

Manual | EN

TS6310

TwinCAT 2 | TCP/IP Connection Server

Supplement | Communication



Table of contents

1	Foreword	5
1.1	Notes on the documentation	5
1.2	For your safety	6
1.3	Notes on information security	7
2	Overview	8
3	Introduction	9
4	System requirements	11
5	Installation	12
6	Installation Windows CE	14
7	PLC libraries	18
7.1	Tcplp.lib	18
7.1.1	FB_SocketConnect	18
7.1.2	FB_SocketClose	19
7.1.3	FB_SocketCloseAll	20
7.1.4	FB_SocketListen	22
7.1.5	FB_SocketAccept	23
7.1.6	FB_SocketSend	24
7.1.7	FB_SocketReceive	25
7.1.8	FB_SocketUdpCreate	26
7.1.9	FB_SocketUdpSendTo	27
7.1.10	FB_SocketUdpReceiveFrom	29
7.1.11	FB_SocketUdpAddMulticastAddress	30
7.1.12	FB_SocketUdpDropMulticastAddress	31
7.1.13	F_GetVersionTcplp	32
7.1.14	ST_SockAddr	33
7.1.15	T_HSOCKET	33
7.1.16	E_WinsockError	34
7.1.17	Global Variables	36
7.2	TcSocketHelper.lib	36
7.2.1	FB_ServerClientConnection	37
7.2.2	FB_ClientServerConnection	40
7.2.3	F_CreateServerHnd	42
7.2.4	F_GetVersionTcSocketHelper	43
7.2.5	T_HSERVER	43
7.2.6	E_SocketAcceptMode	44
7.2.7	E_SocketConnectionState	44
7.2.8	Global constants	44
7.3	TcSnmp.lib	45
7.3.1	FB_SEND_TRAP	45
7.3.2	FB_GetSnmp	46
7.3.3	F_GetVersionTcSNMP	48
7.3.4	SNMP_ST_VariableBinding	48
7.3.5	E_SNMP_GenericTrapNumber	49

7.3.6	E_SNMP_DataTypes	49
7.3.7	Global Variables	49
8	Samples	51
8.1	Tcplp.lib	51
8.1.1	TCP example	51
8.1.2	UDP example	70
8.2	TcSocketHelper.lib examples	80
8.3	TcSnmp.lib	82
8.3.1	Sample: Client trap	82
8.3.2	Sample: SNMP multiple client trap	83
8.3.3	Sample: SNMP Get request	84
9	Error codes	87
9.1	Internal error codes of the TwinCAT TCP/IP Connection Server	87
9.2	Troubleshooting/diagnostics	87
9.3	SNMP_ErrorCodes	88

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®, TwinCAT/BSD®, TC/BSD®, 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

© Beckhoff Automation GmbH & Co. KG, Germany.

The reproduction, distribution and utilization of this document as well as the communication of its contents to others without express authorization are prohibited.

Offenders will be held liable for the payment of damages. All rights reserved in the event of the grant of a patent, utility model or design.

1.2 For your safety

Safety regulations

Read the following explanations for your safety.

Always observe and follow product-specific safety instructions, which you may find at the appropriate places in this document.

Exclusion of liability

All the components are supplied in particular hardware and software configurations which are 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 technology who are familiar with the applicable national standards.

Signal words

The signal words used in the documentation are classified below. In order to prevent injury and damage to persons and property, read and follow the safety and warning notices.

Personal injury warnings

DANGER

Hazard with high risk of death or serious injury.

WARNING

Hazard with medium risk of death or serious injury.

CAUTION

There is a low-risk hazard that could result in medium or minor injury.

Warning of damage to property or environment

NOTICE

The environment, equipment, or data may be damaged.

Information on handling the product



This information includes, for example:
recommendations for action, assistance or further information on the product.

1.3 Notes on information security

The products of Beckhoff Automation GmbH & Co. KG (Beckhoff), insofar as they can be accessed online, are equipped with security functions that support the secure operation of plants, systems, machines and networks. Despite the security functions, the creation, implementation and constant updating of a holistic security concept for the operation are necessary to protect the respective plant, system, machine and networks against cyber threats. The products sold by Beckhoff are only part of the overall security concept. The customer is responsible for preventing unauthorized access by third parties to its equipment, systems, machines and networks. The latter should be connected to the corporate network or the Internet only if appropriate protective measures have been set up.

In addition, the recommendations from Beckhoff regarding appropriate protective measures should be observed. Further information regarding information security and industrial security can be found in our <https://www.beckhoff.com/secguide>.

Beckhoff products and solutions undergo continuous further development. This also applies to security functions. In light of this continuous further development, Beckhoff expressly recommends that the products are kept up to date at all times and that updates are installed for the products once they have been made available. Using outdated or unsupported product versions can increase the risk of cyber threats.

To stay informed about information security for Beckhoff products, subscribe to the RSS feed at <https://www.beckhoff.com/secinfo>.

2 Overview

The TwinCAT TCP/IP Connection Server enables the implementation/realisation of one or several TCP/IP server/clients in the TwinCAT PLC.

Product components

- TcIp.Lib (implements basic TCP/IP and UDP functions);
- TcSocketHelper.Lib (implements TCP/IP helper functions);
- TcSnmp.Lib (implements SNMPv1 helper functions since v1.0.59);
- TwinCAT TCP/IP Connection Server (TwinCAT Server);

3 Introduction

TCP

TCP is a connection-oriented analog protocol that can be compared to a phone connection, where participants have to establish the connection first. The TCP protocol is used in applications where confirmation is required for the data sent by the client or server. Data integrity is managed by the protocol, which requires more resources. The TCP protocol is well suited for sending larger data quantities.

TCP is a stream-oriented transport protocol, i.e. it transports a data stream without defined start and end. No information about length, start and end of a message is transferred. 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).

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.

The message structure for transferring binary data with variable length is often specified as follows: The first data bytes contain a special control character (start delimiter) and the data length of the subsequent data. This enables the receiver to detect the start and end of the message.

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

- An instance of the [FB_SocketConnect \[▶ 18\]](#) and [FB_SocketClose \[▶ 19\]](#) function blocks for establishing and closing the connection to the remote server (Hint: [FB_ClientServerConnection \[▶ 40\]](#) encapsulates the functionality of both function blocks in one function block);
- An instance of the [FB_SocketSend \[▶ 24\]](#) and/or [FB_SocketReceive \[▶ 25\]](#) function block for the data exchange with the remote server;

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

- An instance of the [FB_SocketListen \[▶ 22\]](#) function block for opening the listener socket. An instance of the [FB_SocketAccept \[▶ 23\]](#) and [FB_SocketClose \[▶ 19\]](#) function blocks for establishing and closing the connection(s) to the remote clients (Hint: [FB_ServerClientConnection \[▶ 37\]](#) encapsulates the functionality of all three function block in one function block);
- An instance of the [FB_SocketSend \[▶ 24\]](#) and/or [FB_SocketReceive \[▶ 25\]](#) function block for the data exchange with the remote clients;

An instance of the [FB_SocketCloseAll \[▶ 20\]](#) function block is required in each PLC runtime system in which a socket is opened.

The instances of the [FB_SocketAccept \[▶ 23\]](#) and [FB_SocketReceive \[▶ 25\]](#) function blocks are called cyclically (polling), all others are called as required.

UDP

UDP is a connection-less protocol. Data are sent between devices without an explicit connection. The UDP protocol is well suited for sending small data quantities. An UDP application can be both a client or a server. The UDP protocol does not guarantee that data that were sent actually reach the target (no confirmation for the packets received is sent). The individual data packets may arrive in a different order or may be lost.

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 server/client implementation:

- An instance of the [FB_SocketUdpCreate](#) [[▶ 26](#)] and [FB_SocketClose](#) [[▶ 19](#)] function blocks for opening and closing an UDP socket;
- An instance of the [FB_SocketUdpSendTo](#) [[▶ 27](#)] and/or [FB_SocketUdpReceiveFrom](#) [[▶ 29](#)] function blocks for the data exchange with other devices;
- In each PLC runtime system in which a UDP socket is opened an instance of the [FB_SocketCloseAll](#) [[▶ 20](#)] function block is required;

The instances of the [FB_SocketUdpReceiveFrom](#) [[▶ 29](#)] function block are called cyclically (polling), all others are called as required.

Further information can be found on the following documentation pages.

Glossary

Term	Description
TwinCAT TCP/IP Connection Server	A TwinCAT Server that enables opening, closing, sending and receiving of data via the Windows sockets.
Remote-Client	A client on a remote computer (from a server point of view) with which the server wants to communicate.
Remote-Server	A server on a remote computer (from a client point of view) with which the client wants to communicate.
Local-Client	A client on the local computer.
Local-Server	A server on the local computer.
Connection handle (socket handle)	A PLC variable of type T_HSOCKET [▶ 33]

4 System requirements

The following system requirements need to be met for the TwinCAT TCP/IP Server to run properly.

Windows XP-platform

Currently all Beckhoff Embedded-PC/IPC devices running one of the following operating systems are supported: Windows XP, Windows XP Embedded, Windows Embedded Standard 2009, Windows Vista, Windows 7. Any difference between these two systems will be described in the corresponding [installation manual](#) [► 12]. Additionally, at least TwinCAT2 PLC needs to be installed on the TCP/IP Server and needs to be either in Config- or Run-Mode during operation.

Windows CE-platform

All Beckhoff Embedded-PC/IPC devices running one of the following operating systems are supported: Windows CE5, Windows CE6, Windows CE7.

5 Installation

This part of the documentation gives a step-by-step explanation of the TwinCAT TCP/IP Server setup process for Windows XP based operating systems. The following topics are part of this document:

- Downloading the setup file
- Starting the installation

Downloading the setup file

Like many other TwinCAT Supplement products, TCP/IP Server is available for download via the Beckhoff FTP-Server. The download represents the most current version, which can be licensed either as a 30-Day Demo or as a full version. To download the setup file, please perform the following steps:

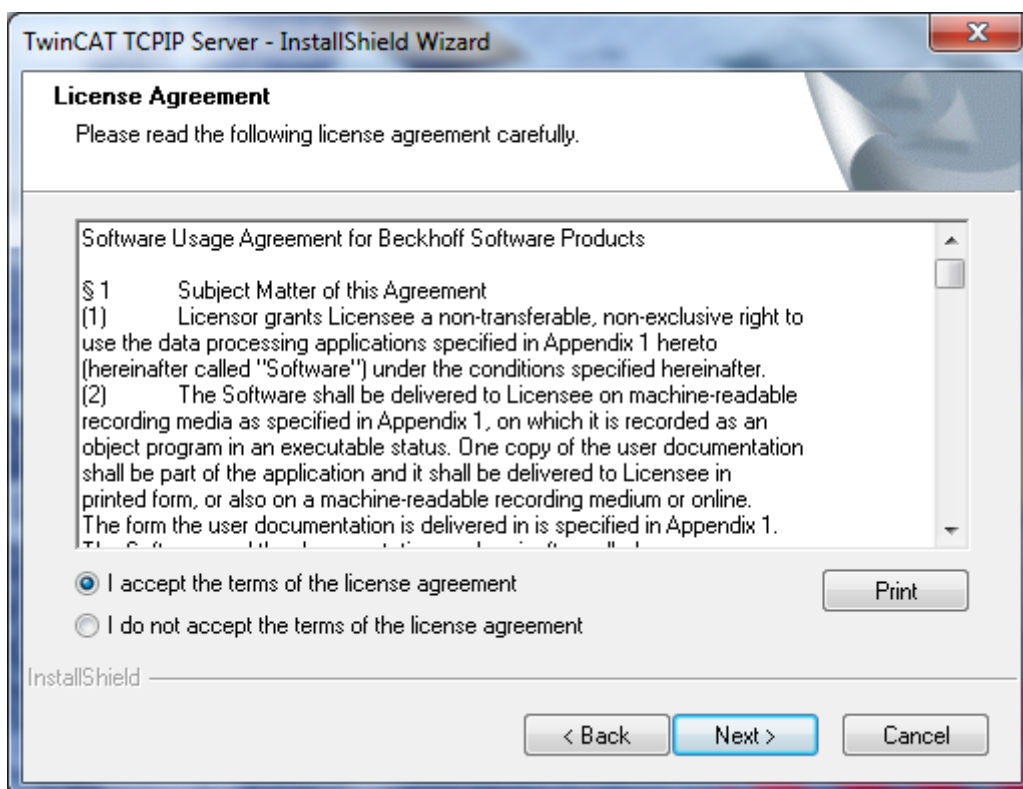
- Select the [TS6310 | TwinCAT TCP/IP Server](#) from the BECKHOFF website.
- (Optional) Transfer the downloaded file to the TwinCAT runtime system, where you would like to install the Supplement.

Starting the installation

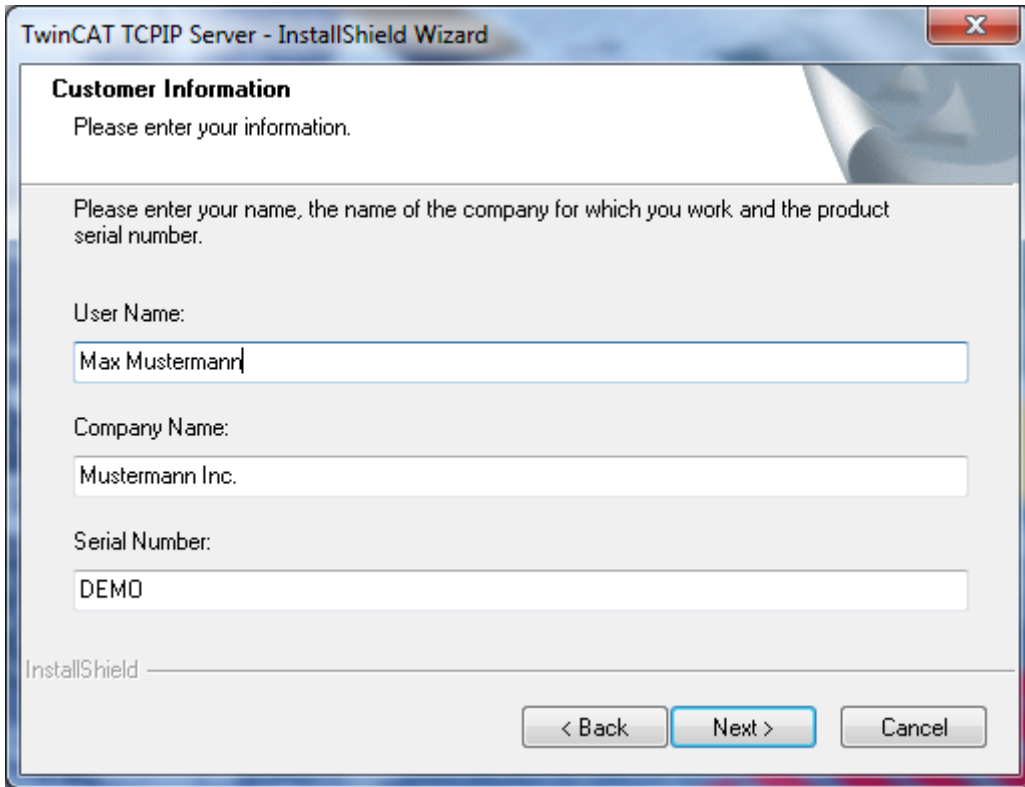
To install the supplement, please follow the steps below:

i Please start the installation under Windows 7 32-bit/64-bit using "Run as administrator" by clicking on the setup file with the right mouse button and selecting the appropriate option in the context menu.

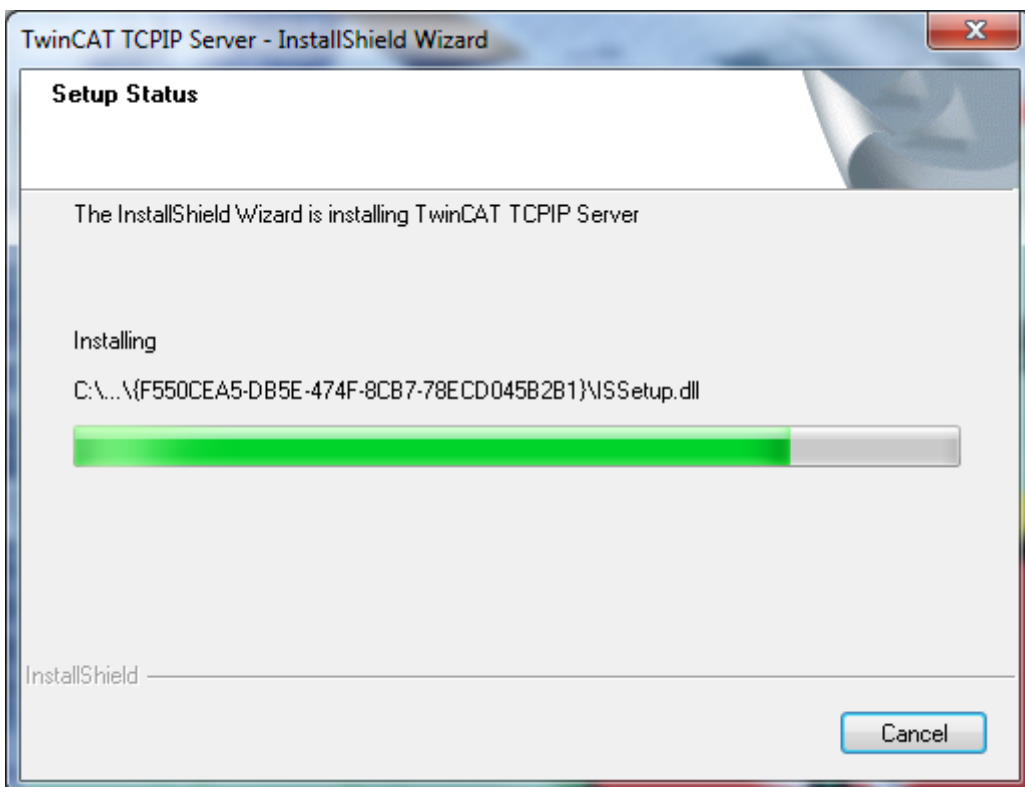
- Double-click the downloaded file "**TcplpServer.exe**".
- Select the **language** in which you wish to install the software.
- Click on "Next" and then accept the **license agreement**.



- Enter your **user data**. All visible fields are mandatory fields. If you want to install a 30-day demo version, please enter "DEMO" as license key.



- Click on **"Install"** to start the installation.



- To complete installation, **please restart your computer**.

6 Installation Windows CE

This part of the documentation describes, how you can install the supplement TwinCAT TCP/IP Server on a Beckhoff Embedded PC Controller based on Windows CE, for example CX1000, CX1020, CX9000, CX9001, CX9010, CP62xx, C69xx, ...

The setup process consists of four steps:

- Downloading the setup file
- Installation on a host computer
- Transferring the setup file to the Windows CE device
- Executing the setup on the Windows CE device

Downloading the setup file

Like many other TwinCAT Supplement products, TCP/IP Server for CE is available for download via the Beckhoff FTP-Server. The download represents the most current version. To download the setup file, please perform the following step:

- Select the TwinCAT TCP/IP Server CE from the [BECKHOFF website](#).

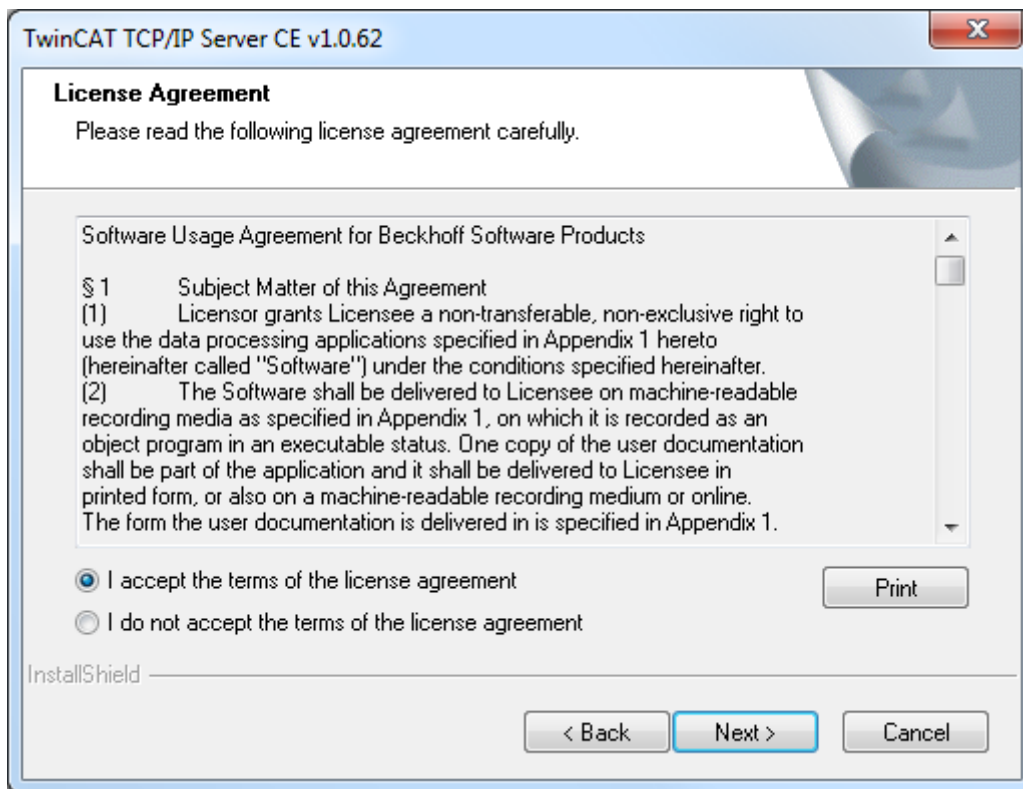
Installation on the host PC

To access the installation files for Windows CE, the downloaded setup file must first be installed on a host PC. This can be any Windows XP-based system. To do this, perform the following steps:

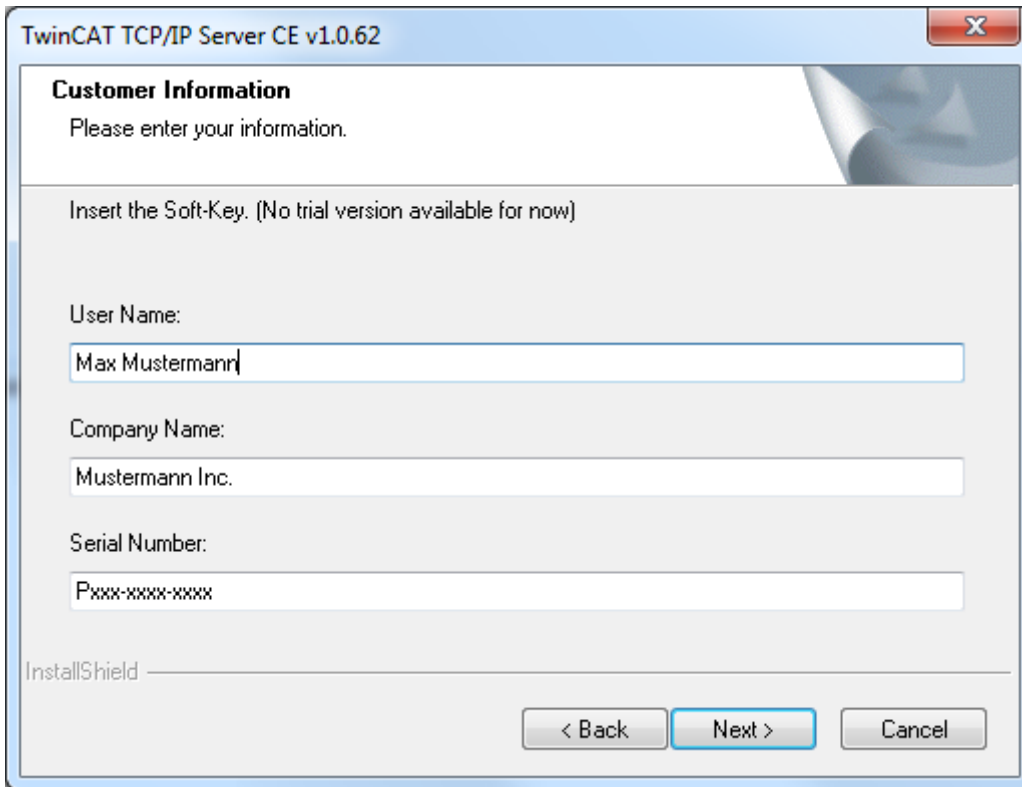


Please note: a 30-day demo version of TCP/IP Server for Windows CE is currently not available. So you need a valid product key for the installation.

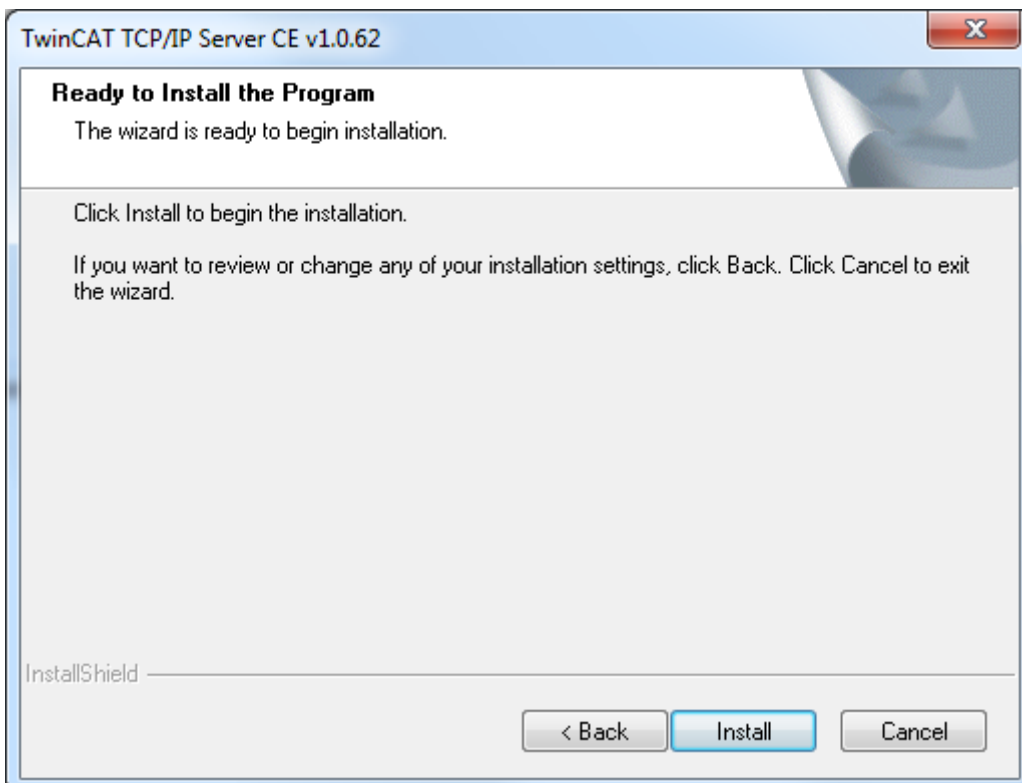
- Double-click the downloaded file "**TcTCPIPSvrCE.exe**".
- Select the **language** in which you wish to install the software.
- Click on "Next" and then accept the **license agreement**.



- Enter your **user data**. All visible fields are mandatory fields.



- Then click **"Install"** to start the installation.



After the installation, the Windows CE installation files are now located in the folder ".\TwinCAT\CE". This folder contains a variety of different CE installation files in the form of CAB files:

- **TCPIP\Install\TcTCPIPSvrCe.I586.cab**: TCP/IP server for x86 based CPUs (like CX10xx, CP62xx, C69xx, ...)
- **TCPIP\Install\TcTCPIPSvrCe.ARMV4I.cab**: TCP/IP server for ARM based CPUs (like CX9001, CX9010, CP6608, ...)

Transferring the setup file to the Windows CE device

Transfer the corresponding setup file to you CE device. This can be done in the following ways:

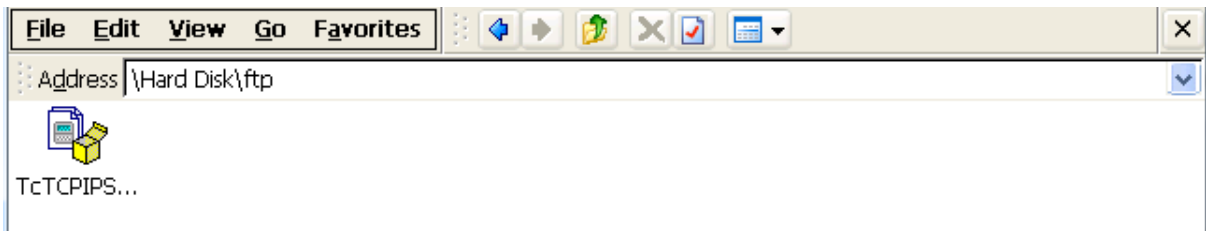
- via a Shared Folder
- via the integrated FTP-Server
- via ActiveSync
- via a CF card

For more information, please consult the "Windows CE" section in our Infosys documentation system.

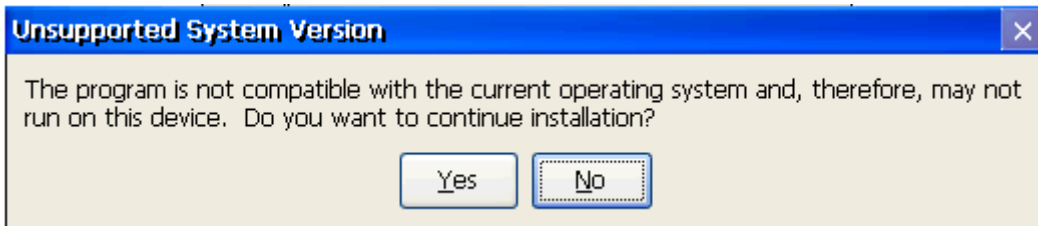
Running the installation on the Windows CE device

The installation file "TcTCPIPSvrCe.xxxx.CAB" transferred to the controller must now be installed. To do this, please perform the following steps on the CE device:

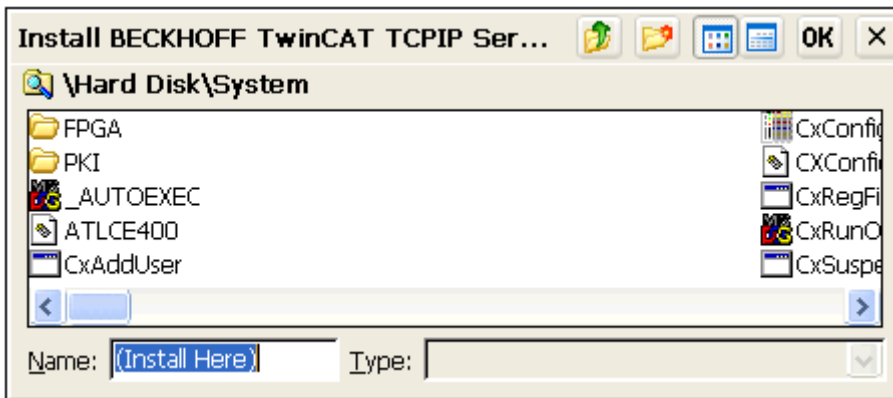
- Navigate to the folder that you transferred the installation file to,



- Double-click the CAB file. If you get a MessageBox saying "Program is not compatible with current operating system", check that you have used the correct CAB file (ARM, I586) for your platform.
- If you are sure that the CAB file is correct, confirm the message with "Yes".



- Confirm the destination directory "\Hard Disk\System" with "Ok".



- To start the installation, click in the upper right corner on "Ok"



After installation, the installation file deletes itself automatically.



The TCP/IP server will not be available until the next system reboot.

7 PLC libraries

7.1 Tcplp.lib

The **Tcplp.Lib** function blocks can be used to realise client or server applications in the TwinCAT PLC. These can exchange data with other communication devices either via **User Datagram Protocol (UDP)** or via **Transmission Control Protocol (TCP)**.

System requirements:

Development environment:

- NT4, W2K, XP, XPe;
- TwinCAT System version 2.9 or higher;
- TwinCAT installation level: TwinCAT PLC or higher;

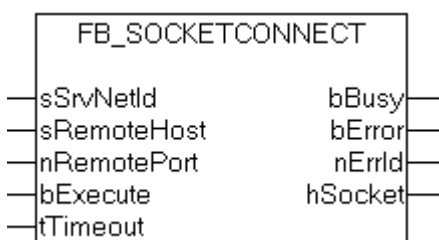
Target system:

- TwinCAT PLC runtime system version 2.8 or higher;
- PC or CX (x86)
 - TwinCAT TCP/IP Connection Server **v1.0.0.0** or higher.
 - NT4, W2K, XP, XPe, CE (image v1.75 or higher);
- CX (ARM)
 - TwinCAT TCP/IP Connection Server **v1.0.0.44** or higher.
 - CE (image v2.13 or higher);

Installation:

The PLC library is supplied with the TwinCAT TCP/IP Connection Server and copied into folder ... \TwinCAT\PLC\Lib during installation.

7.1.1 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** [▶ 24] and **FB_SocketReceive** [▶ 25], 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** [▶ 19]. 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    : STRING(15);
  nRemotePort    : UDINT;
  bExecute       : BOOL;
  tTimeout       : TIME := T#45s; (*!!!*)
END_VAR
```

sSrvNetId: string containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.

sRemoteHost: IP address (Ipv4) of the remote server as a string (e.g. '172.33.5.1'). An empty string can be entered on the local computer for a server.

nRemotePort: IP port number of the remote server (e.g. 200).

bExecute: the function block is enabled via a positive edge at this input.

tTimeout: maximum time that may not be exceeded when the function block is executed.

i The *tTimeout* 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;
END_VAR
```

bBusy: When the function block is activated this output is set. It remains set until an acknowledgement is received.

bError: If an error should occur during the transfer of the command, then this output is set once the bBusy output was reset.

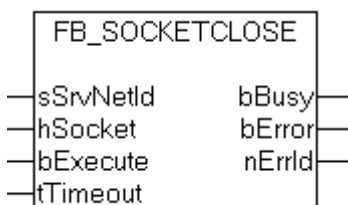
nErrId : If the bError output is set, this parameter returns the TwinCAT TCP/IP Connection Server error number [▶ 87].

hSocket: TCP/IP connection handle [▶ 33] for the newly opened local client socket.

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	Tcplp.Lib
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	(Standard.Lib; TcBase.Lib; TcSystem.Lib are included automatically)

7.1.2 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 [▶ 22], a local client socket with FB_SocketConnect [▶ 18] and a remote client socket with FB_SocketAccept [▶ 23].

UDP: The UDP socket is opened with the function block: FB_SocketUdpCreate [▶ 26].

VAR_INPUT

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

sSrvNetId: String containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.

hSocket: TCP/IP: [Connection handle \[▶ 33\]](#) of the listener, remote or local client socket to be closed. UDP: Connection handle of the UDP socket.

bExecute: The block is activated by a rising edge at this input.

tTimeout: Maximum time allowed for the execution of the function block.

VAR_OUTPUT

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

bBusy: When the function block is activated this output is set. It remains set until an acknowledgement is received.

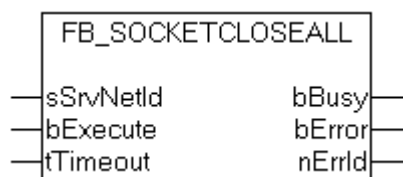
bError: If an error should occur during the transfer of the command, then this output is set once the bBusy output was reset.

nErrId : If the bError output is set, this parameter returns the [TwinCAT TCP/IP Connection Server error number \[▶ 87\]](#).

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	Tcplp.Lib
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	(Standard.Lib; TcBase.Lib; TcSystem.Lib are included automatically)

7.1.3 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
```

sSrvNetId: String containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.

bExecute: The block is activated by a rising edge at this input.

tTimeout: Maximum time allowed for the execution of the function block.

VAR_OUTPUT

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

bBusy: When the function block is activated this output is set. It remains set until and acknowledgement is received.

bError: If an error should occur during the transfer of the command, then this output is set once the bBusy output was reset.

nErrId : If the bError output is set, this parameter returns the TwinCAT TCP/IP Connection Server error number [▶ 87].

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.

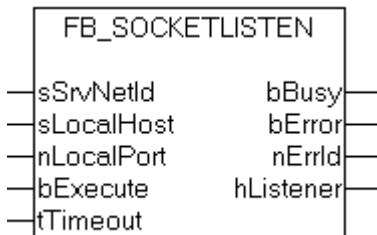
```
PROGRAM MAIN
VAR
  fbSocketCloseAll : FB_SocketCloseAll;
  bCloseAll        : BOOL := TRUE;
END_VAR

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_IFIFNOT fbSocketCloseAll.bBusy THEN(*...
... continue program execution...
...*)
END_IF
```

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	Tcplp.Lib
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	(Standard.Lib; TcBase.Lib; TcSystem.Lib are included automatically)

7.1.4 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 [▶ 23]. If a listener socket is no longer required, it can be closed with the function block FB_SocketClose [▶ 19]. The listener sockets on an individual computer must have unique IP port numbers.

VAR_INPUT

```
VAR_INPUT
  sSrvNetId      : T_AmsNetId := '';
  sLocalHost     : STRING(15);
  nLocalPort     : UDINT;
  bExecute       : BOOL;
  tTimeout       : TIME := T#5s;
END_VAR
```

sSrvNetId: String containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.

sLocalHost: Local server IP address (Ipv4) as a string (e.g. '172.13.15.2'). For a server on the local computer (default), an empty string may be entered.

nLocalPort: Local server IP port (e.g. 200).

bExecute: The block is activated by a rising edge at this input.

tTimeout: 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
```

bBusy: When the function block is activated this output is set. It remains set until an acknowledgement is received.

bError: If an error should occur during the transfer of the command, then this output is set once the bBusy output was reset.

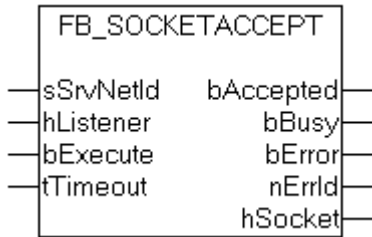
nErrId : If the bError output is set, this parameter returns the TwinCAT TCP/IP Connection Server error number [▶ 87].

hListener: Connection handle [▶ 33] for the new listener socket.

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	Tcplp.Lib
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	(Standard.Lib; TcBase.Lib; TcSystem.Lib are included automatically)

7.1.5 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` [► 24] and `FB_SocketReceive` [► 25] 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` [► 19].

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 block can be activated cyclically via a rising 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
```

sSrvNetId: String containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.

hListener: [Connection handle](#) [► 33] of the listener sockets. This handle must first be requested via the function block `FB_SocketListen` [► 22].

bExecute: The block is activated by a rising edge at this input.

tTimeout: 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
```

bAccepted: This output is set if a new connection to a remote client was established.

bBusy: When the function block is activated this output is set. It remains set until an acknowledgement is received.

bError: If an error should occur during the transfer of the command, then this output is set once the `bBusy` output was reset.

nErrId : If the bError output is set, this parameter returns the [TwinCAT TCP/IP Connection Server error number \[► 87\]](#).

hSocket: [Connection handle \[► 33\]](#) of a new remote client.

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	Tcplp.Lib
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	(Standard.Lib; TcBase.Lib; TcSystem.Lib are included automatically)

7.1.6 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 \[► 23\]](#), or a remote server connection via the function block [FB_SocketConnect \[► 18\]](#).

VAR_INPUT

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

sSrvNetId: string containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.

hSocket: [connection handle \[► 33\]](#) of the communication partner to which data are to be sent.

cbLen: number of data to be sent in bytes.

pSrc: address (pointer) of the transmit buffer.

bExecute: the function block is enabled via a positive edge at this input.

tTimeout: maximum time that may not be exceeded when the function block is executed.



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

VAR_OUTPUT

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


bBusy: When the function block is activated this output is set. It remains set until an acknowledgement is received.

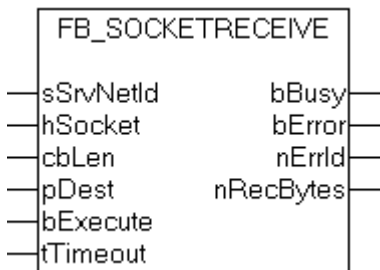
bError: If an error should occur during the transfer of the command, then this output is set once the bBusy output was reset.

nErrId : If the bError output is set, this parameter returns the TwinCAT TCP/IP Connection Server error number [▶ 87].

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	Tcplp.Lib
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	(Standard.Lib; TcBase.Lib; TcSystem.Lib are included automatically)

7.1.7 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 must first be established via the function block FB_SocketAccept [▶ 23], and a remote server connection via the function block FB_SocketConnect [▶ 18]. 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 must be called cyclically (polling) within the PLC task, until all required data have been received. During this process, a 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 == null.

In a simple protocol for receiving, for example, a null-terminated string on a remote server, the function block FB_SocketReceive, for example, must be called repeatedly until the null-termination is detected in the data received.

i 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 function block FB_SocketReceive returns no error and no data. The open socket still exists, but no data are received. The application may then wait endlessly for the remaining frame data bytes. It is recommended to implement timeout monitoring in the application. If not all frame data bytes were received after a certain period, e.g. 10 seconds, the connection must be closed and reinitialized.

VAR_INPUT

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

sSrvNetId: String containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.

hSocket: [Connection handle](#) [[▶ 33](#)] of the communication partner from which data are to be received.

cbLen: Maximum available buffer size in bytes for the data to be read.

pDest: Address (pointer) of the receive buffer.

bExecute: The block is activated by a rising edge at this input.

tTimeout: Maximum time allowed for the execution of the function block.

VAR_OUTPUT

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

bBusy: When the function block is activated this output is set. It remains set until an acknowledgement is received.

bError: If an error should occur during the transfer of the command, then this output is set once the bBusy output was reset.

nErrId : If the bError output is set, this parameter returns the [TwinCAT TCP/IP Connection Server error number](#) [[▶ 87](#)].

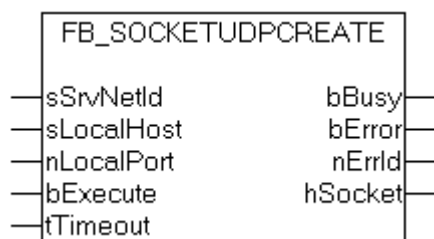
nRecBytes: Number of the last successfully received data bytes.

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	Tcplp.Lib
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	(Standard.Lib; TcBase.Lib; TcSystem.Lib are included automatically)

7.1.8 FB_SocketUdpCreate

From product version: TwinCAT TCP/IP Connection Server v1,0,0,31 and higher



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](#) [[▶ 27](#)] and [FB_SocketUdpReceiveFrom](#) [[▶ 29](#)], for example, in order to exchange data with a remote device. If a UDP socket is no longer required, it can be closed with the function block [FB_SocketClose](#) [[▶ 19](#)]. 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 input variable `sLocalHost` 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.



- 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'.
- 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.
- 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     : STRING(15);
  nLocalPort     : UDINT;
  bExecute      : BOOL;
  tTimeout       : TIME := T#5s;
END_VAR
```

sSrvNetId: String containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.

sLocalHost: The 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: The local IP port number of the UDP client/server socket (e.g. 200).

bExecute: The block is activated by a rising edge at this input.

tTimeout: 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
```

bBusy: When the function block is activated this output is set. It remains set until an acknowledgement is received.

bError: If an error should occur during the transfer of the command, then this output is set once the *bBusy* output was reset.

nErrId : If the *bError* output is set, this parameter returns the TwinCAT TCP/IP Connection Server error number [▶ 87].

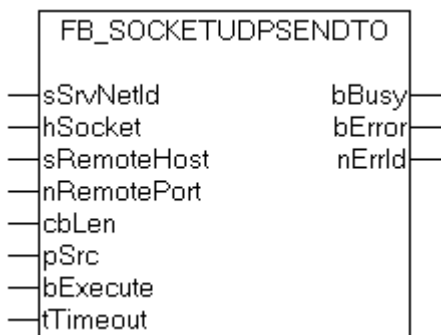
hSocket: The handle [▶ 33] of the newly opened UDP client/server socket.

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	Tcplp.Lib (v1.0.4 or higher)
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	(Standard.Lib; TcBase.Lib; TcSystem.Lib are included automatically)

7.1.9 FB_SocketUdpSendTo

Available in the product version: TwinCAT TCP/IP Connection Server v1,0,0,31 or higher



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](#) [► 26].

VAR_INPUT

```
VAR_INPUT
  sSrvNetId   : T_AmsNetId := '';
  hSocket     : T_HSOCKET;
  sRemoteHost : STRING(15);
  nRemotePort : UDINT;
  cbLen       : UDINT;
  pSrc        : DWORD;
  bExecute    : BOOL;
  tTimeout    : TIME := T#5s;
END_VAR
```

sSrvNetId: string containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.

hSocket: the [handle](#) [► 33] of an opened UDP socket.

sRemoteHost: IP address (IPv4) of the remote device to which data are to be sent as a string (e.g. '172.33.5.1'). An empty string can be entered for a device on the local computer.

nRemotePort: IP port number of the remote device to which data are to be sent (e.g. 200).

cbLen: number of data to be sent in bytes. The maximum number of data bytes to be sent is limited to 8192 bytes by default (by declaring the TCPADS_MAXUDP_BUFFSIZE constant in the library to conserve memory resources).

pSrc: address (pointer) of the transmit buffer.

bExecute: the function block is enabled via a positive edge at this input.

tTimeout: maximum time that may not be exceeded when the function block is executed.



From product version: TwinCAT TCP/IP Connection Server **v1.0.50** and higher, the maximum number of data bytes to be sent can be increased (if absolutely necessary).

1) In the PLC project, redefine the global constant (in our sample we want to increase the maximum number of data bytes to 32000 bytes):

```
VAR_GLOBAL CONSTANT
  TCPADS_MAXUDP_BUFFSIZE : UDINT :=32000;
END_VAR
```

2) Then activate the option "Replace constants" in TwinCAT PLC Control->"Project->Options...->Build" dialog box.

3) Compile the project.

VAR_OUTPUT

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

bBusy: When the function block is activated this output is set. It remains set until an acknowledgement is received.

bError: If an error should occur during the transfer of the command, then this output is set once the bBusy output was reset.

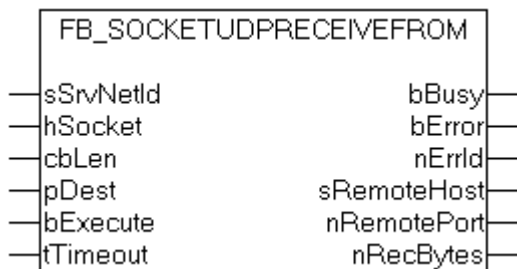
nErrId : If the bError output is set, this parameter returns the TwinCAT TCP/IP Connection Server error number [[▶ 87](#)].

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	Tcplp.Lib (v1.0.4 or higher)
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	(Standard.Lib; TcBase.Lib; TcSystem.Lib are included automatically)

7.1.10 FB_SocketUdpReceiveFrom

Available in the product version: TwinCAT TCP/IP Connection Server v1,0,0,31 or higher



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](#) [[▶ 26](#)]. The instance of the FB_SocketUdpReceive function block must be called cyclically (polling) within the PLC task. During this process, a 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     : DWORD;
  bExecute  : BOOL;
  tTimeout  : TIME := T#5s;
END_VAR
```

sSrvNetId: string containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.

hSocket: the [handle](#) [[▶ 33](#)] of an opened UDP socket whose data are to be received.

cbLen: the 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 by default (by declaring the TCPADS_MAXUDP_BUFFSIZE constant in the library to conserve memory resources).

pDest: the address (pointer) of the receive buffer.

bExecute: the function block is enabled via a positive edge at this input.

tTimeout: maximum time that may not be exceeded when the function block is executed.



From product version: TwinCAT TCP/IP Connection Server **v1.0.50** and higher, the maximum number of data bytes to be sent can be increased (if absolutely necessary).

1) In the PLC project, redefine the global constant (in our sample we want to increase the maximum number of data bytes to 32000 bytes):

```
VAR_GLOBAL CONSTANT
    TCPADS_MAXUDP_BUFFSIZE : UDINT :=32000;
END_VAR
```

2) Then activate the option *"Replace constants"* in TwinCAT PLC Control->"Project->Options...->Build" dialog box.

3) Compile the project.

VAR_OUTPUT

```
VAR_OUTPUT
    bBusy      : BOOL;
    bError     : BOOL;
    nErrId     : UDINT;
    sRemoteHost : STRING(15);
    nRemotePort : UDINT;
    nRecBytes  : UDINT;
END_VAR
```

bBusy: When the function block is activated this output is set. It remains set until and acknowledgement is received.

bError: If an error should occur during the transfer of the command, then this output is set once the bBusy output was reset.

nErrId : If the bError output is set, this parameter returns the TwinCAT TCP/IP Connection Server error number [▶ 87].

sRemoteHost: If successful, IP address (Ipv4) of the remote device whose data were received.

nRemotePort: If successful, IP port number of the remote device whose data were received (e.g. 200).

nRecBytes: Number of the last successfully receive data bytes.

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	Tcplp.Lib (v1.0.4 or higher)
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	(Standard.Lib; TcBase.Lib; TcSystem.Lib are included automatically)

7.1.11 FB_SocketUdpAddMulticastAddress

Available since product version: TwinCAT TCP/IP Connection Server **1.0.64** or higher



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](#) [▶ 26].

VAR_INPUT

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

sSrvNetId: String containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.

hSocket: [Connection handle](#) [▶ 33] of the listener sockets. This handle must first be requested via the function block [FB_SocketUdpCreate](#) [▶ 26].

sMulticastAddr: Multicast address to bind to.

bExecute: The block is activated by a rising edge at this input.

tTimeout: Maximum time allowed for the execution of the function block.

VAR_OUTPUT

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

bBusy: When the function block is activated this output is set. It remains set until an acknowledgement is received.

bError: If an error should occur during the transfer of the command, then this output is set once the bBusy output was reset.

nErrId : If the bError output is set, this parameter returns the [TwinCAT TCP/IP Connection Server error number](#) [▶ 87].

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	Tcplp.lib
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	

7.1.12 FB_SocketUdpDropMulticastAddress

Available since product version: TwinCAT TCP/IP Connection Server 1.0.64 or higher



Removes the binding to a Multicast IP address which has previously been added via the function block [FB_SocketUdpAddMulticastAddress](#) [[▶ 30](#)].

VAR_INPUT

```

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

```

sSrvNetId: String containing the network address of the TwinCAT TCP/IP Connection Server. For the local computer (default) an empty string may be specified.

hSocket: [Connection handle](#) [[▶ 33](#)] of the listener sockets. This handle must first be requested via the function block [FB_SocketUdpCreate](#) [[▶ 26](#)].

sMulticastAddr: Multicast address for which the binding should be removed.

bExecute: The block is activated by a rising edge at this input.

tTimeout: Maximum time allowed for the execution of the function block.

VAR_OUTPUT

```

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

```

bBusy: When the function block is activated this output is set. It remains set until an acknowledgement is received.

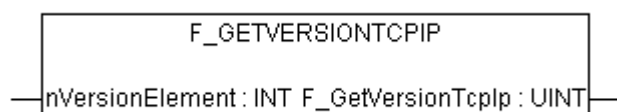
bError: If an error should occur during the transfer of the command, then this output is set once the bBusy output was reset.

nErrId : If the bError output is set, this parameter returns the [TwinCAT TCP/IP Connection Server error number](#) [[▶ 87](#)].

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	Tcplp.lib
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	

7.1.13 F_GetVersionTcplp



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

FUNCTION F_GetVersionTcplp : 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 be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	Tcplp.Lib
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	(Standard.Lib; TcBase.Lib; TcSystem.Lib are included automatically)

7.1.14 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
```

nPort: Internet Protocol (IP) port;

sAddr: Internet protocol address (Ipv4) separated by dots as a string, e.g. "172.34.12.3";

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	Tcplp.Lib
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	(Standard.Lib; TcBase.Lib; TcSystem.Lib are included automatically)

7.1.15 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
```

handle: Internal TwinCAT TCP/IP Connection Server socket handle;

localAddr: Local socket address;

remoteAddr: Remote socket address;

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 \[► 22\]](#) 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 \[► 23\]](#). [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 \[► 24\]](#) and/or [FB_SocketReceive \[► 25\]](#), 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 \[► 19\]](#), 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 \[► 18\]](#) returns the connection handle of a local client socket.
- The connection handle is then transferred to the function blocks [FB_SocketSend \[► 24\]](#) and [FB_SocketReceive \[► 25\]](#), 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 \[► 19\]](#), in order to close a connection that is no longer required.

The function block [FB_SocketCloseAll \[► 20\]](#) 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 be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	Tcplp.Lib
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	(Standard.Lib; TcBase.Lib; TcSystem.Lib are included automatically)

Also see about this

📖 [ST_SockAddr \[► 33\]](#)

7.1.16 E_WinsockError

```

TYPE E_WinsockError :
(
  WSOK,
  WSAEINTR           :=      10004   ,(* A blocking operation was interrupted by a call to
WSACancelBlockingCall. *)
  WSAEBAADF          :=      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 network 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. *)
  WSAESOCKTNSUPPORT  :=      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      ,(* WSASStartup 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 WSASStartup, or WSASStartup 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. *)
WSASYSFAILFAILURE := 10107 , (* A system call that should never fail has failed. *)
WSASERVICE_NOT_FOU ND := 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 canceled. *)
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 data base, but it does not have the correct associated data being resolved for. *)
);
END_TYPE
```

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	Tcplp.Lib
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	(Standard.Lib; TcBase.Lib; TcSystem.Lib are included automatically)

7.1.17 Global Variables

VAR_GLOBAL CONSTANT

```
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; (* TCP/IP Connection Server v1,0,0,31 and higher *)
  TCPADS_IGR_SENDTOBYHDL :UDINT :=16#80000007; (* TCP/IP Connection Server v1,0,0,31 and higher *)

  TCPADS_CONLST_IOF_CONNECT :UDINT :=1;
  TCPADS_CONLST_IOF_LISTEN :UDINT :=2;
  TCPADS_CONLST_IOF_CLOSEALL :UDINT :=3;
  TCPADS_CONLST_IOF_ACCEPT :UDINT :=4;
  TCPADS_CONLST_IOF_UDPBIND :UDINT :=5; (* TCP/IP Connection Server v1,0,0,31 and higher *)

  TCPADS_MAXUDP_BUFFSIZE : UDINT :=1472; (* TCP/IP Connection Server v1,0,0,31 and higher *)
END_VAR
```

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	Tcplp.Lib
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	(Standard.Lib; TcBase.Lib; TcSystem.Lib are included automatically)

7.2 TcSocketHelper.lib

The TcSocketHelper.lib contains usefull TCP/IP helper functions and function blocks.

System requirements:

Development environment:

- TwinCAT System version 2.9 or higher (NT4, W2K, XP, XPe);
- Installation level: TwinCAT PLC or higher.

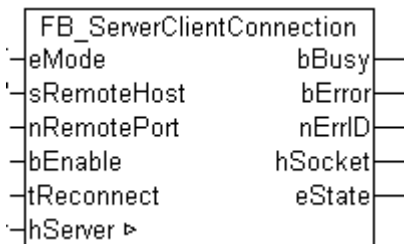
Target system:

- TwinCAT PLC runtime system version 2.9 or higher.
- PC or CX (x86)
 - TwinCAT TCP/IP Connection Server **v1.0.0.41** or higher.
 - NT4, W2K, XP, XPe, CE (image v1.75 or higher);
- CX (ARM)
 - TwinCAT TCP/IP Connection Server **v1.0.0.44** or higher.
 - CE (image v2.13 or higher);

Installation:

The PLC library is supplied with the TwinCAT TCP/IP Connection Server and copied into folder ... \TwinCAT\PLC\Lib during installation.

7.2.1 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 \[▶ 22\]](#), [FB_SocketAccept \[▶ 23\]](#) and [FB_SocketClose \[▶ 19\]](#) 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 \[▶ 24\]](#) function block and/or an instance of the [FB_SocketReceive \[▶ 25\]](#) 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
```

hServer: [server handle \[▶ 43\]](#). This input variable must be initialized via the [F_CreateServerHnd \[▶ 42\]](#) function.

VAR_INPUT

```
VAR_INPUT
    eMode      : E_SocketAcceptMode := eACCEPT_ALL;
    sRemoteHost : STRING(15) := '';
    nRemotePort : UDINT := 0;
```

```

    bEnable      : BOOL;
    tReconnect   : TIME := T#1s;
END_VAR

```

mMode: Determines whether all or only certain connections should be [accepted](#) [► 44].

sRemoteHost: IP address (Ipv4) of the remote client whose connection is to be accepted as a string (e.g.: '172.33.5.1'). For a client on the local computer an empty string may be specified.

nRemotePort: IP port number of the remote client whose connection is to be accepted (e.g. 200).

bEnable: As long as this input is TRUE, the system attempts to establish a connection at regular intervals until a connection was established successfully. Once established, a connection can be closed again with FALSE.

tReconnect: 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

```

bBusy: This output is TRUE as long as the function block is active.

bError: Becomes TRUE as soon as an error has occurred.

nErrId : If the bError output is set, this parameter returns the [TwinCAT TCP/IP Connection Server error number](#) [► 87].

hSocket: Connection handle for the newly opened remote client socket. If successful, this variable is transferred to the instances of the function blocks [FB_SocketSend](#) [► 24] and/or [FB_SocketReceive](#) [► 25].

eState: Returns the current [connection status](#) [► 44].

Example in FBD:

The following example illustrates initialisation 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`), although the connection has not yet been established (passive open).

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

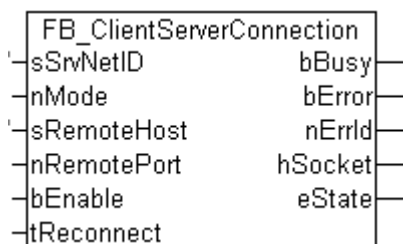
The third connection was activated (`bConnect3=TRUE`), and a connection to the remote client has been established.

Further application examples (including source code) can be found here: [Examples \[► 80\]](#)

Requirements

Development Environment	Target System	PLC libraries to include
TwinCAT v2.9.0 Build >= 1030	PC or CX (x86)	TcSocketHelper.Lib
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	(Standard.Lib; TcBase.Lib; TcSystem.Lib; TcUtilities.Lib; Tcplp.Lib are included automatically)

7.2.2 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` [► 18] and `FB_SocketClose` [► 19] 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` [► 24] function block and/or an instance of the `FB_SocketReceive` [► 25] 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    : STRING(15) := '';
  nRemotePort    : UDINT;
  bEnable        : BOOL;
  tReconnect     : TIME := T#45s;
END_VAR
```

sSrvNetID: 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: parameter flags (modes). The permissible parameters are listed in the table and can be combined via an OR operation:

Flag	Description
CONNECT_MODE_ENABLEDBG	Activates logging of debug messages in the application log. In order to view the debugging messages open the TwinCAT System Manager and activate log view.

sRemoteHost: IP address (Ipv4) of the remote server as a string (e.g. '172.33.5.1'). An empty string can be entered on the local computer for a server.

nRemotePort: IP port number of the remote server (e.g. 200).

bEnable: as long as this input is TRUE, the system attempts to establish a connection at regular intervals until a connection was established successfully. Once established, a connection can be closed again with FALSE.

tReconnect: cycle time with which the function block attempts to establish the connection. After this time at the latest, the attempt is aborted and a new one is started.



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
```

bBusy: This output is TRUE as long as the function block is active.

bError: Becomes TRUE as soon as an error has occurred.

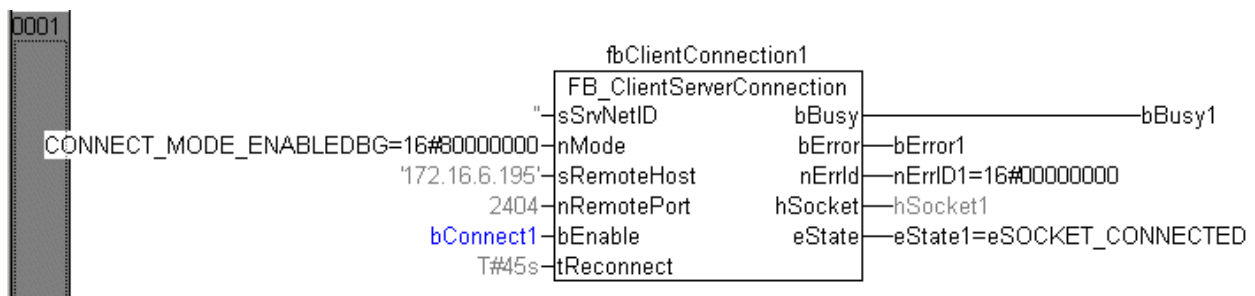
nErrID: If the bError output is set, this parameter returns the [TwinCAT TCP/IP Connection Server error number \[▶ 87\]](#).

hSocket: Connection handle for the newly opened local client socket. If successful, this variable is transferred to the instances of the function blocks [FB SocketSend \[▶ 24\]](#) and/or [FB SocketReceive \[▶ 25\]](#).

eState: Returns the current [connection status \[▶ 44\]](#).

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
```

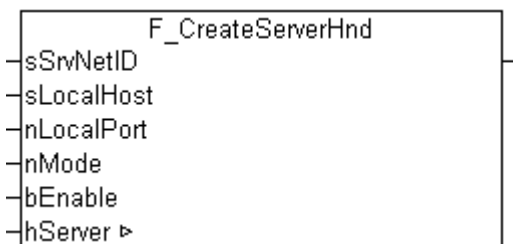


Further application examples (including source code) can be found here: [Examples \[▶ 80\]](#)

Requirements

Development Environment	Target System	PLC libraries to include
TwinCAT v2.9.0 Build >= 1030	PC or CX (x86)	TcSocketHelper.Lib
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	(Standard.Lib; TcBase.Lib; TcSystem.Lib; TcUtilities.Lib; Tcplp.Lib are included automatically)

7.2.3 F_CreateServerHnd



The function F_CreateServerHnd is used to initialize/set the internal parameters of a server handle variable *hServer*. The server handle is then transferred to the instances of the [FB_ServerClientConnection \[▶ 37\]](#) 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
    
```

hServer: [Server handle \[▶ 43\]](#) variable whose internal parameters are to be initialised.

sSrvNetID: 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: Local server IP address (Ipv4) as a string (e.g., '172.13.15.2'). For a server on the local computer (default), an empty string may be entered.

nLocalPort: Local server IP port (e.g., 200).

nMode: parameter flags (modes). The permissible parameters are listed in the table and can be combined via an OR operation:

Flag	Description
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: This input determines the behavior of the listener socket. Once opened, a listener socket remains open until this input becomes 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	Description
TRUE	No error
FALSE	Error, invalid parameter value

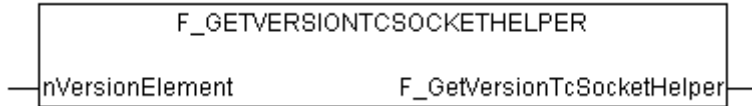
Example:

See in the [FB_ServerClientConnection \[▶ 37\]](#) function block description.

Requirements

Development Environment	Target System	PLC libraries to include
TwinCAT v2.9.0 Build >= 1030	PC or CX (x86)	TcSocketHelper.Lib
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	(Standard.Lib; TcBase.Lib; TcSystem.Lib; TcUtilities.Lib; Tcplp.Lib are included automatically)

7.2.4 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	PLC libraries to include
TwinCAT v2.9.0 Build >= 1030	PC or CX (x86)	TcSocketHelper.Lib
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	(Standard.Lib; TcBase.Lib; TcSystem.Lib; TcUtilities.Lib; Tcplp.Lib are included automatically)

7.2.5 T_HSERVER

A variable of this type represents a TCP/IP server handle. The handle must be initialized before use with the function [F_CreateServerHnd](#) [► 42]. This sets the internal parameters of the T_HSERVER variables.



The structural elements should not be written to or modified directly.

Requirements

Development environment	Target platform	PLC libraries to include
TwinCAT v2.9.0 Build >= 1030	PC or CX (x86)	TcSocketHelper.Lib
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	(Standard.Lib; TcBase.Lib; TcSystem.Lib; TcUtilities.Lib; Tcplp.Lib are included automatically)

7.2.6 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	PLC libraries to include
TwinCAT v2.9.0 Build >= 1030	PC or CX (x86)	TcSocketHelper.Lib
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	(Standard.Lib; TcBase.Lib; TcSystem.Lib; TcUtilities.Lib; Tcplp.Lib are included automatically)

7.2.7 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	PLC libraries to include
TwinCAT v2.9.0 Build >= 1030	PC or CX (x86)	TcSocketHelper.Lib
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	(Standard.Lib; TcBase.Lib; TcSystem.Lib; TcUtilities.Lib; Tcplp.Lib are included automatically)

7.2.8 Global constants

Constant	Value	Description
LISTEN_MODE_CLOSEALL	16#00000001	FORCED close of all previous opened sockets
LISTEN_MODE_USEOPENED	16#00000002	Try to use already opened listener socket
CONNECT_MODE_ENABLEDBG	16#80000000	Enables/Disables debugging messages

Requirements

Development Environment	Target System	PLC libraries to include
TwinCAT v2.9.0 Build >= 1030	PC or CX (x86)	TcSocketHelper.Lib
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	(Standard.Lib; TcBase.Lib; TcSystem.Lib; TcUtilities.Lib; Tcplp.Lib are included automatically)

7.3 TcSnmp.lib

The **TcSnmp.Lib** function blocks can be used to send SNMP Traps or receive SNMP GET using the **TwinCAT TCP/IP Connection Server**. A TwinCAT TCP/IP Connection Server must be running.

Product components

- TcSnmp.Lib

Installation

Windows NT (NT4, W2K, XP, XPe, 7)

Please do the following steps:

- Copy the library into ...TwinCAT\Lib folder.
- Install the TwinCAT TCP/IP Connection Server. The Server will start automatically after starting TwinCAT

7.3.1 FB_SEND_TRAP

The function block allows to send SNMPv1 Traps. You can add variable bindings for sending additional information to the manager along with the trap. The following SNMP Data Types are supported by the function block for the variable bindings: OCTET_STRING, INTEGER32, COUNTER32, GAUGE32, TIMETICKS, OBJECT_ID

The uptime field is always zero.

The maximum packet size is limited to 2000 byte. If more bytes are committed the function block will return a error.

The function block FB_SEND_TRAP should be called in each cycle for a proper operation.

```

VAR_INPUT
  bEnable          :BOOL;
  bExecute         :BOOL;
  sLocalHostIp    :STRING(15);
  sLocalHostPort  :UDINT;
  sManagerIP      :STRING(15);
  sTcIpConnSvrAddr :T_AmsNetId;
  sObjectID       :T_MAXSTRING;
  sCommunity      :STRING(60);
  iGenericTrapNumber :INT;
  bySpecificTrapNumber :BYTE;
  nVarBindings    :USINT;
  pArrVarBinding  :POINTER TO ARRAY[1..iMAX_TRAPBUF_SIZE] OF ST_SNMP_VariableBinding;
END_VAR

```

bEnable: With a rising edge on this input the system attempts to create a Socket. Once established the Output bEnabled is set to TRUE. The Socket can be closed again with a falling edge.

bExecute: Send one Trap with a rising edge on bExecute. A rising edge will clear the output nErrID and bError. A opened Socket is required.

sLocalHostIP: String containing the (IPv4) dotted network address of the local host (e.g. '172.33.5.1'). If there is more than one network adapter present on the machine, the sLocalHostIP parameter allows you to specify which adapter to use.

sLocalHostPort(optional): The local IP port number.

sManagerIP: IP address (IPv4) of the SNMP Manager.

sTcIpConnSvrAddr: not supported

sObjectID: String containing the dotted numerical value that represents the MIB (Management Information Base) of the device. Maximum length of the string is 255. Valid values for each number between the dots is 0...65535. (e.g. '1.3.1.3.2555.3')

sCommunity: String containing the SNMP Community String (e.g. public)

iGenericTrapNumber: The SNMP Generic Trap Number defined in E_SNMP_GenericTrapNumber.

bySpecificTrapNumber: The SNMP Specific Trap Number. Automatically set to 0 if iGenericTrapNumber is not 0x06(E_SNMP_EnterpriseSpecific). Valid values are 1...255.

pArrVarBinding (optional): Pointer to an array of SNMP_ST_VariableBinding.

nVarBindings (optional): Number of the elements in pArrVarBinding. Maximum is iMAX_TRAPBUF_SIZE (defined in Global_Variables)

```
VAR_OUTPUT
  bBusy          :BOOL;
  bEnabled       :BOOL;
  bError         :BOOL;
  nErrID        :DINT;
END_VAR
```

bBusy: When the function block is activated this output is set while sending.

bEnabled: This output is set if a Socket is open.

bError: Becomes TRUE as soon as an error has occurred

nErrID: If the bError output is set, this parameter returns a [TwinCAT TCP/IP Connection Server error](#) [▶ 87] or E_SNMP_ErrorCodes.

The function block is tested with the following software:

SNMP Trap Watcher (BTT Software)

Wireshark 1.2.5

iReasoning MIB Browser Personal Edition 7.0

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	Tcplp.Lib (Standard.Lib; TcBase.Lib; TcSystem.Lib are included automatically)

7.3.2 FB_GetSnmP

The Function Block allows to receive SNMPv1 GET-commands. You can add variable bindings for sending additional information to the SNMP. The following SNMP Data Types are supported by the function block for the variable bindings: OCTET_STRING, INTEGER32, COUNTER32, GAUGE32, TIMETICKS, OBJECT_ID

```
VAR_INPUT
  bEnable          :BOOL;
  bReceive         :BOOL;
  bSendTrap       :BOOL;
  bSendResponse   :BOOL;
  sLocalHostIp    :STRING(15);
  nLocalHostPort  :UDINT;
  sManagerIP      :STRING(15);
  sTcIpConnSvrAddr :T_AmsNetId := '';
  sObjectID       :T_MAXSTRING;
  sCommunity      :STRING(60);
  iGenericTrapNumber :INT;
  iSpecificTrapNumber :BYTE;
  nVarBindings    :USINT := 0;
  pArrVarBinding  :POINTER TO ARRAY[1..iMAX_TRAPBUF_SIZE] OF ST_SNMP_VariableBinding;
  iError          :INT;
END_VAR
```

bEnable: With a rising edge on this input the system attempts to create a Socket. Once established the Output bEnabled is set to TRUE. The Socket can be closed again with a falling edge.

bReceive: Signal a received GET command.

bSendTrap: Signal a sended TRAP.

bSendResponse: Signal an answer is ready to send.

sLocalHostIP: String containing the (IPv4) dotted network address of the local host (e.g. '172.33.5.1'). If there is more than one network adapter present on the machine, the sLocalHostIP parameter allows you to specify which adapter to use.

sLocalHostPort(optional): The local IP port number.

sManagerIP: IP address (IPv4) of the SNMP Manager.

sTclpConnSvrAddr: not supported

sObjectID: String containing the dotted numerical value that represents the MIB (Management Information Base) of the device. Maximum length of the string is 255. Valid values for each number between the dots are 0...65535. (e.g., '1.3.1.3.2555.3')

sCommunity: String containing the SNMP Community String (e.g. public)

iGenericTrapNumber: The SNMP Generic Trap Number defined in E_SNMP_GenericTrapNumber.

iSpecificTrapNumber: The SNMP Specific Trap Number. Automatically set to 0 if iGenericTrapNumber is not 0x06(E_SNMP_EnterpriseSpecific). Valid values are 1...255.

pArrVarBinding (optional): Pointer to an array of SNMP_ST_VariableBinding.

nVarBindings (optional): Number of the elements in pArrVarBinding. Maximum is iMAX_TRAPBUF_SIZE (defined in Global_Variables)

iError: Returns specific SNMP error code.

```
VAR_OUTPUT
  bBusy           :BOOL;
  bError          :BOOL;
  bEnabled        :BOOL;
  bReceived       :BoOL;
  iRecSNMPVersion :INT; (* SNMP Version, sample SNMPv1 = 0 *)
  sRecCommunity   :STRING(60); (*Community Name *)
  iRecPDUType     :INT;(* SNMP PDU Type, sample GET = A0 *)
  arrRecPDURequestID :ARRAY[0..3] OF BYTE; (* SNMP PDU Request ID *)
  iRecPDUError    :INT; (*SNMP PDU Error *)
  iRecPDUErrorIndex :INT; (* SNMP PDU Error Index*)
  sRecObjectID    :T_MAXSTRING; (* Object Identifier *)
  arrRecObjectID  :ARRAY[0..128] OF UDINT;
  nErrID          :UDINT;
END_VAR
```

bBusy: When the function block is activated this output is set while sending.

bError: Becomes TRUE as soon as an error has occurred

bEnabled: This output is set if a Socket is open.

bReceived: This output is set if a get command has occurred..

iRecSNMPVersion: Info about SNMP version. SNMPv1 = 0.

sRecCommunity: Info about the received community strings.

iRecPDUType: Info about the PDU type.

arrRecPDURequestID: Array with PDU request IDs.

iRecPDUError: Output of the PDU errorID.

iRecPDUErrorIndex: Output of error index.

sRecObjectID: String with the objectID.

arrRecObjectID: Array with the received objectIDs .

nErrID: If the bError output is set, this parameter returns a [TwinCAT TCP/IP Connection Server error \[► 87\]](#) or E_SNMP_ErrorCodes.

The function block is tested with:

SNMP Trap Watcher (BTT Software)

Wireshark 1.2.5

iReasoning MIB Browser Personal Edition 7.0

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	Tcplp.Lib (Standard.Lib; TcBase.Lib; TcSystem.Lib are included automatically)

7.3.3 F_GetVersionTcSNMP

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

FUNCTION F_GetVersionTcSNMP : 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 be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	Tcplp.Lib (Standard.Lib; TcBase.Lib; TcSystem.Lib are included automatically)

7.3.4 SNMP_ST_VariableBinding

Structure for SNMP variable bindings

```
TYPE ST_SNMP_VariableBinding :
STRUCT
    iType :INT;
    iLength :INT;
    pArrValue :POINTER TO ARRAY[1..10000] OF BYTE;
    sOID :STRING(300) ;
END_STRUCT
END_TYPE
```

iType: The SNMP Data Type defined in E_SNMP_DataTypes.

iLength: The number of elements of pArrValue.

pArrValue: A pointer to the array filled with the values.

sOID: String containing the dotted numerical value of the variable binding.

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	Tcplp.Lib (Standard.Lib; TcBase.Lib; TcSystem.Lib are included automatically)

7.3.5 E_SNMP_GenericTrapNumber

```

TYPE E_SNMP_GenericTrapNumber :
(
  E_SNMP_ColdStart:= 16#00,
  E_SNMP_WarmStart:= 16#01,
  E_SNMP_LinkDown:=16#02,
  E_SNMP_LinkUp:= 16#03,
  E_SNMP_AuthenticationFailure:= 16#04,
  E_SNMP_EgpNeighborLoss:= 16#05,
  E_SNMP_EnterpriseSpecific:= 16#06
);
END_TYPE
    
```

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	TcSnmp.lib, Tcplp.Lib (Standard.Lib; TcBase.Lib; TcSystem.Lib are included automatically)

7.3.6 E_SNMP_DataTypes

```

TYPE E_SNMP_DataTypes :
(
  E_SNMP_INTEGER:= 16#02,
  E_SNMP_OCTETSTRING:= 16#04,
  E_SNMP_OBJECTID:=16#06,
  E_SNMP_SEQUENCE := 16#30,
  E_SNMP_IPADDRESS:= 16#40,
  E_SNMP_COUNTER32:= 16#41,
  E_SNMP_GAUGE32:= 16#42,
  E_SNMP_TIMETICKS:= 16#43,
  E_SNMP_TRAPTYPE:= 16#A4
);
END_TYPE
    
```

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	Tcplp.Lib (Standard.Lib; TcBase.Lib; TcSystem.Lib werden automatisch hinzugefügt)

7.3.7 Global Variables

VAR_GLOBAL CONSTANT

```

VAR_GLOBAL CONSTANT
  iMAX_TRAPBUF_SIZE :USINT := 255;
END_VAR
    
```

iMAX_TRAPBUF_SIZE: Maximum Number of elements in pArrVarBinding (POINTER TO ARRAY[1..iMAX_TRAPBUF_SIZE] OF ST_SNMP_VariableBinding;)

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	Tcplp.Lib (Standard.Lib; TcBase.Lib; TcSystem.Lib wird automatisch hinzugefügt)

8 Samples

8.1 Tcplp.lib

8.1.1 TCP example

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

The client should be implemented as a function block, of which several instances can be created. The server should be able to communicate with several clients.

Several instances of the server may be created. Each server instance is then addressed via a different port number. The server implementation is more difficult if the server has to communicate with more than one client. An implementation of a suitable client in .NET is also presented. The example can be used as a basis for realising more complex implementations.

System requirements

- TwinCAT v2.8 or higher. Level: TwinCAT PLC as a minimum.
- Installed TwinCAT TCP/IP Connection Server. If two PCs are used for the test, the TwinCAT TCP/IP Connection Server should be installed on both PCs.

Project sources

- <https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383886219/.zip>
- <https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383887627/.zip>
- <https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383889035/.zip>

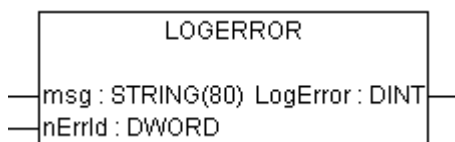
Project description

- [FB_LocalClient function block \[► 55\]](#)
- [FB_LocalServer function block \[► 60\]](#)
- [Testing the function blocks \[► 52\]](#)
- [.NET client project \[► 66\]](#)

Auxiliary functions in the project example

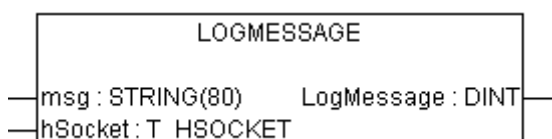
In the example several functions, constants and function blocks are used, which are briefly described below:

```
FUNCTION LogError : DINT
```



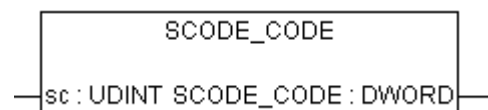
The function writes a message with the error code into the log book of the operating system (Event Viewer). The global variable *bLogDebugMessages* must first be set to TRUE.

```
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.

FUNCTION SCODE_CODE : DWORD



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

Global constants/variables

Name	Default value	Description
bLogDebugMessages	TRUE	Activates/deactivates writing of messages into the log book of the operating system;
MAX_CLIENT_CONNECTIONS	5	Maximum number of remote clients that can simultaneously establish a connection with the server;
MAX_PLCPRJ_RXBUFFER_SIZE	1000	Max. length of the internal receive buffer;
PLCPRJ_RECONNECT_TIME	T#3s	SERVER: After this time has elapsed, the local server will attempt to re-open the listener socket; CLIENT: 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	SERVER and CLIENT: The client and server reads (polls) data from the server or client using this cycle;
PLCPRJ_RECEIVE_TIMEOUT	T#50s (SERVER) T#10s (CLIENT)	SERVER: After this time has elapsed, the local server aborts the reception if no data bytes could be received during this time; CLIENT: After this time has elapsed, the local client aborts the reception if no data bytes could be received during this time;
PLCPRJ_ACCEPT_POLLING_TIME	T#1s	At these intervals, the local server will attempt to accept the connection requests of the remote client;
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).

8.1.1.1 Testing the client and server function blocks

1. Open the <https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383886219/.zip> on the local PC. The IP address of the server has to be adapted to your remote system (initialisation values of the *sRemoteHost* variables). Load the project into the PLC runtime system on the local PC. Start the PLC.

```
PROGRAM MAIN
VAR
    fbClient1          : FB_LocalClient := ( sRemoteHost:= '172.16.11.83' (* IP address of remote s
erver! *), nRemotePort:= 200 );
```

```

    fbClient2      : FB_LocalClient := ( sRemoteHost:= '172.16.11.83', nRemotePort:= 200 );
    fbClient3      : FB_LocalClient := ( sRemoteHost:= '172.16.11.83', nRemotePort:= 2
00 );
    fbClient4      : FB_LocalClient := ( sRemoteHost:= '172.16.11.83', nRemotePort:= 2
00 );
    fbClient5      : FB_LocalClient := ( sRemoteHost:= '172.16.11.83', nRemotePort:= 2
00 );

    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 5 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 server). 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.

2. Open the <https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383887627/.zip> also on the local PC. Select a PLC runtime system on a remote PC as the target system. Load the PLC program into the runtime system. Start the PLC.

```

PROGRAM MAIN
VAR
    fbServer      : FB_LocalServer := ( sLocalHost := '172.16.11.83' (*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

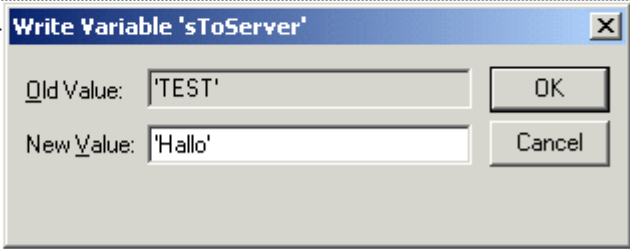
IF NOT fbSocketCloseAll.bBusy THEN
    fbServer( bEnable := bEnableServer );
END_IF

```

3. Set the *bEnableClient4* variable in the client project to TRUE. The second client 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.

```

+---fbConnect
+---fbClose
+---fbClientDataExcha
+---fbConnectTON
+---fbDataExchaTON
    eStep = CLIENT_STATE_DATAEXCHA_WAIT
    sRemoteHost = '172.16.5.36'
    nRemotePort = 200
    sToServer = 'TEST'
    bEnable = TRUE
    bConnected = TRUE
+---hSocket
    bBusy = TRUE
    bError = FALSE
    nErrId = 0
    sFromServer = 'TEST'
    
```



Double-click to open the dialog for writing the *sToServer* variable. Change the value of the string variable, for example to 'Hallo'. Close the dialog with OK. Write the new value into the PLC (CTRL+F7). Shortly afterwards, the value read back by the server can also be seen online.

```

+---fbClient4
    +---fbConnect
    +---fbClose
    +---fbClientDataExcha
    +---fbConnectTON
    +---fbDataExchaTON
        eStep = CLIENT_STATE_DATAEXCHA_WAIT
        sRemoteHost = '172.16.5.36'
        nRemotePort = 200
        sToServer = 'Hallo'
        bEnable = TRUE
        bConnected = TRUE
    +---hSocket
        bBusy = TRUE
        bError = FALSE
        nErrId = 0
        sFromServer = 'Hallo'
+---fbClient5
    bEnableClient1 = TRUE
    
```

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

```

+---fbListen
+---fbClose
+---fbConnectTON
  eStep = SERVER_STATE_IDLE
----fbRemoteClient
  +---fbRemoteClient[1]
  ----fbRemoteClient[2]
    +---fbAccept
    +---fbClose
    +---fbServerDataExcha
    +---fbAcceptTON
    eStep = CLIENT_STATE_DATAEXCHA_WAIT
    +---hListener
    bEnable = TRUE
    bAccepted = TRUE
    +---hSocket
    bBusy = TRUE
    bError = FALSE
    nErrId = 0
    sFromClient = 'Hallo'
  +---fbRemoteClient[3]
  +---fbRemoteClient[4]
  +---fbRemoteClient[5]
  i = 6
  sLocalHost = "
  nLocalPort = 200
  bEnable = TRUE
  bListening = TRUE
+---hListener
  nAcceptedClients = 2
  bBusy = TRUE
  bError = FALSE
  nErrId = 0
  
```

5. In the server and client examples, messages are written into the log book of the operating system during establishment/closing of the connection and in the event of an error. This facilitates troubleshooting. These messages can be displayed in the logger output of the TwinCAT System Manager. Start the TwinCAT System Manager on the local system and activate the logger output. Now deactivate the two clients (set *bEnableClient1* and *bEnableClient4* to FALSE).

Server (Port)	Timestamp	Meldung
TCPLC (801)	10.02.2004 09:38:52 838 ms	LOCAL client CLOSED! Internal handle: 1
TCPLC (801)	10.02.2004 09:38:52 118 ms	LOCAL client CLOSED! Internal handle: 2

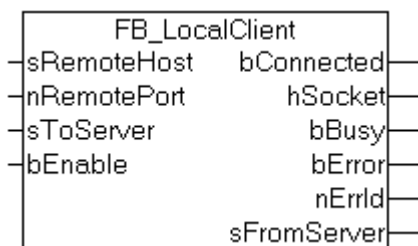
Ready

The server messages can also be displayed on the local PC. To this end, you have to open a second instance of the TwinCAT System Manager on the local PC and select the remote PC as the target system in the TwinCAT System Manager.

8.1.1.2 PLC Client

8.1.1.2.1 FB_LocalClient

Here you can unpack the complete source for the client project: <https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383886219/.zip>;



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](#) [58]). 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 *sFromServer*. 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

```

Also see about this

- 📄 FB_SocketConnect [▶ 18]
- 📄 FB_SocketClose [▶ 19]
- 📄 FB_ClientDataExcha [▶ 58]

8.1.1.2.2 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 immediately *)
            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 remaining 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_OV
ERFLOW );
        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
    END_IF

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

```

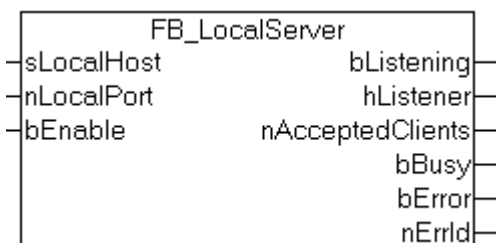
Also see about this

-  [FB_SocketSend \[▶ 24\]](#)
-  [FB_SocketReceive \[▶ 25\]](#)

8.1.1.3 PLC Server

8.1.1.3.1 FB_LocalServer

Here you can unpack the complete source for the server project: <https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383887627/.zip>;



The server must first be allocated a unique *sLocalHost* IP address and an *nLocalPort* 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* [▶ 62]. 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
returned 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
END_IF

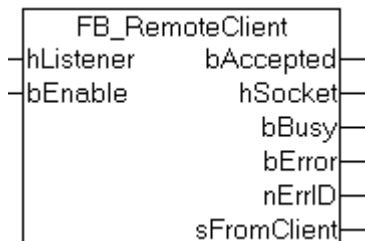
SERVER_STATE_ERROR:
  bError := TRUE;
  IF bListening THEN
    eStep := SERVER_STATE_REMOTE_CLIENTS_CLOSE;
  ELSE
    bBusy := FALSE;
    eStep := SERVER_STATE_IDLE;
  END_IF
END_IF
END_CASE

```

Also see about this

-  [FB_SocketListen](#) [▶ 22]
-  [FB_SocketClose](#) [▶ 19]
-  [FB_RemoteClient](#) [▶ 62]

8.1.1.3.2 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](#) [▶ 64]). 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

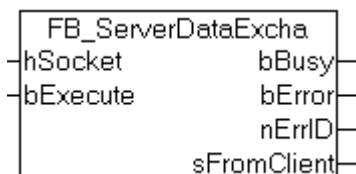
CLIENT_STATE_ERROR:
  bError := TRUE;
  IF bAccepted THEN
    eStep := CLIENT_STATE_CLOSE_START;
  ELSE
    eStep := CLIENT_STATE_IDLE;
    bBusy := FALSE;
  END_IF
END_CASE

```

Also see about this

-  [FB_SocketAccept \[▶ 23\]](#)
-  [FB_SocketClose \[▶ 19\]](#)
-  [FB_ServerDataExcha \[▶ 64\]](#)

8.1.1.3.3 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 read (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 remaining data bytes *)
                                cbReceived := cbReceived - cbFrame;
                            (* recalculate the remaining 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 terminated *)
                            LogError( 'FB_SocketReceive (remote client)', PLCPRJ_ERROR_RECEIVE_BUFFER_OVERFLOW );
                            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_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 (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

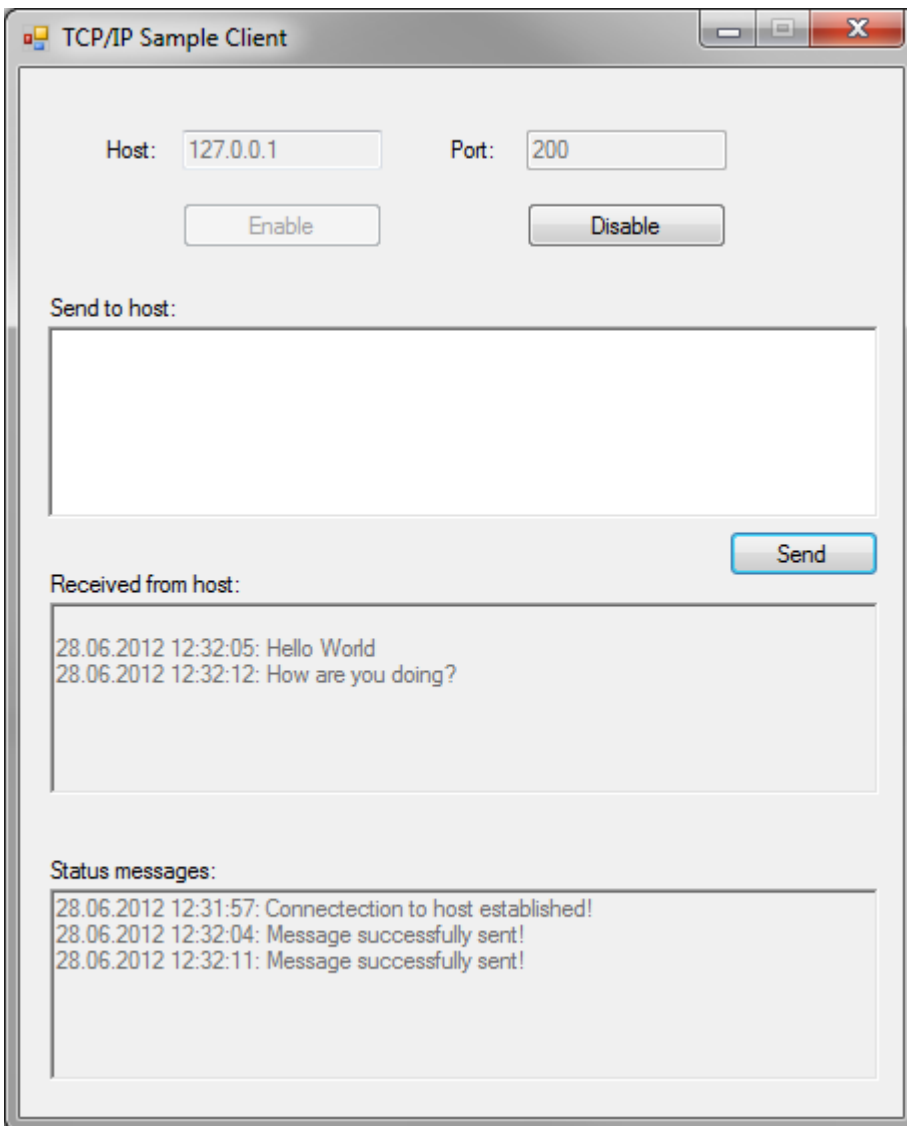
```

Also see about this

-  [FB_SocketReceive \[► 25\]](#)
-  [FB_SocketSend \[► 24\]](#)

8.1.1.4 .NET sample client

This project example shows how a client for the PLC TCP/IP server can be realised 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
{
    public partial class Form1 : Form
    {
        /* #####
         * Constants
         * ##### */
        private const int RCVBUFFERSIZE = 256; // buffer size for receive buffer
    }
}
```

```

private const string DEFAULTTIP = "127.0.0.1";
private const string DEFAULTPORT = "200";
private const int TIMERTICK = 100;

/* #####
 * Global variables
 * ##### */
private static bool _isConnected; // signals whether socket connection is active or not
private static Socket _socket; // object used for socket connection to TCP/IP-Server
private static IPEndPoint _ipAddress; // contains IP address as entered in text field
private static byte[] _rcvBuffer; // receive buffer used for receiving response from TCP/IP-
Server
public Form1()
{
    InitializeComponent();
}

private void 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 = DEFAULTTIP;
    txt_port.Text = DEFAULTPORT;

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

private void 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
 * ##### */
private void timer1_Tick(object sender, EventArgs e)
{
    if (!_isConnected)
        connect();
}

private void 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 establishe
d!\n");
    else
        rtb_statMsg.AppendText(DateTime.Now.ToString() + ": A connection to the host could n
ot be established!\n");
}
catch (Exception ex)
{
    MessageBox.Show("An error occured while establishing a connection to the server: " + ex)
;
}
}

private void 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 = new ASCIIEncoding();
    byte[] tempBuffer = enc.GetBytes(rtb_sendMsg.Text);
    byte[] sendBuffer = new byte[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)
            throw new Exception();
        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, nu
ll, null);

            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);
    }
}

private void 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() + ": Connection to host closed!\n");
rtb_rcvMsg.Clear();
rtb_statMsg.Clear();
}
}
}
}

```

8.1.2 UDP example

The following example shows the implementation of a simple peer-to-peer application in the PLC. The PLC application presented can send a test string to a remote PC and at the same time receive test strings from a remote PC. The test strings are displayed in a message box on the monitor of the target computer. A simple implementation of a suitable communication partner in .NET is also presented. The example can be used as a basis for realizing more complex implementations.

System requirements

- TwinCAT v2.8 or higher. Level: TwinCAT PLC as a minimum.
- Installed TwinCAT TCP/IP connection server (v1.0.0.31 or higher). If two PCs are used for the test, the TwinCAT TCP/IP Connection Server should be installed on both PCs.
- TwinCAT PLC library Tcplp.Lib (v1.0.4 or higher).

Project sources

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

- PLC project: <https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383890443/.zip>
- PLC project: <https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383891851/.zip>
- .NET: <https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383893259/.zip>

Project description

- [peer-to-peer PLC application \[► 73\]](#)
- [.NET communication partner for the PLC \[► 77\]](#)
- [Testing the applications \[► 72\]](#)

Auxiliary functions in the project example

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

```

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).

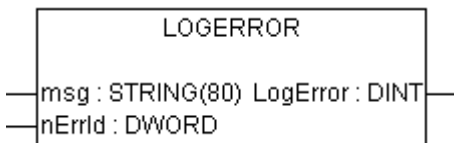
A Fifo entry consists of the following components:

```

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

FUNCTION LogError : DINT

```

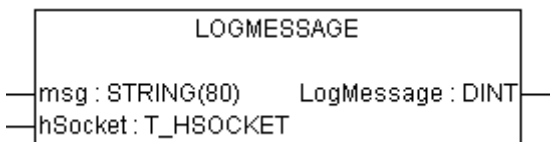


The function writes a message with the error code into the log book of the operating system (Event Viewer). The global variable *bLogDebugMessages* must first be set to TRUE.

```

FUNCTION LogMessage : DINT

```

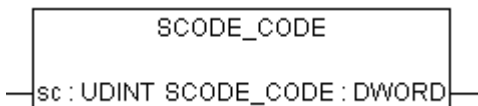


The function writes a message into the log book of the operating system (Event Viewer) if a new socket was opened or closed. The global variable *bLogDebugMessages* must first be set to TRUE.

```

FUNCTION SCORE_CODE : DWORD

```



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

Global constants/variables

Name	Default value	Description
g_sTclpConnSvrAddr	"	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_RECFFIFO_OVERFLOW	16#8104	Sample project error code: The receive Fifo is full.

8.1.2.1 Testing the peer-to-peer applications

The test requires two PCs. Alternatively, the test may be carried out with two runtime systems on a single PC. The constants with the port numbers and the IP addresses of the communication partners must be modified accordingly.

Example test configuration with 2 PCs:

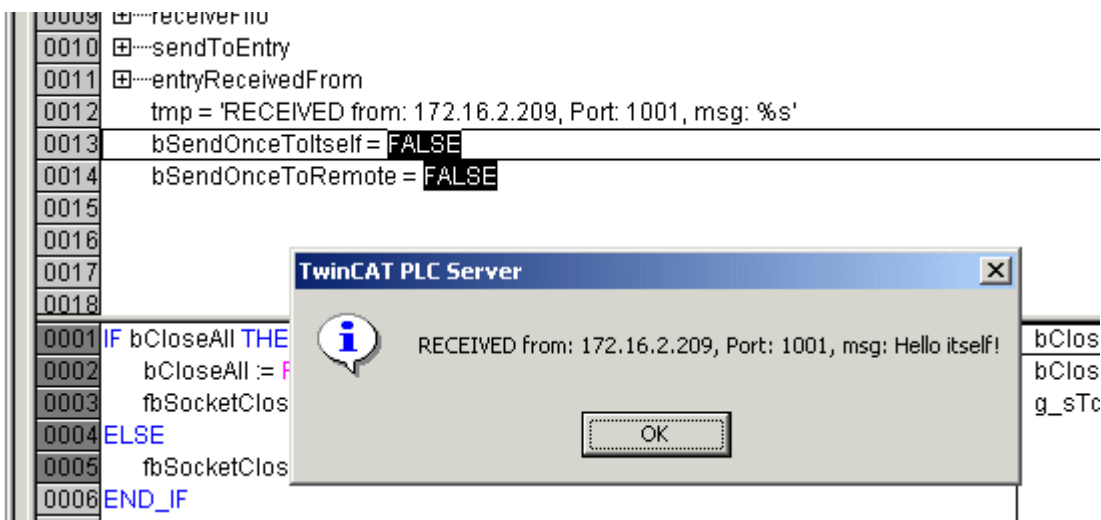
Device A is located on the local PC and has the IP address '172.16.2.209'

Device B is located on the remote PC and has the IP address '172.16.6.195'

Testing the PLC devices

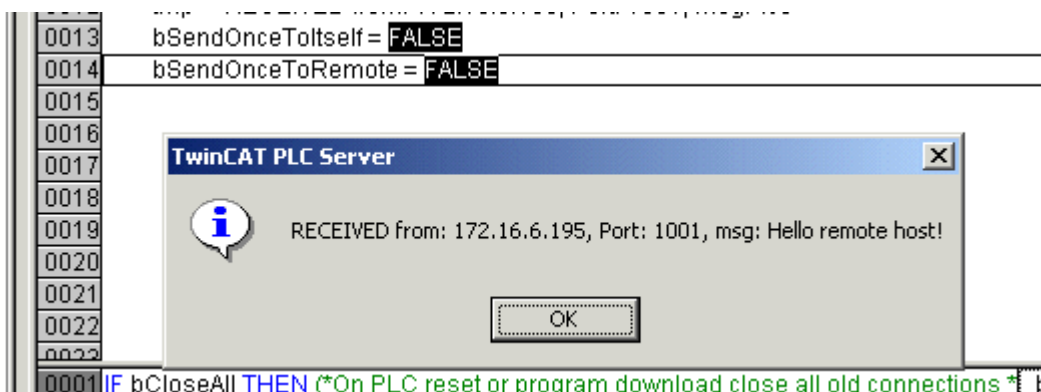
1. Open the <https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383890443/.zip> on the local PC. The constant `REMOTE_HOST_IP` in MAIN has to be adapted to the real IP address of your remote system (in our example: '172.16.6.195'). Load the project into the PLC runtime system. Start the PLC.

2. Local PC: In online mode, write the value `TRUE` to the boolean variable `bSendOnceToItself` in MAIN. Shortly afterwards, a message box with the test string should appear. The UDP data were sent to the local port and IP address.



3. Remote PC: Open the <https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383891851/.zip> on the remote PC. The constant `REMOTE_HOST_IP` in MAIN has to be adapted to the IP address of your local PC (in our example: '172.16.2.209'). Load the project into the PLC runtime system. Start the PLC.

4. Remote PC: In online mode, write the value `TRUE` to the boolean variable `bSendOnceToRemote` in MAIN. Shortly afterwards, a message box with the test string should appear on the local PC.

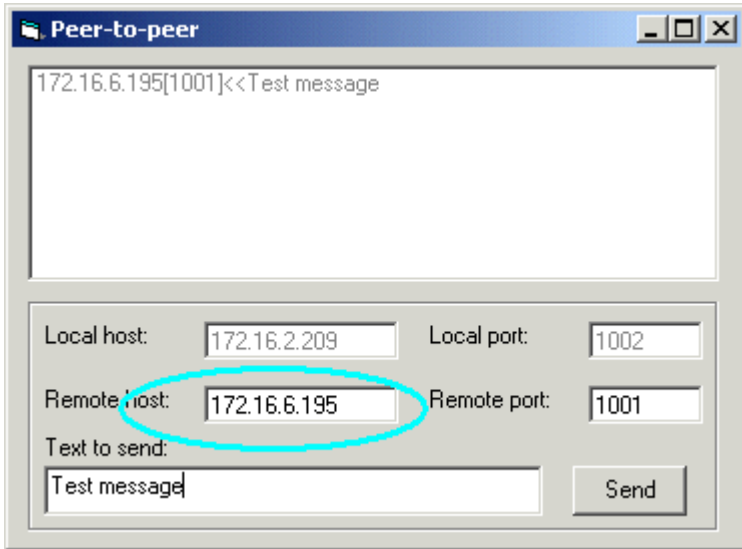


Test with the Visual Basic application

Here you can unpack the Visual Basic sources: <https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383894667/.zip>.

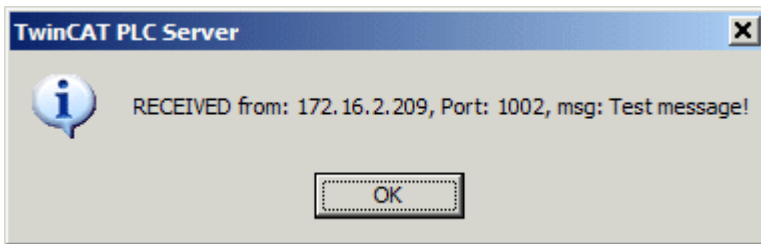
5. Local PC: Start the Visual Basic application (PeerToPeer.exe).

6. Local PC: In the VB dialog, the IP address of the remote host has to be adapted to the real IP address of the remote PC (in our example '172.16.6.195').



7. Local PC: Click the send button. A test string is sent to the remote device with port number 1001. In our case it is the PLC application.

8. Remote PC: Shortly afterwards, a message box with the test string should appear.



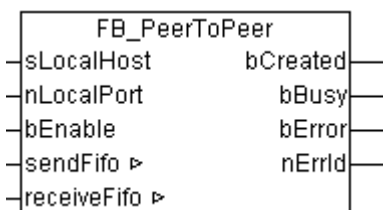
8.1.2.2 PLC Client/Server

8.1.2.2.1 UDP example: peer-to-peer PLC devices A and B

Here you can unpack the complete sources: <https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383890443/.zip>, and <https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383891851/.zip>.

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`.

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_RECFFIFO_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
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` [► 20] function block. If one of the variables `bSendOnceToItself` or `bSendOnceToRemote` has an 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 =>
        send it to itself *)
        sendToEntry.sRemoteHost     := LOCAL_HOST_IP;      (* sRemoteHost == sLocalHost
        t =>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

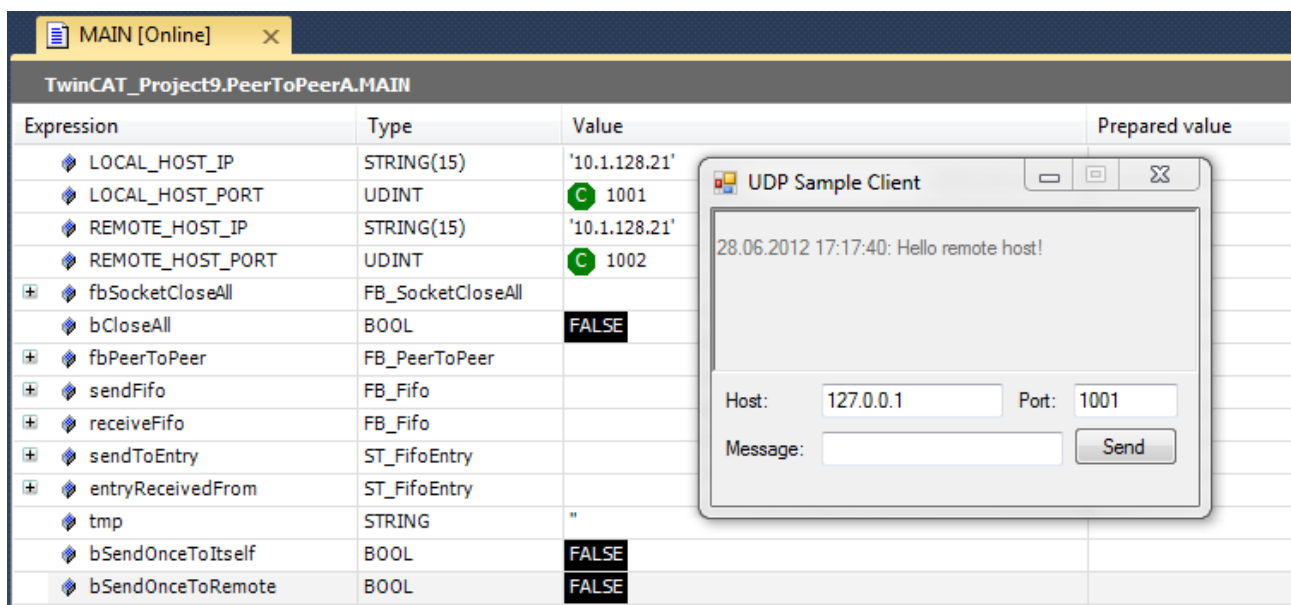
```

Also see about this

- 📖 FB_SocketUdpCreate [▶ 26]
- 📖 FB_SocketClose [▶ 19]
- 📖 FB_SocketUdpReceiveFrom [▶ 29]
- 📖 FB_SocketUdpSendTo [▶ 27]
- 📖 T_HSOCKET [▶ 33]

8.1.2.3 .NET Peer-to-Peer communication

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



How it works

The sample uses the includes 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

How to set up the PLC sample

This .NET sample may be used together with the PLC samples Peer-to-Peer device A or B. If you run the application on separate computer than the PLC runtime, you need to configure IP addresses in both applications according to the assumed specifications from above.

Setting	Type	Description
LOCAL_HOST_IP	Global constant	Needs to be set to 10.1.128.21 (IP of computer which runs PLC)
LOCAL_HOST_PORT	Global constant	Set this constant to 1001 (default value)

Setting	Type	Description
REMOTE_HOST_IP	Global constant	Needs to be set to 10.1.128.30 (IP of computer which runs .NET sample)
REMOTE_HOST_PORT	Global constant	Set this constant to 1002 (default value)
bSendOnceToRemote	Global variable	If set to TRUE, a UDP packet will be send to REMOTE_HOST_IP

How to set up the .NET sample

Setting	Type	Description
DEFAULTIP	Global constant	Will be used in text field "Host" as default value. Set it to 10.1.128.21.
DEFAULTDESTPORT	Global constant	Will be used in text field "Port" as default value. Set it to 1001.
DEFAULTTOWNPORT	Global constant	Will be used by PLC sample in REMOTE_HOST_PORT. Set it to 1002.
DEFAULTSOURCEPORT	Global constant	Used as source port for sending out UDP packet. Not important in this sample.

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

using System.Net;
using System.Net.Sockets;
using System.Threading;

namespace TcpIpServer_SampleClientUdp
{
    public partial class Form1 : Form
    {
        /
        * ##### * C
        onstants * #####
        ##### */privateconststring DEFAULTIP = "127.0.0.1";
        privateconstint DEFAULTDESTPORT = 1001; // Used as destination portprivateconstint DEFAULTTOWNPOR
        T = 1002; // Used for binding when listening for UDP messagesprivateconstint DEFAULTSOURCEPORT = 110
        00; // Only used as source port when sending UDP messages/
        * ##### * G
        lobal variables * #####
        ##### */privatestaticUdpClient _udpClient;
        privatestaticIPEndPoint _ipAddress; // Contains IP address as entered in text fieldprivatestatic
        Thread _rcvThread; // Background thread used to listen for incoming UDP packetspublic Form1()
        {
            InitializeComponent();
        }

        /
        * ##### * E
        vent handler method called when button "Send" is pressed * #####
        ##### */
        privatevoid cmd_send_Click(object sender, EventArgs e)
        {
            byte[] sendBuffer = null;

            /
            * #####
            * Preparing UdpClient, connecting to UDP server and sending content of text field * #####
            ##### */try
            {
                _ipAddress = newIPEndPoint(IPAddress.Parse(txt_host.Text), Convert.ToInt32(txt_port.Text));
                _udpClient = newUdpClient(DEFAULTSOURCEPORT);
                _udpClient.Connect(_ipAddress);

                sendBuffer = Encoding.ASCII.GetBytes(txt_send.Text);
            }
            catch { }
        }
    }
}
```

```

_udpClient.Send(sendBuffer, sendBuffer.Length);

_udpClient.Close();
}
catch (Exception ex)
{
    MessageBox.Show("An unknown error occurred: " + ex);
}
}

/
* ##### * E
vent handler method called when application starts * #####
##### */privatevoid Form1_Load(object sender, EventArgs e)
{
    txt_host.Text = DEFAULTTIP;
    txt_port.Text = DEFAULTTDESTPORT.ToString();
    rtb_rcv.Enabled = false;

    /
* ##### * #####
* Creating background thread which synchronously listens for incoming UDP packets * #####
##### */
    _rcvThread = newThread(rcvThreadMethod);
    _rcvThread.Start();
}

/
* ##### * D
elegate, so that background thread may write into text field on GUI * #####
##### */
publicdelegatevoidrcvThreadCallback(string text);

/
* ##### * M
ethod called by background thread * #####
##### */privatevoid rcvThreadMethod()
{
    /
* ##### * #####
* Listen on any available local IP address and specified port (DEFAULTTOWNPORT) * #####
##### */
byte[] rcvBuffer = null;
    IPEndPoint ipEndPoint = newIPEndPoint(IPAddress.Any, DEFAULTTOWNPORT);;
    UdpClient udpClient = newUdpClient(ipEndPoint);

    /
* ##### * #####
* Continously start a synchronous listen for incoming UDP packets. If a packet has arrived,
* write its content to receive buffer and then into the text field. After that, start circle
* again. * #####
##### */while (true)
    {
        rcvBuffer = udpClient.Receive(ref ipEndPoint); // synchronous call
        rtb_rcv.Invoke(newrcvThreadCallback(this.AppendText), newobject[] { "\n" + DateTime.Now.ToSt
ring() + ": " + Encoding.ASCII.GetString(rcvBuffer) });
    }
}

/
* ##### * H
elper method for delegate * #####
##### */privatevoid AppendText(string text)
{
    rtb_rcv.AppendText(text);
}

/
* ##### * S
top background thread when application closes * #####
##### */
privatevoid Form1_FormClosed(object sender, FormClosedEventArgs e)
{
    _rcvThread.Abort();
}

```

```

}
}
}

```

8.2 TcSocketHelper.lib examples

The examples presented here are based on the functionality offered by TcSocketHelper.Lib.

System requirements:

- TwinCAT version 2.10 Build 1331 (or higher)
- TwinCAT Connection Server v1.0.0.47 or higher installed on the client and server PC;

Communication settings used in the examples:

- 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 address and the IP address accordingly (use the PING command in the prompt to check the connection).

You can test the client and server with the default values on one PC by loading the client application into the first PLC runtime system (801) and the server application into the second PLC runtime system (811).

The behaviour of the PLC project example is determined by the following 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 examples define and use the following internal error codes:

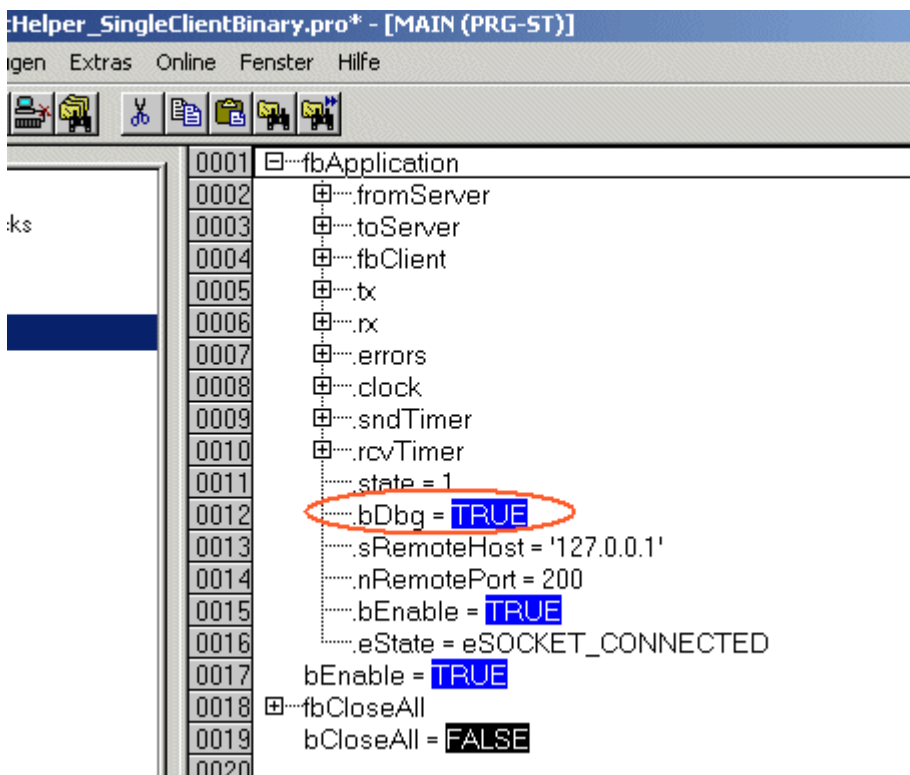
Error code	Value	Description
PLCPRJ_ERROR_RECEIVE_BUFFER_OVERFLOW	16#8101	The internal receive buffer reports an overflow.
PLCPRJ_ERROR_SEND_BUFFER_OVERFLOW	16#8102	The internal send buffer reports an overflow.
PLCPRJ_ERROR_RESPONSE_TIMEOUT	16#8103	The server has not sent the response within the specified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

PLC project	Description
https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383896075/.zip (Client) https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383897483/.zip (Server)	Implementation of an "echo" client/server. The client cyclically sends a test string (sToServer) to the remote server. The server returns the same string to the client unchanged (sFromServer).

PLC project	Description
<p>https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383898891/.zip (Client)</p> <p>https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383900299/.zip (Server)</p>	<p>As above, with the difference that the server can establish up to 5 connections. The client application has 5 client instances. Each instance establishes a connection to the server.</p>
<p>https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383901707/.zip (Client)</p> <p>https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383903115/.zip (Server)</p>	<p>A client-server application for the exchanging of binary data.</p> <p>A simple sample protocol is used. The length of the binary data and a frame counter for the sent and received telegrams are transferred in the protocol header.</p> <p>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.</p> <p>The structure declaration on the client and server sides can be adapted as required. The structure declaration must be identical on both sides.</p> <p>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.</p> <p>The server functionality is implemented in the function block FB_ServerApplication and the client functionality in the function block FB_ClientApplication.</p> <p>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.</p> <p>If you require a functionality, you have to modify the function blocks FB_ServerApplication and FB_ClientApplication accordingly.</p>
<p>https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383904523/.zip (Client)</p> <p>https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383905931/.zip (Server)</p>	<p>As above, with the difference that the server can establish up to 5 connections. The client application has 5 client instances. Each instance establishes a connection to the server.</p>

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

For troubleshooting purposes you can set the input variable *bDbg* to TRUE, thereby activating the debugging output for the sent data in the TwinCAT System Manager Log View:



8.3 TcSnmp.lib

8.3.1 Sample: Client trap

This sample describes a simple Trap send from a PLC to a SNMP management server. Traps can be used to alert thresholds. On every hundred increment of the counter a trap will be send.

Download: <https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383907339/.zip>

Variable Declaration

```
PROGRAM MAIN
VAR
  SendTrap:      FB_SendTrap;
  iState:        INT := 0;
  iCounter:      UDINT:= 0;
  stVarBind:    ST_SNMP_VariableBinding;
END_VAR
```

PLC Program

```
CASE iState OF

  0: (* send trap on every hundred cycle *)
    iCounter := iCounter + 1;
    IF ((iCounter MOD 100) = 0) THEN
      iState := 10;
    END_IF

  10: (* enable FB *)
    SendTrap.bEnable := TRUE;
    IF SendTrap.bEnabled THEN
      iState := 20;
    END_IF

  20: (* set SNMP trap parameter *)
    stVarBind.iType := E_SNMP_INTEGER;
    stVarBind.iLength := SIZEOF(iCounter);
    stVarBind.pArrValue := ADR(iCounter);

    (* TODO: assign object ID of management information base (MIB) *)
    SendTrap.sObjectID := '1.3.6.1.2.1.1.5.0';
```

```

    (* TODO: check default community string *)
    SendTrap.sCommunity:= 'public';
    SendTrap.iGenericTrapNumber:= E_SNMP_WarmStart;
    SendTrap.bExecute := TRUE;
    iState := 30;

30: (* reset *)
    SendTrap.bExecute := FALSE;
    SendTrap.bEnable := FALSE;
    IF NOT SendTrap.bBusy THEN
        iState := 0;
    END_IF

    IF SendTrap.bError THEN
        SendTrap.bEnable := FALSE;
        iState := 99;
    END_IF

99: (* Error case *)
    ;
END_CASE

SendTrap(
    (* TODO: add device IP of TwinCAT device *)
    sLocalHostIp := '',
    (* TODO: add SNMP Manager IP *)
    sManagerIP := ''
);

```

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	TcSnmplib (Tcplp.Lib;
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	Standard.Lib; TcBase.Lib; TcSystem.Lib are included automatically)

8.3.2 Sample: SNMP multiple client trap

This Sample describes the sending of a trap by multiple values. Traps can be used to alert thresholds. On every hundred increment of the counter a trap will be send.

Download: <https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383908747/.zip>

Variablen Declaration

```

PROGRAM MAIN
VAR
    SendTrap:      FB_SendTrap;
    iState:        INT := 0;
    iCounter:      UINT:= 0;
    sObjectId:     STRING :='1.3.6.1.4.1.2';
    sMessage:      T_MaxString:='Information string';
    iGauge:        UDINT:= 4294967295;
    iTimeTicks:    UDINT:= 1000000;
    arrVarBind:    ARRAY[1..5] OF ST_SNMP_VariableBinding;
END_VAR

```

PLC Program

```

CASE iState OF

0: (* send trap on every hundred cycle *)
    iCounter := iCounter + 1;
    IF ((iCounter MOD 100) = 0) THEN
        iState := 10;
    END_IF

10: (* enable FB *)
    SendTrap.bEnable := TRUE;
    IF SendTrap.bEnabled THEN
        iState := 20;
    END_IF

```

```

20: (* set SNMP trap parameter *)
arrVarBind[1].iType := E_SNMP_INTEGER;
arrVarBind[1].iLength := 4;
arrVarBind[1].pArrValue := ADR(iCounter);
arrVarBind[1].sOID := '1.3.1.3.255.1.1';

(*Variable Binding 2*)
arrVarBind[2].iType := E_SNMP_OBJECTID;
arrVarBind[2].iLength := LEN(sObjectId);
arrVarBind[2].pArrValue := ADR(sObjectId);
arrVarBind[2].sOID := '1.3.6.1.3.255.2';

(*Variable Binding 3*)
arrVarBind[3].iType := E_SNMP_OCTETSTRING;
arrVarBind[3].iLength := LEN(sMessage);
arrVarBind[3].pArrValue := ADR(sMessage);
arrVarBind[3].sOID := '1.3.6.1.3.255.3';

(*Variable Binding 4*)
arrVarBind[4].iType := E_SNMP_GAUGE32;
arrVarBind[4].iLength := 4;
arrVarBind[4].pArrValue := ADR(iGauge);
arrVarBind[4].sOID := '1.3.6.1.3.255.4';

(*Variable Binding 5*)
arrVarBind[5].iType := E_SNMP_TIMETICKS;
arrVarBind[5].iLength := 4;
arrVarBind[5].pArrValue := ADR(iTimeTicks);
arrVarBind[5].sOID := '1.3.6.1.3.255.5';

SendTrap.sCommunity := 'public';
SendTrap.iGenericTrapNumber := E_SNMP_WarmStart;
SendTrap.bExecute := TRUE;
iState := 30;

30: (* reset *)
SendTrap.bExecute := FALSE;
SendTrap.bEnable := FALSE;
IF NOT SendTrap.bBusy THEN
    iState := 0;
END_IF

IF SendTrap.bError THEN
    SendTrap.bEnable := FALSE;
    iState := 99;
END_IF

99: (* Error case *)
;
END_CASE

SendTrap(
(* TODO: add device IP of TwinCAT device *)
sLocalHostIp := '',
(* TODO: add SNMP Manager IP *)
sManagerIP := '',
pArrVarBinding := ADR(arrVarBind),
nVarBindings := 5
);

```

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	TcSnmp.lib (Tcplp.Lib; Standard.Lib; TcBase.Lib; TcSystem.Lib are included automatically)
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	

8.3.3 Sample: SNMP Get request

This samples describes how a TwinCAT device can answer on a SNMP GET-request from SNMP manager. If a GET-Request arrived in the PLC, the requested information will be sent if the object id match.

Download: <https://infosys.beckhoff.com/content/1033/tcpipserver/Resources/11383910155/.zip>

Variablen Declaration

```
PROGRAM MAIN
VAR
  GetSnmplib:    FB_GetSnmplib;
  iValue:        DINT;
  sValue:        T_MaxString;
  arrVarBind:    ARRAY[0..1] OF ST_SNMP_VariableBinding;
  iState : INT := 0;
END_VAR
```

PLC Program

```
CASE iState OF
  0: (* Enable *)
    GetSnmplib.bEnable := TRUE;
    IF GetSnmplib.bEnabled THEN
      iState := 10;
    END_IF

  10: (* Wait for SNMP GET request *)
    GetSnmplib.bReceive := TRUE;
    IF GetSnmplib.bReceived THEN
      GetSnmplib.bReceive := FALSE;
      iState := 20;
    END_IF

  20: (* Compare oid's *)

    (* TODO: Assign your own ObjectID (oid) of the management information base (MIB) *)
    IF GetSnmplib.sRecObjectID = '1.3.6.1.2.1.1.5.0' THEN
      sValue := 'BECKHOFF_DEVICE';
      arrVarBind[0].iType := E_SNMP_OCTETSTRING;
      arrVarBind[0].iLength := LEN(sValue);
      arrVarBind[0].pArrValue := ADR(sValue);
      arrVarBind[0].sOID := GetSnmplib.sRecObjectID;
      GetSnmplib.pArrVarBinding := ADR(arrVarBind);
      GetSnmplib.nVarBindings := 1;
      GetSnmplib.iError := 0;
      GetSnmplib.bSendResponse := TRUE;
    ELSE
      (* The requested ObjectID was not found *)
      GetSnmplib.nVarBindings := 0;
      GetSnmplib.iError := 2;
      GetSnmplib.bSendResponse := TRUE;
    END_IF
    iState := 30;

  30: (* reset *)
    GetSnmplib.bSendResponse := FALSE;
    GetSnmplib.bSendTrap := FALSE;
    IF NOT GetSnmplib.bBusy THEN
      iState := 10;
    END_IF
    IF GetSnmplib.bError THEN
      GetSnmplib.bEnable := FALSE;
      iState := 0;
    END_IF
END_CASE

GetSnmplib(
  (* TODO: check community string *)
  sCommunity := 'public',
  iGenericTrapNumber:= E_SNMP_WarmStart,
  (* TODO: add device IP of TwinCAT device *)
  sLocalHostIp := '',
  (* use SNMP Port 163, if default port 161 is in use by OS *)
  sLocalHostPort := 163,
  (* TODO: ADD SNMP Manager IP *)
  sManagerIP := '',
);
```

Requirements

Development environment	Target system type	PLC libraries to be linked
TwinCAT version 2.8.0 or higher	PC or CX (x86)	TcSnmp.lib (Tcplp.Lib; Standard.Lib; TcBase.Lib; TcSystem.Lib are included automatically)
TwinCAT v2.10.0 Build >= 1301	CX (ARM)	

9 Error codes

Requirements

Codes (hex)	Codes (dec)	Error source	Description
0x00000000-0x0007800	0-30720	TwinCAT system error codes	TwinCAT system error (including ADS error codes)
0x00008000-0x00080FF	32768-33023	Internal TwinCAT TCP/IP Connection Server error codes [▶ 87]	Internal error of the TwinCAT TCP/IP Connection Server
0x00009000-0x00090FF	36864-37119	Internal SNMP error codes [▶ 88]	Internal SNMP error codes
0x80070000-0x8007FFF	2147942400-2148007935	Error source = Code - 0x80070000 = Win32 system error codes	Win32 system error (including Windows sockets error codes)

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

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

9.2 Troubleshooting/diagnostics

1. 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.
2. Sniffer tools such as Wireshark enable logging of the entire network communication. The log can then be analyzed by Beckhoff support staff.
3. Check the hardware and software requirements described in this documentation (TwinCAT version, CE image version etc.).
4. Check the software installation hints described in this documentation (e.g. installation of CAB files on CE platform).
5. 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](#) [[▶ 87](#)].

6. Check if the other communication partner/software/device issues an error code.
7. 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.

9.3 SNMP_ErrorCodes

Hex	Dec	Description
0x9001	36865	INCORRECT PARAMETER SIZE
0x9002	36866	INVALID PARAMETER

More Information:
www.beckhoff.com/ts6310

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

