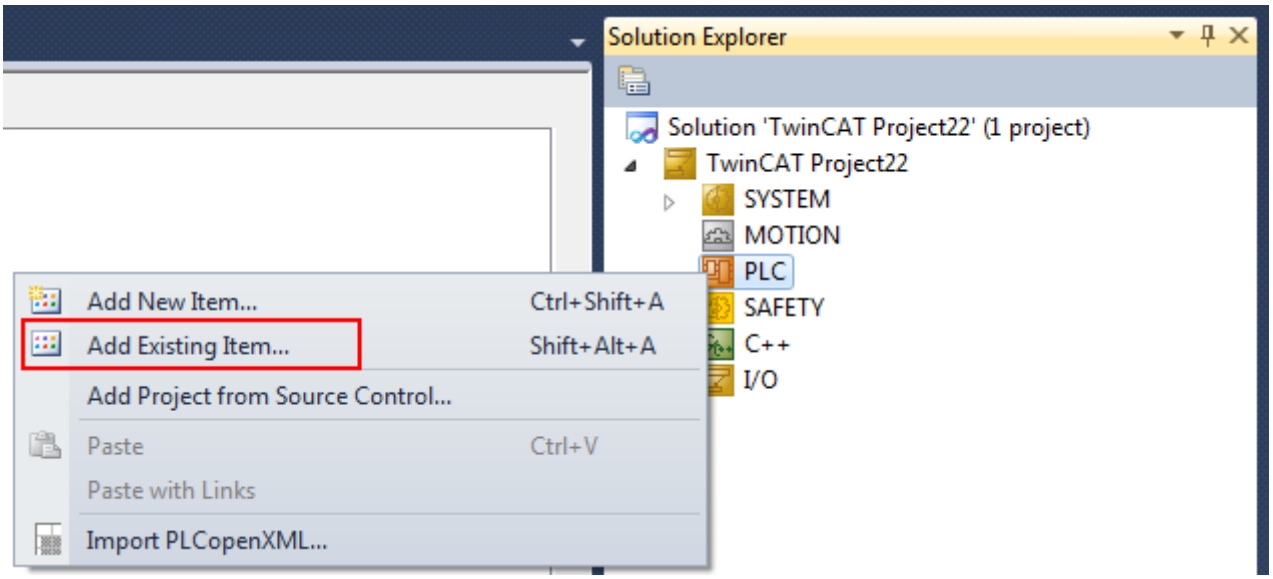
```
     * Connect to TCP/IP-Server using the IP address specified in the text field
     * ############################################################################################ */
try
     {
        _socket = newSocket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.IP);
        _socket.Connect(_ipAddress);
        _isConnected = true;
        if (_socket.Connected)
         rtb_statMsg.AppendText(DateTime.Now.ToString() + ": Connectection to host established!\n");
        else
         rtb_statMsg.AppendText(DateTime.Now.ToString() + ": A connection to the host could not be e
stablished!\n");
     }
     catch (Exception ex)
     {
        MessageBox.Show("An error occured while establishing a connection to the server: " + ex);
     }
    }

    privatevoid cmd_send_Click(object sender, EventArgs e)
    {
     /
* ############################################################################################
     * Read message from text field and prepare send buffer, which is a byte[] array. The last
     * character in the buffer needs to be a termination character, so that the TCP/IP-
Server knows
     * when the TCP stream ends. In this case, the termination character is '0'.
     * ############################################################################################ */
ASCIIEncoding enc = newASCIIEncoding();
     byte[] tempBuffer = enc.GetBytes(rtb_sendMsg.Text);
     byte[] sendBuffer = newbyte[tempBuffer.Length + 1];
     for (int i = 0; i < tempBuffer.Length; i++)
        sendBuffer[i] = tempBuffer[i];
     sendBuffer[tempBuffer.Length] = 0;

     /
* ############################################################################################
       * Send buffer content via TCP/IP connection
     * ############################################################################################ */
try
     {
        int send = _socket.Send(sendBuffer);
        if (send == 0)
         thrownewException();
        else
        {
         /
* ############################################################################################

         * As the TCP/IP-
Server returns a message, receive this message and store content in receive buffer.
         * When message receive is complete, show the received message in text field.
         * ############################################################################################
# */
         rtb_statMsg.AppendText(DateTime.Now.ToString() + ": Message successfully sent!\n");
         IAsyncResult asynRes = _socket.BeginReceive(_rcvBuffer, 0, 256, SocketFlags.None, null, nul
l);
         if (asynRes.AsyncWaitHandle.WaitOne())
         {
            int res = _socket.EndReceive(asynRes);
            char[] resChars = newchar[res + 1];
            Decoder d = Encoding.UTF8.GetDecoder();
            int charLength = d.GetChars(_rcvBuffer, 0, res, resChars, 0, true);
            String result = newString(resChars);
            rtb_rcvMsg.AppendText("\n" + DateTime.Now.ToString() + ": " + result);
            rtb_sendMsg.Clear();
         }
        }
     }
     catch (Exception ex)
       {
        MessageBox.Show("An error occured while sending the message: " + ex);
     }
    }

    privatevoid cmd_disable_Click(object sender, EventArgs e)
    {
     /
* ############################################################################################
     * Disconnect from TCP/IP-Server, stop the timer and prepare GUI
```

```
     * ########################################################################################### */
     timer1.Enabled = false;
     _socket.Disconnect(true);
     if (!_socket.Connected)
     {
        _isConnected = false;
        cmd_disable.Enabled = false;
        cmd_enable.Enabled = true;
        txt_host.Enabled = true;
        txt_port.Enabled = true;
        rtb_sendMsg.Enabled = false;
        cmd_send.Enabled = false;
        rtb_statMsg.AppendText(DateTime.Now.ToString() + ": Connectection to host closed!\n");
        rtb_rcvMsg.Clear();
        rtb_statMsg.Clear();
     }
    }
 }
}
```

6.1.2 Sample02: “Echo“ client /server

This sample is based on the functionality offered by the former TcSocketHelper.Lib, which is now part of Tc2_TcpIp library. It realizes a Client/Server PLC application based on the functionality provided by the former SocketHelper library.

The client cyclically sends a test string (sToServer) to the remote server. The server returns the same string unchanged to the client (sFromServer).



System requirements

- TwinCAT 3 Build 3093 or higher
- TwinCAT 3 Function TF6310 TCP/IP
- If two computers are used to execute the sample (one client and one server), the Function TF6310 needs to be installed on both computers
- If one computer is used to execute the sample, e.g. client and server running in two separate PLC runtimes, both PLC runtimes need to run in separate tasks.

Project downloads

https://github.com/Beckhoff/TF6310_Samples/tree/master/PLC/TCP/Sample02

Project information

The default communication settings used in the above samples are as follows:

- PLC client application: Port and IP address of the remote server: 200, '127.0.0.1'
- PLC server application: Port and IP address of the local server: 200, '127.0.0.1'

To test the client and server application on two different PCs, you have to adjust the port and the IP address accordingly.

However, you can also test the client and server samples with the default values on a single computer by loading the client application into the first PLC runtime system and the server application into the second PLC runtime system.

The behavior of the PLC project sample is determined by the following global variables/constants.

Constant	Value	Description
PLCPRJ_MAX_CONNECTIONS	5	Max. number of server → client connections. A server can establish connections to more than one client. A client can establish a connection to only one server at a time.
PLCPRJ_SERVER_RESPONSE_TIMEOUT	T#10s	Max. delay time (timeout time) after which a server should send a response to the client.
PLCPRJ_CLIENT_SEND_CYCLE_TIME	T#1s	Cycle time based on which a client sends send data (TX) to the server.
PLCPRJ_RECEIVER_POLLING_CYCLE_TIME	T#200ms	Cycle time based on which a client or server polls for receive data (RX).
PLCPRJ_BUFFER_SIZE	10000	Max. internal buffer size for RX/TX data.

The PLC samples define and use the following internal error codes:

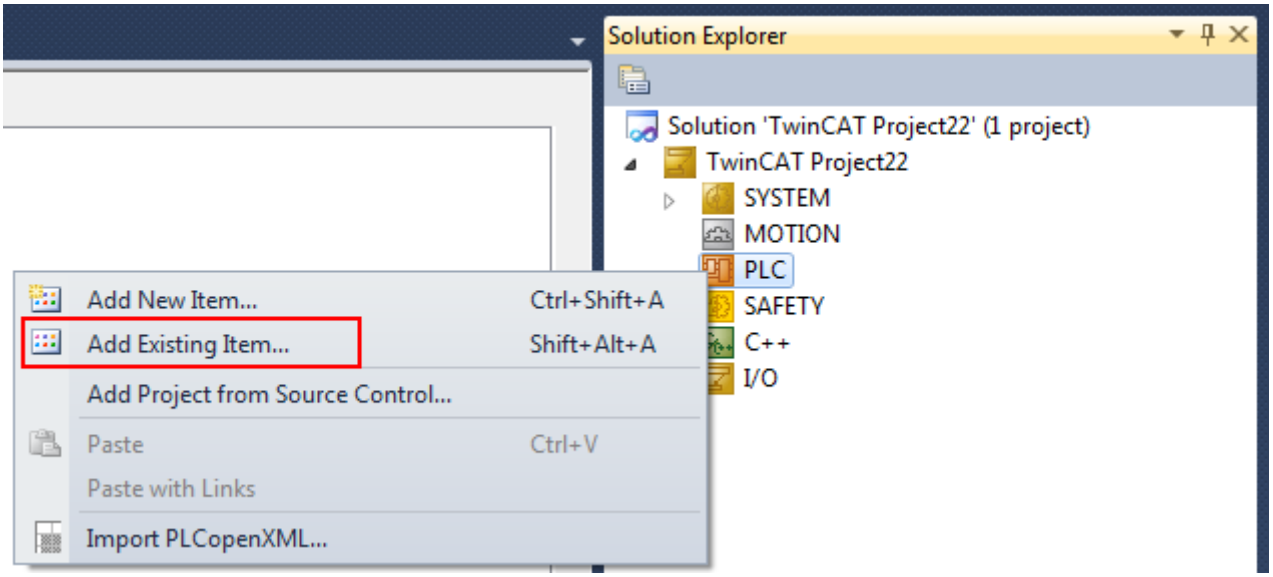
Error code	Value	Description
PLCPRJ_ERROR_RECEIVE_BUFFER_OVERFLOW	16#8101	The internal receive buffer reports an overflow.
PLCPRJ_ERROR_SEND_BUFFER_OVERFLOW	16#8102	The internal send buffer reports an overflow.
PLCPRJ_ERROR_RESPONSE_TIMEOUT	16#8103	The server has not sent the response within the specified timeout time.
PLCPRJ_ERROR_INVALID_FRAME_FORMAT	16#8104	The telegram formatting is incorrect (size, faulty data bytes etc.).

The client and server applications (FB_ServerApplication, FB_ClientApplication) were implemented as function blocks. The application and the connection can thus be instanced repeatedly.

6.1.3 Sample03: “Echo” client/server

This sample is based on the functionality offered by the former TcSocketHelper.Lib, which is now part of Tc2_TcpIp library. It realizes a Client/Server PLC application based on the functionality provided by the former SocketHelper library.

The client cyclically sends a test string (sToServer) to the remote server. The server returns the same string unchanged to the client (sFromServer). The difference between this sample and sample02 is that the server can establish up to five connections and the client application may start five client instances. Each instance establishes a connection to the server.



System requirements

- TwinCAT 3 Build 3093 or higher
- TwinCAT 3 Function TF6310 TCP/IP
- If two computers are used to execute the sample (one client and one server), the Function TF6310 needs to be installed on both computers
- If one computer is used to execute the sample, e.g. client and server running in two separate PLC runtimes, both PLC runtimes need to run in separate tasks

Project downloads

https://github.com/Beckhoff/TF6310_Samples/tree/master/PLC/TCP/Sample03

Project information

The default communication settings used in the above samples are as follows:

- PLC client application: Port and IP address of the remote server: 200, '127.0.0.1'
- PLC server application: Port and IP address of the local server: 200, '127.0.0.1'

To test the client and server application on two different PCs, you have to adjust the port and the IP address accordingly.

However, you can also test the client and server samples with the default values on a single computer by loading the client application into the first PLC runtime system and the server application into the second PLC runtime system.

The behavior of the PLC project sample is determined by the following global variables/constants.

Constant	Value	Description
PLCPRJ_MAX_CONNECTIONS	5	Max. number of server->client connections. A server can establish connections to more than one client. A client can establish a connection to only one server at a time.
PLCPRJ_SERVER_RESPONSE_TIMEOUT	T#10s	Max. delay time (timeout time) after which a server should send a response to the client.
PLCPRJ_CLIENT_SEND_CYCLE_TIME	T#1s	Cycle time based on which a client sends send data (TX) to the server.
PLCPRJ_RECEIVER_POLLING_CYCLE_TIME	T#200ms	Cycle time based on which a client or server polls for receive data (RX).
PLCPRJ_BUFFER_SIZE	10000	Max. internal buffer size for RX/TX data.

The PLC samples define and use the following internal error codes:

Error code	Value	Description
PLCPRJ_ERROR_RECEIVE_BUFFER_OVERFLOW	16#8101	The internal receive buffer reports an overflow.
PLCPRJ_ERROR_SEND_BUFFER_OVERFLOW	16#8102	The internal send buffer reports an overflow.
PLCPRJ_ERROR_RESPONSE_TIMEOUT	16#8103	The server has not sent the response within the specified timeout time.
PLCPRJ_ERROR_INVALID_FRAME_FORMAT	16#8104	The telegram formatting is incorrect (size, faulty data bytes etc.).

The client and server applications (FB_ServerApplication, FB_ClientApplication) were implemented as function blocks. The application and the connection can thus be instanced repeatedly.

6.1.4 Sample04: Binary data exchange

This sample is based on the functionality offered by the former TcSocketHelper.Lib, which is now part of Tc2_TcpIp library. It realizes a Client/Server PLC application based on the functionality provided by the former SocketHelper library.

This sample offers a client-server application for the exchange of binary data. To achieve this, a simple sample protocol is implemented. The length of the binary data and a frame counter for the sent and received telegrams are transferred in the protocol header.

The structure of the binary data is defined by the PLC structure ST_ApplicationBinaryData. The binary data are appended to the headers and transferred. The instances of the binary structure are called toServer, fromServer on the client side and toClient, fromClient on the server side.

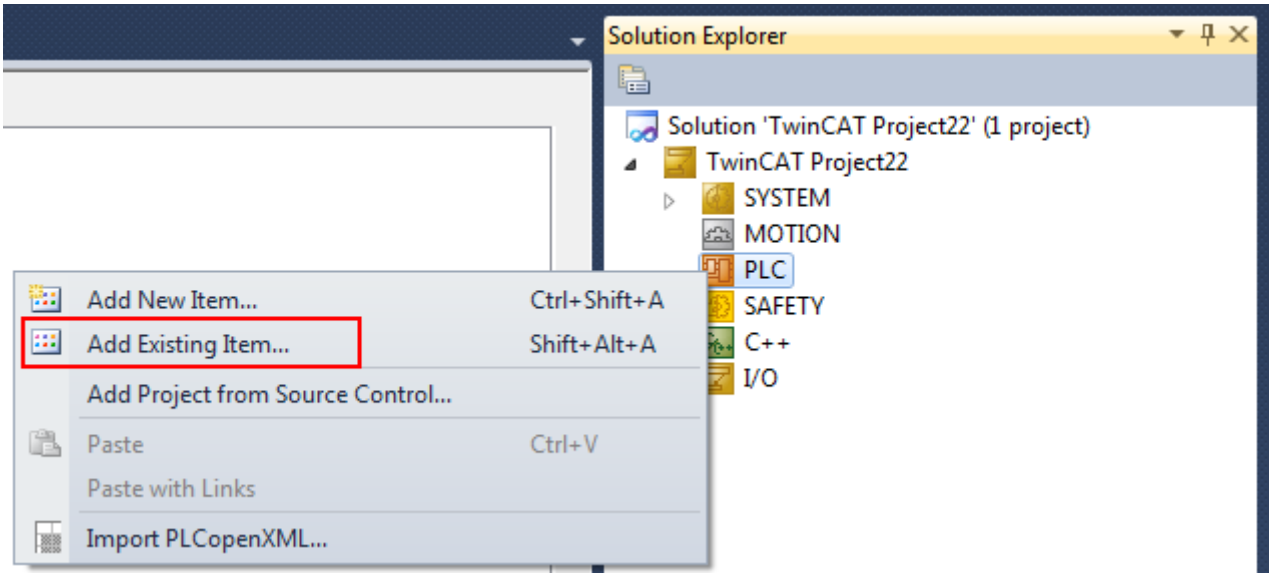
The structure declaration on the client and server sides can be adapted as required. The structure declaration must be identical on both sides.

The maximum size of the structure must not exceed the maximum buffer size of the send/receive Fifos. The maximum buffer size is determined by a constant.

The server functionality is implemented in the function block FB_ServerApplication and the client functionality in the function block FB_ClientApplication.

In the standard implementation the client cyclically sends the data of the binary structure to the server and waits for a response from the server. The server modifies some data and returns them to the client.

If you require a functionality, you have to modify the function blocks FB_ServerApplication and FB_ClientApplication accordingly.



System requirements

- TwinCAT 3 Build 3093 or higher
- TwinCAT 3 Function TF6310 TCP/IP
- If two computers are used to execute the sample (one client and one server), the Function TF6310 needs to be installed on both computers
- If one computer is used to execute the sample, e.g. client and server running in two separate PLC runtimes, both PLC runtimes need to run in separate tasks.

Project downloads

https://github.com/Beckhoff/TF6310_Samples/tree/master/PLC/TCP/Sample04

Project information

The default communication settings used in the above samples are as follows:

- PLC client application: Port and IP address of the remote server: 200, '127.0.0.1'
- PLC server application: Port and IP address of the local server: 200, '127.0.0.1'

To test the client and server application on two different PCs, you have to adjust the port and the IP address accordingly.

However, you can also test the client and server samples with the default values on a single computer by loading the client application into the first PLC runtime system and the server application into the second PLC runtime system.

The behavior of the PLC project sample is determined by the following global variables/constants.

Constant	Value	Description
PLCPRJ_MAX_CONNECTIONS	5	Max. number of server->client connections. A server can establish connections to more than one client. A client can establish a connection to only one server at a time.
PLCPRJ_SERVER_RESPONSE_TIMEOUT	T#10s	Max. delay time (timeout time) after which a server should send a response to the client.
PLCPRJ_CLIENT_SEND_CYCLE_TIME	T#1s	Cycle time based on which a client sends send data (TX) to the server.
PLCPRJ_RECEIVER_POLLING_CYCLE_TIME	T#200ms	Cycle time based on which a client or server polls for receive data (RX).
PLCPRJ_BUFFER_SIZE	10000	Max. internal buffer size for RX/TX data.

The PLC samples define and use the following internal error codes:

Error code	Value	Description
PLCPRJ_ERROR_RECEIVE_BUFFER_OVERFLOW	16#8101	The internal receive buffer reports an overflow.
PLCPRJ_ERROR_SEND_BUFFER_OVERFLOW	16#8102	The internal send buffer reports an overflow.
PLCPRJ_ERROR_RESPONSE_TIMEOUT	16#8103	The server has not sent the response within the specified timeout time.
PLCPRJ_ERROR_INVALID_FRAME_FORMAT	16#8104	The telegram formatting is incorrect (size, faulty data bytes etc.).

The client and server applications (FB_ServerApplication, FB_ClientApplication) were implemented as function blocks. The application and the connection can thus be instanced repeatedly.

6.1.5 Sample05: Binary data exchange

This sample is based on the functionality offered by the former TcSocketHelper.Lib, which is now part of Tc2_TcpIp library. It realizes a Client/Server PLC application based on the functionality provided by the former SocketHelper library.

This sample offers a client-server application for the exchange of binary data. To achieve this, a simple sample protocol is implemented. The length of the binary data and a frame counter for the sent and received telegrams are transferred in the protocol header.

The structure of the binary data is defined by the PLC structure ST_ApplicationBinaryData. The binary data are appended to the headers and transferred. The instances of the binary structure are called toServer, fromServer on the client side and toClient, fromClient on the server side.

The structure declaration on the client and server sides can be adapted as required. The structure declaration must be identical on both sides.

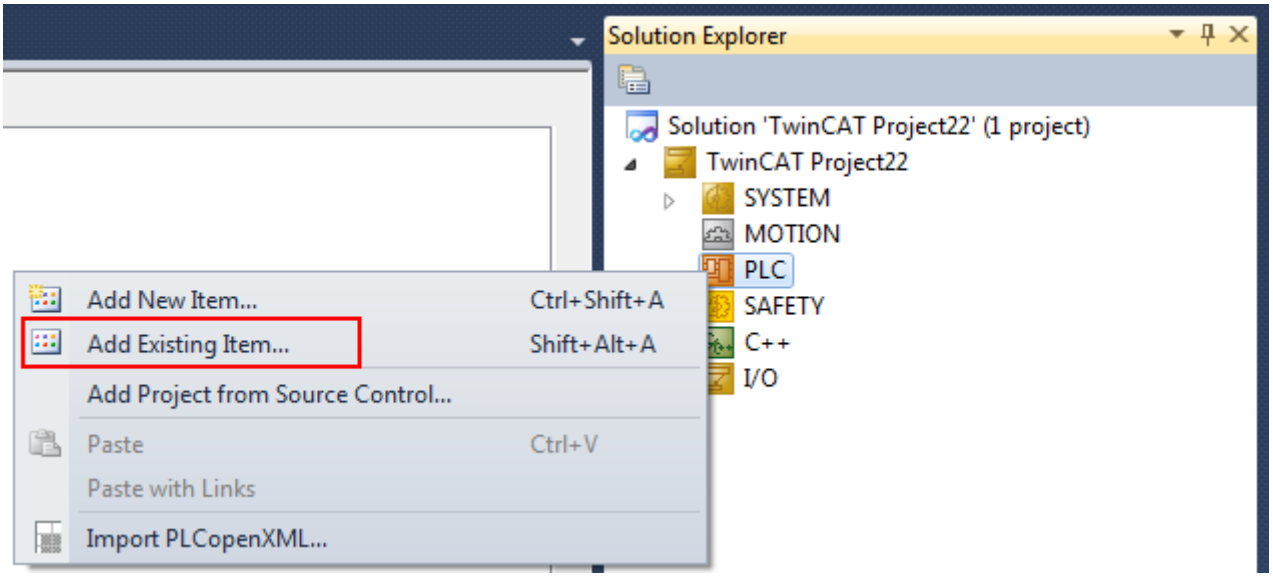
The maximum size of the structure must not exceed the maximum buffer size of the send/receive Fifos. The maximum buffer size is determined by a constant.

The server functionality is implemented in the function block FB_ServerApplication and the client functionality in the function block FB_ClientApplication.

In the standard implementation the client cyclically sends the data of the binary structure to the server and waits for a response from the server. The server modifies some data and returns them to the client.

If you require a functionality, you have to modify the function blocks FB_ServerApplication and FB_ClientApplication accordingly.

The difference between this sample and sample04 is that the server can establish up to 5 connections and the client application may have 5 client instances. Each instance establishes a connection to the server.



System requirements

- TwinCAT 3 Build 3093 or higher
- TwinCAT 3 Function TF6310 TCP/IP
- If two computers are used to execute the sample (one client and one server), the Function TF6310 needs to be installed on both computers
- If one computer is used to execute the sample, e.g. client and server running in two separate PLC runtimes, both PLC runtimes need to run in separate tasks.

Project downloads

https://github.com/Beckhoff/TF6310_Samples/tree/master/PLC/TCP/Sample05

Project information

The default communication settings used in the above samples are as follows:

- PLC client application: Port and IP address of the remote server: 200, '127.0.0.1'
- PLC server application: Port and IP address of the local server: 200, '127.0.0.1'

To test the client and server application on two different PCs, you have to adjust the port and the IP address accordingly.

However, you can also test the client and server samples with the default values on a single computer by loading the client application into the first PLC runtime system and the server application into the second PLC runtime system.

The behavior of the PLC project sample is determined by the following global variables/constants.

Constant	Value	Description
PLCPRJ_MAX_CONNECTIONS	5	Max. number of server->client connections. A server can establish connections to more than one client. A client can establish a connection to only one server at a time.
PLCPRJ_SERVER_RESPONSE_TIMEOUT	T#10s	Max. delay time (timeout time) after which a server should send a response to the client.
PLCPRJ_CLIENT_SEND_CYCLE_TIME	T#1s	Cycle time based on which a client sends send data (TX) to the server.
PLCPRJ_RECEIVER_POLLING_CYCLE_TIME	T#200ms	Cycle time based on which a client or server polls for receive data (RX).
PLCPRJ_BUFFER_SIZE	10000	Max. internal buffer size for RX/TX data.

The PLC samples define and use the following internal error codes:

Error code	Value	Description
PLCPRJ_ERROR_RECEIVE_BUFFER_OVERFLOW	16#8101	The internal receive buffer reports an overflow.
PLCPRJ_ERROR_SEND_BUFFER_OVERFLOW	16#8102	The internal send buffer reports an overflow.
PLCPRJ_ERROR_RESPONSE_TIMEOUT	16#8103	The server has not sent the response within the specified timeout time.
PLCPRJ_ERROR_INVALID_FRAME_FORMAT	16#8104	The telegram formatting is incorrect (size, faulty data bytes etc.).

The client and server applications (FB_ServerApplication, FB_ClientApplication) were implemented as function blocks. The application and the connection can thus be instanced repeatedly.

6.2 UDP

6.2.1 Sample01: Peer-to-peer communication

6.2.1.1 Overview

The following example demonstrates the implementation of a simple Peer-to-Peer application in the PLC and consists of two PLC projects (PeerA and PeerB) plus a .NET application which also acts as a separate peer. All peer applications send a test string to a remote peer and at the same time receive strings from a remote peer. The received strings are displayed in a message box on the monitor of the target computer. Feel free to use and customize this sample to your needs.

System requirements

- TwinCAT 3 Build 3093 or higher
- TwinCAT 3 Function TF6310 TCP/IP
- If two computers are used to execute the sample, the Function TF6310 needs to be installed on both computers
- If one computer is used to execute the sample, e.g. Peer A und Peer B running in two separate PLC runtimes, both PLC runtimes need to run in separate tasks
- To run the .NET sample client, only .NET Framework 4.0 is needed

Project downloads

The sources of the two PLC devices only differ in terms of different IP addresses of the remote communication partners.

https://github.com/Beckhoff/TF6310_Samples/tree/master/PLC/UDP/Sample01

https://github.com/Beckhoff/TF6310_Samples/tree/master/C%23/SampleClientUdp

Project description

The following links provide documentation for each component. Additionally, an own article explains how to start the PLC samples with step-by-step instructions.

- Integration in TwinCAT and Test [▶ 83] (Starting the PLC samples)
- PLC devices A and B [▶ 84] (Peer-to-Peer PLC application)
- .NET communication [▶ 88] (.NET sample client)

Auxiliary functions in the PLC sample projects

In the PLC samples, several functions, constants and function blocks are used, which are briefly described below:

Fifo function block

```
FUNCTION_BLOCK FB_Fifo
VAR_INPUT
    new : ST_FifoEntry;
END_VAR
VAR_OUTPUT
    bOk : BOOL;
    old : ST_FifoEntry;
END_VAR
```

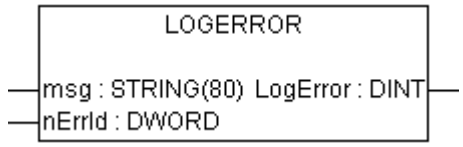
A simple Fifo function block. One instance of this block is used as "send Fifo", another one as "receive Fifo". The messages to be sent are stored in the send Fifo, the received messages are stored in the receive Fifo. The bOk output variable is set to FALSE if errors occurred during the last action (AddTail or RemoveHead) (Fifo empty or overfilled).

A Fifo entry consists of the following components:

```
TYPE ST_FifoEntry :
STRUCT
    sRemoteHost : STRING(15); (* Remote address. String containing an (Ipv4) Internet Protocol dotte
d address. *)
    nRemotePort : UDINT; (* Remote Internet Protocol (IP) port. *)
    msg         : STRING; (* Udp packet data *)
END_STRUCT
END_TYPE
```

LogError function

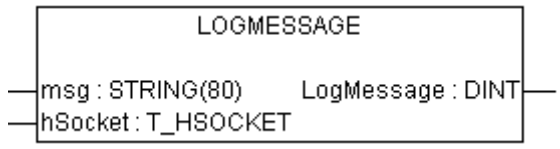
```
FUNCTION LogError : DINT
```



The function writes a message with the error code into the log book of the operating system (Event Viewer). The global variable bLogDebugMessages must first be set to TRUE.

LogMessage function

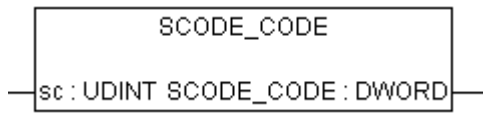
```
FUNCTION LogMessage : DINT
```



The function writes a message into the log book of the operating system (Event Viewer) if a new socket was opened or closed. The global variable bLogDebugMessages must first be set to TRUE.

SCODE_CODE function

```
FUNCTION SCODE_CODE : DWORD
```



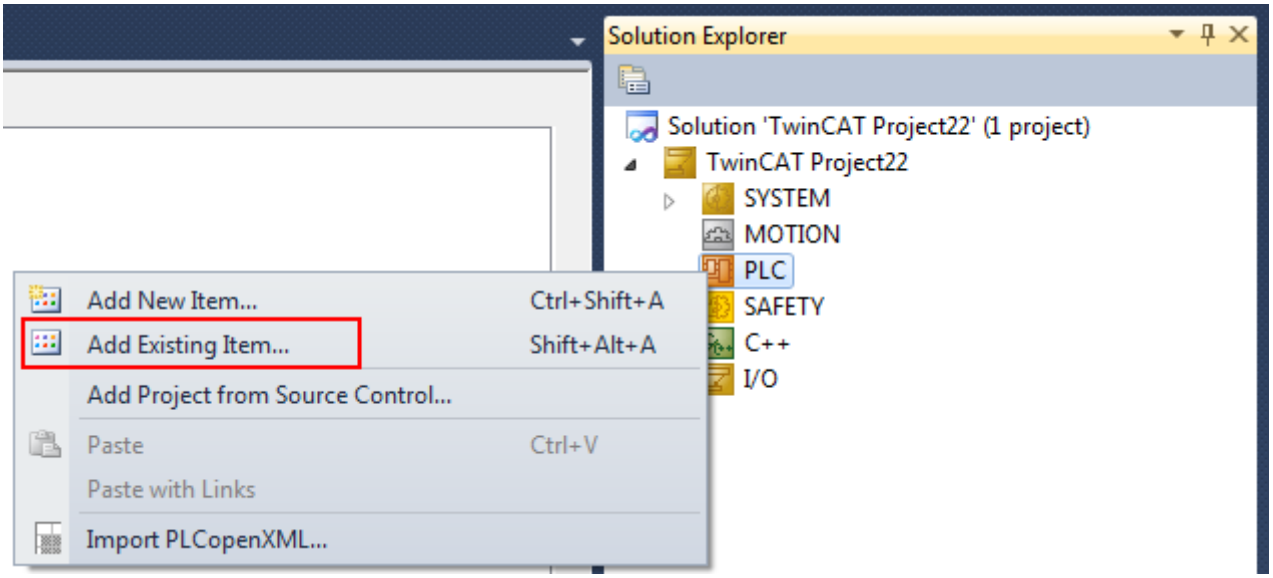
The function masks the lower 16 bits of a Win32 error code returns them.

Also see about this

.NET communication [▶ 88]

6.2.1.2 Integration in TwinCAT and Test

The PLC samples are delivered as a TwinCAT 3 PLC project file. Therefore you need to create a new TwinCAT 3 solution before importing a sample. You can then import the PLC sample in TwinCAT XAE by right-clicking on the PLC node, selecting **Add existing item** and then navigating to the downloaded sample file (please choose *Plc 3.x Project archive (*.tpzip)* as the file type).



Starting this sample requires two computers. Alternatively, the test may also be carried out with two runtime systems on a single computer. The constants with the port numbers and the IP addresses of the communication partners have to be modified accordingly.

Sample configuration with two computers:

- Device A is located on the local computer and has the IP address '10.1.128.21'
- Device B is located on the remote computer and has the IP address '172.16.6.195' 10.1.128.

Device A

Please perform the following steps to configure the sample on device A:

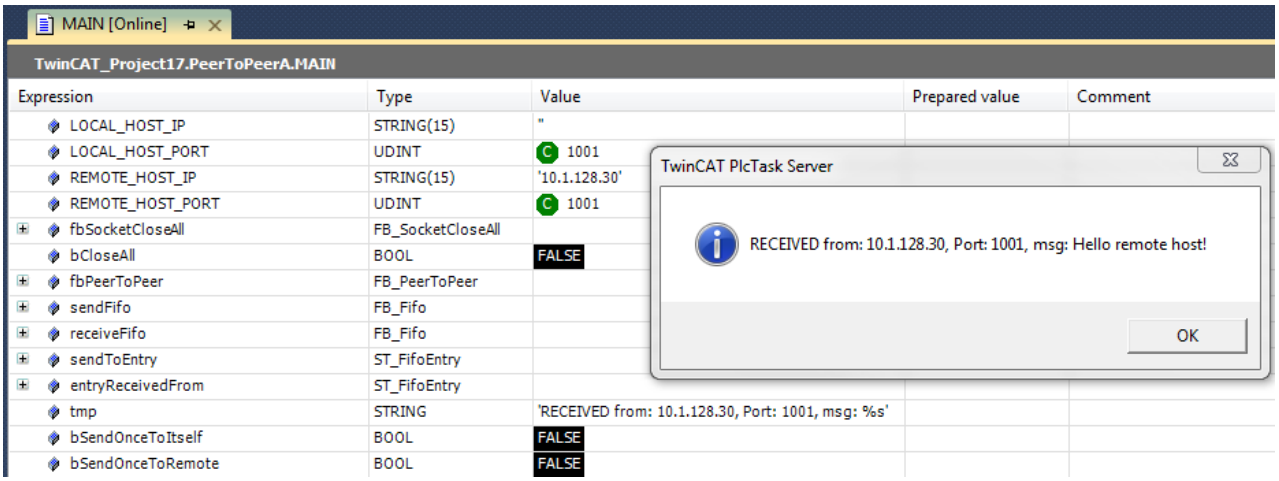
- Create a new TwinCAT 3 solution in TwinCAT XAE and import the Peer-to-Peer PLC project for device A.
- Set the constant REMOTE_HOST_IP in POU MAIN to the real IP address of the remote system (device B - in our example: '10.1.128.').
- Activate the configuration and start the PLC runtime. (Don't forget to create a license for TF6310 TCP/IP)

Device B

Please perform the following steps to configure the sample on device B:

- Create a new TwinCAT 3 solution in TwinCAT XAE and import the Peer-to-Peer PLC project for device B.
- Set the constant REMOTE_HOST_IP in POU MAIN to the IP address of device A (in our example: '10.1.128.21').
- Activate the configuration and start the PLC runtime. (Don't forget to create a license for TF6310 TCP/IP.)
- Login to the PLC runtime and write the value TRUE to the Boolean variable bSendOnceToRemote in POU MAIN.

- Shortly afterwards, a message box with the test string should appear on device A. You can now also repeat the same step on device A. As a result, the message box should then appear on device B.

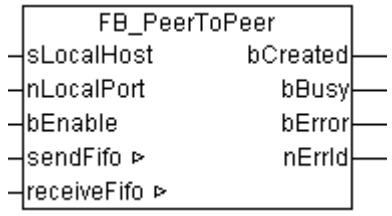


6.2.1.3 PLC devices A and B

The required functionality was encapsulated in the function block FB_PeerToPeer. Each of the communication partners uses an instance of the FB_PeerToPeer function block. The block is activated through a rising edge at the bEnable input. A new UDP socket is opened, and data exchange commences. The socket address is specified via the variables sLocalHost and nLocalPort. A falling edge stops the data exchange and closes the socket. The data to be sent are transferred to the block through a reference (VAR_IN_OUT) via the variable sendFifo. The data received are stored in the variable receiveFifo.

Name	Default value	Description
g_sTcIpConnSvrAddr	''	Network address of the TwinCAT TCP/IP Connection Server. Default: Empty string (the server is located on the local PC);
bLogDebugMessages	TRUE	Activates/deactivates writing of messages into the log book of the operating system;
PLCPRJ_ERROR_SENDFIFO_OVERFLOW	16#8103	Sample project error code: The send Fifo is full.
PLCPRJ_ERROR_RECFIFO_OVERFLOW	16#8104	Sample project error code: The receive Fifo is full.

FUNCTION_BLOCK FB_PeerToPeer



Interface

```
VAR_IN_OUT
    sendFifo    : FB_Fifo;
    receiveFifo : FB_Fifo;
END_VAR
VAR_INPUT
    sLocalHost  : STRING(15);
    nLocalPort  : UDINT;
    bEnable     : BOOL;
END_VAR
VAR_OUTPUT
```

```
    bCreated    : BOOL;
    bBusy       : BOOL;
    bError      : BOOL;
    nErrId      : UDINT;
END_VAR
VAR
    fbCreate    : FB_SocketUdpCreate;
    fbClose     : FB_SocketClose;
    fbReceiveFrom : FB_SocketUdpReceiveFrom;
    fbSendTo    : FB_SocketUdpSendTo;
    hSocket     : T_HSOCKET;
    eStep       : E_ClientServerSteps;
    sendTo      : ST_FifoEntry;
    receivedFrom : ST_FifoEntry;
END_VAR
```

Implementation

```
CASE eStep OF
    UDP_STATE_IDLE:
        IF bEnable XOR bCreated THEN
            bBusy := TRUE;
            bError := FALSE;
            nErrid := 0;
            IF bEnable THEN
                eStep := UDP_STATE_CREATE_START;
            ELSE
                eStep := UDP_STATE_CLOSE_START;
            END_IF
        ELSIF bCreated THEN
            sendFifo.RemoveHead( old => sendTo );
            IF sendFifo.bOk THEN
                eStep := UDP_STATE_SEND_START;
            ELSE (* empty *)
                eStep := UDP_STATE_RECEIVE_START;
            END_IF
        ELSE
            bBusy := FALSE;
        END_IF

    UDP_STATE_CREATE_START:
        fbCreate( bExecute := FALSE );
        fbCreate( sSrvNetId:= g_sTcIpConnSvrAddr,
                sLocalHost:= sLocalHost,
            nLocalPort:= nLocalPort,
            bExecute:= TRUE );
        eStep := UDP_STATE_CREATE_WAIT;

    UDP_STATE_CREATE_WAIT:
        fbCreate( bExecute := FALSE );
        IF NOT fbCreate.bBusy THEN
            IF NOT fbCreate.bError THEN
                bCreated := TRUE;
                hSocket := fbCreate.hSocket;
                eStep := UDP_STATE_IDLE;
                LogMessage( 'Socket opened (UDP)!', hSocket );
            ELSE
                LogError( 'FB_SocketUdpCreate', fbCreate.nErrId );
                nErrId := fbCreate.nErrId;
                eStep := UDP_STATE_ERROR;
            END_IF
        END_IF

    UDP_STATE_SEND_START:
        fbSendTo( bExecute := FALSE );
        fbSendTo( sSrvNetId:=g_sTcIpConnSvrAddr,
            sRemoteHost := sendTo.sRemoteHost,
                nRemotePort := sendTo.nRemotePort,
            hSocket:= hSocket,
            pSrc:= ADR( sendTo.msg ),
            cbLen:= LEN( sendTo.msg ) + 1, (* include the end delimiter *)
         bExecute:= TRUE );
        eStep := UDP_STATE_SEND_WAIT;

    UDP_STATE_SEND_WAIT:
        fbSendTo( bExecute := FALSE );
        IF NOT fbSendTo.bBusy THEN
            IF NOT fbSendTo.bError THEN
                eStep := UDP_STATE_RECEIVE_START;
```

```
            ELSE
                LogError( 'FB_SocketSendTo (UDP)', fbSendTo.nErrId );
                nErrId := fbSendTo.nErrId;
                eStep := UDP_STATE_ERROR;
            END_IF
        END_IF

    UDP_STATE_RECEIVE_START:
        MEMSET( ADR( receivedFrom ), 0, SIZEOF( receivedFrom ) );
     fbReceiveFrom( bExecute := FALSE );
        fbReceiveFrom( sSrvNetId:=g_sTcIpConnSvrAddr,
                hSocket:= hSocket,
                pDest:= ADR( receivedFrom.msg ),
                      cbLen:= SIZEOF( receivedFrom.msg ) - 1, (*without string delimiter *)
             bExecute:= TRUE );
        eStep := UDP_STATE_RECEIVE_WAIT;

    UDP_STATE_RECEIVE_WAIT:
        fbReceiveFrom( bExecute := FALSE );
        IF NOT fbReceiveFrom.bBusy THEN
            IF NOT fbReceiveFrom.bError THEN
                IF fbReceiveFrom.nRecBytes > 0 THEN
                    receivedFrom.nRemotePort := fbReceiveFrom.nRemotePort;
                    receivedFrom.sRemoteHost := fbReceiveFrom.sRemoteHost;
                    receiveFifo.AddTail( new := receivedFrom );
                    IF NOT receiveFifo.bOk THEN(* Check for fifo overflow *)
                     LogError( 'Receive fifo overflow!', PLCPRJ_ERROR_RECFIFO_OVERFLOW );
                    END_IF
                END_IF
                eStep := UDP_STATE_IDLE;
            ELSIF fbReceiveFrom.nErrId = 16#80072746 THEN
                      LogError( 'The connection is reset by remote side.', fbReceiveFrom.nErrId );
                eStep := UDP_STATE_IDLE;
            ELSE
                LogError( 'FB_SocketUdpReceiveFrom (UDP client/server)', fbReceiveFrom.nErrId );
                nErrId := fbReceiveFrom.nErrId;
                eStep := UDP_STATE_ERROR;
            END_IF
        END_IF

    UDP_STATE_CLOSE_START:
        fbClose( bExecute := FALSE );
        fbClose(    sSrvNetId:= g_sTcIpConnSvrAddr,
            hSocket:= hSocket,
            bExecute:= TRUE );
        eStep := UDP_STATE_CLOSE_WAIT;

    UDP_STATE_CLOSE_WAIT:
        fbClose( bExecute := FALSE );
        IF NOT fbClose.bBusy THEN
            LogMessage( 'Socket closed (UDP)!', hSocket );
            bCreated := FALSE;
            MEMSET( ADR(hSocket), 0, SIZEOF(hSocket));
            IF fbClose.bError THEN
                LogError( 'FB_SocketClose (UDP)', fbClose.nErrId );
                     nErrId := fbClose.nErrId;
                eStep := UDP_STATE_ERROR;
            ELSE
                bBusy := FALSE;
                bError := FALSE;
                nErrId := 0;
                eStep := UDP_STATE_IDLE;
            END_IF
        END_IF

    UDP_STATE_ERROR: (* Error step *)
        bError := TRUE;
        IF bCreated THEN
            eStep := UDP_STATE_CLOSE_START;
        ELSE
            bBusy := FALSE;
            eStep := UDP_STATE_IDLE;
        END_IF
END_CASE
```

MAIN program

Previously opened sockets must be closed after a program download or a PLC reset. During PLC start-up, this is done by calling an instance of the FB_SocketCloseAll [▶ 25] function block. If one of the variables bSendOnceToItself or bSendOnceToRemote has a raising edge, a new Fifo entry is generated and stored in the send Fifo. Received messages are removed from the receive Fifo and displayed in a message box.

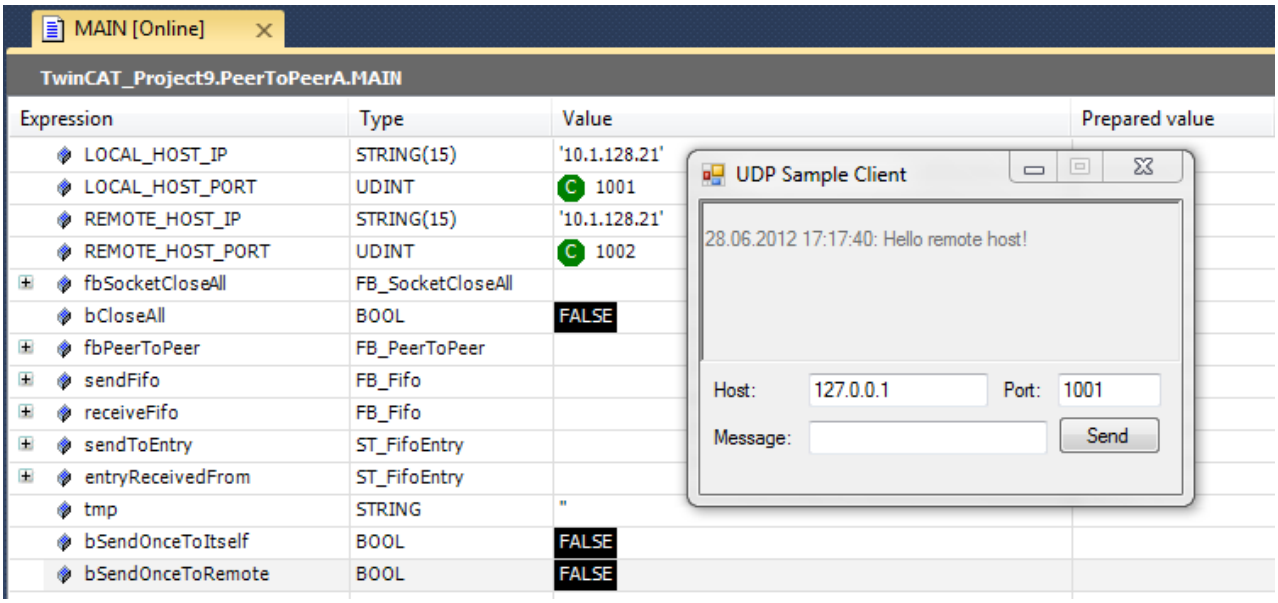
```
PROGRAM MAIN
VAR CONSTANT
    LOCAL_HOST_IP      : STRING(15)     := '';
    LOCAL_HOST_PORT    : UDINT         := 1001;
    REMOTE_HOST_IP     : STRING(15)     := '172.16.2.209';
    REMOTE_HOST_PORT   : UDINT         := 1001;
END_VAR
VAR
    fbSocketCloseAll    : FB_SocketCloseAll;
    bCloseAll     : BOOL := TRUE;

    fbPeerToPeer     : FB_PeerToPeer;
    sendFifo        : FB_Fifo;
    receiveFifo     : FB_Fifo;
    sendToEntry     : ST_FifoEntry;
    entryReceivedFrom : ST_FifoEntry;
    tmp         : STRING;

    bSendOnceToItself : BOOL;
    bSendOnceToRemote : BOOL;
END_VAR
```

```
IF bCloseAll THEN (*On PLC reset or program download close all old connections *)
 bCloseAll := FALSE;
    fbSocketCloseAll( sSrvNetId:= g_sTcIpConnSvrAddr, bExecute:= TRUE, tTimeout:= T#10s );
ELSE
    fbSocketCloseAll( bExecute:= FALSE );
END_IF

IF NOT fbSocketCloseAll.bBusy AND NOT fbSocketCloseAll.bError THEN

    IF bSendOnceToRemote THEN
        bSendOnceToRemote         := FALSE;                  (* clear flag *)
     sendToEntry.nRemotePort         := REMOTE_HOST_PORT;         (* remote host port number*)
     sendToEntry.sRemoteHost         := REMOTE_HOST_IP;             (* remote host IP address *)
     sendToEntry.msg             := 'Hello remote host!';            (* message text*);
     sendFifo.AddTail( new := sendToEntry );                 (* add new entry to the send queue*)
     IF NOT sendFifo.bOk THEN                            (* check for fifo overflow*)
         LogError( 'Send fifo overflow!', PLCPRJ_ERROR_SENDFIFO_OVERFLOW );
        END_IF
    END_IF

    IF bSendOnceToItself THEN
        bSendOnceToItself         := FALSE;                  (* clear flag *)
     sendToEntry.nRemotePort         := LOCAL_HOST_PORT;         (* nRemotePort == nLocalPort => sen
d it to itself *)
       sendToEntry.sRemoteHost         := LOCAL_HOST_IP;             (* sRemoteHost == sLocalHost =>
send it to itself *)
     sendToEntry.msg             := 'Hello itself!';         (* message text*);
     sendFifo.AddTail( new := sendToEntry );                 (* add new entry to the send queue*)
     IF NOT sendFifo.bOk THEN                            (* check for fifo overflow*)
         LogError( 'Send fifo overflow!', PLCPRJ_ERROR_SENDFIFO_OVERFLOW );
        END_IF
    END_IF

 (* send and receive messages *)
 fbPeerToPeer( sendFifo := sendFifo, receiveFifo := receiveFifo, sLocalHost := LOCAL_HOST_IP, nLocal
Port := LOCAL_HOST_PORT, bEnable := TRUE );

 (* remove all received messages from receive queue *)
 REPEAT
        receiveFifo.RemoveHead( old => entryReceivedFrom );
        IF receiveFifo.bOk THEN
            tmp := CONCAT( 'RECEIVED from: ', entryReceivedFrom.sRemoteHost );
            tmp := CONCAT( tmp, ', Port: ' );
            tmp := CONCAT( tmp, UDINT_TO_STRING( entryReceivedFrom.nRemotePort ) );
            tmp := CONCAT( tmp, ', msg: %s' );
            ADSLOGSTR( ADSLOG_MSGTYPE_HINT OR ADSLOG_MSGTYPE_MSGBOX, tmp, entryReceivedFrom.msg );
        END_IF
    UNTIL NOT receiveFifo.bOk
```

```
    END_REPEAT

END_IF
```

6.2.1.4 .NET communication

This sample demonstrates how a .NET communication partner for PLC samples Peer-to-Peer device A or B can be realized.



The .NET Sample Client can be used to send single UPD data packages to a UPD Server, in this case the PLC project PeerToPeerA.

Download

Download the test client.

Unpack the ZIP file; the .exe file runs on a Windows system.

How it works

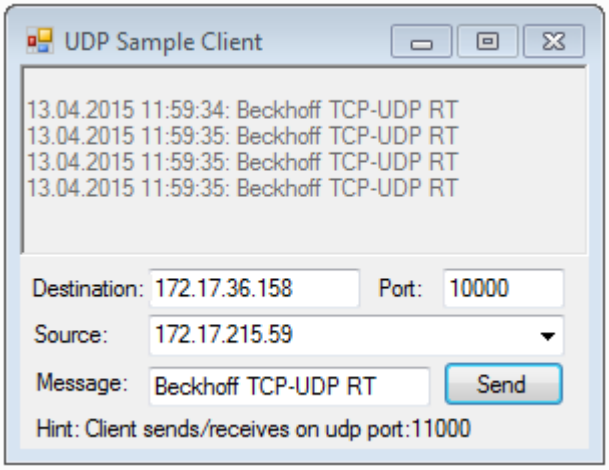
The sample uses the .Net libraries System.Net and System.Net.Sockets to implement a UDP client (class UdpClient). While listening for incoming UDP packets in a background thread, a string can be sent to a remote device by specifying its IP address and port number and clicking the Send button.

For a better understanding of this article, imagine the following setup:

- The PLC project Peer-to-Peer device A is running on a computer with IP address 10.1.128.21
- The .NET application is running on a computer with IP address 10.1.128.30

Description

The client itself uses port 11000 for sending. At the same time it opens this port and displays received messages in the upper part of the interface as a log:



Together with the PLC / C++ examples, this results in an echo example:
A UDP message is sent from the client port 11000 to the server port 10000, which returns the same data to the sender.

The client can be configured via the interface:

- Destination: IP address
- Port: The port that is addressed in the destination
- Source: Sender network card (IP address).
  "OS-based" operating system deals with selection of the appropriate network card.
- Message

TF6311 “TCP/UDP Realtime” does not allow local communication. However, for testing purposes a different network interface can be selected via "Source", so that the UDP packet leaves the computer through one network card and arrives on the other network card ("loop cable").

6.2.2 Sample02: Multicast

This sample demonstrates how to send and receive Multicast packages via UDP.

Client and Server cyclically send a value to each other via a Multicast IP address.

Client and Server are realized by two PLC applications and delivered within a single TwinCAT 3 solution.

System requirements

- TwinCAT 3 Build 3093 or higher
- TwinCAT 3 Function TF6310 TCP/IP version 1.0.64 or higher
- TwinCAT 3 Library Tc2_TcpIp version 3.2.64.0 or higher
- If one computer is used to execute the sample, e.g. client and server running in two separate PLC runtimes, both PLC runtimes need to run in separate tasks.

Project download

https://github.com/Beckhoff/TF6310_Samples/tree/master/PLC/UDP/Sample02

7 Appendix

7.1 OSI model

The following article is a short introduction into the OSI model and describes how this model takes part in our everyday network communication. Note that the ambition to create this article was not to replace more detailed documentations or books about this topic, therefore please only consider it to be a very superficial introduction.

The OSI (Open Systems Interconnection) model describes a standardization of the functionalities in a communication system via abstract layers. Each layer defines an own set of functionalities during the communication between network devices and only communicates with the layer above and below.

OSI model

Layer	Name	Example protocols
7	Application Layer	HTTP, FTP, DNS, SNMP, Telnet
6	Presentation Layer	SSL, TLS
5	Session Layer	NetBIOS, PPTP
4	Transport Layer	TCP, UDP
3	Network Layer	IP, ARP, ICMP, IPSec
2	Data Link Layer	PPP, ATM, Ethernet
1	Physical Layer	Ethernet, USB, Bluetooth, IEEE802.11

Example: If you use your web browser to navigate to http://wwwbeckhoff.com, this communication uses the following protocols from each layer, starting at layer 7: HTTP → TCP → IP → Ethernet. On the other hand, entering https://www.beckhoff.com would use HTTP → SSL → TCP → IP → Ethernet.

The TwinCAT 3 Function TF6310 TCP/IP provides functionalities to develop network-enabled PLC programs using either the transport protocols TCP or UDP. Therefore, PLC programmers may implement their own application layer protocol, defining an own message structure to communicate with remote systems.

7.2 KeepAlive configuration

The transmission of TCP KeepAlive messages verifies if an idle TCP connection is still active. Since version 1.0.47 of the TwinCAT TCP/IP Server (TF6310), the KeepAlive configuration of the Windows operating system is used, which can be configured via the following registry keys:

The following documentation is an excerpt of a Microsoft Technet article.

KeepAliveTime

HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters

Data type	Range	Default value
REG_DWORD	0x1–0xFFFFFFFF (*milliseconds*)	0x6DDD00 (*7,200,000 milliseconds = 2 hours*)

Description

Determines how often TCP sends keep-alive transmissions. TCP sends keep-alive transmissions to verify that an idle connection is still active.This entry is used when the remote system is responding to TCP. Otherwise, the interval between transmissions is determined by the value of the KeepAliveInterval entry. By default, keep-alive transmissions are not sent. The TCP keep-alive feature must be enabled by a program, such as Telnet, or by an Internet browser, such as Internet Explorer.

KeepAliveInterval

HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters

Data type	Range	Default value
REG_DWORD	0x1–0xFFFFFFFF (*milliseconds*)	0x3E8 (*1,000 milliseconds = 1 second*)

Description

Determines how often TCP repeats keep-alive transmissions when no response is received. TCP sends keep-alive transmissions to verify that idle connections are still active. This prevents TCP from inadvertently disconnecting active lines.

7.3 Error codes

7.3.1 Overview of the error codes

Codes (hex)	Codes (dec)	Error source	Description
0x00000000-0x00007800	0-30720	TwinCAT system error codes	TwinCAT system error (including ADS error codes)
0x00008000-0x000080FF	32768-33023	Internal TwinCAT TCP/IP Connection Server error codes [▶ 92]	Internal error of the TwinCAT TCP/IP Connection Server
0x80070000-0x8007FFFF	2147942400-2148007935	Error source = Code - 0x80070000 = Win32 system error codes	Win32 system error (including Windows sockets error codes)

Requirements

Development environment	Target system type	PLC libraries to include
TwinCAT v3.1	PC, CX (x86) or CX (ARM)	Tc2_TcpIp

7.3.2 Internal error codes of the TwinCAT TCP/IP Connection Server

Code (hex)	Code (dec)	Symbolic constant	Description
0x00008001	32769	TCPADSERROR_NOMOREENTRIES	No new sockets can be created (for FB_SocketListen and FB_SocketConnect).
0x00008002	32770	TCPADSERROR_NOTFOUND	Socket handle is invalid (for FB_SocketReceive, FB_SocketAccept, FB_SocketSend etc.).
0x00008003	32771	TCPADSERROR_ALREADYEXISTS	Is returned when FB_SocketListen is called, if the TcpIp port listener already exists.
0x00008004	32772	TCPADSERROR_NOTCONNECTED	Is returned when FB_SocketReceive is called, if the client socket is no longer connected with the server.
0x00008005	32773	TCPADSERROR_NOTLISTENING	Is returned when FB_SocketAccept is called, if an error was registered in the listener socket.

Requirements

Development environment	Target system type	PLC libraries to include
TwinCAT v3.1	PC, CX (x86) or CX (ARM)	Tc2_TcpIp

7.3.3 Troubleshooting/diagnostics

- In the event of connection problems the PING command can be used to ascertain whether the external communication partner can be reached via the network connection. If this is not the case, check the network configuration and firewall settings.
- Sniffer tools such as Wireshark enable logging of the entire network communication. The log can then be analysed by Beckhoff support staff.
- Check the hardware and software requirements described in this documentation (TwinCAT version, CE image version etc.).
- Check the software installation hints described in this documentation (e.g. installation of CAB files on CE plattform).
- Check the input parameters that are transferred to the function blocks (network address, port number, data etc, connection handle.) for correctness. Check whether the function block issues an error code. The documentation for the error codes can be found here: Overview of error codes [▶ 91].
- Check if the other communication partner/software/device issues an error code.
- Activate the debug output integrated in the TcSocketHelper.Lib during connection establishment/ disconnect process (keyword: CONNECT_MODE_ENABLEDBG). Open the TwinCAT System Manager and activate the LogView window. Analyze/check the debug output strings.

Requirements

Development environment	Target system type	PLC libraries to include
TwinCAT v3.1	PC, CX (x86) or CX (ARM)	Tc2_TcpIp

7.4 Support and Service

Beckhoff and their partners around the world offer comprehensive support and service, making available fast and competent assistance with all questions related to Beckhoff products and system solutions.

Beckhoff's branch offices and representatives

Please contact your Beckhoff branch office or representative for local support and service on Beckhoff products!

The addresses of Beckhoff's branch offices and representatives round the world can be found on her internet pages:
http://www.beckhoff.com

You will also find further documentation for Beckhoff components there.

Beckhoff Headquarters

Beckhoff Automation GmbH & Co. KG

Huelshorstweg 20
33415 Verl
Germany

Phone:	+49(0)5246/963-0
Fax:	+49(0)5246/963-198
e-mail:	info@beckhoff.com

Beckhoff Support

Support offers you comprehensive technical assistance, helping you not only with the application of individual Beckhoff products, but also with other, wide-ranging services:

- support
- design, programming and commissioning of complex automation systems
- and extensive training program for Beckhoff system components

Hotline:	+49(0)5246/963-157
Fax:	+49(0)5246/963-9157
e-mail:	support@beckhoff.com

Beckhoff Service

The Beckhoff Service Center supports you in all matters of after-sales service:

- on-site service
- repair service
- spare parts service
- hotline service

Hotline:	+49(0)5246/963-460
Fax:	+49(0)5246/963-479
e-mail:	service@beckhoff.com

More Information:
www.beckhoff.com/tf6310

Beckhoff Automation GmbH & Co. KG
Hülshorstweg 20
33415 Verl
Germany
Phone: +49 5246 9630
info@beckhoff.com
www.beckhoff.com