The concept of Sockets and basic Function Blocks for communication over Ethernet Part 2 TCP Server and TCP Client

Article · July 2014

CITATIONS
0

READS
9,179

1 author:

Wojciech Gomolka
Festo France
13 PUBLICATIONS **14** CITATIONS

SEE PROFILE

GOMOLKA Wojciech
FESTO France
8, rue du Clos de St.Catherine
94363 Bry sur Marne
wojciech.gomolka@fr.festo.com

CoDeSys® v.2.3.9

Library SysLibSockets.lib

The concept of Sockets and basic Function Blocks for communication over Ethernet

Part 2
TCP Server and TCP Client

1. Introduction

In the first part of this publication [1] we introduced the basic concepts and some details of the implementation of the Ethernet Socket for CoDeSys platform, details needed to achieve the basic Functional Blocks for communication under UDP protocol.

This part will present an implementation of functions from SysLibSockets.lib library to realize an application with TCP session under CoDeSys v.2.3.x. We present a pragmatic approach to the creation of Functional Blocks TCPServer / TCPClient, the blocks that can be easily used to realize this kind of applications.

The Transmission Control Protocol (TCP) is one of the core protocols of the Internet protocol suite (IP). TCP provides reliable, ordered and error-checked transfer of data between two applications running on equipment connected to a local area network, Intranet or Internet. A big variety of standard protocols could be typically encapsulated in TCP: HTTP, HTTPS, SMTP, POP3, IMAP, SSH, FTP, Telnet... And the most popular industrial protocol Modbus TCP.

TCP connections are full duplex and End-to-End. Full duplex means that traffic can go in both directions at the same time. End-to-End means that each connection has exactly two end points. So, unlike UDP protocol, TCP does not support multicasting and broadcasting.

Because TCP allows the reliable transfer of data between two partners, it is very often used for specific tasks as the configuration of the remote equipment (transfer of parameters), data exchange between robot and vision system, RFID management, transfer (upload/download) of programs...

Of course, if applications do not require the reliability of a TCP connection or need the multicasting, they may instead use the connectionless User Datagram Protocol (UDP).

2. Relationship between Server and Client in the session TCP.

The figure 1 illustrates the sequence of the typical TCP session.

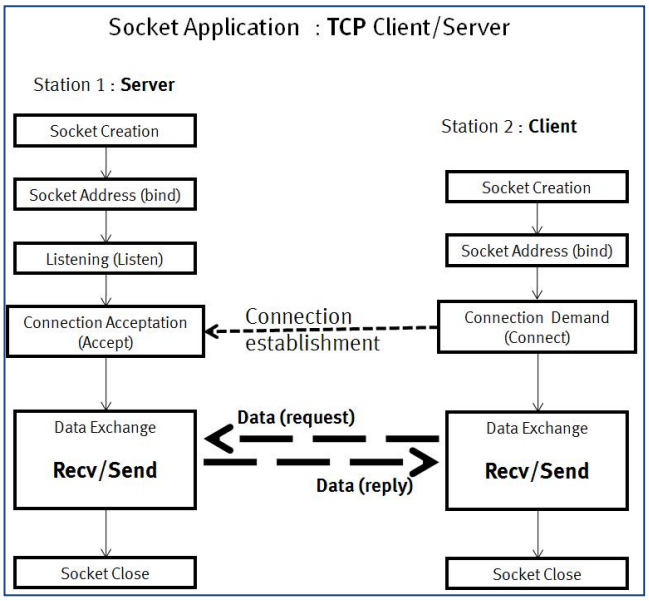

Fig.1 *Relationship between the Server and the Client in TCP session*

TCP session may be divided into three phases ([2], [3], [5]):

- **Connection Establishment** phase: the connection between two equipments (Client/Server) must be properly established before starting the transfer of data
- **Data Transfer** phase : the exchange of data between connected Client and Server
- **Connection Termination** phase : where connection is terminated and closed, and all allocated resources are released

From the point of view of the TCP Server, the starting point for the connection is “Listen” state where a Server waits for connection request from any remote TCP Client. But before listening, a Server must first create the socket and bind to (to open) a communication port.

When the Server is in the “Listen” state, a Client may initiate connection (the function Connect()). If Client’s request is accepted by Server, the connection is established and switches directly to Data Transfer phase. In this phase, both Client and Server can communicate.

The function Recv() is used to receive data, and the function Send() is used to send data. Theoretically, the order of call of functions Send() and Recv() doesn't matter. You can call one before the other or vice versa. But, for the same process, the sequence Recv() / Send() could be blocking and the application will not be able to send something if before, it does not receive something.

Either the Server or the Client could decide to close connection and to terminate TCP session. When the connection must be closed (or is lost), the function Close() is called for closing all open sockets and to release all allocated resources.

Note:

If an event that is not allowed occurs, the connection can enter into FAILURE state and can stay there (blocking state). For this, we propose to close the session and to switch into Connection Termination phase.

So, the following functions are used for the implementation of the functionality TCPServer and/or TCPClient.

TCP Server:

C/C++	CoDeSys	TCP phase
socket()	**SysSockCreate()**	*Connection Establishment*
bind()	**SysSockBind()**	
listen()	**SysSockListen()**	
accept()	**SysSockAccept()**	
send()	**SysSockSend()**	*Data Transfer*
recv()	**SysSockRecv()**	
close()	**SysSockClose()** **SysSockShutdown()**	*Connection Termination*

TCP Client:

C/C++	CoDeSys	TCP phase
socket()	**SysSockCreate()**	*Connection Establishment*
connect()	**SysSockConnect()**	
send()	**SysSockSend()**	*Data Transfer*
recv()	**SysSockRecv()**	
close()	**SysSockClose()** **SysSockShutDown()**	*Connection Termination*

3. Implementation of TCP functionality in the PLC application.

The socket is basically created and exploited in the blocking mode. This may cause trouble in PLC application because the program will stop on the function call where it waits the function finished.

By default, the call of socket function could be blocking for the following “TCP oriented” functions:

- SysSockRecv(), SysSockSend(),
- SysSockConnect(), SysSockAccept(),
- SysSockClose();

Theoretically, in Data Transfer phase of the TCP connection, the order of call of functions Send() and Recv() doesn't matter. But, for the same process (task), the sequence:

```
SysSockRecv();
SysSockSend();
```

could be critical. The program will not be able to send something if before, it does not receive something.

There are different ways to solve this problem in the PLC application:

1. The use of one task for PLC control and some several separate tasks for socket and communication handling
2. Changing of the socket mode into no blocking mode via function SysSockIoctl()
3. The use the function SysSockSelect()

Two last methods could be the simplest way to have no blocking mode of the socket behaviour. But, unfortunately, its implementation depends of PLC operating system. Sometimes, these options are not supported by OS.

So, the approach 1 could be an effective solution.

3.1 Functionality TCP Server

For TCP Server functionality we propose a solution based on the distribution of TCP functions into three separate tasks (Fig. 2)

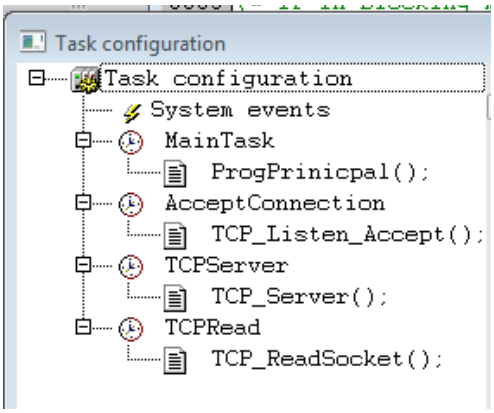

Fig.2 Main tasks for the functionality ***TCPServer***

Task 1: TCPServer

The management of main phases of the TCP connection; the task done by FB_TCP_Server()

a. Connection Establishment phase
    i. creation and opening of the socket (SysSockCreate(), SysSocketOption(), SysSocketBind())
    ii. Initiation of the TCP session (SysSockListen())
b. Data Transfer phase
    i. Transmission of messages to the connected TCP Client (SysSockSend())
c. Connection Termination phase
    i. Closing of the TCP connection TCP (SysSockShutDown(), SysSockClose())

Task 2: AcceptConnection

Asynchronous task to complete Connection Establishment phase (Listen state) when connection request is received; the task done by the program: TCP_Listen_Accept()

a. The reference of the client socket is automatically transmitted to the server in client's request. The Server initializes the new local socket (Sockaddres).
b. The Server calls SysSockAccept() function to accept the connection request and returns the new socket descriptor for established connection. So now, the server is able to answer client.
c. Optional : external timer for the timeout of the TCP connection

Task 3: TCPRead

Asynchronous task to complete Data Transfer phase; this task, done by FB TCP_ReadSocket(), allows to receive and read messages sent by connected Client

Note: If the message of Client is not present, SysSockRecv will block the task and wait for a new packet. So this function must be implemented in any separate asynchronous task.

3.1.1 CoDeSys (Server) Function Block: FB_TCP_Server()

The Function Block ***FB_TCP_Server*** gives a possibility of the Server functionality in Ethernet dialogue with any TCP Client.

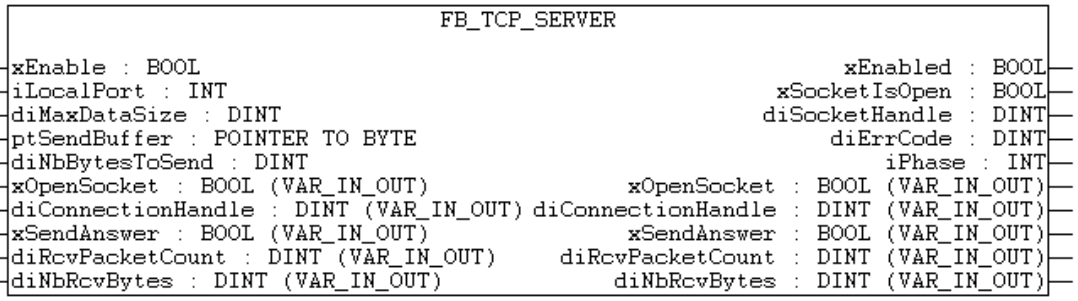

Fig.3 Function Block FB_TCP_Server

Firstly, this FB creates the main socket and initiates the connection (it launches the function SysSockListen()) and waits for the demand of connection sent by TCP Client. After reception of this demand, it waits for the acceptance of connection (SysSockAccept() launched by a separate asynchronous task), and afterwards it goes to the state where it waits for the reception of the TCP packet sent by the TCP Client. Then, the TCP Server may respond to the Client.

By setting the variable ***xSendAnswer*** a message can be transmitted to the Client for which the socket parameters are "pointed" by input variable ***diConnectionHandle***. After, the variable *xSendAnswer* will be reset automatically by Function Block.

IMPORTANT:

- In our implementation, after acceptance of the connection, the Server sends automatically a simple message "OK" to the connected Client
- After, Server waits for first message sent by Client and if ***diRcvPacketCount*** is › 0, it is able to answer.

Tab. 3.1-1 FB_TCP_Server : Input variables

	Type	Description
xEnable	BOOL	Enable FB, if TRUE this FB will be evaluated completely normally
iLocalPort	INT	Local Ethernet port assigned for the socket of the Server
diMaxDataSize	DINT	Max size of data to transmit,
ptSendBuffer	POINTER TO BYTE	Points to a buffer (ARRAY OF BYTE) where transmit (send) data should be stored
diNbBytesToSend	DINT	Specifies the number of data (bytes) to send

Tab.3.1-2 FB_TCP_Server : Output variables

	Type	Description
xEnabled	BOOL	TRUE if FB enabled and is evaluated completely
xSocketIsOpen	BOOL	TRUE if the socket of the Server is correctly open
diSocketHandle	DINT	The socket descriptor for TCP Server
diErrCode	DINT	Error code
iPhase	INT	Nb of internal step of FB execution (phase of the TCP connection)

Tab.3.1-3 FB_TCP_Server : IN_OUT variables

	Type	Description
xOpenSocket	BOOL	IF TRUE the Socket is open and TCP connection launched; ***If FALSE the socket is closed automatically***
xSendAnswer	BOOL	If TRUE the message with answer can be transmitted to the client. After execution, the FB will reset this variable
diRcvPacketCount	DINT	Number of received UDP telegrams
diNbRcvBytes	DINT	Number of the bytes (length) in the last received message

3.1.2 CoDeSys (Server) Function Block: FB_TCP_Read()

The Function Block ***FB_TCP_Read*** completes Data Transfer phase and gives a possibility to receive messages sent by TCP Client. When the input variable ***xExecute*** is set (TRUE), it starts listening of the opened communication port.

When the new message is received, it is read and the FB indicates the reception by setting the ***xReady*** output variable. Then, it gives information on the number of received bytes (***diNbRcvBytes***) and increments the variable ***diNbRcvPackets***.

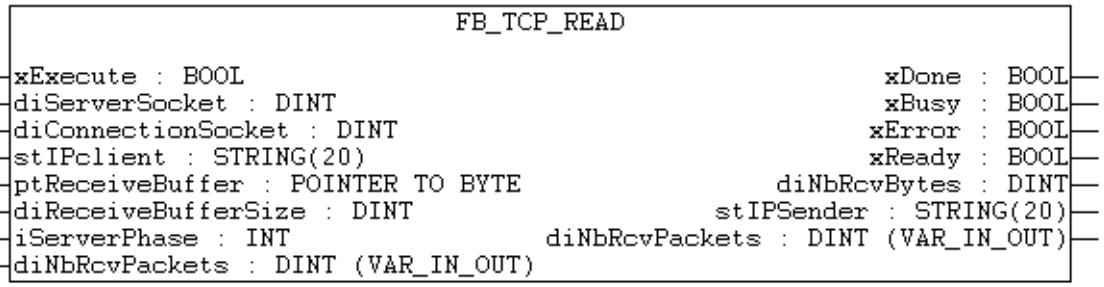

Fig.4 Function Block FB_TCP_Read

Tab. 3.1-4 FB_TCP_Read : Input variables

	Type	Description
xExecute	BOOL	Launch FB, if TRUE this FB waits for the message from connected client
iServerSocket	DINT	The socket descriptor returned by TCP Server
diConnectionSocket	DINT	The socket descriptor for connected Client, returned after connection acceptance
stIPclient	STRING(20)	Address IP of the connected TCP Client
ptReceiveBuffer	POINTER TO BYTE	Pointer to a buffer (ARRAY OF BYTE) of received data
diReceiveBufferSize	DINT	The length of the reception buffer (in bytes)
iServerPhase	INT	The phase (step) of the FB_TCP_Server used for synchronization of tasks and for connection control

Tab.3.1-5 FB_TCP_Read : Output variables

	Type	Description
xDone	BOOL	TRUE if FB enabled and data are received
xBusy	BOOL	TRUE if FB active and waits for the data from Client
xError	BOOL	TRUE in the case of reception error
xReady	BOOL	TRUE if FB data are correctly received and transferred to the buffer
diNbRcvBytes	DINT	Number of received bytes (message length)
stIPSender	STRING(20)	Address IP of the sender of received message (connected TCP Client)

Tab.3.1-6 FB_TCP_Read : IN_OUT variables

	Type	Description
diNbRcvPackets	DINT	Number of the received packets (messages); incremented byFB

3.1.3 CoDeSys (Server) Program: TCP_Listen_Accept()

In Data Transfer phase, FB_TCP_Server and FB_TCP_Read are able to communicate with remote Client. But before, this Client must be connected.

TCP Connections are identified by the socket identifiers at both ends; that is Server socket and Client socket. The reference of the Client Socket is automatically transmitted to the Server in the connection request.

To establish the connection with the Client, the Server must be in "Listen" state where it waits for the request to open the connection. When this request is arrived, it must be accepted by Server.
In order not to block the process, this phase of the Connection Establishment is carried out by two asynchronous tasks (see Fig.2).

The first task "***TCPServer***" initiates the establishment of the connection by calling SysSockListen() function and waits in "Listen" state that the Client's request will be accepted by Server. It waits for the valid descriptor of the socket for established connection (here IN_OUT variable: ***diConnectionHandle***).

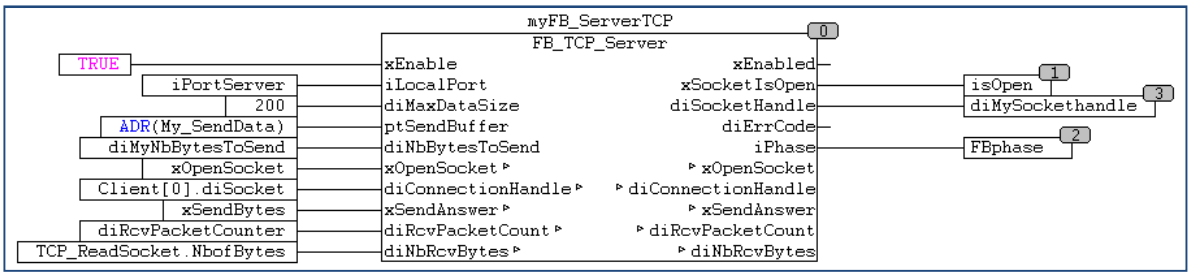

*Fig.5 Listing of the program **TCP_Server***

This descriptor is returned by the second task "***AcceptConnection***" which creates the new socket and accepts the connection request (the call of the function SysSosckAccept()). This task is assigned to the program ***TCP_Listen_Accept*** (see Fig.6) and the value of the descriptor is given by the global variable ***Client***.

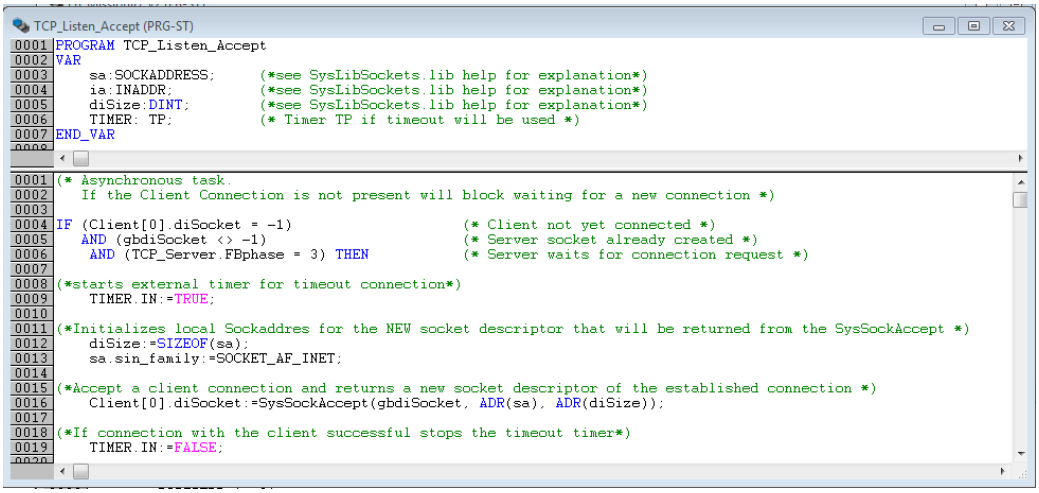

*Fig.6 Listing of the program **TCP_Listen_Accept***

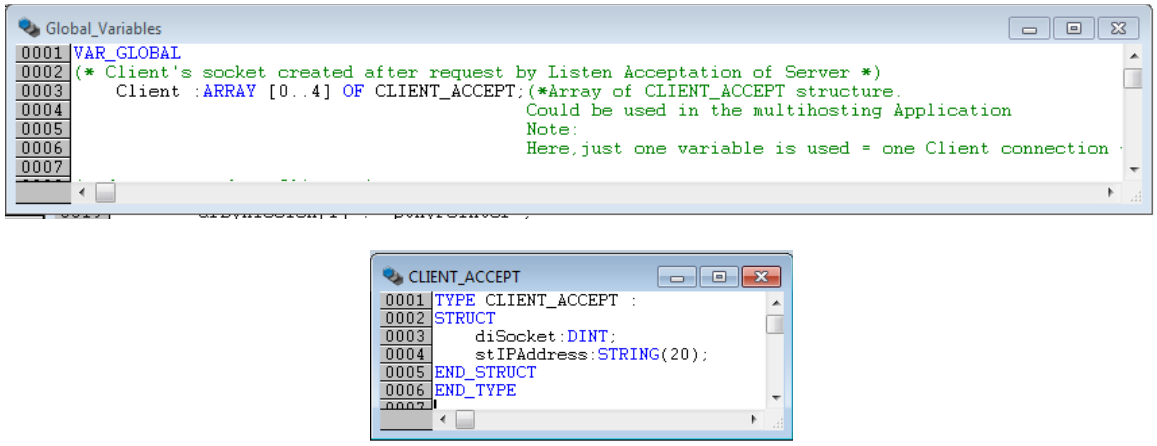

*Fig.7 Global Variable **"Client"** and Data Type **"CLIENT_ACCEPT"***

3.2 Functionality TCP Client

For the functionality TCP Client we can provide the solution also based on the distribution of TCP functions into separate tasks.

But, looking at the TCP state diagram ([2], [3], [5]), TCPClient is not obliged to go through the state "Listen" (passive open). After creating the socket it goes directly to the SYN_SEN state (active open) which represents waiting for a confirmation (acknowledgement) of the connection request after having sent this request to the Server.

Note:

Of course, before the Client attempts to connect with a server, the server must first to create the socket, bind to and listen at communication port open for the connection; this is called a passive open.
Once this passive open is established, a client may initiate a connection (active open) by sending connection request to the Server.

When both the Client and the Server acknowledge the connection, the "point-to-point" communication is established. But, as for the Server, data exchange functions Recv() and Send() must be implemented in two separate tasks.

3.2.1 CoDeSys Function Block: FB_TCP_Client()

The Function Block ***FB_TCP_Client*** provides some services of the establishment of the Ethernet dialogue with one TCP Server. These services are :

- The establishment of the connexion with TCP Server
- The sending of messages to the Server
- The (normal) termination of the connection

Note:

The receiving of the messages from Server is managed by Function Block FB_TCPclient_Read instanced in the separated task.

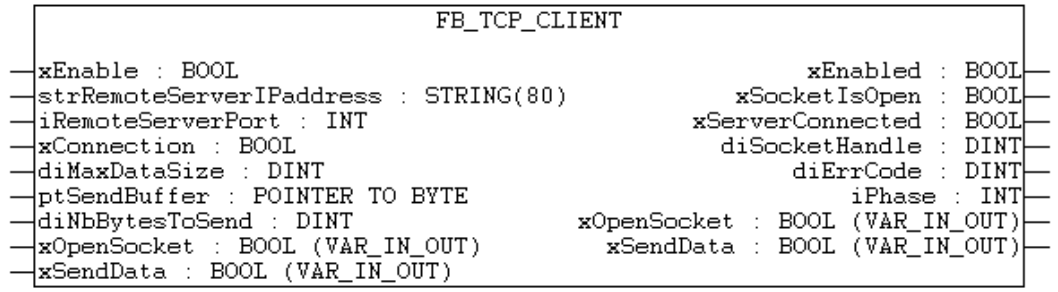

Fig.8 Function Block FB_TCP_Client

First, this FB creates his socket and starts the connection with the Server described by input variables ***stRemoteServerIPaddress*** and ***iRemoteServerPort***.

When the input variable ***xConnection*** is set, it sends the connection request (***SysSockConnect()***) in direction of the Server.

After acceptance of the connection by the Server, the output variable ***xServerConnected*** is set (TRUE) and FB goes to the phase of Data Transfer where it is ready to dialogue with the Server.

By setting the variable ***xSendData*** a message can be transmitted to the Server. After complete transmission, the variable ***xSendData*** will be reset automatically by Function Block.

Tab. 3.2-1 FB_TCP_Client : Input variables

	Type	Description
xEnable	BOOL	Enable FB, if TRUE this FB will be evaluated completely normally
strRemoteServerIPaddress	STRING	Address IP of the Server
iRemoteServerPort	INT	No of the communication port of the Server
xConnection	BOOL	Demand of the TCP connection
diMaxDataSize	DINT	Max size (bytes) of transmitted data (max length of messages)
ptSendBuffer	POINTER TO BYTE	Pointer to transmission (Send) buffer : Array of Bytes
diNbBytesToSend	DINT	Number of data (bytes) to send

Tab.3.2-2 FB_TCP_Client : Output variables

	Type	Description
xEnabled	BOOL	TRUE if FB enabled and is evaluated completely
xSocketIsOpen	BOOL	TRUE if the socket is correctly open
xServerConnected	BOOL	TRUE if Server is connected
diSocketHandle	DINT	The socket descriptor for open socket
diErrCode	DINT	Error code
iPhase	INT	Nb of internal step of FB execution (phase of the TCP connection)

Tab.3.2-3 FB_TCP_Client : IN_OUT variables

	Type	Description
xOpenSocket	BOOL	IF TRUE the Socket is open and TCP connection launched; ***If FALSE the socket is closed automatically***
xSendData	BOOL	If TRUE the message is transmitted to the Server. After execution, the FB will reset this variable

3.2.2 CoDeSys (Client) Function Block: FB_TCP_ClientRead()

The Function Block ***FB_TCP_ClientRead*** completes Data Transfer phase from TCP Client side and gives a possibility to receive messages sent by TCP Server. When the input variable ***xExecute*** is set (TRUE), it starts listening and waits for the data sent by Server.

When the new data are received, they are read and the FB indicates the reception by setting the ***xReady*** output variable. Then, it gives information on the number of received bytes (***diNbRcvBytes***) and increments the variable ***diNbRcvPackets***.

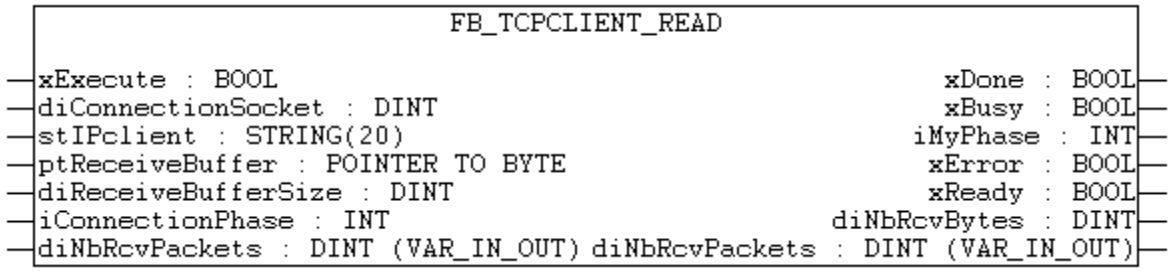

Fig.9 Function Block FB_TCP_ClientRead

Tab. 3.2-4 FB_TCP_ClientRead : Input variables

	Type	Description
xExecute	BOOL	If TRUE, FB is started and waits for the message from connected Server
diConnectionSocket	DINT	The Client socket descriptor
stIPclient	STRING(20)	Address IP of the connected TCP Client
ptReceiveBuffer	POINTER TO BYTE	Pointers to the buffer (ARRAY OF BYTE) of received data
diReceiveBufferSize	DINT	The length of the reception buffer (in bytes)
iConnectionPhase	INT	The phase (step) of the FB_TCP_Client used for synchronization of Data Transfer phase

Tab.3.2-5 FB_TCP_ClientRead : Output variables

	Type	Description
xDone	BOOL	TRUE if FB enabled and data are received
xBusy	BOOL	TRUE if FB active and waits for the data from Server
xError	BOOL	TRUE in the case of reception error
xReady	BOOL	TRUE if FB data are correctly received and transferred to the buffer
diNbRcvBytes	DINT	Number of received bytes (message length)

Tab.3.2-6 FB_TCP_ClientRead : IN_OUT variables

	Type	Description
diNbRcvPackets	DINT	Number of the received packets (messages); incremented byFB

4. Implementation in the PLC application.

Very often, in industrial applications, the TCP is merely used as a transport protocol which does not interpret the contents of the data. This is the job of the application protocol, as, for example, Modbus TCP which is widely used in industrial application.

And, very often, this is also the case of the specific protocols based on TCP approach as, for example, in our robotic application (Fig.10)

.

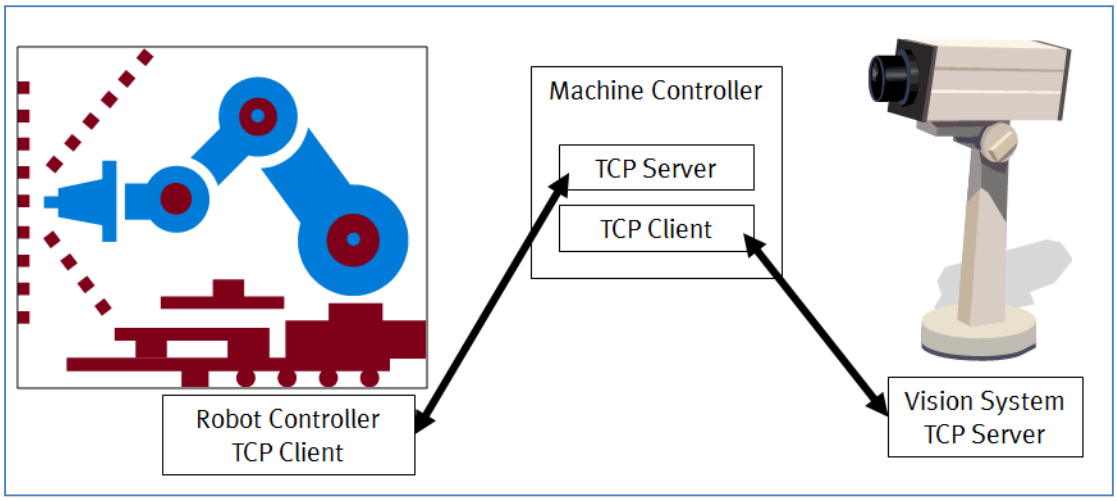

Fig.10 Application of TCP Client/Server function inside Machine Controller

In this application, the machine controller (PLC) exchanges data via TCP protocol:

- As TCP Server with the controller of the robots
- As TCP Client with controller of the camera (quality inspection)

The assembly robot sends identifiers and parameters of the needed part. Machine produces these parts and sends some data to the controller of the camera, the data used for the quality control of the mounted part

During assembly process, the robot picks the part, shows to camera and waits the results of the quality control. The camera gives binary information on the part (bad/good) directly to the robot.

For the both, TCP connections are implemented as transport protocol to encapsulate two proprietary application protocols.

NOTE:
For the complete source code of the CoDeSys project and described POU's: please contact the author of this notice.

5. Discussion

TCP is a complex protocol. While significant enhancements in the hardware have been made and proposed over the years, the most basic principles of TCP have not changed significantly since its first specification.

However, the direct implementation of the basic functionality Client/Server in the PLC application, under "not informatics" language, is not easy.

For many industrial applications, the complexity of TCP can be a problem. E.g. TCP could be not appropriate for "hard" real time application where the network booting or huge numbers of participants (clients/servers) influence directly the throughput and transmission speed on the network.

And, unlike the UDP protocol, TCP does not support multicasting and broadcasting.

Of course, if applications do not require the reliability of a TCP connection or need the multicasting, they may instead use the connectionless User Datagram Protocol (UDP).

But the primary function of TCP is to ensure that all packets of data are transferred correctly. TCP is reliable and it has been optimized for wired networks. So it could be used for specific tasks which do not need “hard” real time behaviour but reliable data exchange and/or for applications where multicasting/broadcasting traffic is forbidden.

The configuration of the remote equipment (transfer of parameters), the exchange of large packets of data (e.g. between robot and vision system, RFID management), the transfer (upload/download) of programs could be example of tasks where TCP can be useful.

In our application we used the concept proposed by CANLogix1 in their application note [6]. Of course, we adapted this approach for our PLC (CPX-CEC Festo) and for our application under CoDeSys v.2.3.

Of course, there are a lot of ways to ameliorate our approach, especially in the management of the connection termination and in error treating.

But for our application, it was enough. Nevertheless we remain open to any suggestions for improvement and optimization.

NOTE:
For the complete source code of the CoDeSys project and described POU’s: please contact the author of this notice.

6. References:

[1] W.Gomolka, “CoDeSys and Ethernet communication. Concept of Sockets and basic Function Blocks for the communication over Ethernet. Part 1: UPD ClientServer”; Researchgate.net; publication 262198350; 2014

[2] “Transmission Control Protocol (TCP) specification”; RFC 793

[3] “Wiki: Transmission Control Protocol”; http://en.wikipedia.org/wiki/Transmission_Control_Protocol

[4] “Client server applications on the 758-870 using SysLibSocket and WagoLibEthernet_01 library”; WAGO Application note A113501; 2005

[5] J.Treurniet and J.H.Lefebvre ; “A Finite State Machine Model of TCP Connections in the Transport Layer”; TECHNICAL MEMORANDUM ; DRDC Ottawa TM 2003-139; November 2003

[6] “TCP/IP Server—Client example with CANLogix-1”; Application Note APN00004, Doc Version: 1.0 Date: May 23, 2008

... and a lot of good articles and examples on the Net ...

Some interesting links:

- Oscat library
http://www.oscat.de/downloadmanager/viewcategory/4-oscatnetwork.html

- 3S forum
http://forum.codesys.com/viewtopic.php?f=1&t=1981
ftp://ftp2.3s-software.com/pub/Examples/Projects/CoDeSysV2.3/Communication/TcpIp/

- presentation
http://lionne.cnam.fr/Cours/CPI/TCP_IP.pdf