Protocol API

TCP/IP

Packet Interface

V2.4

Hilscher Gesellschaft für Systemautomation mbH

www.hilscher.com

DOC050201API14EN | Revision 14 | English | 2017-01 | Released | Public

Table of contents

1 Introduction

1.1 About this document

This manual describes the application interface of the TCP/IP and UDP/IP protocol stack. Use this manual to support and guide you through the integration process of the given stack into your own application.

This stack was developed based upon Hilscher's Task Layer Reference Programming Model. This programming model is a description of how to develop a task in general, which is a convention defining a combination of appropriate functions belonging to the same task. Furthermore, it defines how different tasks have to communicate with each other in order to exchange their data. The Reference Model is commonly used by all developers at Hilscher and shall be used by you as well when writing your application task on top of the stack.

1.2 List of revisions

Rev	Date	Name	Chapter	Revision
13	2015-09-10			TCP/IP stack version V2.2.x.x
		KM	4.4.1	ACD indication parameters updated.
		RG	4.2.1	Note added
14	2017-01-25	AM	3	Section *Application implementation requirements* added.
		RH	4.1.5	Host name and domain name added.
		KM	4.2.11	Example revised.
		HH	4.4.1	Description for packet field "ulReason" extended.
			5.2	Section *TCP/IP startup parameters* updated.
			6	Section *Status/error codes*: Tables splitted.

Table 1: List of revisions

1.3 Functional overview

The stack has been written in order to meet the corresponding RFCs (Request for Comments). See section *Specifications* at page *6* for further information. The main functionality from application view is:

- TCP and
- UDP communication.

1.4 System requirements

This software package has following system requirements to its environment:

- netX-Chip as CPU hardware platform
- operating system for task scheduling required
- operating system independency, rcX or Windows CE are implemented as a reference
- Stack portable to any other processor technology

1.5 Intended audience

This manual is suitable for software developers with the following background:

- Knowledge of the programming language C
- Knowledge of the use of the real-time operating system rcX
- Knowledge of the TCP/IP protocol suite

1.6 Specifications

The data below applies to TCP/IP stack version V2.4.

1.6.1 Supported protocols

- ARP - Address Resolution Protocol (RFC 826)
- IP - Internet Protocol (RFC 791)
- ICMP - Internet Control Message Protocol (RFC 792)
- TCP - Transmission Control Protocol (RFC 793, RFC 896)
- IGMPv2 - Internet Group Management Protocol, Version 2 (RFC 2236)
- UDP - User Datagram Protocol (RFC 768)
- BOOTP - Bootstrap Protocol (RFC 951, RFC 1542, RFC 2132)
- DHCP - Dynamic Host Configuration Protocol (RFC 2131, RFC 2132)
- Ethernet frame types: Ethernet II (RFC 894), IEEE 802.3 receive only (RFC 1042)

1.6.2 Technical data

Feature	Value
Number of sockets	Max. 128 (Startup parameter)
ARP cache size	Max. 512 entries (Startup parameter)
ARP timeout	600 seconds by default (Startup parameter 60 … 3600 seconds)
Route cache size	32 entries
Route timeout	900 seconds
IP multicast groups	Receive: 64 Send: not limited
IP multicast groups	Receive: 64 Send: not limited
IP datagram size	Up to 1500 bytes

Table 2: Technical data – TCP/IP

1.6.3 Limitations

- IP fragmentation not supported
- TCP urgent data not supported
- TCP port 0 not supported
- UDP port 67 reserved for BOOTP and DHCP
- UDP port 25383 reserved for Hilscher NetIdent Protocol
- IP multicast receive always enabled from Ethernet Device Driver (EDD)

1.7 Terms, abbreviations and definitions

Term	Description
AP (-task)	Application (-task) on top of the stack
ARP	Address Resolution Protocol
BOOTP	Bootstrap Protocol
DHCP	Dynamic Host Configuration Protocol
EDD	Ethernet Device Driver
ICMP	Internet Control Message Protocol
IP	Internet Protocol
MAC (address)	Media Access Control (address) = Ethernet address
MSS	Maximum segment size (of TCP data), normally = 1460 byte on Ethernet (Maximum); *MSS = MTU - sizeof(IP header) - sizeof (TCP header) = 1500 –20 –20 = 1460*
MTU	Maximum Transmission Unit, normally 1500 byte = Data part of Ethernet frame
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

Table 3: Terms, Abbreviations and Definitions

All variables, parameters, and data used in this manual have the LSB/MSB (“Intel”) data format. This corresponds to the convention of the Microsoft C Compiler.

All IP addresses in this document have host byte order.

1.8 References to documents

This document refers to the following documents:

[1] Hilscher Gesellschaft für Systemautomation mbH: rcX - Realtime Communication System for netX - Kernel API Function Reference, Revision 7, English, 2013.

[2] Hilscher Gesellschaft für Systemautomation mbH: Dual-Port Memory Interface Manual, netX based products, Revision 12, English, 2012.

Table 4: References to Documents

1.9 Legal notes

1.9.1 Copyright

1.9.2 Important notes

The user manual, accompanying texts and the documentation were created for the use of the products by qualified experts, however, errors cannot be ruled out. For this reason, no guarantee can be made and neither juristic responsibility for erroneous information nor any liability can be assumed. Descriptions, accompanying texts and documentation included in the user manual do not present a guarantee nor any information about proper use as stipulated in the contract or a warranted feature. It cannot be ruled out that the user manual, the accompanying texts and the documentation do not correspond exactly to the described features, standards or other data of the delivered product. No warranty or guarantee regarding the correctness or accuracy of the information is assumed.

We reserve the right to change our products and their specification as well as related user manuals, accompanying texts and documentation at all times and without advance notice, without obligation to report the change. Changes will be included in future manuals and do not constitute any obligations. There is no entitlement to revisions of delivered documents. The manual delivered with the product applies.

Hilscher Gesellschaft für Systemautomation mbH is not liable under any circumstances for direct, indirect, incidental or follow-on damage or loss of earnings resulting from the use of the information contained in this publication.

1.9.3 Exclusion of liability

The software was produced and tested with utmost care by Hilscher Gesellschaft für Systemautomation mbH and is made available as is. No warranty can be assumed for the performance and flawlessness of the software for all usage conditions and cases and for the results produced when utilized by the user. Liability for any damages that may result from the use of the hardware or software or related documents, is limited to cases of intent or grossly negligent violation of significant contractual obligations. Indemnity claims for the violation of significant contractual obligations are limited to damages that are foreseeable and typical for this type of contract.

It is strictly prohibited to use the software in the following areas:

- for military purposes or in weapon systems;
- for the design, construction, maintenance or operation of nuclear facilities;
- in air traffic control systems, air traffic or air traffic communication systems;
- in life support systems;
- in systems in which failures in the software could lead to personal injury or injuries leading to death.

We inform you that the software was not developed for use in dangerous environments requiring fail-proof control mechanisms. Use of the software in such an environment occurs at your own risk. No liability is assumed for damages or losses due to unauthorized use.

1.9.4 Export

The delivered product (including the technical data) is subject to export or import laws as well as the associated regulations of different counters, in particular those of Germany and the USA. The software may not be exported to countries where this is prohibited by the United States Export Administration Act and its additional provisions. You are obligated to comply with the regulations at your personal responsibility. We wish to inform you that you may require permission from state authorities to export, re-export or import the product.

2 Getting started

2.1 Sockets-based programming model

The TCP_UDP-task utilizes a sockets-based handling of TCP and UDP communication. The socket model implemented in the stack is loosely related to the well-known BSD sockets or Winsock programming model. However, only the basic features known from these models are implemented in order to simplify the communication and to adapt to the restrictions of an embedded device. The main difference is the command-based communication provided by the TCP/IP stack as opposed to the function calls implemented by BSD sockets or Winsock libraries. The function calls are represented by request command and confirmation command pairs in the TCP_UDP-task.

The following sections describe how to make use of the socket communication and how to handle the commands involved. Chapter *The application* (page *24*) describe all commands in detail.

2.1.1 TCP client example

The TCP client example illustrates the command handling required by a simple TCP client application. This application establishes a TCP connection to a remote server, and it sends some request data. The client then reads the response data supplied by the server followed by a shutdown of the connection.

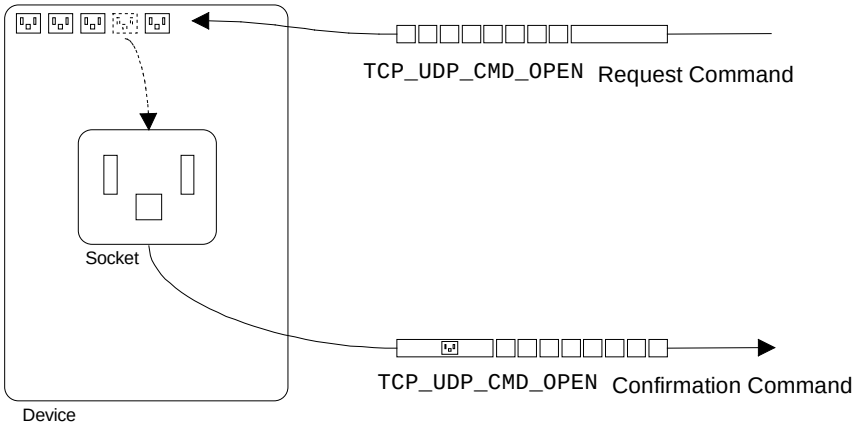

Figure 1: TCP client example - TCPIP_TCP_UDP_CMD_OPEN_REQ/CNF

The first step of any socket-based communication is to open a socket of the desired protocol type. Therefor a TCPIP_TCP_UDP_CMD_OPEN request command should be sent to the stack. The stack will return a confirmation command containing the **handle** of the socket just opened.

When the confirmation command is returned the TCP client application can proceed to the next step which establishes a connection to the TCP server. This will be accomplished by sending a TCPIP_TCP_CMD_CONNECT command to the stack.

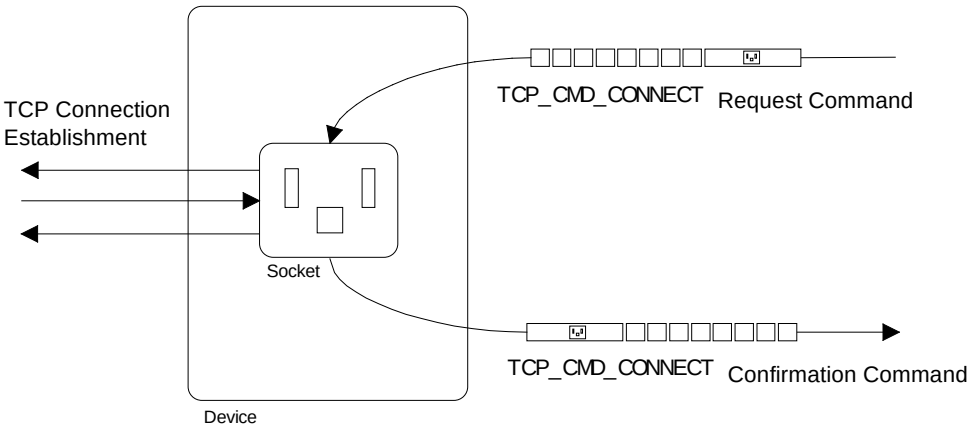

Figure 2: TCP client example - TCPIP_TCP_CMD_CONNECT_REQ/CNF

The device then contacts the server owning the IP address given in the TCPIP_TCP_CMD_CONNECT request command. If the connection could be established a confirmation command reporting success will be returned to the application. In case the IP address cannot be found or the server refuses the connection the confirmation command will contain an error code indicating the reason of failure.

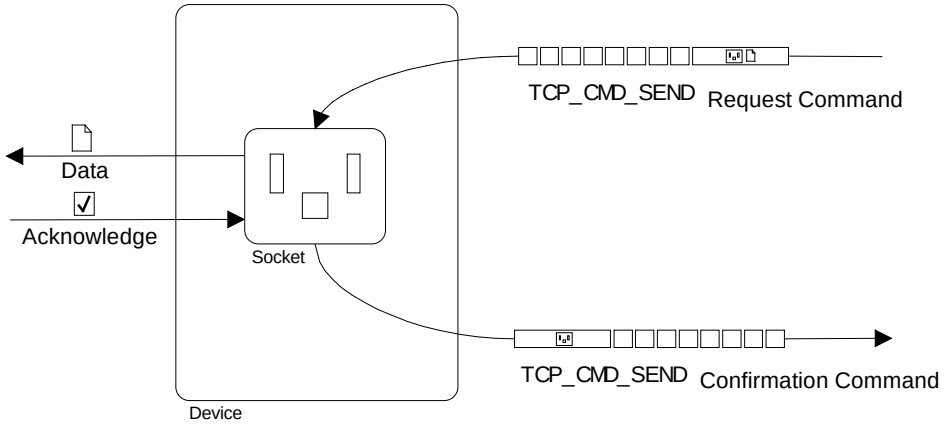

Figure 3: TCP client example - TCPIP_TCP_CMD_SEND_REQ/CNF

Once the connection is established the application can then start sending its request data using a TCPIP_TCP_CMD_SEND request command. The TCP/IP stack forwards the data to the server, and waits for the server's TCP/IP stack to acknowledge it. When the acknowledgement arrives the stack returns a TCPIP_TCP_CMD_SEND confirmation command to the application.

Remark: The application must not wait for the TCPIP_TCP_CMD_SEND confirmation command to send the next TCPIP_TCP_CMD_SEND request command.

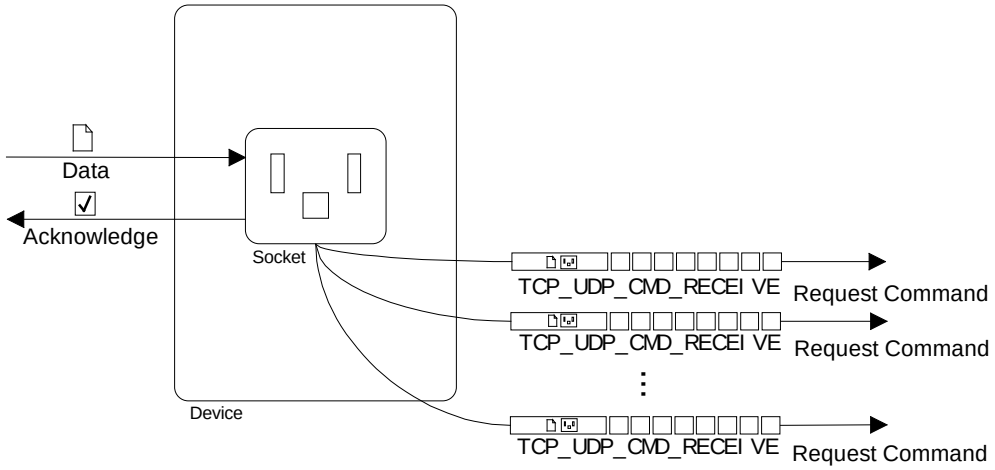

Figure 4: TCP client example - TCPIP_TCP_UDP_CMD_RECEIVE_IND

When the server has finished processing, it sends its response data through the TCP connection. The stack receives the data from the network, and sends it to the application using TCPIP_TCP_UDP_CMD_RECEIVE indication commands. There will be as many TCPIP_TCP_UDP_CMD_RECEIVE indication commands as required to transport the server's data.

Please note, that the stack sends TCPIP_TCP_UDP_CMD_RECEIVE indication commands automatically, whenever data is received from the network. There is no explicit receive request command, because the application is assumed to be ready to handle received data once it has established a connection. Thus, the application should check for available indication commands from the stack on a regular basis.

Once the application has received all response data from the server it terminates the TCP connection by sending a TCPIP_TCP_UDP_CMD_CLOSE request command.

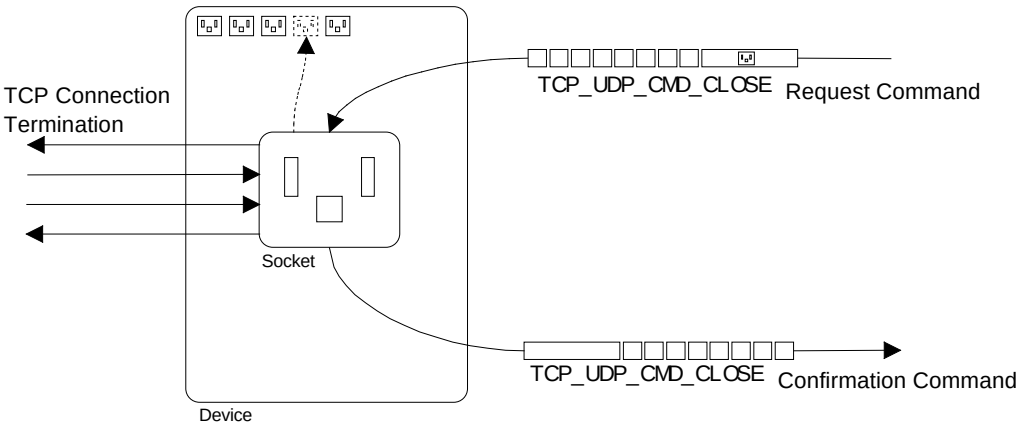

Figure 5: TCP client example - TCPIP_TCP_UDP_CMD_CLOSE_REQ/CNF

When the connection is terminated the stack closes the socket, and a TCPIP_TCP_UDP_CMD_CLOSE confirmation command is returned to the application.

2.1.2 TCP server example

The example shown in this section shall demonstrate the operation of a very simple TCP server application. The server waits for an incoming request from the network. When the request is received some response data is sent as a reply and the connection is closed upon completion.

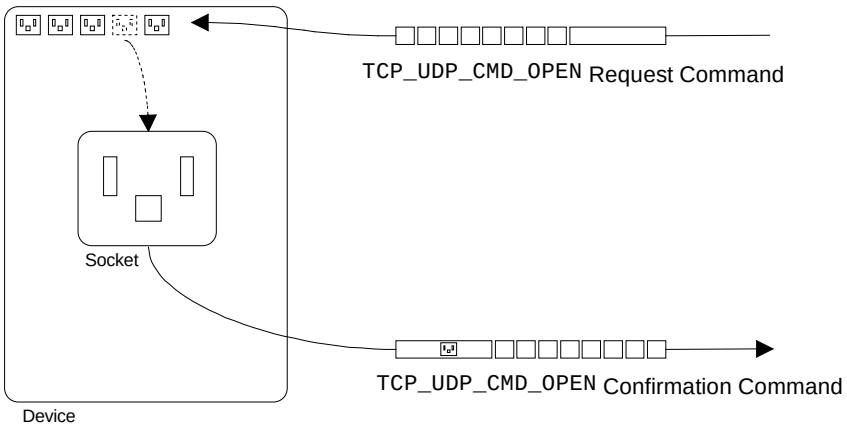

Figure 6: TCP server example - TCPIP_TCP_UDP_CMD_OPEN_REQ/CNF

A socket must be opened before any TCP communication on the network can start. The server application sends a TCPIP_TCP_UDP_CMD_OPEN request command to the stack, and waits for the corresponding confirmation command which returns a **socket handle**.

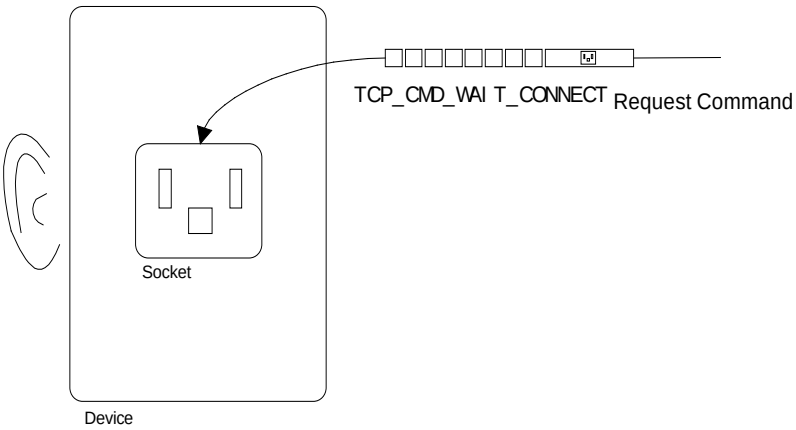

Figure 7: TCP server example - TCPIP_TCP_CMD_WAIT_CONNECT_REQ

When a socket handle can be obtained, the server application puts this socket into listening state by sending a TCPIP_TCP_CMD_WAIT_CONNECT request command. The stack defers the confirmation command to this request command as long as the socket is waiting for an incoming connection.

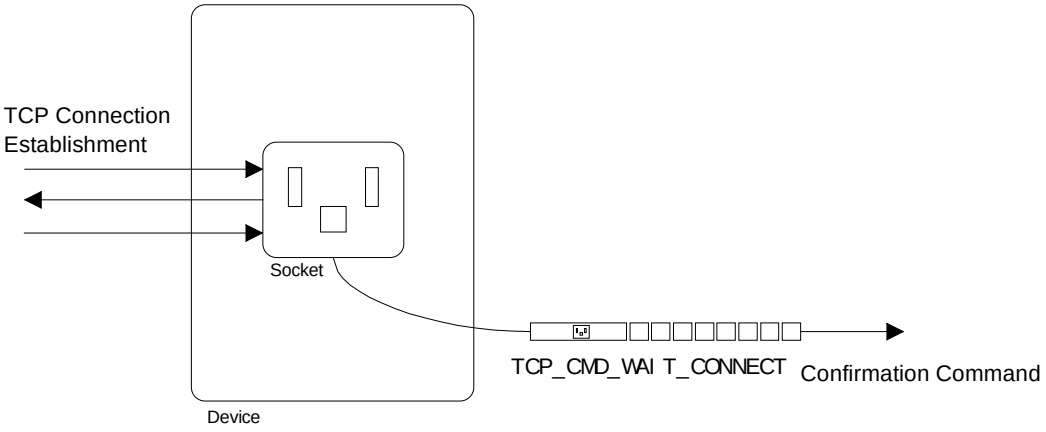

Figure 8: TCP server example - TCPIP_TCP_CMD_WAIT_CONNECT_CNF

Once a connection request is detected, it is processed, and the socket is no longer listening. The result of the connection establishment is returned in the TCPIP_TCP_CMD_WAIT_CONNECT confirmation command then. If successful, the socket will be connected to the client that initiated the connection.

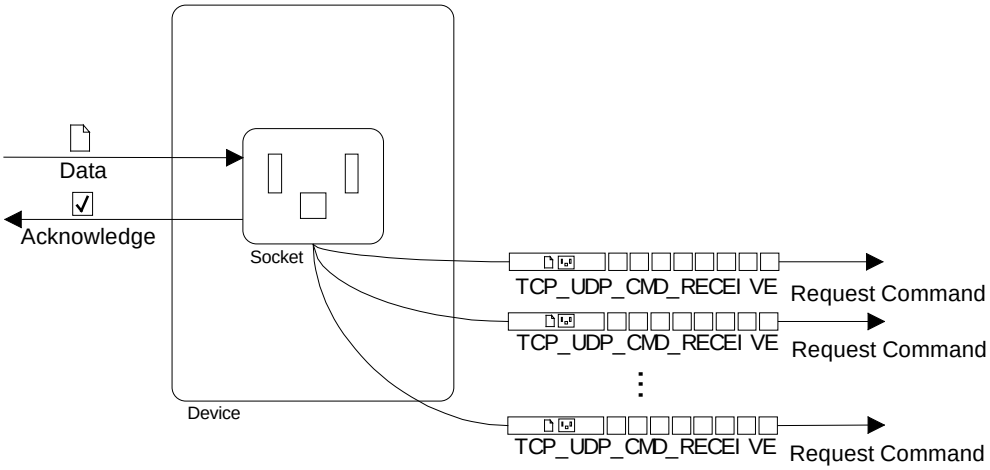

Figure 9: TCP server example - TCPIP_TCP_UDP_CMD_RECEIVE_IND

The server then awaits the client's request data, which is transferred to the server application using TCPIP_TCP_UDP_CMD_RECEIVE indication commands.

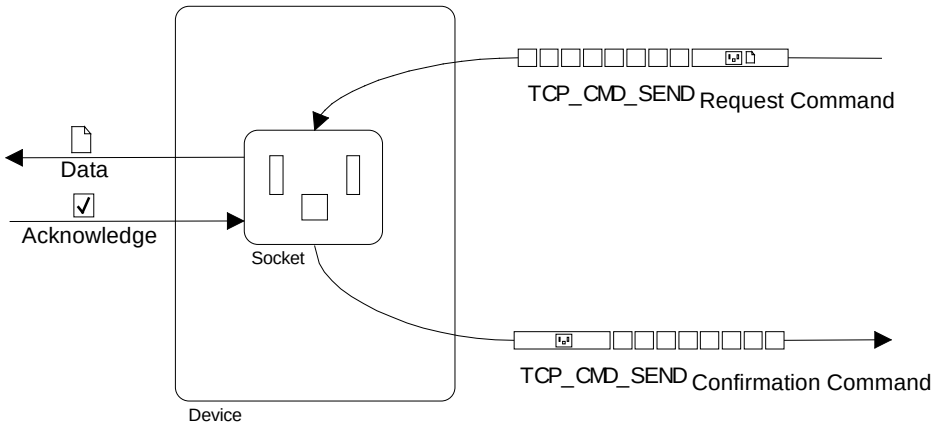

Figure 10: TCP server example - TCPIP_TCP_CMD_SEND_REQ/CNF

After processing the request the server sends response data back to the client by means of TCPIP_TCP_CMD_SEND commands.

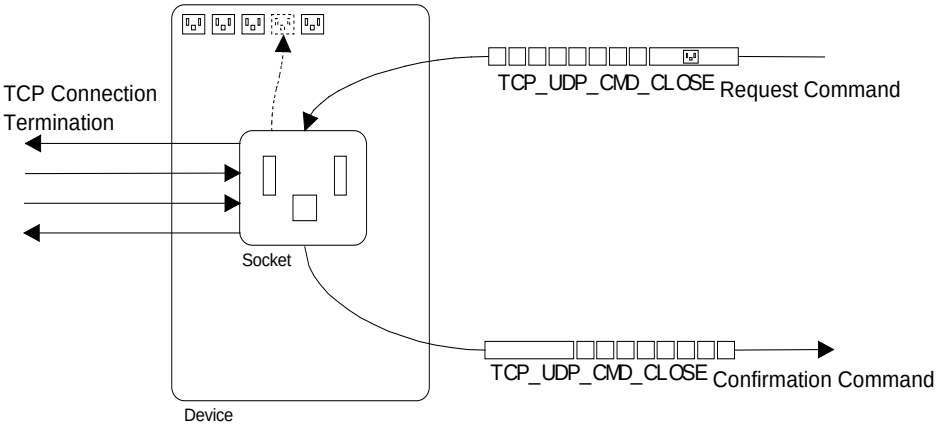

Figure 11: TCP server example - TCPIP_TCP_UDP_CMD_CLOSE_REQ/CNF

Once the data is transferred the server sends a TCPIP_TCP_UDP_CMD_CLOSE request command to the stack in order to terminate the TCP connection and to close the socket.

2.1.3 UDP communication example

UDP is a very simple datagram-oriented protocol layer, which doesn't guarantee data to be delivered reliably. The concept of a connection does not exist within UDP, which leads to a simplified communication model compared to TCP. As a result the number of commands that need to be exchanged between application and stack is reduced, too.

The example shown below illustrates which commands need to be handled by an application that communicates using UDP.

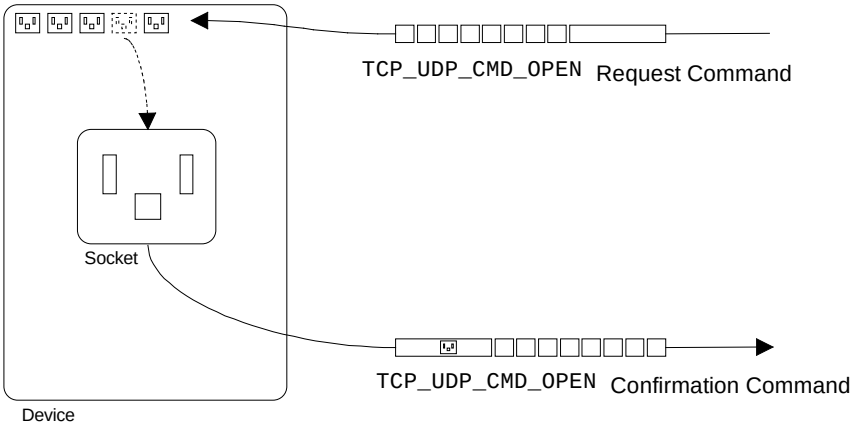

Figure 12: UDP communication example - TCPIP_TCP_UDP_CMD_OPEN_REQ/CNF

Most simply, only a UDP socket needs to be opened in order to enable an application to receive UDP data. The application sends a TCPIP_TCP_UDP_CMD_OPEN request command to the stack, and waits for the corresponding confirmation command to be returned.

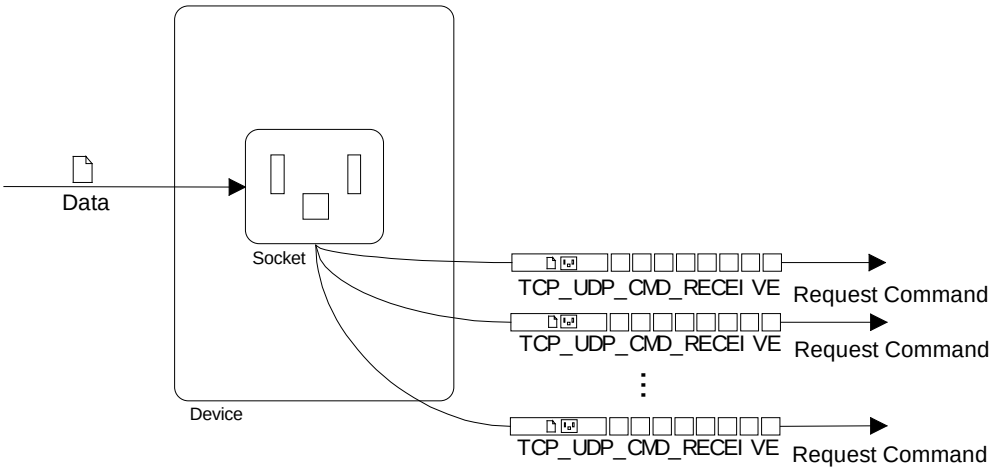

Figure 13: UDP communication example - TCPIP_TCP_UDP_CMD_RECEIVE_IND

Once the socket is opened all incoming UDP data for this socket will be sent to the application by means of TCPIP_TCP_UDP_CMD_RECEIVE commands.

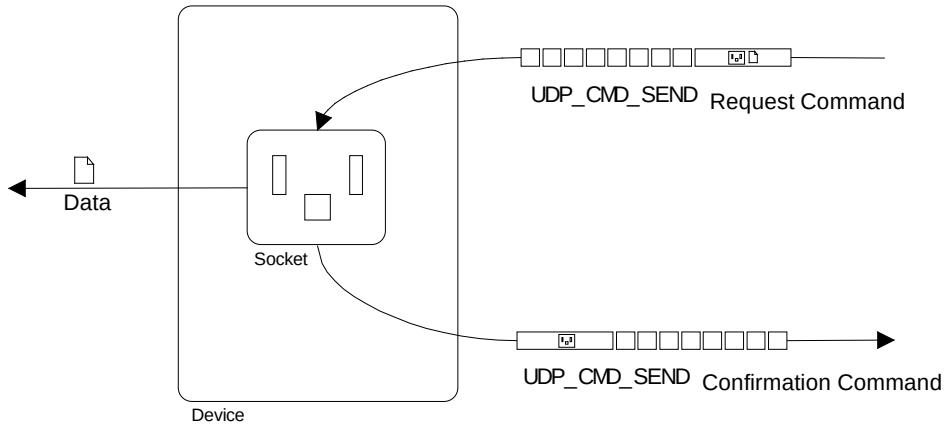

Figure 14: UDP communication example - TCPIP_UDP_CMD_SEND_REQ/CNF

The application processes the data, and sends its response data using TCPIP_UDP_CMD_SEND commands.

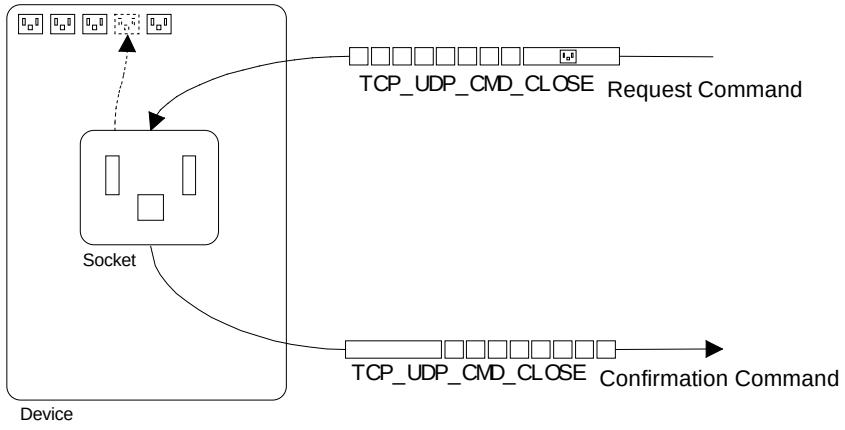

Figure 15: UDP communication example - TCPIP_TCP_UDP_CMD_CLOSE_REQ/CNF

When the application wants to stop UDP communication, it sends a TCPIP_TCP_UDP_CMD_CLOSE request command in order to close the corresponding socket.

2.2 Configuration

The TCP/IP stack needs basic configuration data to be able to start. Normally, this configuration data will be read from a database (DBM file) residing in the device's flash memory.

Alternatively, it is possible to provide the configuration data dynamically by the AP-task. Configuration data sent to the TCP/IP stack this way will have precedence over the data read from flash, but will not be stored in flash memory. Hence, the dynamic configuration procedure must be executed again after each reset or power cycle of the stack.

It is possible to send configuration command to the TCP/IP stack (TCP_UDP task). The commands are explained in detail in section *The application* starting at page *24*.

2.3 Start-up of the TCP/IP stack

After power on or reset the stack performs a start-up initialization. During this phase several steps are taken to bring the stack from uninitialized state to operation.

First, the hardware will be self-tested and the internal operating system will be started.

During the second step the initialization of the TCP/IP stack will be completed. Initially, the TCP_UDP-task will report an error code `TLR_E_TCP_TASK_F_NOT_INITIALIZED` (diagnostic code `TLR_DIAG_E_TCP_TASK_F_NOT_INITIALIZED`), which indicates that the task is still initializing. This error (diagnostic) code will change to `TLR_S_OK` (`TLR_DIAG_STA_OK`) once the task has finished the initialization successfully. In case of an initialization error an error (diagnostic) code which is different from `TLR_E_TCP_TASK_F_NOT_INITIALIZED` (`TLR_DIAG_E_TCP_TASK_F_NOT_INITIALIZED`) will be issued.

The TCP/IP stack (task) initialization can take up to 209 seconds dependent on the current configuration. The actual time is mainly determined by the DHCP and BOOTP parameters. The DHCP configuration mechanism requires up to 136 seconds and BOOTP requires up to 68 seconds to decide whether it succeeded or failed. If the IP address is configured manually only the check for duplicate IP address assignment will be performed which takes about 3 seconds.

According to the fallback procedure described in section *Protocol parameters* (on page 24) all three IP address configuration mechanisms can be combined. Please look at the table below for a list of all combinations.

IP Configuration Mechanism			Maximum Time Required (seconds)
DHCP	**BOOTP**	**Manually**	
		X	3
	x		68
	x	X	71
x			136
x		X	139
x	x		204
x	x	X	209

Table 5: Start-up of the TCP/IP stack

3 Application implementation requirements

If the TCP/IP packet interface is used in an application, the application must follow several rules, as the IP protocol is a core component of many protocols. A wrong behavior of the application can cause unexpected behaviors of the protocol stack. The following sections describes the required behaviour of an application.

Every socket opened successfully by an application must be closed explicitly by the application. The TCP/IP protocol will not close any socket implicitly under any condition. If the application does not obey this rule, the TCP/IP protocol will run out of socket resources at some time. This will also affect the industrial communication protocol.

3.1 UDP sockets

If UDP sockets are used by the application, special care must be taken for the use case of an IP address change within the protocol stack. The Hilscher TCP/IP protocol delays the IP address change until all sockets have been closed. Therefor the application must close the socket when an IP address change is required. If some sockets are not closed, the TCP/IP protocol will force the IP address change after a certain timeout (30 seconds by default). All unclosed sockets will be moved into an disfunctional state and must be closed later.

A minimal statemachine has to be implemented in the application in order to properly interface with the TCP/IP protocol as shown below. The following events are used in this statemachine:

Name	Description
CLOSE_SOCKET	The application requests to close the socket.
OPEN_SOCKET	The application request to open a socket.
TCP_UDP_CLOSE_CNF	The application received a TCPIP_TCP_UDP_CMD_CLOSE_CNF packet from TCP/IP protocol.
TCP_UDP_OPEN_CNF	The application received a TCPIP_TCP_UDP_CMD_OPEN_CNF packet from TCP/IP protocol.
TCP_UDP_SHUTDOWN_IND	The application received a TCPIP_TCP_UDP_CMD_SHUTDOWN_IND packet from TCP/IP protocol.
TIMEOUT	The retry timer timed out.

Table 6 Events used in UDP socket statemachine

The following actions are used in the statemachine:

Name	Description
send_TCP_UDP_OPEN_REQ	The application sends a TCPIP_TCP_UDP_CMD_OPEN_REQ packet to TCP/IP protocol
send_TCP_UDP_CLOSE_REQ	The application sends a TCPIP_TCP_UDP_CMD_CLOSE_REQ packet to TCP/IP protocol
startTimer()	The application starts an internal timer associated with the socket. Timeout should be greater than 10 ms.
stopTimer()	The application stops the internal timer associated with the socket.

Table 7 Actions used in UDP socket statemachine

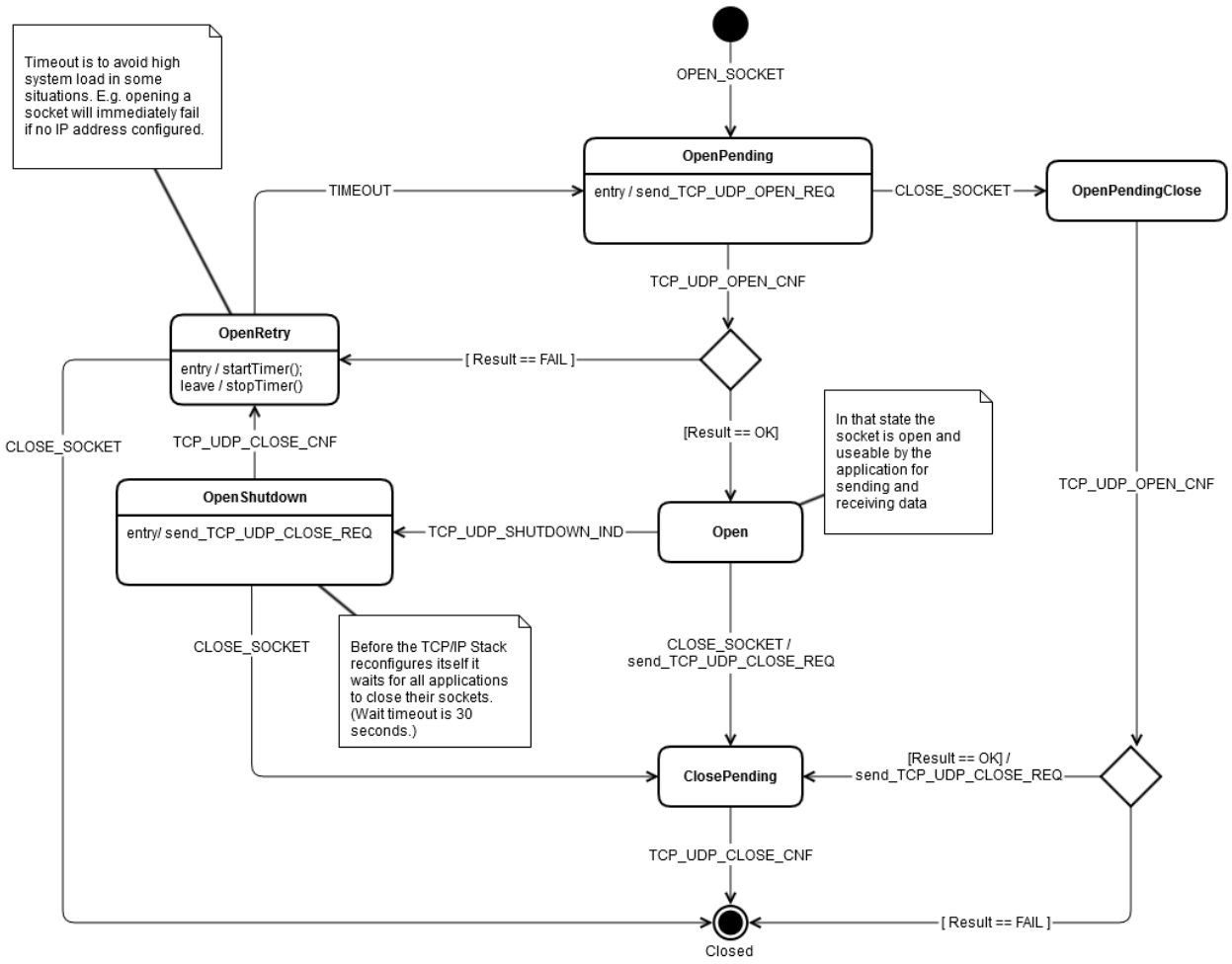

Figure 16 Application UDP socket statemachine

3.2 TCP sockets

Two different statemachines are required acording to the TCP protocol. The listen socket statemachine organizes the incomming connections on a listen socket, while the connection socket statemachine organizes a connection socket. On server side, one instance of the listen socket statemachine and multiple instances of the connection socket statemachines are required. On the client side one instance of the connection socket statemachine is required. The statemachines are shown below. Please note, that the connection socket statemachine and listen socket statemachine have a special interaction when a new incomming connection is to be accepted.

The following events are used by these statemachines:

Name	Description
CLOSE_SOCKET	The application requests to close the socket.
OPEN_SOCKET	The application requests to open a socket.
TIMEOUT	The retry timer timed out.
CLIENT_CLOSED	A connection socket previously created was closed (List socket only).
CREATE_CLIENT	A new inccoming connection was accepted (connection socket only).
TCP_UDP_CLOSE_CNF	The application received a TCPIP_TCP_UDP_CMD_CLOSE_CNF packet from the TCP/IP protocol.
TCP_UDP_OPEN_CNF	The application received a TCPIP_TCP_UDP_CMD_OPEN_CNF packet from the TCP/IP protocol.
TCP_UDP_SHUTDOWN_IND	The application received a TCPIP_TCP_UDP_CMD_SHUTDOWN_IND packet from the TCP/IP protocol.
TCP_WAIT_CONNECT_CNF	The application received a TCPIP_TCP_CMD_WAIT_CONNECT_CNF packet from the TCP/IP protocol (Listen socket only).
TCP_CONNECT_CNF	The application received a TCPIP_TCP_CMD_CONNECT_CNF packet from the TCP/IP protocol (Connection socket only).

Table 8 Events used in TCP socket statemachines

The following actions are used in the statemachines and are to be implemented in the application:

Name	Description
send_TCP_UDP_OPEN_REQ	The application sends a TCPIP_TCP_UDP_CMD_OPEN_REQ packet to the TCP/IP protocol.
send_TCP_UDP_CLOSE_REQ	The application sends a TCPIP_TCP_UDP_CMD_CLOSE_REQ packet to the TCP/IP protocol.
send_TCP_WAIT_CONNECT_REQ	The application sends a TCPIP_TCP_CMD_WAIT_CONNECT_REQ packet to the TCP/IP protocol (Listen socket only).
send_TCP_CONNECT_REQ	The application sends a TCPIP_TCP_CMD_CONNECT_REQ packet to the TCP/IP protocol (Connection socket only).
startTimer()	The application starts an internal timer associated with the socket. Timeout hast to be greater than 10 ms.
stopTimer()	The application stops the internal timer associated with the socket.
newClient()	The application creates a new connection socket using the initial event CREATE_CLIENT.

Table 9 Actions used in TCP socket statemachines

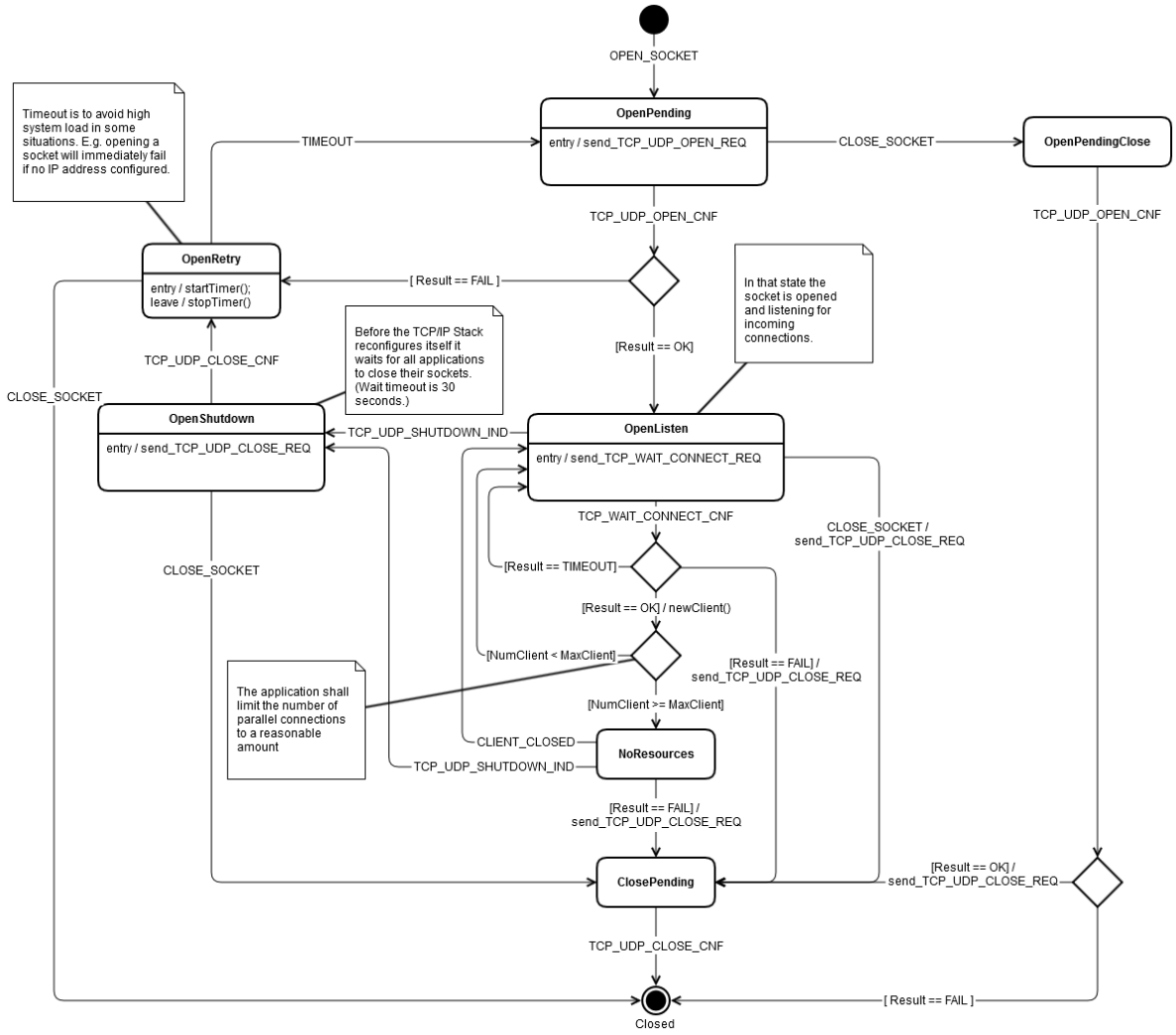

Figure 17 TCP listen socket statemachine

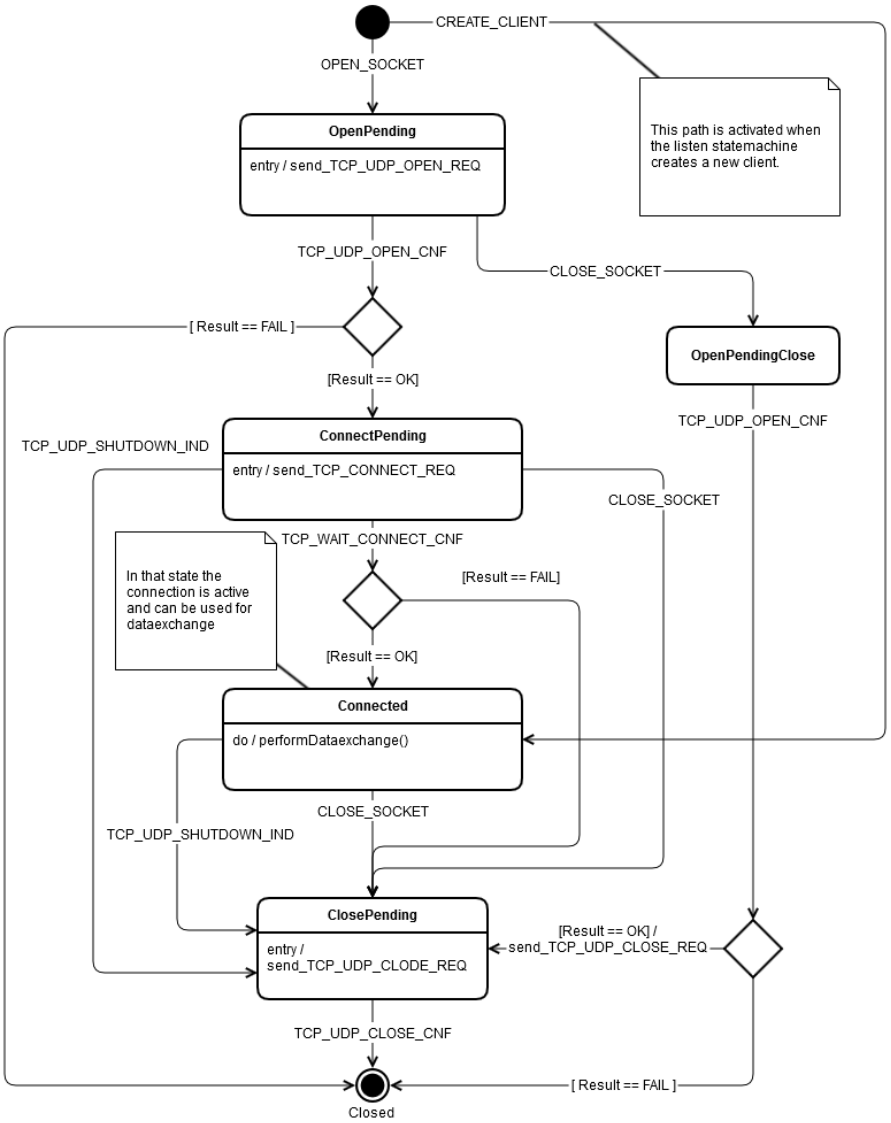

Figure 18 TCP connection socket statemachine

4 The application interface

This chapter describes the application interface of the TCP/IP stack.

4.1 Configuration

Normally the configuration is stored into the non-volatile flash memory. The TCP/IP stack reads this data block during its start-up initialization and performs consistency checks.

The application can add, change or delete parameters during run-time. These parameters will be kept in the RAM area of the device. For this purpose, the command `TCPIP_IP_CMD_SET_CONFIG_REQ` has to be used.

Note: Parameter sets stored in the RAM area will get lost if the device is powered down or if a restart is performed.

4.1.1 Protocol parameters

The parameter name (for example `ulFlags`) corresponds direct with the ***Packet structure*** respectively ***Packet description*** fields in section *The application* (starting on page *24*).

Parameter `ulFlags`

The `ulFlags` parameter holds bit-oriented flags according to the following table:

Bits	Bit mask / Description
31 ... 15	Reserved for future use
14	`IP_CFG_FLAG_NO_GRATUITOUS_ARP` Enable/Disable sending of gratuitous ARP frames. If set, sending is disabled.
13 … 6	Reserved for future use
5	`IP_CFG_FLAG_ETHERNET_ADDR` Set Ethernet address (MAC address): If set, the `abEthernetAddr` area will be evaluated.
4	`IP_CFG_FLAG_DHCP` Enable DHCP: If set, the stack obtains its configuration from a DHCP server.
3	`IP_CFG_FLAG_BOOTP` Enable BOOTP: If set, the stack obtains its configuration from a BOOTP server.
2	`IP_CFG_FLAG_GATEWAY` Gateway available: If set, the content of the `ulGateway` parameter will be evaluated. If the flag is not set the stack will assume that there exists no gateway.
1	`IP_CFG_FLAG_NET_MASK` Netmask available: If set, the content of the `ulNetMask` parameter will be evaluated. If the flag is not set the stack will assume to be an isolated host which is not connected to any network. The `ulGateway` parameter will be ignored in this case.
0	`IP_CFG_FLAG_IP_ADDR` IP address available: If set, the content of the `ulIpAddr` parameter will be evaluated.

Table 10: Parameter `ulFlags`

The valid combinations of flags for manual IP configuration (IP_CFG_FLAG_IP_ADDR, IP_CFG_FLAG_NET_MASK and IP_CFG_FLAG_GATEWAY) are:

- No flag set: No manual configuration - only DHCP and/or BOOTP
- IP_CFG_FLAG_IP_ADDR + IP_CFG_FLAG_NET_MASK: Local network without gateway
- IP_CFG_FLAG_IP_ADDR + IP_CFG_FLAG_NET_MASK + IP_CFG_FLAG_GATEWAY: Network with gateway.

Please note, that there exists a fallback procedure between the different configuration methods, if more than one is enabled in the ulFlags parameter. If enabled, the stack will first try to configure using DHCP. If DHCP configuration fails, the stack wills fallback to BOOTP, if this is enabled. In case of a BOOTP failure, the values found in the ulIpAddr, ulNetMask and ulGateway parameters will be used, if enabled in ulFlags. If none of these configuration mechanisms succeed, the stack will report an error. For a description of the timing implied by the different IP configuration mechanisms, please refer to section *Start-up of the TCP/IP* at page *18*.

Parameter ulIpAddr

The stacks IP address can be configured using the ulIpAddr parameter. Additionally, the IP_CFG_FLAG_IP_ADDR flag must be set in ulFlags.

Parameter ulNetMask

The ulNetMask parameter holds the Netmask for the subnet the device is connected to. Additionally, the IP_CFG_FLAG_NET_MASK flag must be set in ulFlags.

Parameter ulGateway

The ulGateway parameter stores the IP address of the default gateway. Additionally, the IP_CFG_FLAG_GATEWAY flag must be set in ulFlags.

Parameter abEthernetAddr

The abEthernetAddr area can be used to overwrite the device's default Ethernet address (MAC address). Additionally, the IP_CFG_FLAG_ETHERNET_ADDR flag must be set in ulFlags.

4.1.2 TCPIP_IP_CMD_SET_CONFIG_REQ/CNF - Providing configuration data

Using this command, the IP layer can be provided with new configuration parameters. If any sockets are open when the TCPIP_IP_CMD_SET_CONFIG_REQ command is received by the stack it will send a TCPIP_TCP_UDP_CMD_SHUTDOWN_IND command to the owner of the socket. Please refer to the section *TCPIP_TCP_UDP_CMD_SHUTDOWN_IND/RES - Shutdown of the* at page *94* to learn more about the handling of the TCPIP_TCP_UDP_CMD_SHUTDOWN_IND/RES command.

Packet structure

```
typedef struct TCPIP_DATA_IP_CMD_SET_CONFIG_REQ_Ttag
{
  TLR_UINT32  ulFlags;
  TLR_UINT32  ulIpAddr;
  TLR_UINT32  ulNetMask;
  TLR_UINT32  ulGateway;
  TLR_UINT8   abEthernetAddr[6];

} TCPIP_DATA_IP_CMD_SET_CONFIG_REQ_T;

#define TCPIP_DATA_IP_CMD_SET_CONFIG_REQ_SIZE \
                            (sizeof(TCPIP_DATA_IP_CMD_SET_CONFIG_REQ_T))

typedef struct TCPIP_PACKET_IP_CMD_SET_CONFIG_REQ_Ttag
{
  TLR_PACKET_HEADER_T                 tHead;
  TCPIP_DATA_IP_CMD_SET_CONFIG_REQ_T  tData;

} TCPIP_PACKET_IP_CMD_SET_CONFIG_REQ_T;
```

Packet description

structure TCPIP_PACKET_IP_CMD_SET_CONFIG_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Head - structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of TCP_UDP-task process queue
ulSrc	UINT32		Source queue handle of AP-task process queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for TCPIP_IP_xx packets.
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	22	TCPIP_DATA_IP_CMD_SET_CONFIG_REQ_SIZE - Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0x00000000	-
ulCmd	UINT32	0x200	TCPIP_IP_CMD_SET_CONFIG_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
Data - structure TCPIP_DATA_IP_CMD_SET_CONFIG_REQ_T			
ulFlags	UINT32		Flags: See chapter *Protocol parameters* on page 24.
ulIpAddr	UINT32		IP address of the stack
ulNetMask	UINT32		Netmask of local subnet
ulGateway	UINT32		IP address of default gateway
abEthernetAddr[6]	UINT8[]		Ethernet address (MAC address) of the device

Table 11: `TCPIP_IP_CMD_SET_CONFIG_REQ` – Request command for providing configuration data

Source code example

```
#define LOCAL_IP_ADDR     (0xC0A80ACF)  /* Own IP address: 192.168.10.207       */
#define LOCAL_NET_MASK    (0xFFFFFF00)  /* Own Netmask   : 255.255.255.0        */
#define LOCAL_GATEWAY     (0xC0A80A0A)  /* Gateway       : 192.168.10.10        */


TLR_RESULT
ApIpCmdSetConfigReq( TCPIP_AP_TASK_RSC_T FAR*  ptRsc )
{
  TCPIP_PACKET_IP_CMD_SET_CONFIG_REQ_T*  ptPck;


  if( TLR_POOL_PACKET_GET( ptRsc->tLoc.hPool, &ptPck ) != TLR_S_OK )
  {
    return( TLR_E_FAIL );
  }

  TLS_QUE_LINK_SET_NEW_DESTID( ptRsc->tRem.tQueTcpTask, 0 );
  TLS_QUE_LINK_SET_PACKET_SRC( ptPck, ptRsc->tLoc.tLnkSrc );

  ptPck->tHead.ulLen   = TCPIP_DATA_IP_CMD_SET_CONFIG_REQ_SIZE;
  ptPck->tHead.ulId    = ++ptRsc->tLoc.ulSndTcpId;
  ptPck->tHead.ulSta   = 0;
  ptPck->tHead.ulCmd   = TCPIP_IP_CMD_SET_CONFIG_REQ;
  ptPck->tHead.ulExt   = 0;
  ptPck->tHead.ulRout  = 0;

  ptPck->tData.ulFlags   = IP_CFG_FLAG_IP_ADDR | \
                           IP_CFG_FLAG_NET_MASK | IP_CFG_FLAG_GATEWAY;

  ptPck->tData.ulIpAddr  = LOCAL_IP_ADDR;

  ptPck->tData.ulNetMask = LOCAL_NET_MASK;
  ptPck->tData.ulGateway = LOCAL_GATEWAY;

  if( TLS_QUE_SENDPACKET_FIFO( ptRsc->tRem.tQueTcpTask,
                               ptPck,
                               100                     ) != TLR_S_OK )
  {
    TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool, ptPck );
    return( TLR_E_FAIL );
  }

  return( TLR_S_OK );
}
```

Packet structure

```
typedef struct TCPIP_PACKET_IP_CMD_SET_CONFIG_CNF_Ttag
{
  TLR_PACKET_HEADER_T  tHead;

} TCPIP_PACKET_IP_CMD_SET_CONFIG_CNF_T;
```

Packet description

structure TCPIP_PACKET_IP_CMD_SET_CONFIG_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle, untouched
ulSrc	UINT32		Source queue handle, untouched
ulDestId	UINT32	0	Destination End Point Identifier, untouched
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, untouched
ulLen	UINT32	0	Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification, untouched
ulSta	UINT32		See section *Status/error codes* (page 117).
ulCmd	UINT32	0x201	TCPIP_IP_CMD_SET_CONFIG_CNF - Command
ulExt	UINT32	0	Extension, untouched
ulRout	UINT32	0	Routing, do not touch

Table 12: TCPIP_IP_CMD_SET_CONFIG_CNF – Confirmation command for providing configuration data

Source code example

```
TLR_RESULT
ApIpCmdSetConfigCnf( TCPIP_AP_TASK_RSC_T                  FAR*  ptRsc,
                     TCPIP_PACKET_IP_CMD_SET_CONFIG_CNF_T FAR*  ptPck )
{
  TLR_RESULT  eRslt = ptPck->tHead.ulSta;


  TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool,ptPck );

  return( eRslt );
}
```

4.1.3 TCPIP_IP_CMD_GET_CONFIG_REQ/CNF - Obtaining configuration data

The TCPIP_IP_CMD_GET_CONFIG command can be used to obtain the current configuration from the IP layer. Parameters returned include IP address, netmask, IP address of default gateway, and several flags.

Packet structure

```
typedef struct TCPIP_PACKET_IP_CMD_GET_CONFIG_REQ_Ttag
{
  TLR_PACKET_HEADER_T  tHead;

} TCPIP_PACKET_IP_CMD_GET_CONFIG_REQ_T;
```

Packet description

structure TCPIP_PACKET_IP_CMD_GET_CONFIG_REQ_T			**Type: Request**
Variable	**Type**	**Value / Range**	**Description**
structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of TCP_UDP-task process queue
ulSrc	UINT32		Source queue handle of AP-task process queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for TCPIP_IP_xx packets.
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	0	Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0x00000000	-
ulCmd	UINT32	0x202	TCPIP_IP_CMD_GET_CONFIG_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons

Table 13: TCPIP_IP_CMD_GET_CONFIG_REQ – Request command for obtaining configuration data

Source code example

```
TLR_RESULT
ApIpCmdGetConfigReq( TCPIP_AP_TASK_RSC_T FAR*  ptRsc )
{
  TCPIP_PACKET_IP_CMD_GET_CONFIG_REQ_T*  ptPck;


  if( TLR_POOL_PACKET_GET( ptRsc->tLoc.hPool, &ptPck ) != TLR_S_OK )
  {
    return( TLR_E_FAIL );
  }

  TLS_QUE_LINK_SET_NEW_DESTID( ptRsc->tRem.tQueTcpTask, 0 );
  TLS_QUE_LINK_SET_PACKET_SRC( ptPck, ptRsc->tLoc.tLnkSrc );

  ptPck->tHead.ulLen   = 0;  /* No .tData for this packet */
  ptPck->tHead.ulId    = ++ptRsc->tLoc.ulSndTcpId;
  ptPck->tHead.ulSta   = 0;
  ptPck->tHead.ulCmd   = TCPIP_IP_CMD_GET_CONFIG_REQ;
  ptPck->tHead.ulExt   = 0;
  ptPck->tHead.ulRout  = 0;

  if( TLS_QUE_SENDPACKET_FIFO( ptRsc->tRem.tQueTcpTask,
                               ptPck,
                               100                    ) != TLR_S_OK )
  {
    TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool, ptPck );
    return( TLR_E_FAIL );
  }

  return( TLR_S_OK );
}
```

Packet structure

```
typedef struct TCPIP_DATA_IP_CMD_GET_CONFIG_CNF_Ttag
{
  TLR_UINT32  ulFlags;
  TLR_UINT32  ulIpAddr;
  TLR_UINT32  ulNetMask;
  TLR_UINT32  ulGateway;
  TLR_UINT8   abEthernetAddr[6];

} TCPIP_DATA_IP_CMD_GET_CONFIG_CNF_T;

#define TCPIP_DATA_IP_CMD_GET_CONFIG_CNF_SIZE \
                           (sizeof(TCPIP_DATA_IP_CMD_GET_CONFIG_CNF_T))

typedef struct TCPIP_PACKET_IP_CMD_GET_CONFIG_CNF_Ttag
{
  TLR_PACKET_HEADER_T                tHead;
  TCPIP_DATA_IP_CMD_GET_CONFIG_CNF_T  tData;

} TCPIP_PACKET_IP_CMD_GET_CONFIG_CNF_T;
```

Packet description

Structure TCPIP_PACKET_IP_CMD_GET_CONFIG_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle, untouched
ulSrc	UINT32		Source queue handle, untouched
ulDestId	UINT32	0	Destination End Point Identifier, untouched
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, untouched
ulLen	UINT32	22	TCPIP_DATA_IP_CMD_GET_CONFIG_CNF_SIZE - Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification, untouched
ulSta	UINT32		See section *Status/error codes* (page 117).
ulCmd	UINT32	0x203	TCPIP_IP_CMD_GET_CONFIG_CNF - Command
ulExt	UINT32	0	Extension, untouched
ulRout	UINT32	0	Routing, do not touch
Data - Structure TCPIP_DATA_IP_CMD_GET_CONFIG_CNF_T			
ulFlags	UINT32		Flags: See chapter *Protocol parameters* on page 24.
ulIpAddr	UINT32		IP address of the stack
ulNetMask	UINT32		Netmask of local subnet
ulGateway	UINT32		IP address of default gateway
abEthernetAddr[6]	UINT8[]		Ethernet address (MAC address) of the device

Table 14: TCPIP_IP_CMD_GET_CONFIG_CNF – Confirmation command for obtaining configuration data

Source code example

```
TLR_RESULT
ApIpCmdGetConfigCnf( TCPIP_AP_TASK_RSC_T                  FAR*  ptRsc,
                     TCPIP_PACKET_IP_CMD_GET_CONFIG_CNF_T FAR*  ptPck )
{
  TLR_RESULT  eRslt = ptPck->tHead.ulSta;

  if( TLR_S_OK == eRslt )
  {
    ptRsc->tLoc.ulFlags   = ptPck->tData.ulFlags;
    ptRsc->tLoc.ulIpAddr  = ptPck->tData.ulIpAddr;
    ptRsc->tLoc.ulNetMask = ptPck->tData.ulNetMask;
    ptRsc->tLoc.ulGateway = ptPck->tData.ulGateway;

    LIB_MEMCPY( &ptRsc->tLoc.abEthernetAddr[0],
                &ptPck->tData.abEthernetAddr[0],
                sizeof( ptPck->tData.abEthernetAddr ) );
  }

  TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool,ptPck );
  return( eRslt );
}
```

4.1.4 TCPIP_IP_CMD_SET_PARAM_REQ/CNF - Setting IP parameters

Some parameters of the IP layer can be modified during run-time. Using the TCPIP_IP_CMD_SET_PARAM command, entries in the stacks ARP cache can be added or removed.

Furthermore, an “ARP send request interface” is implemented (modes IP_PRM_SEND_ARP_REQ, IP_PRM_SEND_ARP_TMT_REQ, IP_PRM_SEND_ARP_TMT_REQ_W_CACHEENTRY and IP_PRM_SET_ARP_REQ_TMT). See the following description.

ARP send request interface

The motivation for mode IP_PRM_SEND_ARP_REQ is, to send an ARP request for checking for an existing IP address on the network. Up to 128 simultaneously active ARP requests are possible. The timeout for every ARP request is 1 second by default and can be changed by mode IP_PRM_SET_ARP_REQ_TMT.

If the checked IP address (remote station) exists, the confirmation packets delivers the error code TLR_S_OK , otherwise the confirmation packets delivers the error code TLR_E_FAIL after the timeout.

For the request packet (struct tSendArpReq):

- Parameter ulIpAddr is the IP address to check
- If parameter abEthernetAddr is set to zero (0x00-0x00-0x00-0x00-0x00-0x00), a broadcast ARP request is send, otherwise an unicast ARP request to the given MAC address is sent.

For the confirmation packet (struct tSendArpCnf):

- Parameter ulIpAddr is unchanged
- Parameter abEthernetAddr is set to the MAC address of the checked remote station, if the station exists (ulSta = TLR_S_OK). Otherwise, the parameter abEthernetAddr is set to zero (ulSta = TLR_E_FAIL).

The motivation for mode IP_PRM_SEND_ARP_TMT_REQ is in principle the same as for mode IP_PRM_SEND_ARP_REQ, but this mode can search for more stations with the given IP address. Up to 128 simultaneously active ARP requests are possible. The timeout for every ARP request is 1 second by default and can be changed by mode IP_PRM_SET_ARP_REQ_TMT. Furthermore, the command is aborted, if a parameterized count of stations has answered (before the timeout is elapsed).

For the request packet (struct tSendArpTmtReq):

- Parameter ulIpAddr is the IP address to check
- If parameter abEthernetAddr is set to zero (0x00-0x00-0x00-0x00-0x00-0x00), a broadcast ARP request is send, otherwise an unicast ARP request to the given MAC address is sent.
- The parameter ulStationCntAbort defines the count of received stations, when the command aborts (before the timeout is elapsed). A value of 0 results to an internal value of 0xFFFF0000. This means in the practice that there is no station limit - only the timeout is active. Furthermore, the value is internal delimited to 0xFFFF0000 (no error occurs).

For the confirmation packet (struct `tSendArpTmtCnf`):

- Parameter `ulIpAddr` is unchanged
- Parameter `abEthernetAddr` is unchanged
- Parameter `ulStationCntAbort` is unchanged
- Parameter `ulStationCnt` is the count of founded stations with the given IP address `ulIpAddr`.
- Parameter `tStation` is the station list (MAC addresses of the received stations). `tStation[0]` is the first received station, `tStation[1]` the second, and so on. The first `SEND_ARP_TMT_STATION_MAX` (100) stations are stored.

Mode `IP_PRM_SEND_ARP_TMT_REQ_W_CACHEENTRY` is the same as mode `IP_PRM_SEND_ARP_TMT_REQ`, but the ARP replies from remotes are additionally integrated in the ARP cache.

With mode `IP_PRM_SET_ARP_REQ_TMT`, the timeout for the ARP requests can be set. The default value is 1 second.

For the request packet (struct `tSetArpReqTmt`):

- Parameter `ulTimeout` is the global timeout for the ARP requests in milliseconds. The range is `ARP_REQ_INTF_TIMEOUT_MIN` (100 ms) … `ARP_REQ_INTF_TIMEOUT_MAX` (60000 ms).

The confirmation packet has no parameter.

Register ACD application

This mode can be used to register an application at the TcpIp stack in order to receive an indication packet ( ) when an address conflict occurred. The ACD mechanism can be enabled using the corresponding flag (`TCPIP_SRT_FLAG_ACTIVATE_ACD`) in the startup parameters of the TCP/IP stack.

Register ICMP Indication

Is the application registered to this service, the TcpIp stack will send an indication (TCPIP_IP_CMD_ICMP_IND) to the application if an ICMP packet of the registered type was received. The request is fully processed by the TcpIp stack, there is no possibility to handle the request by the application.

Packet structure

```
/* Valid modes of packet ulMode variable */
#define IP_PRM_ADD_ARP_ENTRY            (1)
#define IP_PRM_DEL_ARP_ENTRY            (2)
#define IP_PRM_DEL_ARP_ENTRY_IP         (3)
#define IP_PRM_DEL_ARP_ENTRY_MAC        (4)
#define IP_PRM_SEND_ARP_REQ             (5)
#define IP_PRM_SEND_ARP_TMT_REQ         (6)
#define IP_PRM_SET_ARP_REQ_TMT          (7)
#define IP_PRM_REGISTER_ACD_APP         (8)
#define IP_PRM_REGISTER_ICMP_APP        (9)

typedef struct TCPIP_DATA_IP_CMD_SET_PARAM_REQ_Ttag
{
  TLR_UINT32   ulMode;

  union
  {
    struct
    {
      TLR_UINT32  ulIpAddr;
      TLR_UINT8   abEthernetAddr[6];
    } tAddDelArpEntry;
```

```
    struct
    {
      TLR_UINT32  ulIpAddr;
    } tDelArpEntryIp;

    struct
    {
      TLR_UINT8  abEthernetAddr[6];
    } tDelArpEntryMac;

    struct
    {
      TLR_UINT32  ulServices;
    } tRegisterIcmpService;

    struct
    {
      TLR_UINT32  ulIpAddr;
      TLR_UINT8   abEthernetAddr[6];
    } tSendArpReq;

    struct
    {
      TLR_UINT32  ulIpAddr;
      TLR_UINT8   abEthernetAddr[6];
      TLR_UINT32  ulStationCntAbort;  /* Abort command, if this count of      */
    } tSendArpTmtReq; /* stations has reached (e.g. 2)                        */

    struct
    {
      TLR_UINT32  ulTimeout;
    } tSetArpReqTmt;

  } unParam;

} TCPIP_DATA_IP_CMD_SET_PARAM_REQ_T;

#define TCPIP_DATA_IP_CMD_SET_PARAM_REQ_SIZE_ADD_ARP_ENTRY      \
              (sizeof(TLR_UINT32) + 10 /*sizeof(tAddDelArpEntry)*/)
#define TCPIP_DATA_IP_CMD_SET_PARAM_REQ_SIZE_DEL_ARP_ENTRY      \
              (sizeof(TLR_UINT32) + 10 /*sizeof(tAddDelArpEntry)*/)
#define TCPIP_DATA_IP_CMD_SET_PARAM_REQ_SIZE_DEL_ARP_ENTRY_IP   \
              (sizeof(TLR_UINT32) + 4  /*sizeof(tDelArpEntryIp )*/)
#define TCPIP_DATA_IP_CMD_SET_PARAM_REQ_SIZE_DEL_ARP_ENTRY_MAC  \
              (sizeof(TLR_UINT32) + 6  /*sizeof(tDelArpEntryMac)*/)

#define TCPIP_DATA_IP_CMD_SET_PARAM_REQ_SIZE_SEND_ARP_REQ       \
              (sizeof(TLR_UINT32) + 10 /*sizeof(tSendArpReq    )*/)

#define TCPIP_DATA_IP_CMD_SET_PARAM_REQ_SIZE_SEND_ARP_TMT_REQ   \
              (sizeof(TLR_UINT32) + 14 /*sizeof(tSendArpTmtReq )*/)

#define TCPIP_DATA_IP_CMD_SET_PARAM_REQ_SIZE_SET_ARP_REQ_TMT    \
              (sizeof(TLR_UINT32) + 4  /*sizeof(tSetArpReqTmt  )*/)

typedef struct TCPIP_PACKET_IP_CMD_SET_PARAM_REQ_Ttag
{
  TLR_PACKET_HEADER_T                tHead;
  TCPIP_DATA_IP_CMD_SET_PARAM_REQ_T  tData;
} TCPIP_PACKET_IP_CMD_SET_PARAM_REQ_T;
```

Packet description

Structure TCPIP_PACKET_IP_CMD_SET_PARAM_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of TCP_UDP-task process queue
ulSrc	UINT32		Source queue handle of AP-task process queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for TCPIP_IP_xx packets.
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32		Packet data length in bytes: See Table 16: TCPIP_IP_CMD_SET_PARAM_REQ – Mode description for parameter data unParam
ulId	UINT32	0 ... 2^{32} -1	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0x00000000	-
ulCmd	UINT32	0x204	TCPIP_IP_CMD_SET_PARAM_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
Data - Structure TCPIP_DATA_IP_CMD_SET_PARAM_REQ_T			
ulMode	UINT32		Mode: Type of parameter to configure: See Table 16: `TCPIP_IP_CMD_SET_PARAM_REQ` – Mode description for parameter data `unParam`
unParam	union		Union unParam: Parameter data to set: See Table 16: `TCPIP_IP_CMD_SET_PARAM_REQ` – Mode description for parameter data `unParam` and the following tables

Table 15: `TCPIP_IP_CMD_SET_PARAM_REQ` – Request command for setting IP parameters

The structure of the parameter data unParam (union in the *Packet structure*) depends on the Mode parameter ulMode:

Mode ulMode	Description	union element of unParam	ulLen *
IP_PRM_ ADD_ARP_ENTRY (1)	Add static entry to ARP cache	tAddDelArpEntry	TCPIP_DATA_ IP_CMD_SET_PARAM_REQ_SIZE_ ADD_ARP_ENTRY (14)
IP_PRM_ DEL_ARP_ENTRY (2)	Delete entry from ARP cache	tAddDelArpEntry	TCPIP_DATA_ IP_CMD_SET_PARAM_REQ_SIZE_ DEL_ARP_ENTRY (14)
IP_PRM_ DEL_ARP_ENTRY_IP (3)	Delete entry from ARP cache	tDelArpEntryIp	TCPIP_DATA_ IP_CMD_SET_PARAM_REQ_SIZE_ DEL_ARP_ENTRY_IP (8)
IP_PRM_ DEL_ARP_ENTRY_MAC (4)	Delete all entries with specified MAC address from ARP cache	tDelArpEntryMac	TCPIP_DATA_ IP_CMD_SET_PARAM_REQ_SIZE_ DEL_ARP_ENTRY_MAC (10)
IP_PRM_ SEND_ARP_REQ (5)	Sends an ARP request (“ARP send request interface“)	tSendArpReq	TCPIP_DATA_ IP_CMD_SET_PARAM_REQ_SIZE_ SEND_ARP_REQ (14)
IP_PRM_ SEND_ARP_TMT_REQ (6)	Sends an ARP request (“ARP send request interface“). Searches for more stations.	tSendArpTmtReq	TCPIP_DATA_ IP_CMD_SET_PARAM_REQ_SIZE_ SEND_ARP_TMT_REQ (18)
IP_PRM_ SET_ARP_REQ_TMT (7)	Set timeout for ARP requests (global value for every ARP request).	tSetArpReqTmt	TCPIP_DATA_ IP_CMD_SET_PARAM_REQ_SIZE_ SET_ARP_REQ_TMT (8)
IP_PRM_ REGISTER_ACD_APP (8)	Register Application for ACD conflict indications	No element necessary	TCPIP_DATA_ IP_CMD_SET_PARAM_REQ_SIZE_ REGISTER_ACD_APP (4)
IP_PRM_ REGISTER_ICMP_APP (9)	Register application for ICMP service	tRegisterIcmp Service	TCPIP_DATA_ IP_CMD_SET_PARAM_REQ_SIZE_ REGISTER_ICMP_APP (8)

Table 16: TCPIP_IP_CMD_SET_PARAM_REQ – Mode description for parameter data unParam

* The allocated packet size must consider the maximum used packet data length ulLen by memory allocation! This must be considered also for the **confirmation packet**, see Table 26 on 43.

The following table describes the union unParam in structure TCPIP_DATA_IP_CMD_SET_PARAM_REQ_T. The union elements are described in the following tables.

Union unParam			
Variable	**Type**	**Value / Range**	**Description**
tAddDelArpEntry	struct		Request structure for both, *Add static entry to ARP cache* and *Delete entry from ARP cache* For definition, see Table 18: TCPIP_IP_CMD_SET_PARAM_REQ – Struct tAddDelArpEntry of union unParam
tDelArpEntryIp	struct		Request structure for *Delete entry from ARP cache* For definition, see Table 19: TCPIP_IP_CMD_SET_PARAM_REQ – Struct tDelArpEntryIp of union unParam
tDelArpEntryMac	struct		Request structure for *Delete all entries with specified MAC address from ARP cache* For definition, see Table 20: TCPIP_IP_CMD_SET_PARAM_REQ – Struct tDelArpEntryMac of union unParam
tSendArpReq	struct		Request structure for “ARP send request interface” For definition, see Table 21: TCPIP_IP_CMD_SET_PARAM_REQ – Struct tSendArpReq of union unParam
tSendArpTmtReq	struct		Request structure for “ARP send request interface”. Searches for more stations. For definition, see Table 22: TCPIP_IP_CMD_SET_PARAM_REQ – Struct tSendArpTmtReq of union unParam
tSetArpReqTmt	struct		Request structure for set timeout of “ARP send request interface” (global value for every ARP request). For definition, see Table 23: TCPIP_IP_CMD_SET_PARAM_REQ – Struct tSetArpReqTmt of union unParam
tRegisterIcmp Service	Struct		Request structure for registering a ICMP service For definition, see Table 21: TCPIP_IP_CMD_SET_PARAM_REQ – Struct tSendArpReq of union unParam

Table 17: TCPIP_IP_CMD_SET_PARAM_REQ – Union unParam

Structure tAddDelArpEntry			
Variable	**Type**	**Value / Range**	**Description**
ulIpAddr	UINT32		IP address
abEthernetAddr	UINT8[6]		Ethernet address (MAC address)

Table 18: TCPIP_IP_CMD_SET_PARAM_REQ – Struct tAddDelArpEntry of union unParam

Structure tDelArpEntryIp			
Variable	**Type**	**Value / Range**	**Description**
ulIpAddr	UINT32		IP address

Table 19: TCPIP_IP_CMD_SET_PARAM_REQ – Struct tDelArpEntryIp of union unParam

Structure tDelArpEntryMac			
Variable	**Type**	**Value / Range**	**Description**
abEthernetAddr	UINT8[6]		Ethernet address (MAC address)

Table 20: TCPIP_IP_CMD_SET_PARAM_REQ – Struct tDelArpEntryMac of union unParam

Structure tSendArpReq			
Variable	**Type**	**Value / Range**	**Description**
ulIpAddr	UINT32		IP address
abEthernetAddr	UINT8[6]		Ethernet address (MAC address)

Table 21: TCPIP_IP_CMD_SET_PARAM_REQ – Struct tSendArpReq of union unParam

Structure tSendArpTmtReq			
Variable	**Type**	**Value / Range**	**Description**
ulIpAddr	UINT32		IP address
abEthernetAddr	UINT8[6]		Ethernet address (MAC address)
ulStationCntAbort	UINT32	0 … 0xFFFF0000	Count of received stations, when the command aborts 0 = No station limit

Table 22: TCPIP_IP_CMD_SET_PARAM_REQ – Struct tSendArpTmtReq of union unParam

Structure tSetArpReqTmt			
Variable	**Type**	**Value / Range**	**Description**
ulTimeout	UINT32	100 … 60000	Timeout for the ARP Requests (milliseconds)

Table 23: TCPIP_IP_CMD_SET_PARAM_REQ – Struct tSetArpReqTmt of union unParam

Structure tRegisterIcmpService			
Variable	**Type**	**Value / Range**	**Description**
ulService	UINT32	0x00000001	IP_PRM_REGISTER_ICMP_SERVICE_ECHO_REQUEST Register an application for a ICMP ping indication service

Table 24: TCPIP_IP_CMD_SET_PARAM_REQ – Struct tRegisterIcmpService of union unParam

Source code example

```
#define REMOTE_IP_ADDR    (0xC0A80A6A)  /* IP address of remote host:          */
                                        /* 192.168.10.106                      */


TLR_RESULT
ApIpCmdSetParamReq( TCPIP_AP_TASK_RSC_T FAR*  ptRsc )
{
  TCPIP_PACKET_IP_CMD_SET_PARAM_REQ_T*  ptPck;


  if( TLR_POOL_PACKET_GET( ptRsc->tLoc.hPool, &ptPck ) != TLR_S_OK )
  {
    return( TLR_E_FAIL );
  }

  TLS_QUE_LINK_SET_NEW_DESTID( ptRsc->tRem.tQueTcpTask, 0 );
  TLS_QUE_LINK_SET_PACKET_SRC( ptPck, ptRsc->tLoc.tLnkSrc );

  ptPck->tHead.ulLen   = TCPIP_DATA_IP_CMD_SET_PARAM_REQ_SIZE_ADD_ARP_ENTRY;
  ptPck->tHead.ulId    = ++ptRsc->tLoc.ulSndTcpId;
  ptPck->tHead.ulSta   = 0;
  ptPck->tHead.ulCmd   = TCPIP_IP_CMD_SET_PARAM_REQ;
  ptPck->tHead.ulExt   = 0;
  ptPck->tHead.ulRout  = 0;

  ptPck->tData.ulMode  = IP_PRM_ADD_ARP_ENTRY;

  ptPck->tData.unParam.tAddDelArpEntry.ulIpAddr = REMOTE_IP_ADDR;

  /* MAC address of REMOTE_IP_ADDR: 00-04-E2-C5-00-8A (Example!!) */
  ptPck->tData.unParam.tAddDelArpEntry.abEthernetAddr[0] = 0x00;
  ptPck->tData.unParam.tAddDelArpEntry.abEthernetAddr[1] = 0x04;
  ptPck->tData.unParam.tAddDelArpEntry.abEthernetAddr[2] = 0xE2;
  ptPck->tData.unParam.tAddDelArpEntry.abEthernetAddr[3] = 0xC5;
  ptPck->tData.unParam.tAddDelArpEntry.abEthernetAddr[4] = 0x00;
  ptPck->tData.unParam.tAddDelArpEntry.abEthernetAddr[5] = 0x8A;

  if( TLS_QUE_SENDPACKET_FIFO( ptRsc->tRem.tQueTcpTask,
                               ptPck,
                               100                    ) != TLR_S_OK )
  {
    TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool, ptPck );
    return( TLR_E_FAIL );
  }

  return( TLR_S_OK );
}
```

Packet structure

```
typedef struct TCPIP_DATA_IP_CMD_SET_PARAM_CNF_Ttag
{
  TLR_UINT32  ulMode;

  union
  {
    struct
    {
      TLR_UINT32  ulIpAddr;
      TLR_UINT8   abEthernetAddr[6];
    } tSendArpCnf;

    struct
    {
      TLR_UINT32  ulIpAddr;           /* Struct of Request  */
      TLR_UINT8   abEthernetAddr[6];  /*                    */
      TLR_UINT32  ulStationCntAbort;  /*                    */

      TLR_UINT32  ulStationCnt;         /* Count of founded stations  */
      MAC_ADDR_T  tStation[SEND_ARP_TMT_STATION_MAX]; /* Station list */
    } tSendArpTmtCnf;

  } unParam;

} TCPIP_DATA_IP_CMD_SET_PARAM_CNF_T;

#define TCPIP_DATA_IP_CMD_SET_PARAM_CNF_SIZE \
                           (sizeof(TLR_UINT32))

#define TCPIP_DATA_IP_CMD_SET_PARAM_CNF_SIZE_SEND_ARP_REQ       \
              (sizeof(TLR_UINT32) + 10 /*sizeof(tSendArpCnf    )*/)

#define TCPIP_DATA_IP_CMD_SET_PARAM_CNF_SIZE_SEND_ARP_TMT_REQ_MIN \
              (sizeof(TLR_UINT32) + 18 /*sizeof(tSendArpTmtCnf )*/)

typedef struct TCPIP_PACKET_IP_CMD_SET_PARAM_CNF_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
  TCPIP_DATA_IP_CMD_SET_PARAM_CNF_T  tData;

} TCPIP_PACKET_IP_CMD_SET_PARAM_CNF_T;
```

Packet description

Structure TCPIP_PACKET_IP_CMD_SET_PARAM_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle, untouched
ulSrc	UINT32		Source queue handle, untouched
ulDestId	UINT32	0	Destination End Point Identifier, untouched
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, untouched
ulLen	UINT32		Packet data length in bytes: See Table 26: `TCPIP_IP_CMD_SET_PARAM_CNF` – Mode Description for Parameter Data `unParam`
ulId	UINT32	0 ... 2^{32} -1	Packet identification, untouched
ulSta	UINT32		See section *Status/error codes* (page 117).
ulCmd	UINT32	0x205	TCPIP_IP_CMD_SET_PARAM_CNF - Command
ulExt	UINT32	0	Extension, untouched
ulRout	UINT32	0	Routing, do not touch
Data - Structure TCPIP_DATA_IP_CMD_SET_PARAM_CNF_T			
ulMode	UINT32		Mode: Type of parameter to configure: See Table 26: `TCPIP_IP_CMD_SET_PARAM_CNF` – Mode Description for Parameter Data `unParam`
unParam	union		Union unParam: Confirmation Parameter data: See Table 26: `TCPIP_IP_CMD_SET_PARAM_CNF` – Mode Description for Parameter Data `unParam` and the following tables

Table 25: `TCPIP_IP_CMD_SET_PARAM_CNF` – Confirmation command for setting IP parameters

The structure of the parameter data unParam (union in the *Packet structure*) depends on the Mode parameter ulMode:

Mode ulMode	Description	union element of unParam	ulLen *
IP_PRM_ ADD_ARP_ENTRY (1)	Add static entry to ARP cache	n/a	TCPIP_DATA_ IP_CMD_SET_PARAM_CNF_SIZE (4)
IP_PRM_ DEL_ARP_ENTRY (2)	Delete entry from ARP cache	n/a	TCPIP_DATA_ IP_CMD_SET_PARAM_CNF_SIZE (4)
IP_PRM_ DEL_ARP_ENTRY_IP (3)	Delete entry from ARP cache	n/a	TCPIP_DATA_ IP_CMD_SET_PARAM_CNF_SIZE (4)
IP_PRM_ DEL_ARP_ENTRY_MAC (4)	Delete all entries with specified MAC address from ARP cache	n/a	TCPIP_DATA_ IP_CMD_SET_PARAM_CNF_SIZE (4)
IP_PRM_ SEND_ARP_REQ (5)	Sends an ARP request (“ARP send request interface“)	tSendArpCnf	TCPIP_DATA_ IP_CMD_SET_PARAM_CNF_SIZE_ SEND_ARP_REQ (14)
IP_PRM_ SEND_ARP_TMT_REQ (6)	Sends an ARP request (“ARP send request interface“). Searches for more stations.	tSendArpTmtCnf	TCPIP_DATA_ IP_CMD_SET_PARAM_CNF_SIZE_ SEND_ARP_TMT_REQ_MIN (22 + 6* ulStationCnt)
IP_PRM_ SET_ARP_REQ_TMT (7)	Set timeout for ARP requests (global value for every ARP request).	n/a	TCPIP_DATA_ IP_CMD_SET_PARAM_CNF_SIZE (4)
IP_PRM_ REGISTER_ACD_APP (8)	Register Application for ACD conflict indications	n/a	TCPIP_DATA_ IP_CMD_SET_PARAM_CNF_SIZE (4)
IP_PRM_ REGISTER_ICMP_APP (9)	Register application for ICMP service	n/a	TCPIP_DATA_ IP_CMD_SET_PARAM_CNF_SIZE (4)

Table 26: TCPIP_IP_CMD_SET_PARAM_CNF – Mode Description for Parameter Data unParam

* In error case, ulLen is zero.

The following table describes the union unParam in structure TCPIP_DATA_IP_CMD_SET_PARAM_CNF_T. The union elements are described in the following tables.

Union unParam			
Variable	**Type**	**Value / Range**	**Description**
tSendArpCnf	struct		Confirmation structure for “ARP send request interface” For definition, see Table 28: TCPIP_IP_CMD_SET_PARAM_CNF – Struct tSendArpCnf of union unParam
tSendArpTmtCnf	struct		Confirmation structure for “ARP send request interface”. Searches for more stations. For definition, see Table 29: TCPIP_IP_CMD_SET_PARAM_CNF – Struct tSendArpTmtCnf of union unParam

Table 27: TCPIP_IP_CMD_SET_PARAM_CNF – Union unParam

Structure tSendArpCnf			
Variable	**Type**	**Value / Range**	**Description**
ulIpAddr	UINT32		IP address
abEthernetAddr	UINT8[6]		Ethernet address (MAC address)

Table 28: TCPIP_IP_CMD_SET_PARAM_CNF – Struct tSendArpCnf of union unParam

Structure tSendArpTmtCnf			
Variable	**Type**	**Value / Range**	**Description**
ulIpAddr	UINT32		IP address
abEthernetAddr	UINT8[6]		Ethernet address (MAC address)
ulStationCntAbort	UINT32		Count of received stations, when the command aborts 0 = No station limit
ulStationCnt	UINT32		Count of founded stations with the given IP address
tStation	struct		Station list (MAC addresses of the received stations)

Table 29: TCPIP_IP_CMD_SET_PARAM_CNF – Struct tSendArpTmtCnf of union unParam

Source code example

```
TLR_RESULT
ApIpCmdSetParamCnf( TCPIP_AP_TASK_RSC_T                 FAR*  ptRsc,
                    TCPIP_PACKET_IP_CMD_SET_PARAM_CNF_T FAR*  ptPck )
{
  TLR_RESULT  eRslt = ptPck->tHead.ulSta;


  TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool,ptPck );

  return( eRslt );
}
```

4.1.5 TCPIP_IP_CMD_GET_PARAM_REQ/CNF - Obtaining IP parameters

The TCPIP_IP_CMD_GET_PARAM command provides a method of obtaining the current parameters from the IP layer. Access to the IP stacks ARP cache is currently implemented.

Packet structure

```
/* Valid modes of packet ulMode variable */
#define IP_PRM_GET_ARP_ENTRY_INDEX      (1)
#define IP_PRM_GET_ARP_ENTRY_IP         (2)
#define IP_PRM_GET_ARP_ENTRY_MAC        (3)

#define IP_PRM_GET_HOST_NAME            (10)
#define IP_PRM_GET_DOMAIN_NAME          (11)

typedef struct TCPIP_DATA_IP_CMD_GET_PARAM_REQ_Ttag
{
  TLR_UINT32  ulMode;

  union
  {
    struct
    {
      TLR_UINT32  ulIndex;
    } tArpEntryIndex;

    struct
    {
      TLR_UINT32  ulIpAddr;
    } tArpEntryIp;

    struct
    {
      TLR_UINT8   abEthernetAddr[6];
    } tArpEntryMac;
  } unParam;

} TCPIP_DATA_IP_CMD_GET_PARAM_REQ_T;

#define TCPIP_DATA_IP_CMD_GET_PARAM_REQ_SIZE_GET_ARP_ENTRY_INDEX  \
              (sizeof(TLR_UINT32) + 4 /*sizeof(tArpEntryIndex)*/)

#define TCPIP_DATA_IP_CMD_GET_PARAM_REQ_SIZE_GET_ARP_ENTRY_IP     \
              (sizeof(TLR_UINT32) + 4 /*sizeof(tArpEntryIp)   */)

#define TCPIP_DATA_IP_CMD_GET_PARAM_REQ_SIZE_GET_ARP_ENTRY_MAC    \
              (sizeof(TLR_UINT32) + 6 /*sizeof(tArpEntryMac)  */)

typedef struct TCPIP_PACKET_IP_CMD_GET_PARAM_REQ_Ttag
{
  TLR_PACKET_HEADER_T                tHead;
  TCPIP_DATA_IP_CMD_GET_PARAM_REQ_T  tData;

} TCPIP_PACKET_IP_CMD_GET_PARAM_REQ_T;
```

Packet description

Structure TCPIP_PACKET_IP_CMD_GET_PARAM_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of TCP_UDP-task process queue
ulSrc	UINT32		Source queue handle of AP-task process queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for TCPIP_IP_xx packets.
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32		Packet data length in bytes: See Table 31: `TCPIP_IP_CMD_GET_PARAM_REQ` – Mode description for parameter data `unParam`
ulId	UINT32	0 ... 2^{32} -1	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0x00000000	-
ulCmd	UINT32	0x206	TCPIP_IP_CMD_GET_PARAM_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
Data - Structure TCPIP_DATA_IP_CMD_GET_PARAM_REQ_T			
ulMode	UINT32		Mode: Type of parameter to be obtained: See Table 31: `TCPIP_IP_CMD_GET_PARAM_REQ` – Mode description for parameter data `unParam`
unParam	union		Union unParam: Additional data required by parameter: See Table 31: `TCPIP_IP_CMD_GET_PARAM_REQ` – Mode description for parameter data `unParam` and the following tables

Table 30: `TCPIP_IP_CMD_GET_PARAM_REQ` – Request command for obtaining IP parameters

The structure of parameter request data unParam (union in the *Packet structure*) depends on the Mode parameter ulMode:

Mode ulMode	Description	union element of unParam	ulLen *
IP_PRM_GET_ ARP_ENTRY_INDEX (1)	Get ARP entry at specified cache index	tArpEntryIndex	TCPIP_DATA_ IP_CMD_GET_PARAM_REQ_SIZE_ GET_ARP_ENTRY_INDEX (8)
IP_PRM_GET_ ARP_ENTRY_IP (2)	Get entry with specified IP address from ARP cache	tArpEntryIp	TCPIP_DATA_ IP_CMD_GET_PARAM_REQ_SIZE_ GET_ARP_ENTRY_IP (8)
IP_PRM_GET_ ARP_ENTRY_MAC (3)	Get first entry with specified MAC address from ARP cache	tArpEntryMac	TCPIP_DATA_ IP_CMD_GET_PARAM_REQ_SIZE_ GET_ARP_ENTRY_MAC (10)
IP_PRM_GET_ HOST_NAME (10)	Get host name	none	(4)
IP_PRM_GET_ DOMAIN_NAME (11)	Get domain name	none	(4)

Table 31: TCPIP_IP_CMD_GET_PARAM_REQ – Mode description for parameter data unParam

* The allocated packet size must consider the maximum used packet data length ulLen by memory allocation!

The following table describes the union unParam in structure TCPIP_DATA_IP_CMD_GET_PARAM_REQ_T. The union elements are described in the following tables.

Union unParam			
Variable	**Type**	**Value / Range**	**Description**
tArpEntryIndex	struct		Request structure for *Get ARP entry at specified cache index* For definition, see Table 33: TCPIP_IP_CMD_GET_PARAM_REQ – Struct tArpEntryIndex of union unParam
tArpEntryIp	struct		Request structure for *Get entry with specified IP address from ARP cache* For definition, see Table 34: TCPIP_IP_CMD_GET_PARAM_REQ – Struct tArpEntryIp of union unParam
tArpEntryMac	struct		Request structure for *Get first entry with specified MAC address from ARP cache* For definition, see Table 35: TCPIP_IP_CMD_GET_PARAM_REQ – Struct tArpEntryMac of union unParam

Table 32: TCPIP_IP_CMD_GET_PARAM_REQ – Union unParam

Structure tArpEntryIndex			
Variable	**Type**	**Value / Range**	**Description**
ulIndex	UINT32	0 … 63 *	ARP cache index * This value is the ARP cache size minus 1. By default configuration, the ARP cache size is TCPIP_SRT_ARP_CACHE_SIZE_DEFAULT (64) (see startup parameter ulArpCacheSize).

Table 33: TCPIP_IP_CMD_GET_PARAM_REQ – Struct tArpEntryIndex of union unParam

Structure tArpEntryIp			
Variable	**Type**	**Value / Range**	**Description**
ulIpAddr	UINT32		IP address

Table 34: TCPIP_IP_CMD_GET_PARAM_REQ – Struct tArpEntryIp of union unParam

Structure tArpEntryMac			
Variable	**Type**	**Value / Range**	**Description**
abEthernetAddr	UINT8[6]		Ethernet address (MAC address)

Table 35: TCPIP_IP_CMD_GET_PARAM_REQ – Struct tArpEntryMac of union unParam

Source code example

```
TLR_RESULT
ApIpCmdGetParamReq( TCPIP_AP_TASK_RSC_T FAR*  ptRsc )
{
  TCPIP_PACKET_IP_CMD_GET_PARAM_REQ_T*  ptPck;


  if( TLR_POOL_PACKET_GET( ptRsc->tLoc.hPool, &ptPck ) != TLR_S_OK )
  {
    return( TLR_E_FAIL );
  }

  TLS_QUE_LINK_SET_NEW_DESTID( ptRsc->tRem.tQueTcpTask, 0 );
  TLS_QUE_LINK_SET_PACKET_SRC( ptPck, ptRsc->tLoc.tLnkSrc );

  ptPck->tHead.ulLen   = \
                       TCPIP_DATA_IP_CMD_GET_PARAM_REQ_SIZE_GET_ARP_ENTRY_INDEX;
  ptPck->tHead.ulId    = ++ptRsc->tLoc.ulSndTcpId;
  ptPck->tHead.ulSta   = 0;
  ptPck->tHead.ulCmd   = TCPIP_IP_CMD_GET_PARAM_REQ;
  ptPck->tHead.ulExt   = 0;
  ptPck->tHead.ulRout  = 0;

  ptPck->tData.ulMode  = IP_PRM_GET_ARP_ENTRY_INDEX;
  ptPck->tData.unParam.tArpEntryIndex.ulIndex = 0;  /* First ARP cache entry */

  if( TLS_QUE_SENDPACKET_FIFO( ptRsc->tRem.tQueTcpTask,
                               ptPck,
                               100                    ) != TLR_S_OK )
  {
    TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool, ptPck );
    return( TLR_E_FAIL );
  }

  return( TLR_S_OK );
}
```

Packet structure

```
typedef struct TCPIP_DATA_IP_CMD_GET_PARAM_CNF_Ttag
{
  TLR_UINT32  ulMode;

  union
  {
    struct
    {
      TLR_UINT32  ulIpAddr;
      TLR_UINT8   abEthernetAddr[6];
    } tArpEntry;
    TLR_CHAR szDomainName[TCP_MAX_DOMAIN_NAME];
    TLR_CHAR szHostName[TCP_MAX_HOST_NAME];

  } unParam;

} TCPIP_DATA_IP_CMD_GET_PARAM_CNF_T;

#define TCPIP_DATA_IP_CMD_GET_PARAM_CNF_SIZE_GET_ARP_ENTRY_INDEX  \
              (sizeof(TLR_UINT32) + 10 /*sizeof(tArpEntry)*/)

#define TCPIP_DATA_IP_CMD_GET_PARAM_CNF_SIZE_GET_ARP_ENTRY_IP     \
              (sizeof(TLR_UINT32) + 10 /*sizeof(tArpEntry)*/)

#define TCPIP_DATA_IP_CMD_GET_PARAM_CNF_SIZE_GET_ARP_ENTRY_MAC    \
              (sizeof(TLR_UINT32) + 10 /*sizeof(tArpEntry)*/)

typedef struct TCPIP_PACKET_IP_CMD_GET_PARAM_CNF_Ttag
{
  TLR_PACKET_HEADER_T                tHead;
  TCPIP_DATA_IP_CMD_GET_PARAM_CNF_T  tData;

} TCPIP_PACKET_IP_CMD_GET_PARAM_CNF_T;
```

Packet description

Structure TCPIP_PACKET_IP_CMD_GET_PARAM_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle, untouched
ulSrc	UINT32		Source queue handle, untouched
ulDestId	UINT32	0	Destination End Point Identifier, untouched
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, untouched
ulLen	UINT32		Packet data length in bytes: See Table 37: `TCPIP_IP_CMD_GET_PARAM_CNF` – Mode description for parameter data `unParam`
ulId	UINT32	0 ... 2^{32} -1	Packet identification, untouched
ulSta	UINT32		See section *Status/error codes* (page 117).
ulCmd	UINT32	0x207	TCPIP_IP_CMD_GET_PARAM_CNF - Command
ulExt	UINT32	0	Extension, untouched
ulRout	UINT32	0	Routing, do not touch
Data - Structure TCPIP_DATA_IP_CMD_GET_PARAM_CNF_T			
ulMode	UINT32		Mode: Type of parameter: See Table 37: `TCPIP_IP_CMD_GET_PARAM_CNF` – Mode description for parameter data `unParam`
unParam	union		Union unParam Result data: See Table 37: `TCPIP_IP_CMD_GET_PARAM_CNF` – Mode description for parameter data `unParam`

Table 36: `TCPIP_IP_CMD_GET_PARAM_CNF` – Confirmation Command for obtaining IP Parameters

The structure of parameter result data `unParam` (union in the *Packet structure*) depends on the Mode parameter `ulMode`:

Mode `ulMode`	**Description**	**union element of `unParam`**	**`ulLen` ***
`IP_PRM_GET_ARP_ENTRY_INDEX (1)`	Entry from ARP cache	`tArpEntry`	`TCPIP_DATA_IP_CMD_GET_PARAM_CNF_SIZE_GET_ARP_ENTRY_INDEX (14)`
`IP_PRM_GET_ARP_ENTRY_IP (2)`	Entry from ARP cache	`tArpEntry`	`TCPIP_DATA_IP_CMD_GET_PARAM_CNF_SIZE_GET_ARP_ENTRY_IP (14)`
`IP_PRM_GET_ARP_ENTRY_MAC (3)`	Entry from ARP cache	`tArpEntry`	`TCPIP_DATA_IP_CMD_GET_PARAM_CNF_SIZE_GET_ARP_ENTRY_MAC (14)`
`IP_PRM_GET_HOST_NAME (10)`	host name	`abHostName`	`4 + host name length`
`IP_PRM_GET_DOMAIN_NAME (11)`	domain name	`abDomainName`	`4 + domain name length`

Table 37: `TCPIP_IP_CMD_GET_PARAM_CNF` – Mode description for parameter data `unParam`

* In case of error, `ulLen` is zero.

Union unParam			
Variable	**Type**	**Value / Range**	**Description**
tArpEntry	struct		Confirmation structure for all three modes For definition, see Table 39: TCPIP_IP_CMD_GET_PARAM_CNF – Struct tArpEntry of union unParam

Table 38: TCPIP_IP_CMD_GET_PARAM_CNF – Union unParam

Structure tArpEntry			
Variable	**Type**	**Value / Range**	**Description**
ulIpAddr	UINT32		IP address
abEthernetAddr	UINT8[6]		Ethernet address (MAC address)

Table 39: TCPIP_IP_CMD_GET_PARAM_CNF – Struct tArpEntry of union unParam

Source code example

```
TLR_RESULT
ApIpCmdGetParamCnf( TCPIP_AP_TASK_RSC_T                 FAR*  ptRsc,
                    TCPIP_PACKET_IP_CMD_GET_PARAM_CNF_T FAR*  ptPck )
{
  TLR_RESULT  eRslt = ptPck->tHead.ulSta;
  if( TLR_S_OK == eRslt )
  {
    ptRsc->tLoc.ulIpAddr = ptPck->tData.unParam.tArpEntry.ulIpAddr;

    LIB_MEMCPY( &ptRsc->tLoc.abEthernetAddr[0],
                &ptPck->tData.unParam.tArpEntry.abEthernetAddr[0],
                sizeof( ptPck->tData.unParam.tArpEntry.abEthernetAddr ) );
  }
  TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool,ptPck );
  return( eRslt );
}
```

4.2 TCP and UDP socket and communication services

4.2.1 TCPIP_TCP_UDP_CMD_OPEN_REQ/CNF - Opening a socket

The TCPIP_TCP_UDP_CMD_OPEN command can be used by the application to obtain a handle for a socket of the specified protocol type. This is always the first step to be taken for any socket-based communication. The newly created socket will be bound to the specified local IP address and local port number.

Note: In general, the binding to a port number can be accomplished as described below at parameter *ulPort* in *Table 40: TCPIP_TCP_UDP_CMD_OPEN_REQ – Request command for opening a* . However, the usage of TCP port addresses within the firmware can be restricted to a specific range of available port addresses (such as 1024…2048, for instance) by the "Ethernet Interface TCP Port Numbers" tag list entry. If this entry has been applied to the firmware file using the tag list editor, only the restricted range of port addresses is applicable.

With this command the AP-task and the TCP_UDP-task have to exchange their End Point Identifier of this socket connection that shall be established between both. The End Point Identifier is a 32-Bit value specified by each process in order to associate incoming and address outgoing packets to the right Communication End Point. The End Point Identifier of the TCP_UDP-task is used as **socket handle** in the socket-based communication. The symbolic name for this socket handle for the following TCPIP_TCP/UDP_xx commands is *ulSocket*.

The AP-task has to specify its End Point Identifier in the ulSrcId of the initializing TCPIP_TCP_UDP_CMD_OPEN request packet. In the event of a successful Initialization, the TCP_UDP-task uses this specified value in the ulSrcId variable of each Confirmation Packet and in the ulDestId variable for Indication Packets (for example receive data indication command TCPIP_TCP_UDP_CMD_RECEIVE) sent to the AP-task context from now on. This covers all following socket-based packets (TCPIP_TCP/UDP_xx).

The TCP_UDP-task returns its End Pointer Reference (socket handle *ulSocket*) in ulDestId of the Confirmation Packet of the TCPIP_TCP_UDP_CMD_OPEN packet. The AP-task has to use this back coming value as ulDestId value from now on in all request and response TCPIP_TCP/UDP_xx packets that are sent to the TCP_UDP-task context.

Using the Macro TLS_QUE_SENDPACKET_FIFO() will send the packet to the TCP_UDP-task Process Queue.

When working with an UDP socket the application can start sending data right away (TCPIP_UDP_CMD_SEND), once the socket is successfully opened. Some more steps are required for TCP sockets: A connection must be established using either the TCPIP_TCP_CMD_WAIT_CONNECT or TCPIP_TCP_CMD_CONNECT command before being able transfer data.

Packet structure

```
/* Protocol types of packet ulProtocol variable */
#define TCP_PROTO_TCP                   (1)
#define TCP_PROTO_UDP                   (2)

typedef struct TCPIP_DATA_TCP_UDP_CMD_OPEN_REQ_Ttag
{
  TLR_UINT32  ulIpAddr;
  TLR_UINT32  ulPort;
  TLR_UINT32  ulProtocol;

} TCPIP_DATA_TCP_UDP_CMD_OPEN_REQ_T;

#define TCPIP_DATA_TCP_UDP_CMD_OPEN_REQ_SIZE \
                            (sizeof(TCPIP_DATA_TCP_UDP_CMD_OPEN_REQ_T))

typedef struct TCPIP_PACKET_TCP_UDP_CMD_OPEN_REQ_Ttag
{
  TLR_PACKET_HEADER_T                tHead;
  TCPIP_DATA_TCP_UDP_CMD_OPEN_REQ_T  tData;

} TCPIP_PACKET_TCP_UDP_CMD_OPEN_REQ_T;
```

Packet description

Structure TCPIP_PACKET_TCP_UDP_CMD_OPEN_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of TCP_UDP-task process queue
ulSrc	UINT32		Source queue handle of AP-task process queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for request packet.
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	12	TCPIP_DATA_TCP_UDP_CMD_OPEN_REQ_SIZE - Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0x00000000	-
ulCmd	UINT32	0x300	TCPIP_TCP_UDP_CMD_OPEN_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
Data - Structure TCPIP_DATA_TCP_UDP_CMD_OPEN_REQ_T			
ulIpAddr	UINT32	 0 (0.0.0.0) ≠ 0	Local IP address to use with socket: Bind socket to currently configured IP address Bind socket to specified IP address, must match currently configured IP address
ulPort	UINT32	 0 1 2 ... 65534 65535	Local port number to use with socket: Bind socket to next available port in range 1024 ... 32257 Bind socket to next available port in range 513 …1023 Bind socket to specified port Bind socket to next available port in range 513 …1023
ulProtocol	UINT32	 1 2	Protocol type to use: TCP_PROTO_TCP - Transmission Control Protocol (TCP) TCP_PROTO_UDP - User Datagram Protocol (UDP)

Table 40: TCPIP_TCP_UDP_CMD_OPEN_REQ – Request command for opening a socket

Source code example

```
#define LOCAL_PORT        (1024)        /* Local TCP/UDP port                    */


TLR_RESULT
ApTcpUdpCmdOpenReq( TCPIP_AP_TASK_RSC_T FAR*  ptRsc )
{
  TCPIP_PACKET_TCP_UDP_CMD_OPEN_REQ_T*  ptPck;


  if( TLR_POOL_PACKET_GET( ptRsc->tLoc.hPool, &ptPck ) != TLR_S_OK )
  {
    return( TLR_E_FAIL );
  }

  TLS_QUE_LINK_SET_NEW_DESTID( ptRsc->tRem.tQueTcpTask, 0 );
  TLS_QUE_LINK_SET_PACKET_SRC( ptPck, ptRsc->tLoc.tLnkSrc );

  ptPck->tHead.ulLen   = TCPIP_DATA_TCP_UDP_CMD_OPEN_REQ_SIZE;
  ptPck->tHead.ulId    = ++ptRsc->tLoc.ulSndTcpId;
  ptPck->tHead.ulSta   = 0;
  ptPck->tHead.ulCmd   = TCPIP_TCP_UDP_CMD_OPEN_REQ;
  ptPck->tHead.ulExt   = 0;
  ptPck->tHead.ulRout  = 0;

  ptPck->tData.ulIpAddr  = 0;     /* 0 = Bind socket to currently configured IP */
                                  /*     address                                */
  ptPck->tData.ulPort    = LOCAL_PORT;  /* 2 ... 65534 = Bind socket to         */
                                  /*       specified port                       */
#if defined( TCP_CLIENT ) || defined( TCP_SERVER )
  ptPck->tData.ulProtocol = TCP_PROTO_TCP;
#else
  ptPck->tData.ulProtocol = TCP_PROTO_UDP;
#endif

  if( TLS_QUE_SENDPACKET_FIFO( ptRsc->tRem.tQueTcpTask,
                               ptPck,
                               100                    ) != TLR_S_OK )
  {
    TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool, ptPck );
    return( TLR_E_FAIL );
  }

  return( TLR_S_OK );
}
```

Packet structure

```
typedef struct TCPIP_DATA_TCP_UDP_CMD_OPEN_CNF_Ttag
{
  TLR_UINT32  ulIpAddr;
  TLR_UINT32  ulPort;
  TLR_UINT32  ulProtocol;

} TCPIP_DATA_TCP_UDP_CMD_OPEN_CNF_T;

#define TCPIP_DATA_TCP_UDP_CMD_OPEN_CNF_SIZE \
                            (sizeof(TCPIP_DATA_TCP_UDP_CMD_OPEN_CNF_T))

typedef struct TCPIP_PACKET_TCP_UDP_CMD_OPEN_CNF_Ttag
{
  TLR_PACKET_HEADER_T                tHead;
  TCPIP_DATA_TCP_UDP_CMD_OPEN_CNF_T  tData;

} TCPIP_PACKET_TCP_UDP_CMD_OPEN_CNF_T;
```

Packet description

Structure TCPIP_PACKET_TCP_UDP_CMD_OPEN_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle, untouched
ulSrc	UINT32		Source queue handle, untouched
ulDestId	UINT32	1 ... 256	Destination End Point Identifier, socket handle *ulSocket*
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, untouched
ulLen	UINT32	12	TCPIP_DATA_TCP_UDP_CMD_OPEN_CNF_SIZE - Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification, untouched
ulSta	UINT32		See section *Status/error codes* (page 117).
ulCmd	UINT32	0x301	TCPIP_TCP_UDP_CMD_OPEN_CNF - Command
ulExt	UINT32	0	Extension, untouched
ulRout	UINT32	0	Routing, do not touch
Data - Structure TCPIP_DATA_TCP_UDP_CMD_OPEN_CNF_T			
ulIpAddr	UINT32		Local IP address assigned to this socket
ulPort	UINT32	2 ... 65534	Local port number actually assigned to this socket
ulProtocol	UINT32	 1 2	Protocol type used with this socket: TCP_PROTO_TCP - Transmission Control Protocol (TCP) TCP_PROTO_UDP - User Datagram Protocol (UDP)

Table 41: TCPIP_TCP_UDP_CMD_OPEN_CNF – Confirmation command for opening a socket

Source code example

```
TLR_RESULT
ApTcpUdpCmdOpenCnf( TCPIP_AP_TASK_RSC_T                 FAR*  ptRsc,
                    TCPIP_PACKET_TCP_UDP_CMD_OPEN_CNF_T FAR*  ptPck )
{
  TLR_RESULT  eRslt = ptPck->tHead.ulSta;


  if( TLR_S_OK == eRslt  )
  {
    ptRsc->tLoc.ulIpAddr   = ptPck->tData.ulIpAddr;
    ptRsc->tLoc.ulPort     = ptPck->tData.ulPort;
    ptRsc->tLoc.ulProtocol = ptPck->tData.ulProtocol;
    ptRsc->tLoc.ulSocket   = ptPck->tHead.ulDestId;
  }

  TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool,ptPck );

  return( eRslt );
}
```

4.2.2 TCPIP_TCP_UDP_CMD_CLOSE_REQ/CNF - Closing a socket

The TCPIP_TCP_UDP_CMD_CLOSE command works for both TCP and UDP sockets. It will close a currently active socket, and will destroy its socket handle. If the socket was connected to a remote TCP communication partner the connection will be terminated.

The request command expects a socket handle and a timeout value to be provided. The timeout parameter applies to TCP sockets, and it will work the following way: The stack will send its connection termination request, and will then wait up to the specified time for the remote TCP stack to close the connection, too (graceful close). If the timeout is exceeded a TCP connection reset will be forced (hard close). A connection reset will always be performed if the timeout value is set to -1 (0xFFFFFFFF).

Note: Please note, that the TCPIP_TCP_UDP_CMD_CLOSE confirmation command will be delayed for 2 seconds for TCP sockets, if the local TCP stack was the one, which initiated the TCP connection termination (active close).

UDP sockets will always be closed immediately. The timeout value must be set to zero in this case.

Packet structure

```
typedef struct TCPIP_DATA_TCP_UDP_CMD_CLOSE_REQ_Ttag
{
  TLR_UINT32  ulTimeout;

} TCPIP_DATA_TCP_UDP_CMD_CLOSE_REQ_T;

#define TCPIP_DATA_TCP_UDP_CMD_CLOSE_REQ_SIZE \
                            (sizeof(TCPIP_DATA_TCP_UDP_CMD_CLOSE_REQ_T))

typedef struct TCPIP_PACKET_TCP_UDP_CMD_CLOSE_REQ_Ttag
{
  TLR_PACKET_HEADER_T                 tHead;
  TCPIP_DATA_TCP_UDP_CMD_CLOSE_REQ_T  tData;

} TCPIP_PACKET_TCP_UDP_CMD_CLOSE_REQ_T;
```

Packet description

Structure TCPIP_PACKET_TCP_UDP_CMD_CLOSE_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of TCP_UDP-task process queue
ulSrc	UINT32		Source queue handle of AP-task process queue
ulDestId	UINT32	1 ... 256	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to the socket handle *ulSocket* for request packet (Socket handle to close).
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	4	TCPIP_DATA_TCP_UDP_CMD_CLOSE_REQ_SIZE - Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0x00000000	-
ulCmd	UINT32	0x302	TCPIP_TCP_UDP_CMD_CLOSE_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
Data - Structure TCPIP_DATA_TCP_UDP_CMD_CLOSE_REQ_T			
ulTimeout	UINT32		Close timeout:
		0	Default timeout (13000 milliseconds)
		1 ... 2^{31} -1	Wait up to specified time for completion of the close operation (graceful close), if timeout is exceeded perform connection reset. (timeout in milliseconds). **Applicable to TCP sockets only**
		0xFFFFFFFF	Close socket immediately (connection reset) **Applicable to TCP sockets only**

Table 42: TCPIP_TCP_UDP_CMD_CLOSE_REQ – Request command for closing a socket

Source code example

```
TLR_RESULT
ApTcpUdpCmdCloseReq( TCPIP_AP_TASK_RSC_T FAR*  ptRsc )
{
  TCPIP_PACKET_TCP_UDP_CMD_CLOSE_REQ_T*  ptPck;


  if( TLR_POOL_PACKET_GET( ptRsc->tLoc.hPool, &ptPck ) != TLR_S_OK )
  {
    return( TLR_E_FAIL );
  }

  TLS_QUE_LINK_SET_NEW_DESTID( ptRsc->tRem.tQueTcpTask, ptRsc->tLoc.ulSocket );
  TLS_QUE_LINK_SET_PACKET_SRC( ptPck, ptRsc->tLoc.tLnkSrc );

  ptPck->tHead.ulLen   = TCPIP_DATA_TCP_UDP_CMD_CLOSE_REQ_SIZE;
  ptPck->tHead.ulId    = ++ptRsc->tLoc.ulSndTcpId;
  ptPck->tHead.ulSta   = 0;
  ptPck->tHead.ulCmd   = TCPIP_TCP_UDP_CMD_CLOSE_REQ;
  ptPck->tHead.ulExt   = 0;
  ptPck->tHead.ulRout  = 0;

  ptPck->tData.ulTimeout = 0;  /* 0 = Default timeout (13 s) */

  if( TLS_QUE_SENDPACKET_FIFO( ptRsc->tRem.tQueTcpTask,
                               ptPck,
                               100                    ) != TLR_S_OK )
  {
    TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool, ptPck );
    return( TLR_E_FAIL );
  }

  return( TLR_S_OK );
}
```

Packet structure

```
typedef struct TCPIP_PACKET_TCP_UDP_CMD_CLOSE_CNF_Ttag
{
  TLR_PACKET_HEADER_T  tHead;

} TCPIP_PACKET_TCP_UDP_CMD_CLOSE_CNF_T;
```

Packet description

Structure TCPIP_PACKET_TCP_UDP_CMD_CLOSE_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle, untouched
ulSrc	UINT32		Source queue handle, untouched
ulDestId	UINT32	1 ... 256	Destination End Point Identifier, untouched
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, untouched
ulLen	UINT32	0	Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification, untouched
ulSta	UINT32		See section *Status/error codes* (page 117).
ulCmd	UINT32	0x303	TCPIP_TCP_UDP_CMD_CLOSE_CNF - Command
ulExt	UINT32	0	Extension, untouched
ulRout	UINT32	0	Routing, do not touch

Table 43: `TCPIP_TCP_UDP_CMD_CLOSE_CNF` – Confirmation command for closing a socket

Note: The error codes `TLR_E_TCP_ERR_TIMEOUT_TCP_UDP_CMD_CLOSE` (TCP Close timeout expired), `TLR_E_TCP_ERR_DEST_UNREACHABLE_TCP_UDP_CMD_CLOSE` (Destination is unreachable), and `TLR_E_TCP_ERR_CONN_RESET` (Connection reset) rather should be treated as a notification code, because the socket is closed in these cases, too. However, if one of the other error codes is returned, the socket's state is not affected.

Source code example

```
TLR_RESULT
ApTcpUdpCmdCloseCnf( TCPIP_AP_TASK_RSC_T                  FAR*  ptRsc,
                     TCPIP_PACKET_TCP_UDP_CMD_CLOSE_CNF_T FAR*  ptPck )
{
  TLR_RESULT  eRslt = ptPck->tHead.ulSta;


  TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool,ptPck );

  return( eRslt );
}
```

4.2.3 TCPIP_TCP_UDP_CMD_CLOSE_ALL_REQ/CNF - Closing all sockets

Sending the TCPIP_TCP_UDP_CMD_CLOSE_ALL command will instruct the stack to close all sockets currently in use by the host's application. Thus, all previously obtained socket handles will become invalid.

The timeout value specified in the request command will apply to TCP sockets only. UDP sockets will always be closed immediately.

If a timeout value unequal zero is given, this timeout will be valid for all TCP sockets that have to be closed. If the timeout is set to zero, the current timeout for data send operations on the individual TCP socket will be used. TCP connections that time out during the close operation will automatically be aborted with a TCP connection reset.

A possible use of the TCPIP_TCP_UDP_CMD_CLOSE_ALL command is to free all allocated socket resources, once the application ends communication to the stack.

Packet structure

```
typedef struct TCPIP_DATA_TCP_UDP_CMD_CLOSE_ALL_REQ_Ttag
{
  TLR_UINT32  ulTimeout;

} TCPIP_DATA_TCP_UDP_CMD_CLOSE_ALL_REQ_T;

#define TCPIP_DATA_TCP_UDP_CMD_CLOSE_ALL_REQ_SIZE \
                            (sizeof(TCPIP_DATA_TCP_UDP_CMD_CLOSE_ALL_REQ_T))

typedef struct TCPIP_PACKET_TCP_UDP_CMD_CLOSE_ALL_REQ_Ttag
{
  TLR_PACKET_HEADER_T                      tHead;
  TCPIP_DATA_TCP_UDP_CMD_CLOSE_ALL_REQ_T  tData;

} TCPIP_PACKET_TCP_UDP_CMD_CLOSE_ALL_REQ_T;
```

Packet description

Structure TCPIP_PACKET_TCP_UDP_CMD_CLOSE_ALL_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of TCP_UDP-task process queue
ulSrc	UINT32		Source queue handle of AP-task process queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for request packet, means close all sockets.
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	4	TCPIP_DATA_TCP_UDP_CMD_CLOSE_ALL_REQ_SIZE - Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0x00000000	-
ulCmd	UINT32	0x304	TCPIP_TCP_UDP_CMD_CLOSE_ALL_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
Data - Structure TCPIP_DATA_TCP_UDP_CMD_CLOSE_ALL_REQ_T			
ulTimeout	UINT32	0 1 ... 2^{31} -1 0xFFFFFFFF	Close timeout: Use the individual socket's data send timeout for TCP close operation Timeout value for all TCP close operations (milliseconds) Close TCP sockets immediately (connection reset)

Table 44: TCPIP_TCP_UDP_CMD_CLOSE_ALL_REQ – Request command for closing all sockets

Source code example

```
TLR_RESULT
ApTcpUdpCmdCloseAllReq( TCPIP_AP_TASK_RSC_T FAR*  ptRsc )
{
  TCPIP_PACKET_TCP_UDP_CMD_CLOSE_ALL_REQ_T*  ptPck;


  if( TLR_POOL_PACKET_GET( ptRsc->tLoc.hPool, &ptPck ) != TLR_S_OK )
  {
    return( TLR_E_FAIL );
  }

  TLS_QUE_LINK_SET_NEW_DESTID( ptRsc->tRem.tQueTcpTask, 0 );
  TLS_QUE_LINK_SET_PACKET_SRC( ptPck, ptRsc->tLoc.tLnkSrc );

  ptPck->tHead.ulLen   = TCPIP_DATA_TCP_UDP_CMD_CLOSE_ALL_REQ_SIZE;
  ptPck->tHead.ulId    = ++ptRsc->tLoc.ulSndTcpId;
  ptPck->tHead.ulSta   = 0;
  ptPck->tHead.ulCmd   = TCPIP_TCP_UDP_CMD_CLOSE_ALL_REQ;
  ptPck->tHead.ulExt   = 0;
  ptPck->tHead.ulRout  = 0;

  ptPck->tData.ulTimeout = 0; /* Use the individual socket's data send  */
                              /* timeout for TCP close operation        */

  if( TLS_QUE_SENDPACKET_FIFO( ptRsc->tRem.tQueTcpTask,
                               ptPck,
                               100                    ) != TLR_S_OK )
  {
    TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool, ptPck );
    return( TLR_E_FAIL );
  }

  return( TLR_S_OK );
}
```

Packet structure

```
typedef struct TCPIP_PACKET_TCP_UDP_CMD_CLOSE_ALL_CNF_Ttag
{
  TLR_PACKET_HEADER_T  tHead;

} TCPIP_PACKET_TCP_UDP_CMD_CLOSE_ALL_CNF_T;
```

Packet description

Structure TCPIP_PACKET_TCP_UDP_CMD_CLOSE_ALL_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle, untouched
ulSrc	UINT32		Source queue handle, untouched
ulDestId	UINT32	0	Destination End Point Identifier, untouched
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, untouched
ulLen	UINT32	0	Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification, untouched
ulSta	UINT32		See section *Status/error codes* (page 117).
ulCmd	UINT32	0x305	TCPIP_TCP_UDP_CMD_CLOSE_ALL_CNF - Command
ulExt	UINT32	0	Extension, untouched
ulRout	UINT32	0	Routing, do not touch

Table 45: TCPIP_TCP_UDP_CMD_CLOSE_ALL_CNF – Confirmation command for closing all sockets

Note: The confirmation command returns the result code from the last socket closed in its **status** field `ulSta`. Error codes of `TLR_E_TCP_ERR_TIMEOUT_TCP_UDP_CMD_CLOSE_ALL` (TCP Close timeout expired), `TLR_E_TCP_ERR_DEST_UNREACHABLE_TCP_UDP_CMD_CLOSE_ALL` (Destination is unreachable), and `TLR_E_TCP_ERR_CONN_RESET` (Connection reset) rather should be treated as a notification code, because all sockets were closed in these cases, too. However, if one of the other error codes is returned, the command was rejected immediately and the socket's state is not affected.

Source code example

```
TLR_RESULT
ApTcpUdpCmdCloseAllCnf( TCPIP_AP_TASK_RSC_T                      FAR*  ptRsc,
                        TCPIP_PACKET_TCP_UDP_CMD_CLOSE_ALL_CNF_T FAR*  ptPck )
{
  TLR_RESULT  eRslt = ptPck->tHead.ulSta;


  TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool,ptPck );

  return( eRslt );
}
```

4.2.4 TCPIP_TCP_CMD_WAIT_CONNECT_REQ/CNF - Waiting for an Incoming TCP connection

Using the TCPIP_TCP_CMD_WAIT_CONNECT request command a socket can be put into listening state in order to wait for an incoming TCP connection. A TCP server application waiting for connection requests from remote clients may serve as an example here.

The request command requires the handle of a previously opened socket and two timeout values to be provided. The send timeout value determines a maximum wait time for future TCPIP_TCP_CMD_SEND request command on this socket. Using the connect timeout parameter a maximum time to wait for incoming connections can be given.

Usually, the TCPIP_TCP_CMD_WAIT_CONNECT request command will not be confirmed immediately. An immediate confirmation command will only be returned if invalid packet parameters are detected. In the error free case the confirmation command will be deferred until a connection request comes in. If the **status** field ulSta indicates no error, a connection could be created, and the socket is currently in established state. In any other case no connection could be established, and the socket is no longer listening.

Packet structure

```
typedef struct TCPIP_DATA_TCP_CMD_WAIT_CONNECT_REQ_Ttag
{
  TLR_UINT32  ulTimeoutSend;
  TLR_UINT32  ulTimeoutListen;

} TCPIP_DATA_TCP_CMD_WAIT_CONNECT_REQ_T;

#define TCPIP_DATA_TCP_CMD_WAIT_CONNECT_REQ_SIZE \
                            (sizeof(TCPIP_DATA_TCP_CMD_WAIT_CONNECT_REQ_T))

typedef struct TCPIP_PACKET_TCP_CMD_WAIT_CONNECT_REQ_Ttag
{
  TLR_PACKET_HEADER_T                     tHead;
  TCPIP_DATA_TCP_CMD_WAIT_CONNECT_REQ_T  tData;

} TCPIP_PACKET_TCP_CMD_WAIT_CONNECT_REQ_T;
```

Packet description

Structure TCPIP_PACKET_TCP_CMD_WAIT_CONNECT_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of TCP_UDP-task process queue
ulSrc	UINT32		Source queue handle of AP-task process queue
ulDestId	UINT32	1 ... 256	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to the socket handle *ulSocket* for request packet (Handle of socket to connect).
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	8	TCPIP_DATA_TCP_CMD_WAIT_CONNECT_REQ_SIZE - Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0x00000000	-
ulCmd	UINT32	0x306	TCPIP_TCP_CMD_WAIT_CONNECT_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
Data - Structure TCPIP_DATA_TCP_CMD_WAIT_CONNECT_REQ_T			
ulTimeoutSend	UINT32	 0 1 ... 2^{31} -1	Send timeout: Timeout for future send commands: Default timeout (31000 milliseconds) Wait up to specified time for transfer of data (time in milliseconds)
ulTimeoutListen	UINT32	 0 1 ... 2^{31} -1	Connect timeout: Timeout for wait connect command: Wait until connection comes in Wait up to specified time for incoming connection (time in milliseconds)

Table 46: `TCPIP_TCP_CMD_WAIT_CONNECT_REQ` – Request command for waiting for an incoming TCP connection

Source code example

```
TLR_RESULT
ApTcpCmdWaitConnectReq( TCPIP_AP_TASK_RSC_T FAR*  ptRsc )
{
  TCPIP_PACKET_TCP_CMD_WAIT_CONNECT_REQ_T*  ptPck;


  if( TLR_POOL_PACKET_GET( ptRsc->tLoc.hPool, &ptPck ) != TLR_S_OK )
  {
    return( TLR_E_FAIL );
  }

  TLS_QUE_LINK_SET_NEW_DESTID( ptRsc->tRem.tQueTcpTask, ptRsc->tLoc.ulSocket );
  TLS_QUE_LINK_SET_PACKET_SRC( ptPck, ptRsc->tLoc.tLnkSrc );

  ptPck->tHead.ulLen   = TCPIP_DATA_TCP_CMD_WAIT_CONNECT_REQ_SIZE;
  ptPck->tHead.ulId    = ++ptRsc->tLoc.ulSndTcpId;
  ptPck->tHead.ulSta   = 0;
  ptPck->tHead.ulCmd   = TCPIP_TCP_CMD_WAIT_CONNECT_REQ;
  ptPck->tHead.ulExt   = 0;
  ptPck->tHead.ulRout  = 0;

  ptPck->tData.ulTimeoutSend    = 0; /* 0 = Default timeout (31 s)         */
  ptPck->tData.ulTimeoutListen  = 0; /* 0 = Wait until connection comes in */

  if( TLS_QUE_SENDPACKET_FIFO( ptRsc->tRem.tQueTcpTask,
                               ptPck,
                               100                    ) != TLR_S_OK )
  {
    TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool, ptPck );
    return( TLR_E_FAIL );
  }

  return( TLR_S_OK );
}
```

Packet structure

```
typedef struct TCPIP_DATA_TCP_CMD_WAIT_CONNECT_CNF_Ttag
{
  TLR_UINT32  ulIpAddr;
  TLR_UINT32  ulPort;

} TCPIP_DATA_TCP_CMD_WAIT_CONNECT_CNF_T;

#define TCPIP_DATA_TCP_CMD_WAIT_CONNECT_CNF_SIZE \
                            (sizeof(TCPIP_DATA_TCP_CMD_WAIT_CONNECT_CNF_T))

typedef struct TCPIP_PACKET_TCP_CMD_WAIT_CONNECT_CNF_Ttag
{
  TLR_PACKET_HEADER_T                    tHead;
  TCPIP_DATA_TCP_CMD_WAIT_CONNECT_CNF_T  tData;

} TCPIP_PACKET_TCP_CMD_WAIT_CONNECT_CNF_T;
```

Packet description

Structure TCPIP_PACKET_TCP_CMD_WAIT_CONNECT_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle, untouched
ulSrc	UINT32		Source queue handle, untouched
ulDestId	UINT32	1 ... 256	Destination End Point Identifier, untouched (Handle of socket for the incoming connection)
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, untouched
ulLen	UINT32	8	TCPIP_DATA_TCP_CMD_WAIT_CONNECT_CNF_SIZE - Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification, untouched
ulSta	UINT32		See section *Status/error codes* (page 117).
ulCmd	UINT32	0x307	TCPIP_TCP_CMD_WAIT_CONNECT_CNF - Command
ulExt	UINT32	0	Extension, untouched
ulRout	UINT32	0	Routing, do not touch
Data - Structure TCPIP_DATA_TCP_CMD_WAIT_CONNECT_CNF_T			
ulIpAddr	UINT32		IP address of remote client
ulPort	UINT32	0 ... 65535	Port number of remote client

Table 47: `TCPIP_TCP_CMD_WAIT_CONNECT_CNF` – Confirmation command for waiting for an Incoming TCP connection

Source code example

```
TLR_RESULT
ApTcpCmdWaitConnectCnf( TCPIP_AP_TASK_RSC_T                     FAR*  ptRsc,
                        TCPIP_PACKET_TCP_CMD_WAIT_CONNECT_CNF_T FAR*  ptPck )
{
  TLR_RESULT  eRslt = ptPck->tHead.ulSta;


  if( TLR_S_OK == eRslt  )
  {
    ptRsc->tLoc.ulRemoteIpAddr = ptPck->tData.ulIpAddr;
    ptRsc->tLoc.ulRemotePort   = ptPck->tData.ulPort;
  }

  TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool,ptPck );

  return( eRslt );
}
```

4.2.5 TCPIP_TCP_CMD_CONNECT_REQ/CNF - Establishing a TCP connection

The TCPIP_TCP_CMD_CONNECT request command can be used to actively establish a TCP connection to a remote partner. A typical example would be a TCP client application connecting to a remote server.

The request command takes the IP address and port number of the remote station as well as the handle of a previously opened socket as parameters. Additionally, timeout values for the connection establishment process itself and for later data send commands can be given.

Packet structure

```
typedef struct TCPIP_DATA_TCP_CMD_CONNECT_REQ_Ttag
{
  TLR_UINT32  ulIpAddr;
  TLR_UINT32  ulPort;
  TLR_UINT32  ulTimeoutSend;
  TLR_UINT32  ulTimeoutConnect;

} TCPIP_DATA_TCP_CMD_CONNECT_REQ_T;

#define TCPIP_DATA_TCP_CMD_CONNECT_REQ_SIZE \
                           (sizeof(TCPIP_DATA_TCP_CMD_CONNECT_REQ_T))

typedef struct TCPIP_PACKET_TCP_CMD_CONNECT_REQ_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
  TCPIP_DATA_TCP_CMD_CONNECT_REQ_T  tData;

} TCPIP_PACKET_TCP_CMD_CONNECT_REQ_T;
```

Packet description

Structure TCPIP_PACKET_TCP_CMD_CONNECT_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of TCP_UDP-task process queue
ulSrc	UINT32		Source queue handle of AP-task process queue
ulDestId	UINT32	1 ... 256	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to the socket handle *ulSocket* for request packet (Handle of socket to connect to).
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	16	TCPIP_DATA_TCP_CMD_CONNECT_REQ_SIZE - Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0x00000000	-
ulCmd	UINT32	0x308	TCPIP_TCP_CMD_CONNECT_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
Data - Structure TCPIP_DATA_TCP_CMD_CONNECT_REQ_T			
ulIpAddr	UINT32		IP address of remote server to connect to
ulPort	UINT32	1 ... 65535	Port number of remote server to connect to
ulTimeoutSend	UINT32	 0 1 ... 2^{31} -1	Send timeout: Timeout for future send commands: Default timeout (31000 milliseconds) Wait up to specified time for successful transfer data (time in milliseconds)
ulTimeoutConnect	UINT32	 0 1 ... 2^{31} -1	Connect Timeout: Timeout for connect command Default timeout (31000 milliseconds) Wait up to specified time for connection (time in milliseconds)

Table 48: `TCPIP_TCP_CMD_CONNECT_REQ` – Request command for establishing a TCP connection

Source code example

```
#define REMOTE_IP_ADDR    (0xC0A80A6A)  /* IP address of remote host:          */
                                        /* 192.168.10.106                      */

#define REMOTE_PORT       (1028)        /* TCP/UDP port of remote host.        */


TLR_RESULT
ApTcpCmdConnectReq( TCPIP_AP_TASK_RSC_T FAR*  ptRsc )
{
  TCPIP_PACKET_TCP_CMD_CONNECT_REQ_T*  ptPck;


  if( TLR_POOL_PACKET_GET( ptRsc->tLoc.hPool, &ptPck ) != TLR_S_OK )
  {
    return( TLR_E_FAIL );
  }

  TLS_QUE_LINK_SET_NEW_DESTID( ptRsc->tRem.tQueTcpTask, ptRsc->tLoc.ulSocket );
  TLS_QUE_LINK_SET_PACKET_SRC( ptPck, ptRsc->tLoc.tLnkSrc );

  ptPck->tHead.ulLen   = TCPIP_DATA_TCP_CMD_CONNECT_REQ_SIZE;
  ptPck->tHead.ulId    = ++ptRsc->tLoc.ulSndTcpId;
  ptPck->tHead.ulSta   = 0;
  ptPck->tHead.ulCmd   = TCPIP_TCP_CMD_CONNECT_REQ;
  ptPck->tHead.ulExt   = 0;
  ptPck->tHead.ulRout  = 0;

  ptPck->tData.ulIpAddr         = REMOTE_IP_ADDR;
  ptPck->tData.ulPort           = REMOTE_PORT;
  ptPck->tData.ulTimeoutSend    = 0; /* 0 = Default timeout (31 s) */
  ptPck->tData.ulTimeoutConnect = 0; /* 0 = Default timeout (31 s) */

  if( TLS_QUE_SENDPACKET_FIFO( ptRsc->tRem.tQueTcpTask,
                               ptPck,
                               100                    ) != TLR_S_OK )
  {
    TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool, ptPck );
    return( TLR_E_FAIL );
  }

  return( TLR_S_OK );
}
```

Packet structure

```
typedef struct TCPIP_DATA_TCP_CMD_CONNECT_CNF_Ttag
{
  TLR_UINT32  ulIpAddr;
  TLR_UINT32  ulPort;

} TCPIP_DATA_TCP_CMD_CONNECT_CNF_T;

#define TCPIP_DATA_TCP_CMD_CONNECT_CNF_SIZE \
                           (sizeof(TCPIP_DATA_TCP_CMD_CONNECT_CNF_T))

typedef struct TCPIP_PACKET_TCP_CMD_CONNECT_CNF_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
  TCPIP_DATA_TCP_CMD_CONNECT_CNF_T  tData;

} TCPIP_PACKET_TCP_CMD_CONNECT_CNF_T;
```

Packet description

Structure TCPIP_PACKET_TCP_CMD_CONNECT_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle, untouched
ulSrc	UINT32		Source queue handle, untouched
ulDestId	UINT32	1 ... 256	Destination End Point Identifier, untouched (Handle of socket for the incoming connection)
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, untouched
ulLen	UINT32	8	TCPIP_DATA_TCP_CMD_CONNECT_CNF_SIZE - Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification, untouched
ulSta	UINT32		See section *Status/error codes* (page 117).
ulCmd	UINT32	0x309	TCPIP_TCP_CMD_CONNECT_CNF - Command
ulExt	UINT32	0	Extension, untouched
ulRout	UINT32	0	Routing, do not touch
Data - Structure TCPIP_DATA_TCP_CMD_CONNECT_CNF_T			
ulIpAddr	UINT32		IP address of remote server
ulPort	UINT32	1 ... 65535	Port number of remote server

Table 49: TCPIP_TCP_CMD_CONNECT_CNF – Confirmation command for establishing a TCP connection

Source code example

```
TLR_RESULT
ApTcpCmdConnectCnf( TCPIP_AP_TASK_RSC_T                FAR*  ptRsc,
                    TCPIP_PACKET_TCP_CMD_CONNECT_CNF_T FAR*  ptPck )
{
  TLR_RESULT  eRslt = ptPck->tHead.ulSta;


  if( TLR_S_OK == eRslt  )
  {
    ptRsc->tLoc.ulRemoteIpAddr = ptPck->tData.ulIpAddr;
    ptRsc->tLoc.ulRemotePort   = ptPck->tData.ulPort;
  }

  TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool,ptPck );

  return( eRslt );
}
```

4.2.6 TCPIP_TCP_CMD_SEND_REQ/CNF - Sending TCP data

The TCPIP_TCP_CMD_SEND request command can be used to transfer data to the TCP communication partner. The only required parameter besides the actual data to be transferred is a handle of a socket in established state. That means, this socket has to be connected to the remote communication partner using the TCPIP_TCP_CMD_WAIT_CONNECT or TCPIP_TCP_CMD_CONNECT commands beforehand.

Up to TCPIP_MAX_TCP_DATA_CNT (1460) bytes of data can be sent in a single packet. The request command will be confirmed with a confirmation command by the stack when the data has been acknowledged by the remote TCP/IP stack. Afterwards the next TCPIP_TCP_CMD_SEND request command can be sent to the stack (see note below). Each data block sent this way is subject to timeout supervision. If the data cannot be delivered within the time specified in the TCPIP_TCP_CMD_WAIT_CONNECT or TCPIP_TCP_CMD_CONNECT command the connection will be aborted.

Note: The application must not wait for the TCPIP_TCP_CMD_SEND confirmation command to send the next TCPIP_TCP_CMD_SEND request command. The stack can buffer up to ulQueFreeElemCnt (Startup parameter, Default = TCPIP_SRT_QUE_FREE_ELEM_CNT_DEFAULT = 128) application request commands (over all sockets!). In this case, the stack sent the data depending on the actual window size of the remote TCP/IP stack in a more efficient way.

The stack starts to send the accumulated data, as soon as a full size TCP segment can be assembled. The stack sends the data immediately, if the push flag TCP_SEND_OPT_PUSH in the options field ulOptions is set.

Packet structure

```
/* Options of packet ulOptions variable */
#define TCP_SEND_OPT_PUSH               (0x00000001)

typedef struct TCPIP_DATA_TCP_CMD_SEND_REQ_Ttag
{
  TLR_UINT32  ulOptions;
  TLR_UINT8   abData[TCPIP_MAX_TCP_DATA_CNT];

} TCPIP_DATA_TCP_CMD_SEND_REQ_T;

#define TCPIP_DATA_TCP_CMD_SEND_REQ_SIZE (sizeof(TCPIP_DATA_TCP_CMD_SEND_REQ_T) \
                                          - TCPIP_MAX_TCP_DATA_CNT             )

typedef struct TCPIP_PACKET_TCP_CMD_SEND_REQ_Ttag
{
  TLR_PACKET_HEADER_T            tHead;
  TCPIP_DATA_TCP_CMD_SEND_REQ_T  tData;

} TCPIP_PACKET_TCP_CMD_SEND_REQ_T;
```

Packet description

Structure TCPIP_PACKET_TCP_CMD_SEND_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of TCP_UDP-task process queue
ulSrc	UINT32		Source queue handle of AP-task process queue
ulDestId	UINT32	1 ... 256	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to the socket handle *ulSocket* for request packet.
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	4 + *n*	TCPIP_DATA_TCP_CMD_SEND_REQ_SIZE + *n* - Packet data length in bytes *n* is the Application data count of abData[1460] in bytes *n* = 1 … TCPIP_MAX_TCP_DATA_CNT (1460)
ulId	UINT32	0 ... 2^{32} -1	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0x00000000	-
ulCmd	UINT32	0x30A	TCPIP_TCP_CMD_SEND_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
Data - Structure TCPIP_DATA_TCP_CMD_SEND_REQ_T			
ulOptions	UINT32		Options: See Table 51: `TCPIP_TCP_CMD_SEND_REQ` – TCP send options `ulOptions`
abData[1460]	UINT8[]		Application data with length *n* (see also ulLen)

Table 50: `TCPIP_TCP_CMD_SEND_REQ` – Request command for sending TCP data

The `ulOptions` parameter holds the option data in a bit-oriented format:

Bits	Name (Bit mask)	Description
31 ... 1	Reserved	Reserved for future use
0	`TCP_SEND_OPT_PUSH (0x00000001)`	Push flag: If set, the stack send the data immediate

Table 51: `TCPIP_TCP_CMD_SEND_REQ` – TCP send options `ulOptions`

Source code example

```
TLR_RESULT
ApTcpCmdSendReq( TCPIP_AP_TASK_RSC_T FAR*  ptRsc,
                 TLR_UINT                  uiSendLen )
{
  TCPIP_PACKET_TCP_CMD_SEND_REQ_T*  ptPck;
  TLR_UINT                          uiIdx;            /* Data index     */


  if( TLR_POOL_PACKET_GET( ptRsc->tLoc.hPool, &ptPck ) != TLR_S_OK )
  {
    return( TLR_E_FAIL );
  }

  TLS_QUE_LINK_SET_NEW_DESTID( ptRsc->tRem.tQueTcpTask, ptRsc->tLoc.ulSocket );
  TLS_QUE_LINK_SET_PACKET_SRC( ptPck, ptRsc->tLoc.tLnkSrc );

  ptPck->tHead.ulLen   = TCPIP_DATA_TCP_CMD_SEND_REQ_SIZE + uiSendLen;
  ptPck->tHead.ulId    = ++ptRsc->tLoc.ulSndTcpId;
  ptPck->tHead.ulSta   = 0;
  ptPck->tHead.ulCmd   = TCPIP_TCP_CMD_SEND_REQ;
  ptPck->tHead.ulExt   = 0;
  ptPck->tHead.ulRout  = 0;

  ptPck->tData.ulOptions  = TCP_SEND_OPT_PUSH;  /* Push flag */

#ifdef SET_TEST_DATA
  for( uiIdx = 0; uiIdx < uiSendLen; uiIdx++ )
  {
    ((TCPIP_PACKET_TCP_CMD_SEND_REQ_T*) ptPck)->tData.abData[uiIdx] \
                                                = ptRsc->tLoc.bTestSndData++;
  }
#endif  /* #ifdef SET_TEST_DATA */

  if( TLS_QUE_SENDPACKET_FIFO( ptRsc->tRem.tQueTcpTask,
                               ptPck,
                               100                    ) != TLR_S_OK )
  {
    TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool, ptPck );
    return( TLR_E_FAIL );
  }

  return( TLR_S_OK );
}
```

Packet structure

```
typedef struct TCPIP_PACKET_TCP_CMD_SEND_CNF_Ttag
{
  TLR_PACKET_HEADER_T  tHead;

} TCPIP_PACKET_TCP_CMD_SEND_CNF_T;
```

Packet description

Structure TCPIP_PACKET_TCP_CMD_SEND_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle, untouched
ulSrc	UINT32		Source queue handle, untouched
ulDestId	UINT32	1 ... 256	Destination End Point Identifier, untouched
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, untouched
ulLen	UINT32	0	Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification, untouched
ulSta	UINT32		See section *Status/error codes* (page 117).
ulCmd	UINT32	0x30B	TCPIP_TCP_CMD_SEND_CNF - Command
ulExt	UINT32	0	Extension, untouched
ulRout	UINT32	0	Routing, do not touch

Table 52: TCPIP_TCP_CMD_SEND_CNF – Confirmation command for sending TCP data

Note: If the error code TLR_E_TCP_ERR_TIMEOUT_TCP_CMD_SEND (TCP Send timeout expired) is returned, the timeout for the data send operation expired, and the connection was aborted.

Source code example

```
TLR_RESULT
ApTcpCmdSendCnf( TCPIP_AP_TASK_RSC_T             FAR*  ptRsc,
                 TCPIP_PACKET_TCP_CMD_SEND_CNF_T FAR*  ptPck )
{
  TLR_RESULT  eRslt = ptPck->tHead.ulSta;


  TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool,ptPck );

  return( eRslt );
}
```

4.2.7 TCPIP_UDP_CMD_SEND_REQ/CNF - Sending UDP data

The TCPIP_UDP_CMD_SEND_REQ command can be used to transfer data to an UDP communication partner. The target's IP address, port number, and the handle of an UDP socket opened with the TCPIP_TCP_UDP_CMD_OPEN command must be provided.

The data from this request command is sent in a UDP packet through the network, and a confirmation command is returned immediately.

Note: Please note the following limitations, when sending UDP data: Up to TCPIP_MAX_UDP_DATA_CNT (1472) bytes can be sent per UDP packet.

Packet structure

```
typedef struct TCPIP_DATA_UDP_CMD_SEND_REQ_Ttag
{
  TLR_UINT32  ulIpAddr;
  TLR_UINT32  ulPort;
  TLR_UINT32  ulOptions;
  TLR_UINT8   abData[TCPIP_MAX_UDP_DATA_CNT];

} TCPIP_DATA_UDP_CMD_SEND_REQ_T;

#define TCPIP_DATA_UDP_CMD_SEND_REQ_SIZE (sizeof(TCPIP_DATA_UDP_CMD_SEND_REQ_T) \
                                          - TCPIP_MAX_UDP_DATA_CNT             )

typedef struct TCPIP_PACKET_UDP_CMD_SEND_REQ_Ttag
{
  TLR_PACKET_HEADER_T            tHead;
  TCPIP_DATA_UDP_CMD_SEND_REQ_T  tData;

} TCPIP_PACKET_UDP_CMD_SEND_REQ_T;
```

Packet description

Structure TCPIP_PACKET_UDP_CMD_SEND_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of TCP_UDP-task process queue
ulSrc	UINT32		Source queue handle of AP-task process queue
ulDestId	UINT32	1 ... 256	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to the socket handle *ulSocket* for request packet.
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	12 + *n*	TCPIP_DATA_UDP_CMD_SEND_REQ_SIZE + *n* - Packet data length in bytes *n* is the Application data count of abData[1472] in bytes *n* = 1 … TCPIP_MAX_UDP_DATA_CNT (1472)
ulId	UINT32	0 ... 2^{32} -1	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0x00000000	-
ulCmd	UINT32	0x30C	TCPIP_UDP_CMD_SEND_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
Data - Structure TCPIP_DATA_UDP_CMD_SEND_REQ_T			
ulIpAddr	UINT32		Target IP address
ulPort	UINT32	0 ... 65535	Target Port number
ulOptions	UINT32	0	Options: Reserved for future use
abData[1472]	UINT8[]		Application data with length *n* (see also ulLen)

Table 53: `TCPIP_UDP_CMD_SEND_REQ` – Request command for sending UDP data

Source code example

```
#define REMOTE_IP_ADDR    (0xC0A80A6A)  /* IP address of remote host:           */
                                        /* 192.168.10.106                       */

#define REMOTE_PORT       (1028)        /* TCP/UDP port of remote host.         */


TLR_RESULT
ApUdpCmdSendReq( TCPIP_AP_TASK_RSC_T FAR*  ptRsc,
                 TLR_UINT                  uiSendLen )
{
  TCPIP_PACKET_UDP_CMD_SEND_REQ_T*  ptPck;
  TLR_UINT                          uiIdx;            /* Data index     */


  if( TLR_POOL_PACKET_GET( ptRsc->tLoc.hPool, &ptPck ) != TLR_S_OK )
  {
    return( TLR_E_FAIL );
  }

  TLS_QUE_LINK_SET_NEW_DESTID( ptRsc->tRem.tQueTcpTask, ptRsc->tLoc.ulSocket );
  TLS_QUE_LINK_SET_PACKET_SRC( ptPck, ptRsc->tLoc.tLnkSrc );

  ptPck->tHead.ulLen   = TCPIP_DATA_UDP_CMD_SEND_REQ_SIZE + uiSendLen;
  ptPck->tHead.ulId    = ++ptRsc->tLoc.ulSndTcpId;
  ptPck->tHead.ulSta   = 0;
  ptPck->tHead.ulCmd   = TCPIP_UDP_CMD_SEND_REQ;
  ptPck->tHead.ulExt   = 0;
  ptPck->tHead.ulRout  = 0;

  ptPck->tData.ulIpAddr   = REMOTE_IP_ADDR;
  ptPck->tData.ulPort     = REMOTE_PORT;
  ptPck->tData.ulOptions  = 0;

  for( uiIdx = 0; uiIdx < uiSendLen; uiIdx++ )  /* Set test data also without def
  {
    ((TCPIP_PACKET_UDP_CMD_SEND_REQ_T*) ptPck)->tData.abData[uiIdx] \
                                                 = ptRsc->tLoc.bTestSndData++;
  }

  if( TLS_QUE_SENDPACKET_FIFO( ptRsc->tRem.tQueTcpTask,
                               ptPck,
                               100                    ) != TLR_S_OK )
  {
    TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool, ptPck );
    return( TLR_E_FAIL );
  }

  return( TLR_S_OK );
}
```

Packet structure

```
typedef struct TCPIP_PACKET_UDP_CMD_SEND_CNF_Ttag
{
  TLR_PACKET_HEADER_T  tHead;

} TCPIP_PACKET_UDP_CMD_SEND_CNF_T;
```

Packet description

Structure TCPIP_PACKET_UDP_CMD_SEND_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle, untouched
ulSrc	UINT32		Source queue handle, untouched
ulDestId	UINT32	1 ... 256	Destination End Point Identifier, untouched
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, untouched
ulLen	UINT32	0	Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification, untouched
ulSta	UINT32		See section *Status/error codes* (page 117).
ulCmd	UINT32	0x30D	TCPIP_UDP_CMD_SEND_CNF - Command
ulExt	UINT32	0	Extension, untouched
ulRout	UINT32	0	Routing, do not touch

Table 54: TCPIP_UDP_CMD_SEND_CNF – Confirmation command for sending UDP data

Note: A confirmation command reporting no error does not guarantee that the data was successfully received by the communication partner.

The error code TLR_E_TCP_ERR_DEST_UNREACHABLE_UDP_CMD_SEND (Destination is unreachable) is returned, if the previous UDP send command was targeted to the same destination as the current one, and a destination unreachable notification was received. So the **Status** field ulSta indicates an error from the previous send command in this case. However, the data from the current UDP send command have been transmitted on the line despite of this error.

Source code example

```
TLR_RESULT
ApUdpCmdSendCnf( TCPIP_AP_TASK_RSC_T             FAR*  ptRsc,
                 TCPIP_PACKET_UDP_CMD_SEND_CNF_T FAR*  ptPck )
{
  TLR_RESULT  eRslt = ptPck->tHead.ulSta;


  TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool,ptPck );

  return( eRslt );
}
```

4.2.8 TCPIP_TCP_UDP_CMD_SET_SOCK_OPTION_REQ/CNF - Setting socket options

Socket specific parameters can be provided for each socket by sending a TCPIP_TCP_UDP_CMD_SET_SOCK_OPTION command to the TCP/IP stack.

Packet structure

```
typedef struct TCPIP_DATA_TCP_UDP_CMD_SET_SOCK_OPTION_REQ_Ttag
{
  TLR_UINT32  ulMode;

  union
  {
    TLR_UINT32  ulTtl;
    TLR_UINT32  ulTimeoutSend;
    TLR_UINT32  ulTimeoutInactive;
    TLR_UINT32  ulTimeoutKeepAlive;
    TLR_UINT32  ulMulticastGroup;
    TLR_UINT32  ulMulticastTtl;
    TLR_UINT32  ulMulticastLoop;
    TLR_UINT32  ulTos;
    struct
    {
      TLR_UINT8  bEnable;
      TLR_UINT16 usId;
      TLR_UINT8  bPriority;

    }tVlanTag;

  } unParam;

} TCPIP_DATA_TCP_UDP_CMD_SET_SOCK_OPTION_REQ_T;

#define TCPIP_DATA_TCP_UDP_CMD_SET_SOCK_OPTION_REQ_SIZE  \
              (sizeof(TCPIP_DATA_TCP_UDP_CMD_SET_SOCK_OPTION_REQ_T))

typedef struct TCPIP_PACKET_TCP_UDP_CMD_SET_SOCK_OPTION_REQ_Ttag
{
  TLR_PACKET_HEADER_T                          tHead;
  TCPIP_DATA_TCP_UDP_CMD_SET_SOCK_OPTION_REQ_T  tData;

} TCPIP_PACKET_TCP_UDP_CMD_SET_SOCK_OPTION_REQ_T;
```

Packet description

Structure TCPIP_PACKET_TCP_UDP_CMD_SET_SOCK_OPTION_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of TCP_UDP-task process queue
ulSrc	UINT32		Source queue handle of AP-task process queue
ulDestId	UINT32	1 ... 256	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to the socket handle *ulSocket* for request packet.
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	8	TCPIP_DATA_TCP_UDP_CMD_SET_SOCK_OPTION_REQ_SIZE - Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0x00000000	-
ulCmd	UINT32	0x30E	TCPIP_TCP_UDP_CMD_SET_SOCK_OPTION_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
Data - Structure TCPIP_DATA_TCP_UDP_CMD_SET_SOCK_OPTION_REQ_T			
ulMode	UINT32		Mode: Type of option: See Table 56: `TCPIP_TCP_UDP_CMD_SET_SOCK_OPTION_REQ` – Socket option data `unParam`
unParam	union		Union unParam Data required by option: See Table 56: `TCPIP_TCP_UDP_CMD_SET_SOCK_OPTION_REQ` – Socket option data `unParam`

Table 55: `TCPIP_TCP_UDP_CMD_SET_SOCK_OPTION_REQ` – Request command for setting socket options

The structure of socket option `unParam` (union in the *Packet structure*) depends on the Mode parameter `ulMode` (Default values are **bold and underlined**):

Mode `ulMode`	Description	union element of `unParam`	Data type	Value
`TCP_SOCK_TTL` (1)	Set TTL value for outgoing IP packets	`ulTtl`	UINT32	1 ... **64** … 255
`TCP_SOCK_SEND_TIMEOUT` (2)	Set new send timeout (milliseconds) **Applicable to TCP sockets only**	`ulTimeoutSend`	UINT32	**0** ... 2^{31} -1 (0 – defaults to 31000)
`TCP_SOCK_INACTIVE_TIMEOUT` (5)	Set inactivity timeout (milliseconds) **Applicable to TCP sockets only**	`ulTimeoutInactive`	UINT32	**0** ... 2^{31} -1 (0 – Timeout inactive)
`TCP_SOCK_KEEPALIVE_TIMEOUT` (6)	Set keep-alive timeout (milliseconds) **Applicable to TCP sockets only**	`ulTimeoutKeepAlive`	UINT32	**0** ... 2^{31} -1 (0 – Keep-alive inactive)
`TCP_SOCK_ADD_MEMBERSHIP` (7)	Add IP multicast group membership **Applicable to UDP sockets only**	`ulMulticastGroup`	UINT32	Multicast group address
`TCP_SOCK_DROP_MEMBERSHIP` (8)	Drop IP multicast group membership **Applicable to UDP sockets only**	`ulMulticastGroup`	UINT32	Multicast group address
`TCP_SOCK_MULTICAST_TTL` (9)	Set TTL for multicast packets **Applicable to UDP sockets only**	`ulMulticastTtl`	UINT32	**1** ... 255
`TCP_SOCK_MULTICAST_LOOP` (10)	Set loopback mode of outgoing multicast packets **Applicable to UDP sockets only**	`ulMulticastLoop`	UINT32	**0** ... 1 (0 – Loopback inactive)
`TCP_SOCK_TOS` (11)	Set TOS value for outgoing TCP and UDP packets	`ulTos`	UINT32	**0** ... 255 (See notes below!)

`TCP_SOCK_VLAN` (12)	Enable 802.1Q VLAN tagging	`tVlanTag`	`Struct`
			`bEnable`
			
			`usId`
			
			`bPriority`
				
				0,1 (Off,On)
				Vlan Id
				0 … 0xFFF
				Vlan Priority
				0 … 7

Table 56: `TCPIP_TCP_UDP_CMD_SET_SOCK_OPTION_REQ` – Socket option data `unParam`

Notes: The `TCP_SOCK_INACTIVE_TIMEOUT` and the `TCP_SOCK_KEEPALIVE_TIMEOUT` option influence each other. Only one can be active at a time. Enabling one will automatically disable the other.

If either the inactivity timeout expires or the keep-alive mechanism fails, the connection will be closed automatically. The stack will notify the application by sending a `TCPIP_TCP_UDP_CMD_RECEIVE_STOP` command containing an appropriate error code. Please

refer to section *TCPIP_TCP_UDP_CMD_RECEIVE_STOP_IND - Stop receiving of TCP data and UDP* at page *96* for a description of the command.

The TCP_SOCK_TOS option should be used **very careful**, because all values are allowed! See RFCs 791 (old meaning of this parameter), 2474 (DSCP) and 3168 (ECN). The ECN bits are not supported from the TCP/IP stack and should be set to zero at the moment.

Source code example

```
TLR_RESULT
ApTcpUdpCmdSetSockOptionsReq( TCPIP_AP_TASK_RSC_T FAR*  ptRsc )
{
  TCPIP_PACKET_TCP_UDP_CMD_SET_SOCK_OPTION_REQ_T*  ptPck;


  if( TLR_POOL_PACKET_GET( ptRsc->tLoc.hPool, &ptPck ) != TLR_S_OK )
  {
    return( TLR_E_FAIL );
  }

  TLS_QUE_LINK_SET_NEW_DESTID( ptRsc->tRem.tQueTcpTask, ptRsc->tLoc.ulSocket );
  TLS_QUE_LINK_SET_PACKET_SRC( ptPck, ptRsc->tLoc.tLnkSrc );

  ptPck->tHead.ulLen  = TCPIP_DATA_TCP_UDP_CMD_SET_SOCK_OPTION_REQ_SIZE;
  ptPck->tHead.ulId   = ++ptRsc->tLoc.ulSndTcpId;
  ptPck->tHead.ulSta  = 0;
  ptPck->tHead.ulCmd  = TCPIP_TCP_UDP_CMD_SET_SOCK_OPTION_REQ;
  ptPck->tHead.ulExt  = 0;
  ptPck->tHead.ulRout = 0;

  ptPck->tData.ulMode                = TCP_SOCK_SEND_TIMEOUT;
  ptPck->tData.unParam.ulTimeoutSend = 10000;  /* 10 s */

  if( TLS_QUE_SENDPACKET_FIFO( ptRsc->tRem.tQueTcpTask,
                               ptPck,
                               100                    ) != TLR_S_OK )
  {
    TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool, ptPck );
    return( TLR_E_FAIL );
  }

  return( TLR_S_OK );
}
```

Packet structure

```
typedef struct TCPIP_DATA_TCP_UDP_CMD_SET_SOCK_OPTION_CNF_Ttag
{
  TLR_UINT32  ulMode;

} TCPIP_DATA_TCP_UDP_CMD_SET_SOCK_OPTION_CNF_T;

#define TCPIP_DATA_TCP_UDP_CMD_SET_SOCK_OPTION_CNF_SIZE  \
              (sizeof(TCPIP_DATA_TCP_UDP_CMD_SET_SOCK_OPTION_CNF_T))

typedef struct TCPIP_PACKET_TCP_UDP_CMD_SET_SOCK_OPTION_CNF_Ttag
{
  TLR_PACKET_HEADER_T                           tHead;
  TCPIP_DATA_TCP_UDP_CMD_SET_SOCK_OPTION_CNF_T  tData;

} TCPIP_PACKET_TCP_UDP_CMD_SET_SOCK_OPTION_CNF_T;
```

Packet description

Structure TCPIP_PACKET_TCP_UDP_CMD_SET_SOCK_OPTION_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle, untouched
ulSrc	UINT32		Source queue handle, untouched
ulDestId	UINT32	1 ... 256	Destination End Point Identifier, untouched
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, untouched
ulLen	UINT32	4	TCPIP_DATA_TCP_UDP_CMD_SET_SOCK_OPTION_CNF_SIZE - Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification, untouched
ulSta	UINT32		See section *Status/error codes* (page 117).
ulCmd	UINT32	0x30F	TCPIP_TCP_UDP_CMD_SET_SOCK_OPTION_CNF - Command
ulExt	UINT32	0	Extension, untouched
ulRout	UINT32	0	Routing, do not touch
Data - Structure TCPIP_DATA_TCP_UDP_CMD_SET_SOCK_OPTION_CNF_T			
ulMode	UINT32		Mode: Type of option: See Table 58: `TCPIP_TCP_UDP_CMD_SET_SOCK_OPTION_CNF` – Parameter `ulMode`

Table 57: `TCPIP_TCP_UDP_CMD_SET_SOCK_OPTION_CNF` – Confirmation command for setting socket options

Mode `ulMode`	Description
`TCP_SOCK_TTL` (1)	Set TTL value for outgoing IP packets
`TCP_SOCK_SEND_TIMEOUT` (2)	Set new send timeout
`TCP_SOCK_INACTIVE_TIMEOUT` (5)	Set inactivity timeout
`TCP_SOCK_KEEPALIVE_TIMEOUT` (6)	Set keep-alive timeout
`TCP_SOCK_ADD_MEMBERSHIP` (7)	Add IP multicast group membership
`TCP_SOCK_DROP_MEMBERSHIP` (8)	Drop IP multicast group membership
`TCP_SOCK_MULTICAST_TTL` (9)	Set TTL for multicast packets
`TCP_SOCK_MULTICAST_LOOP` (10)	Set loopback mode of outgoing multicast packets
`TCP_SOCK_TOS` (11)	Set TOS value for outgoing TCP and UDP packets

Table 58: `TCPIP_TCP_UDP_CMD_SET_SOCK_OPTION_CNF` – Parameter `ulMode`

Source code example

```
TLR_RESULT
ApTcpUdpCmdSetSockOptionsCnf( \
                  TCPIP_AP_TASK_RSC_T                              FAR*  ptRsc,
                  TCPIP_PACKET_TCP_UDP_CMD_SET_SOCK_OPTION_CNF_T FAR*  ptPck )
{
  TLR_RESULT  eRslt = ptPck->tHead.ulSta;


  TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool,ptPck );

  return( eRslt );
}
```

4.2.9 TCPIP_TCP_UDP_CMD_GET_SOCK_OPTION_REQ/CNF - Obtaining socket options

Specific information about an individual socket can be obtained from the TCP/IP stack by sending a TCPIP_TCP_UDP_CMD_GET_SOCK_OPTION request command.

Packet structure

```
typedef struct TCPIP_DATA_TCP_UDP_CMD_GET_SOCK_OPTION_REQ_Ttag
{
  TLR_UINT32  ulMode;
} TCPIP_DATA_TCP_UDP_CMD_GET_SOCK_OPTION_REQ_T;

#define TCPIP_DATA_TCP_UDP_CMD_GET_SOCK_OPTION_REQ_SIZE  \
             (sizeof(TCPIP_DATA_TCP_UDP_CMD_GET_SOCK_OPTION_REQ_T))

typedef struct TCPIP_PACKET_TCP_UDP_CMD_GET_SOCK_OPTION_REQ_Ttag
{
  TLR_PACKET_HEADER_T                              tHead;
  TCPIP_DATA_TCP_UDP_CMD_GET_SOCK_OPTION_REQ_T  tData;
} TCPIP_PACKET_TCP_UDP_CMD_GET_SOCK_OPTION_REQ_T;
```

Packet description

Structure TCPIP_PACKET_TCP_UDP_CMD_GET_SOCK_OPTION_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of TCP_UDP-task process queue
ulSrc	UINT32		Source queue handle of AP-task process queue
ulDestId	UINT32	1 ... 256	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to the socket handle *ulSocket* for request packet.
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	4	TCPIP_DATA_TCP_UDP_CMD_GET_SOCK_OPTION_REQ_SIZE - Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0x00000000	-
ulCmd	UINT32	0x310	TCPIP_TCP_UDP_CMD_GET_SOCK_OPTION_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
Data - Structure TCPIP_DATA_TCP_UDP_CMD_GET_SOCK_OPTION_REQ_T			
ulMode	UINT32		Mode: Type of option: See Table 60: TCPIP_TCP_UDP_CMD_GET_SOCK_OPTION_REQ – Parameter ulMode

Table 59: TCPIP_TCP_UDP_CMD_GET_SOCK_OPTION_REQ – Request command for obtaining socket options

Mode ulMode	Description
TCP_SOCK_TTL (1)	Get TTL value of outgoing IP packets
TCP_SOCK_SEND_TIMEOUT (2)	Get current send timeout **Applicable to TCP sockets only**
TCP_SOCK_PROTOCOL (3)	Get protocol type
TCP_SOCK_PORT (4)	Get port number
TCP_SOCK_INACTIVE_TIMEOUT (5)	Get inactivity timeout **Applicable to TCP sockets only**
TCP_SOCK_KEEPALIVE_TIMEOUT (6)	Get keep-alive timeout **Applicable to TCP sockets only**
TCP_SOCK_MULTICAST_TTL (9)	Get TTL of multicast packets **Applicable to UDP sockets only**
TCP_SOCK_MULTICAST_LOOP (10)	Get loopback mode of outgoing multicast packets **Applicable to UDP sockets only**
TCP_SOCK_TOS (11)	Get TOS value of outgoing TCP and UDP packets

Table 60: TCPIP_TCP_UDP_CMD_GET_SOCK_OPTION_REQ – Parameter ulMode

Source code example

```
TLR_RESULT
ApTcpUdpCmdGetSockOptionsReq( TCPIP_AP_TASK_RSC_T FAR*  ptRsc )
{
  TCPIP_PACKET_TCP_UDP_CMD_GET_SOCK_OPTION_REQ_T*  ptPck;
  if( TLR_POOL_PACKET_GET( ptRsc->tLoc.hPool, &ptPck ) != TLR_S_OK )
  {
    return( TLR_E_FAIL );
  }
  TLS_QUE_LINK_SET_NEW_DESTID( ptRsc->tRem.tQueTcpTask, ptRsc->tLoc.ulSocket );
  TLS_QUE_LINK_SET_PACKET_SRC( ptPck, ptRsc->tLoc.tLnkSrc );

  ptPck->tHead.ulLen  = TCPIP_DATA_TCP_UDP_CMD_GET_SOCK_OPTION_REQ_SIZE;
  ptPck->tHead.ulId   = ++ptRsc->tLoc.ulSndTcpId;
  ptPck->tHead.ulSta  = 0;
  ptPck->tHead.ulCmd  = TCPIP_TCP_UDP_CMD_GET_SOCK_OPTION_REQ;
  ptPck->tHead.ulExt  = 0;
  ptPck->tHead.ulRout = 0;
  ptPck->tData.ulMode = TCP_SOCK_SEND_TIMEOUT;
  if( TLS_QUE_SENDPACKET_FIFO( ptRsc->tRem.tQueTcpTask,
                               ptPck,
                               100                    ) != TLR_S_OK )
  {
    TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool, ptPck );
    return( TLR_E_FAIL );
  }
  return( TLR_S_OK );
}
```

Packet structure

```
typedef struct TCPIP_DATA_TCP_UDP_CMD_GET_SOCK_OPTION_CNF_Ttag
{
  TLR_UINT32  ulMode;
  union
  {
    TLR_UINT32  ulTtl;
    TLR_UINT32  ulTimeoutSend;
    TLR_UINT32  ulProtocol;
    TLR_UINT32  ulPort;
    TLR_UINT32  ulTimeoutInactive;
    TLR_UINT32  ulTimeoutKeepAlive;
    TLR_UINT32  ulMulticastTtl;
    TLR_UINT32  ulMulticastLoop;
    TLR_UINT32  ulTos;
  } unParam;
} TCPIP_DATA_TCP_UDP_CMD_GET_SOCK_OPTION_CNF_T;
#define TCPIP_DATA_TCP_UDP_CMD_GET_SOCK_OPTION_CNF_SIZE  \
             (sizeof(TCPIP_DATA_TCP_UDP_CMD_GET_SOCK_OPTION_CNF_T))
typedef struct TCPIP_PACKET_TCP_UDP_CMD_GET_SOCK_OPTION_CNF_Ttag
{
  TLR_PACKET_HEADER_T                              tHead;
  TCPIP_DATA_TCP_UDP_CMD_GET_SOCK_OPTION_CNF_T  tData;
} TCPIP_PACKET_TCP_UDP_CMD_GET_SOCK_OPTION_CNF_T;
```

Packet description

Structure TCPIP_PACKET_TCP_UDP_CMD_GET_SOCK_OPTION_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle, untouched
ulSrc	UINT32		Source queue handle, untouched
ulDestId	UINT32	1 ... 256	Destination End Point Identifier, untouched
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, untouched
ulLen	UINT32	8	TCPIP_DATA_TCP_UDP_CMD_GET_SOCK_OPTION_CNF_SIZE - Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification, untouched
ulSta	UINT32		See section *Status/error codes* (page 117).
ulCmd	UINT32	0x311	TCPIP_TCP_UDP_CMD_GET_SOCK_OPTION_CNF - Command
ulExt	UINT32	0	Extension, untouched
ulRout	UINT32	0	Routing, do not touch
Data - Structure TCPIP_DATA_TCP_UDP_CMD_GET_SOCK_OPTION_CNF_T			
ulMode	UINT32		Mode: Type of option: See Table 62: `TCPIP_TCP_UDP_CMD_GET_SOCK_OPTION_CNF` – Socket option data `unParam`
unParam	union		Result data unParam: See Table 62: `TCPIP_TCP_UDP_CMD_GET_SOCK_OPTION_CNF` – Socket option data `unParam`

Table 61: `TCPIP_TCP_UDP_CMD_GET_SOCK_OPTION_CNF` – Confirmation command for obtaining socket options

The structure of socket option unParam (union in the *Packet structure*) depends on the Mode parameter ulMode (Default values are **bold and underlined**):

Mode ulMode	Description	Data type	Value
TCP_SOCK_TTL (1)	TTL value of outgoing IP packets	UINT32	1 ... **64** … 255
TCP_SOCK_SEND_TIMEOUT (2)	Current send timeout (milliseconds)	UINT32	1 ... **31000** … 2^{31} -1
TCP_SOCK_PROTOCOL (3)	Protocol type	UINT32	TCP_PROTO_TCP (1) TCP_PROTO_UDP (2)
TCP_SOCK_PORT (4)	Local port number	UINT32	0 ... 65535
TCP_SOCK_INACTIVE_TIMEOUT (5)	Inactivity timeout (milliseconds)	UINT32	**0** ... 2^{31} -1
TCP_SOCK_KEEPALIVE_TIMEOUT (6)	Keep-alive timeout (milliseconds)	UINT32	**0** ... 2^{31} -1
TCP_SOCK_MULTICAST_TTL (9)	TTL value of multicast packets	UINT32	**1** ... 255
TCP_SOCK_MULTICAST_LOOP (10)	Loopback mode of outgoing multicast packets	UINT32	**0** ... 1 (0 – Loopback inactive)
TCP_SOCK_TOS (11)	TOS value of outgoing TCP and UDP packets	UINT32	**0** ... 255

Table 62: TCPIP_TCP_UDP_CMD_GET_SOCK_OPTION_CNF – Socket option data unParam

Source code example

```
TLR_RESULT
ApTcpUdpCmdGetSockOptionsCnf( \
                  TCPIP_AP_TASK_RSC_T                              FAR*  ptRsc,
                  TCPIP_PACKET_TCP_UDP_CMD_GET_SOCK_OPTION_CNF_T FAR*  ptPck )
{
  TLR_RESULT  eRslt = ptPck->tHead.ulSta;


  if( TLR_S_OK == eRslt  )
  {
    if( ptPck->tData.ulMode == TCP_SOCK_SEND_TIMEOUT )
    {
      ptRsc->tLoc.ulTimeoutSend = ptPck->tData.unParam.ulTimeoutSend;
    }
    /* else if ... */
  }

  TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool,ptPck );

  return( eRslt );
}
```

4.2.10 TCPIP_TCP_UDP_CMD_RECEIVE_IND - Receiving TCP data and UDP data

Data received from the network for a TCP socket or a UDP socket will be sent to the application in an indication command according to the description below.

The indication packet itself requires no response packet to be built by the application. Using just the Macro `TLS_QUE_RETURNPACKET()` will return the packet back to the TCP_UDP-task context.

An additional command `TCPIP_TCP_UDP_CMD_RECEIVE_STOP` with **Status** field `ulSta` holding an error code will be sent to the application in case of a connection being closed or reset. This command can be treated as an end-of-data marker, because no more receive data commands from this socket will be sent to the application afterwards.

Packet structure

```
/* Options of packet ulOptions variable */
#define TCP_RECV_OPT_BROADCAST          (0x00000001)
#define TCP_RECV_OPT_MULTICAST          (0x00000002)

typedef struct TCPIP_DATA_TCP_UDP_CMD_RECEIVE_IND_Ttag
{
  TLR_UINT32  ulIpAddr;
  TLR_UINT32  ulPort;
  TLR_UINT32  ulOptions;
  TLR_UINT8   abData[TCPIP_MAX_TCP_UDP_DATA_CNT];

} TCPIP_DATA_TCP_UDP_CMD_RECEIVE_IND_T;

#define TCPIP_DATA_TCP_UDP_CMD_RECEIVE_IND_SIZE              \
              (sizeof(TCPIP_DATA_TCP_UDP_CMD_RECEIVE_IND_T)  \
               - TCPIP_MAX_TCP_UDP_DATA_CNT                )

typedef struct TCPIP_PACKET_TCP_UDP_CMD_RECEIVE_IND_Ttag
{
  TLR_PACKET_HEADER_T                   tHead;
  TCPIP_DATA_TCP_UDP_CMD_RECEIVE_IND_T  tData;

} TCPIP_PACKET_TCP_UDP_CMD_RECEIVE_IND_T;
```

Packet description

Structure TCPIP_PACKET_TCP_UDP_CMD_RECEIVE_IND_T			Type: Indication
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of AP-task process queue
ulSrc	UINT32		Source queue handle of TCP_UDP-task process queue
ulDestId	UINT32	0 ... 2^{32} -1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process.
ulSrcId	UINT32	1 ... 256	Source End Point Identifier, specifying the origin of the packet inside the Source Process. Socket handle *ulSocket* of the receiving socket.
ulLen	UINT32	12 + *n*	TCPIP_DATA_TCP_UDP_CMD_RECEIVE_IND_SIZE + *n* - Packet data length in bytes *n* is the Application data count of abData[1472] in bytes *n* = 1 … TCPIP_MAX_TCP_UDP_DATA_CNT (1472)
ulId	UINT32	0 ... 2^{32} -1	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0x00000000	-
ulCmd	UINT32	0x312	TCPIP_TCP_UDP_CMD_RECEIVE_IND - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
Data - Structure TCPIP_DATA_TCP_UDP_CMD_RECEIVE_IND_T			
ulIpAddr	UINT32		Originating IP address
ulPort	UINT32	0 ... 65535	Originating port Number
ulOptions	UINT32		Options of received data: See Table 64: `TCPIP_TCP_UDP_CMD_RECEIVE_IND` – Receive options `ulOptions`
abData[1472]	UINT8[]		Received application data with length *n* (see also ulLen)

Table 63: `TCPIP_TCP_UDP_CMD_RECEIVE_IND` – Indication command for receiving TCP data and UDP data

The `ulOptions` parameter holds the option data in a bit-oriented format:

Bits	Name (Bit mask)	Description
31 ... 2	Reserved	Reserved for future use
1	`TCP_RECV_OPT_MULTICAST` (`0x00000002`)	Multicast flag: UDP multicast telegram
0	`TCP_RECV_OPT_BROADCAST` (`0x00000001`)	Broadcast flag: UDP broadcast telegram

Table 64: `TCPIP_TCP_UDP_CMD_RECEIVE_IND` – Receive options `ulOptions`

Source code example

```
TLR_RESULT
ApTcpUdpCmdReceiveInd( TCPIP_AP_TASK_RSC_T                    FAR*  ptRsc,
                       TCPIP_PACKET_TCP_UDP_CMD_RECEIVE_IND_T FAR*  ptPck )
{
  TLR_RESULT  eRslt = ptPck->tHead.ulSta;


  if( ptPck->tHead.ulSrcId == ptRsc->tLoc.ulSocket )
  { /* Data received, copy to buffer */
    LIB_MEMCPY( &ptRsc->tLoc.abReceiveBuffer[0],
                    /* TLR_UINT8  abReceiveBuffer[TCPIP_MAX_TCP_UDP_DATA_CNT]; */
                &ptPck->tData.abData[0],
                (ptPck->tHead.ulLen - TCPIP_DATA_TCP_UDP_CMD_RECEIVE_IND_SIZE) );

    ptRsc->tLoc.uiReceiveBufferLen = (ptPck->tHead.ulLen - \
                                      TCPIP_DATA_TCP_UDP_CMD_RECEIVE_IND_SIZE);
  }
  /* else: Wrong socket handle */

  TLS_QUE_RETURNPACKET( ptPck );
  /* TLS_QUE_RETURNPACKET( ptPck );                                          */

  return( eRslt );
}
```

4.2.11 TCPIP_TCP_UDP_CMD_SHUTDOWN_IND/RES - Shutdown of the stack

In some situation the stack is required to stop using the current IP address, and thus it has to stop communicating on the network. This can be caused by a DHCP lease expiring, or by a TCPIP_IP_CMD_SET_CONFIG command being received by the stack.

In these cases, the TCP/IP stack will send a TCPIP_TCP_UDP_CMD_SHUTDOWN indication command to the application. This command will tell the application to close all sockets and afterwards return the packet back to the TCP_UDP-task. For this, the macro TLS_QUE_RETURNPACKET( ) should be used.

If the TCP/IP stack doesn't receive the response command within 30 seconds after sending the TCPIP_TCP_UDP_CMD_SHUTDOWN indication command, it will close all sockets automatically.

The TCP/IP stack will then reenter initializing state, and all further request commands will be rejected. Please refer to the section *Start-up of the TCP/IP* at page *18* for a description of the stack's initialization procedure.

Packet structure

```
typedef struct TCPIP_PACKET_TCP_UDP_CMD_SHUTDOWN_IND_Ttag
{
  TLR_PACKET_HEADER_T  tHead;

} TCPIP_PACKET_TCP_UDP_CMD_SHUTDOWN_IND_T;
```

Packet description

Structure TCPIP_PACKET_TCP_UDP_CMD_SHUTDOWN_IND_T			Type: Indication
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of AP-task process queue
ulSrc	UINT32		Source queue handle of TCP_UDP -task process queue
ulDestId	UINT32	0 ... 2^{32} -1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process.
ulSrcId	UINT32	0	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	0	Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0x00000000	-
ulCmd	UINT32	0x314	TCPIP_TCP_UDP_CMD_SHUTDOWN_IND - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons

Table 65: TCPIP_TCP_UDP_CMD_SHUTDOWN_IND – Indication command for shutdown of the stack

Source code example

```
TLR_RESULT
ApTcpUdpCmdShutdownInd( TCPIP_AP_TASK_RSC_T                     FAR*  ptRsc,
                        TCPIP_PACKET_TCP_UDP_CMD_SHUTDOWN_IND_T FAR*  ptPck )
{
  TLR_RESULT                                    eRslt;
  TCPIP_PACKET_TCP_UDP_CMD_CLOSE_ALL_CNF_T FAR* ptPckCloseAllCnf;
  TLR_BOOLEAN fCloseAllFinished = FALSE;

  eRslt = ApTcpUdpCmdCloseAllReq( ptRsc );

  if( TLR_S_OK == eRslt )
  {
    /* Command TCPIP_TCP_UDP_CMD_CLOSE_ALL_REQ send successful */
    while (fCloseAllFinished == FALSE)
    {
      eRslt = TLR_QUE_WAITFORPACKET( ptRsc->tLoc.hQue,
                                     &ptPckCloseAllCnf,
                                     TLR_TIM_TIME_TO_TICK(10000) );
      if( TLR_S_OK == eRslt )
      {
        if( ptPckCloseAllCnf->tHead.ulCmd == TCPIP_TCP_UDP_CMD_CLOSE_ALL_CNF )
        {
          ApTcpUdpCmdCloseAllCnf( ptRsc, ptPckCloseAllCnf );
          fCloseAllFinished = TRUE;
        }
        else
        {
          /* Handle other packet commands */
        }
      }
    }
  }
  TLS_QUE_RETURNPACKET( ptPck );

  return( eRslt );
}
```

4.2.12 TCPIP_TCP_UDP_CMD_RECEIVE_STOP_IND - Stop receiving of TCP data and UDP data

This command TCPIP_TCP_UDP_CMD_RECEIVE_STOP with **Status** field ulSta holding an error code will be sent to the application in case of a connection being closed or reset. This command can be treated as an end-of-data marker, because no more receive data commands from this socket will be sent to the application afterwards.

The indication packet itself requires no response packet to be built by the application. Using just the Macro TLS_QUE_RETURNPACKET( ) will return the packet back to the TCP_UDP-task context.

Note:

However, the socket is **not automatically closed** after receiving this indication. Therefore, the application has to take care that the socket is closed again.

For instance, if a TCP Client establishes a connection, transfers data and finally closes the connection, the port will remain unreachable if the application does not react to this indication with the sequence:

TCPIP_TCP_UDP_CMD_CLOSE_REQ -> (TCP/IP Stack)

TCPIP_TCP_UDP_CMD_CLOSE_CNF <- (TCP/IP Stack)

TCPIP_TCP_UDP_CMD_OPEN_REQ -> (TCP/IP Stack)

TCPIP_TCP_UDP_CMD_OPEN_CNF <- (TCP/IP Stack)

TCPIP_TCP_CMD_WAIT_CONNECT_REQ -> (TCP/IP Stack)

Packet structure

```
typedef struct TCPIP_DATA_TCP_UDP_CMD_RECEIVE_STOP_IND_Ttag
{
  TLR_UINT32  ulIpAddr;
  TLR_UINT32  ulPort;
  TLR_UINT32  ulOptions;

} TCPIP_DATA_TCP_UDP_CMD_RECEIVE_STOP_IND_T;

#define TCPIP_DATA_TCP_UDP_CMD_RECEIVE_STOP_IND_SIZE  \
              (sizeof(TCPIP_DATA_TCP_UDP_CMD_RECEIVE_STOP_IND_T))

typedef struct TCPIP_PACKET_TCP_UDP_CMD_RECEIVE_STOP_IND_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  TCPIP_DATA_TCP_UDP_CMD_RECEIVE_STOP_IND_T  tData;

} TCPIP_PACKET_TCP_UDP_CMD_RECEIVE_STOP_IND_T;
```

Packet description

Structure TCPIP_PACKET_TCP_UDP_CMD_RECEIVE_STOP_IND_T			Type: Indication
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of AP-task process queue
ulSrc	UINT32		Source queue handle of TCP_UDP -task process queue
ulDestId	UINT32	0 ... 2^{32} -1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process.
ulSrcId	UINT32	1 ... 256	Source End Point Identifier, specifying the origin of the packet inside the Source Process. Socket handle *ulSocket* of the receiving socket.
ulLen	UINT32	12	TCPIP_DATA_TCP_UDP_CMD_RECEIVE_STOP_IND_SIZE
ulId	UINT32	0 ... 2^{32} -1	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32		See section *Status/error codes* (page 117).
ulCmd	UINT32	0x316	TCPIP_TCP_UDP_CMD_RECEIVE_STOP_IND - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
Data - Structure TCPIP_DATA_TCP_UDP_CMD_RECEIVE_STOP_IND_T			
ulIpAddr	UINT32		Originating IP address
ulPort	UINT32	0 ... 65535	Originating port Number
ulOptions	UINT32		Options of received data: See Table 67: `TCPIP_TCP_UDP_CMD_RECEIVE_STOP_IND` – Receive options `ulOptions`

Table 66: `TCPIP_TCP_UDP_CMD_RECEIVE_STOP_IND` – Indication command for stop receiving of TCP data and UDP data

The `ulOptions` parameter holds the option data in a bit-oriented format:

Bits	Name (Bit mask)	Description
31 ... 0	Reserved	Reserved for future use

Table 67: `TCPIP_TCP_UDP_CMD_RECEIVE_STOP_IND` – Receive options `ulOptions`

Source code example

```
TLR_RESULT
ApTcpUdpCmdReceiveStopInd(
                          TCPIP_AP_TASK_RSC_T                          FAR*  ptRsc,
                          TCPIP_PACKET_TCP_UDP_CMD_RECEIVE_STOP_IND_T FAR*  ptPck )
{
  TLR_RESULT  eRslt = ptPck->tHead.ulSta;


  if( ptPck->tHead.ulSrcId == ptRsc->tLoc.ulSocket )
  { /* End-of-data marker received - no further data receive */
    if( ptPck->tHead.ulSta == TLR_E_TCP_ERR_CONN_CLOSED )
    { /* Graceful close */
    }
    else if( ptPck->tHead.ulSta == TLR_E_TCP_ERR_CONN_RESET )
    { /* Hard close */
    }
  }
  /* else: Wrong socket handle */

  TLS_QUE_RETURNPACKET( ptPck );

  return( eRslt );
}
```

4.3 ICMP services

4.3.1 TCPIP_IP_CMD_PING_REQ/CNF - Sending a ping

The TCPIP_IP_CMD_PING command can be used to send an ICMP Echo Request packet (Internet Control Message Protocol) to the specified IP address. The target IP stack should answer with an ICMP Echo Reply packet. This command is similar to the commonly known “ping” program command.

Packet structure

```
typedef struct TCPIP_DATA_IP_CMD_PING_REQ_Ttag
{
  TLR_UINT32  ulIpAddr;
  TLR_UINT32  ulTimeout;
} TCPIP_DATA_IP_CMD_PING_REQ_T;
#define TCPIP_DATA_IP_CMD_PING_REQ_SIZE (sizeof(TCPIP_DATA_IP_CMD_PING_REQ_T))

typedef struct TCPIP_PACKET_IP_CMD_PING_REQ_Ttag
{
  TLR_PACKET_HEADER_T           tHead;
  TCPIP_DATA_IP_CMD_PING_REQ_T  tData;
} TCPIP_PACKET_IP_CMD_PING_REQ_T;
```

Packet description

Structure TCPIP_PACKET_IP_CMD_PING_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of TCP_UDP-task process queue
ulSrc	UINT32		Source queue handle of AP-task process queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for TCPIP_IP_xx packets.
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	8	TCPIP_DATA_IP_CMD_PING_REQ_SIZE - Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0x00000000	-
ulCmd	UINT32	0x20A	TCPIP_IP_CMD_PING_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
Data - Structure TCPIP_DATA_IP_CMD_PING_REQ_T			
ulIpAddr	UINT32		IP address of the target system
ulTimeout	UINT32	0 1 ... 2^{31} -1	Timeout for request: Return immediately, don’t wait for an ICMP Echo Reply packet Wait up to specified time for return of an ICMP Echo Reply packet (time in milliseconds)

Table 68: TCPIP_IP_CMD_PING_REQ – Request command for sending a ping

Source code example

```
#define REMOTE_IP_ADDR    (0xC0A80A6A)  /* IP address of remote host:          */
                                        /* 192.168.10.106                      */

TLR_RESULT
ApIpCmdPingReq( TCPIP_AP_TASK_RSC_T FAR*  ptRsc )
{
  TCPIP_PACKET_IP_CMD_PING_REQ_T*  ptPck;


  if( TLR_POOL_PACKET_GET( ptRsc->tLoc.hPool, &ptPck ) != TLR_S_OK )
  {
    return( TLR_E_FAIL );
  }

  TLS_QUE_LINK_SET_NEW_DESTID( ptRsc->tRem.tQueTcpTask, 0 );
  TLS_QUE_LINK_SET_PACKET_SRC( ptPck, ptRsc->tLoc.tLnkSrc );

  ptPck->tHead.ulLen   = TCPIP_DATA_IP_CMD_PING_REQ_SIZE;
  ptPck->tHead.ulId    = ++ptRsc->tLoc.ulSndTcpId;
  ptPck->tHead.ulSta   = 0;
  ptPck->tHead.ulCmd   = TCPIP_IP_CMD_PING_REQ;
  ptPck->tHead.ulExt   = 0;
  ptPck->tHead.ulRout  = 0;

  ptPck->tData.ulIpAddr  = REMOTE_IP_ADDR;
  ptPck->tData.ulTimeout = 1000;      /* 1 s           */

  if( TLS_QUE_SENDPACKET_FIFO( ptRsc->tRem.tQueTcpTask,
                               ptPck,
                               100                    ) != TLR_S_OK )
  {
    TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool, ptPck );
    return( TLR_E_FAIL );
  }

  return( TLR_S_OK );
}
```

Packet structure

```
typedef struct TCPIP_DATA_IP_CMD_PING_CNF_Ttag
{
  TLR_UINT32  ulResponseTime;

} TCPIP_DATA_IP_CMD_PING_CNF_T;

#define TCPIP_DATA_IP_CMD_PING_CNF_SIZE (sizeof(TCPIP_DATA_IP_CMD_PING_CNF_T))

typedef struct TCPIP_PACKET_IP_CMD_PING_CNF_Ttag
{
  TLR_PACKET_HEADER_T           tHead;
  TCPIP_DATA_IP_CMD_PING_CNF_T  tData;

} TCPIP_PACKET_IP_CMD_PING_CNF_T;
```

Packet description

Structure TCPIP_PACKET_IP_CMD_PING_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle, untouched
ulSrc	UINT32		Source queue handle, untouched
ulDestId	UINT32	0	Destination End Point Identifier, untouched
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, untouched
ulLen	UINT32	4	TCPIP_DATA_IP_CMD_PING_CNF_SIZE - Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification, untouched
ulSta	UINT32		See section *Status/error codes* (page 117).
ulCmd	UINT32	0x20B	TCPIP_IP_CMD_PING_CNF - Command
ulExt	UINT32	0	Extension, untouched
ulRout	UINT32	0	Routing, do not touch
Data - Structure TCPIP_DATA_IP_CMD_PING_CNF_T			
ulResponseTime	UINT32		Response time of the Ping answer (milliseconds)

Table 69: TCPIP_IP_CMD_PING_CNF – Confirmation command for sending a ping

Source code example

```
TLR_RESULT
ApIpCmdPingCnf( TCPIP_AP_TASK_RSC_T            FAR*  ptRsc,
                TCPIP_PACKET_IP_CMD_PING_CNF_T FAR*  ptPck )
{
  TLR_RESULT  eRslt = ptPck->tHead.ulSta;


  if( TLR_S_OK == eRslt  )
  {
    ptRsc->tLoc.ulResponseTime = ptPck->tData.ulResponseTime;
  }

  TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool,ptPck );

  return( eRslt );
}
```

4.3.2 TCPIP_IP_CMD_ICMP_IND – ICMP indication has been received

This command TCPIP_IP_CMD_ICMP_IND will be sent to an application in case of an ICMP request has been received by the TcpIp stack. To receive this indication the application needs to register it self using the command TCPIP_IP_CMD_SET_PARAM_REQ with mode IP_PRM_REGISTER_ICMP_APP set.

The indication packet itself requires no response packet to be built by the application. Using just the Macro TLS_QUE_RETURNPACKET( ) will return the packet back to the TCP_UDP-task context.

Packet structure

```
typedef __TCPIP_PACKED_PRE struct TCPIP_DATA_IP_CMD_ICMP_IND_Ttag
{
  TLR_UINT8   bType;
  TLR_UINT8   bCode;
  TLR_UINT16  usChecksum;
  TLR_UINT8   abData[TCPIP_MAX_TCP_UDP_DATA_CNT];

} __TCPIP_PACKED_POST TCPIP_DATA_IP_CMD_ICMP_IND_T;

#define TCPIP_DATA_IP_CMD_ICMP_IND_SIZE (sizeof(TCPIP_DATA_IP_CMD_ICMP_IND_T))

typedef struct TCPIP_PACKET_IP_CMD_ICMP_IND_Ttag
{
  TLR_PACKET_HEADER_T           tHead;
  TCPIP_DATA_IP_CMD_ICMP_IND_T  tData;

} TCPIP_PACKET_IP_CMD_ICMP_IND_T;
```

Packet description

Structure TCPIP_PACKET_IP_CMD_ICMP_IND_T			Type: Indication
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of AP-task process queue
ulSrc	UINT32		Source queue handle of TCP_UDP -task process queue
ulDestId	UINT32	0 ... 2^{32} -1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process.
ulSrcId	UINT32	1 ... 256	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	4 + n	*n* is the Application data count of abData[1472] in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0x00000000	-
ulCmd	UINT32	0x20C	TCPIP_IP_CMD_ICMP_IND - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
Data - Structure TCPIP_DATA_IP_CMD_ICMP_IND_T			
bType	UINT8		ICMP Type field
bCode	UINT8		further specification of the ICMP type
usChecksum	UINT16		This field contains error checking data calculated from the ICMP header+data
abData[]	UINT8[]		Data of the ICMP request

Table 70: `TCPIP_IP_CMD_ICMP_IND` – *Indication command for ICMP*

4.4 Information services

4.4.1 TCPIP_TCP_UDP_CMD_ACD_CONFLICT_IND – Address conflict occurred

This command TCPIP_TCP_UDP_CMD_ACD_CONFLICT_IND will be sent to an application in case of an ip address conflict being detected. To receive this indication, the application needs to register it self using the command TCPIP_IP_CMD_SET_PARAM_REQ with mode IP_PRM_REGISTER_ACD_APP set.

The indication packet itself requires no response packet to be built by the application. Using just the Macro TLS_QUE_RETURNPACKET( ) will return the packet back to the TCP_UDP-task context.

Packet structure

```
/* Address conflict reasons */

#define TCPIP_ACD_CONFL_RSN_DEFAULT       1
#define TCPIP_ACD_CONFL_RSN_DEFENDED_IP   2

enum TCPIP_ACD_CONFLICT_EVENT_E
{
  TCPIP_ACD_CONFLICT_EVENT_CONFLICT_OCCURRED = TCPIP_ACD_CONFL_RSN_DEFAULT,
  TCPIP_ACD_CONFLICT_EVENT_ADDRESS_DEFENDED  = TCPIP_ACD_CONFL_RSN_DEFENDED_IP,
  TCPIP_ACD_CONFLICT_EVENT_CONFLICT_RESOLVED = TCPIP_ACD_CONFL_RSN_DEFENDED_IP + 1,
};

/* ACD States --> bAcdActivity */

#define TCPIP_ACD_CONFLICT_STATE_PROBING           1
#define TCPIP_ACD_CONFLICT_STATE_ONGOING_DETECTION 2
#define TCPIP_ACD_CONFLICT_STATE_SEMI_ACTIVE       3

/* Structure of data area of an Ethernet ARP packet */
typedef struct
{
  TLR_UINT16   usHardwAddrType;
  TLR_UINT16   usProtocolType;
  TLR_UINT8    usHardwSize;
  TLR_UINT8    usProtocolSize;
  TLR_UINT16   usOpCode;
  TLR_UINT8    abSenderMacAddr[6];
  TLR_UINT32   ulSenderIpAddr;
  TLR_UINT8    abTargetMacAddr[6];
  TLR_UINT32   ulTargetIpAddr;

} TCPIP_ARP_PACKET;

Typedef struct TCPIP_DATA_TCP_UDP_CMD_ACD_CONFLICT_IND_Ttag
{
  TLR_UINT32        ulReason;
  TLR_UINT8         bAcdActivity;
  TCPIP_ARP_PACKET  tLastConflictArp;
} TCPIP_DATA_TCP_UDP_CMD_ACD_CONFLICT_IND_T;

#define TCPIP_DATA_TCP_UDP_CMD_ACD_CONFLICT_IND_SIZE  \
              (sizeof(TCPIP_DATA_TCP_UDP_CMD_ACD_CONFLICT_IND_T))
typedef struct TCPIP_PACKET_TCP_UDP_CMD_ACD_CONFLICT_IND_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  TCPIP_DATA_TCP_UDP_CMD_ACD_CONFLICT_IND_T  tData;
} TCPIP_PACKET_TCP_UDP_CMD_ACD_CONFLICT_IND_T;
```

Packet description

Structure TCPIP_PACKET_TCP_UDP_CMD_ACD_CONFLICT_IND_T			Type: Indication
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of AP-task process queue
ulSrc	UINT32		Source queue handle of TCP_UDP -task process queue
ulDestId	UINT32	0 ... 2^{32} -1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process.
ulSrcId	UINT32	1 ... 256	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	33	TCPIP_DATA_TCP_UDP_CMD_ACD_CONFLICT_IND_SIZE
ulId	UINT32	0 ... 2^{32} -1	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0x00000000	-
ulCmd	UINT32	0x31A	TCPIP_TCP_UDP_CMD_ACD_CONFLICT_IND - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
Data - Structure TCPIP_DATA_TCP_UDP_CMD_ACD_CONFLICT_IND_T			
ulReason	UINT32	1, 2, 3	This field holds the reason why this indication has been sent to the application. 1: TCPIP_ACD_CONFLICT_EVENT_CONFLICT_OCCURRED An IP address conflict occurred. The TCP/IP stack ceases using the IP address. 2: TCPIP_ACD_CONFLICT_EVENT_ADDRESS_DEFENDED Tcp stack defended its IP address (IP address is still in use). 3: TCPIP_ACD_CONFLICT_EVENT_CONFLICT_RESOLVED The address conflict has been resolved.
bAcdActivity	UINT8	1, 2, 3	1: TCPIP_ACD_CONFLICT_STATE_PROBING Conflict occurred in state “Probing” (Active Phase) 2: TCPIP_ACD_CONFLICT_STATE_ONGOING_DETECTION Conflict occurred in state “Ongoing Detection” (Passive Phase) 3: TCPIP_ACD_CONFLICT_STATE_SEMI_ACTIVE Conflict occurred in state “Semi active probing” (Semi-active Phase)
tLastConflictArp	Struct TCPIP_ARP_PACKET		ARP packet from conflict partner

Table 71: `TCPIP_TCP_UDP_CMD_ACD_CONFLICT_IND` – Indication command for an address conflict

Structure TCPIP_ARP_PACKET			
Variable	**Type**	**Value / Range**	**Description**
usHardwAddrType	UINT16	1	Hardware address type
usProtocolType	UINT16	0x0800	Protocol type
usHardwSize	UINT16	6	Hardware address protocol size
usProtocolSize	UINT16	4	Protocol size
usOpCode	UINT16	1,2	Op-Code 1: Request 2: Response
abSenderMacAddr[6]	UINT8		Sender MAC address
ulSenderIpAddr	UINT32		Sender IP address
abTargetMacAddr[6]	UINT8		Target MAC address
ulTargetIpAddr	UINT32		Target IP address

Table 72: `TCPIP_ARP_PACKET` – ARP packet

4.4.2 TCPIP_IP_CMD_GET_OPTIONS_REQ/CNF - Obtaining TCP/IP stack capabilities

The TCPIP_IP_CMD_GET_OPTIONS command instructs the IP layer to return information about supported protocols to the application.

Packet structure

```
typedef struct TCPIP_PACKET_IP_CMD_GET_OPTIONS_REQ_Ttag
{
  TLR_PACKET_HEADER_T  tHead;
} TCPIP_PACKET_IP_CMD_GET_OPTIONS_REQ_T;
```

Packet description

Structure TCPIP_PACKET_IP_CMD_GET_OPTIONS_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of TCP_UDP-task process queue
ulSrc	UINT32		Source queue handle of AP-task process queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for TCPIP_IP_xx packets.
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	0	Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0x00000000	-
ulCmd	UINT32	0x208	TCPIP_IP_CMD_GET_OPTIONS_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons

Table 73: TCPIP_IP_CMD_GET_OPTIONS_REQ – Request command for obtaining TCP/IP stack capabilities

Source code example

```
TLR_RESULT
ApIpCmdGetOptionsReq( TCPIP_AP_TASK_RSC_T FAR*  ptRsc )
{
  TCPIP_PACKET_IP_CMD_GET_OPTIONS_REQ_T*  ptPck;


  if( TLR_POOL_PACKET_GET( ptRsc->tLoc.hPool, &ptPck ) != TLR_S_OK )
  {
    return( TLR_E_FAIL );
  }

  TLS_QUE_LINK_SET_NEW_DESTID( ptRsc->tRem.tQueTcpTask, 0 );
  TLS_QUE_LINK_SET_PACKET_SRC( ptPck, ptRsc->tLoc.tLnkSrc );

  ptPck->tHead.ulLen   = 0;
  ptPck->tHead.ulId    = ++ptRsc->tLoc.ulSndTcpId;
  ptPck->tHead.ulSta   = 0;
  ptPck->tHead.ulCmd   = TCPIP_IP_CMD_GET_OPTIONS_REQ;
  ptPck->tHead.ulExt   = 0;
  ptPck->tHead.ulRout  = 0;

  if( TLS_QUE_SENDPACKET_FIFO( ptRsc->tRem.tQueTcpTask,
                               ptPck,
                               100                    ) != TLR_S_OK )
  {
    TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool, ptPck );
    return( TLR_E_FAIL );
  }

  return( TLR_S_OK );
}
```

Packet structure

```
/* Valid options of packet ulOptions variable */
#define IP_OPT_PROTO_TCP               (0x00000001)
#define IP_OPT_PROTO_UDP               (0x00000002)
#define IP_OPT_BOOTP                   (0x00000004)
#define IP_OPT_DHCP                    (0x00000008)
#define IP_OPT_MULTICAST               (0x00000010)

typedef struct TCPIP_DATA_IP_CMD_GET_OPTIONS_CNF_Ttag
{
  TLR_UINT32  ulOptions;

} TCPIP_DATA_IP_CMD_GET_OPTIONS_CNF_T;

#define TCPIP_DATA_IP_CMD_GET_OPTIONS_CNF_SIZE \
                           (sizeof(TCPIP_DATA_IP_CMD_GET_OPTIONS_CNF_T))

typedef struct TCPIP_PACKET_IP_CMD_GET_OPTIONS_CNF_Ttag
{
  TLR_PACKET_HEADER_T                  tHead;
  TCPIP_DATA_IP_CMD_GET_OPTIONS_CNF_T  tData;

} TCPIP_PACKET_IP_CMD_GET_OPTIONS_CNF_T;
```

Packet description

Structure TCPIP_PACKET_IP_CMD_GET_OPTIONS_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
Head - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle, untouched
ulSrc	UINT32		Source queue handle, untouched
ulDestId	UINT32	0	Destination End Point Identifier, untouched
ulSrcId	UINT32	0 ... 2^{32} -1	Source End Point Identifier, untouched
ulLen	UINT32	4	TCPIP_DATA_IP_CMD_GET_OPTIONS_CNF_SIZE - Packet data length in bytes
ulId	UINT32	0 ... 2^{32} -1	Packet identification, untouched
ulSta	UINT32		See section *Status/error codes* (page 117).
ulCmd	UINT32	0x209	TCPIP_IP_CMD_GET_OPTIONS_CNF - Command
ulExt	UINT32	0	Extension, untouched
ulRout	UINT32	0	Routing, do not touch
Data - Structure TCPIP_DATA_IP_CMD_GET_OPTIONS_CNF_T			
ulOptions	UINT32		Options supported by TCP/IP stack: See Table 75: `TCPIP_IP_CMD_GET_OPTIONS_CNF` – Supported options `ulOptions`

Table 74: `TCPIP_IP_CMD_GET_OPTIONS_CNF` – Confirmation command for obtaining TCP/IP stack capabilities

The `ulOptions` parameter holds the option data in a bit-oriented format:

Bits	Name (Bit mask)	Description
31 ... 5	Reserved	Reserved for future use
4	`IP_OPT_MULTICAST (0x00000010)`	IP Multicast supported
3	`IP_OPT_DHCP (0x00000008)`	DHCP supported
2	`IP_OPT_BOOTP (0x00000004)`	BOOTP supported
1	`IP_OPT_PROTO_UDP (0x00000002)`	UDP supported
0	`IP_OPT_PROTO_TCP (0x00000001)`	TCP supported

Table 75: `TCPIP_IP_CMD_GET_OPTIONS_CNF` – Supported options `ulOptions`

Source code example

```
TLR_RESULT
ApIpCmdGetOptionsCnf( TCPIP_AP_TASK_RSC_T                   FAR*  ptRsc,
                      TCPIP_PACKET_IP_CMD_GET_OPTIONS_CNF_T FAR*  ptPck )
{
  TLR_RESULT  eRslt = ptPck->tHead.ulSta;


  if( TLR_S_OK == eRslt  )
  {
    ptRsc->tLoc.ulOptions = ptPck->tData.ulOptions;
  }

  TLR_POOL_PACKET_RELEASE( ptRsc->tLoc.hPool,ptPck );

  return( eRslt );
}
```

5 Special topics

5.1 TCP_UDP task

The TCP_UDP task is the main task of the TCP/IP stack. It is responsible for all application interactions and represents the counterpart of the AP task within the existent TCP/IP stack implementation.

To get the handle of the process queue of the TCP_UDP-task the macro `TLS_QUE_IDENTIFY()` has to be used in conjunction with the following ASCII queue name.

ASCII queue name	Description
"EN_TCPUDP_QUE"	Name of the TCP_UDP task process queue

Table 76: TCP_UDP task process queue

The returned handle is a structure from type `TLR_QUE_LINK_T` and has to be used as queue handle in conjunction with the macros like `TLS_QUE_SENDPACKET_FIFO/LIFO()` for sending a packet to the TCP_UDP task. A source code example for understanding:

```
#define TCP_TASK_NAME                   "TCP_UDP"
#define EN_TCPUDP_PROCESS_QUEUE_NAME    "EN_TCPUDP_QUE"

  /*** Remote Resources from TCP_UDP task ***/

  /* Task TCP_UDP */
  eRslt = TLR_TSK_IDENTIFY( TCP_TASK_NAME,                /* task name      */
                            ptRsc->tLoc.uTskInst,         /* task instance  */
                            &ptRsc->tRem.hTskTcpTask,     /* task handle    */
                            &ptRsc->tRem.uToknTcpTask,    /* task token     */
                            &ptRsc->tRem.uPrioTcpTask );  /* task priority  */
  if( TLR_S_OK != eRslt )
  {
    return eRslt;  /* Error */
  }

  eRslt = TLS_QUE_IDENTIFY( EN_TCPUDP_PROCESS_QUEUE_NAME,   /* queue name     */
                            ptRsc->tLoc.uTskInst,           /* task instance  */
                            &ptRsc->tRem.tQueTcpTask     ); /* queue handle   */
  if( TLR_S_OK != eRslt )
  {
    return eRslt;  /* Error */
  }

TLS_QUE_LINK_SET_NEW_DESTID( ptRsc->tRem.tQueTcpTask, 0 );
                       /* ( tQueLink               , ulDestId ) */

TLS_QUE_SENDPACKET_FIFO( ptRsc->tRem.tQueTcpTask,…
```

Remark: The macro `TLS_QUE_LINK_SET_NEW_DESTID()` set `ulDestId` for all further calls of `TLS_QUE_SENDPACKET_FIFO()`. The parameter `ulDestId` is

- 0 for IP layer commands (TCPIP_IP_xx) and command TCPIP_TCP_UDP_CMD_CLOSE_ALL_REQ
- *ulSocket* (socket handle) for socket-based commands (`TCPIP_TCP/UDP_xx`). *ulSocket* is the `ulDestId` parameter from confirmation command `TCPIP_TCP_UDP_CMD_OPEN_CNF`.

5.2 TCP/IP startup parameters

This chapter provides information for users of linkable object modules (LOM).

The startup parameter structure has to be used to parameterize the TCP/IP stack at compiler link time. These parameters are defined in the rcX configuration file (for example `Config_netX.c`).

The following startup parameter structure is declared in header file TcpipTcpTask_Functionlist.h. The meaning of the parameters is explained directly in the structure below. See also the following *Startup Parameter Limits*.

Note: The actual version of the startup parameters `ulParamVersion` is `TCPIP_STARTUPPARAMETER_VERSION_6` (6). Furthermore, the task identifier `ulTaskIdentifier` must be `TLR_TASK_TCPUDP`. See also the following source code example.

Startup parameter structure

```
typedef struct TCPIP_TCP_TASK_STARTUPPARAMETER_Ttag
{
  TLR_UINT32  ulTaskIdentifier;     /* task identifier see TLR_TaskIdentifier.h */
  TLR_UINT32  ulParamVersion;       /* structure version                        */

  /*** Queue, pool element sizes ***/
  TLR_UINT32    ulQueElemCntAp; /* TCP/IP stacks process queue size for AP      */
                        /* packets                                            */
                        /* Remark: The real queue size is this value plus     */
                        /*         startup-parameter ulEddQuePoolElemCnt (EDD)*/
                        /*         plus 3 (reserve)                           */
                        /*         (ulQueElemCntAp + ulEddQuePoolElemCnt + 3) */
                        /* Range: TCPIP_SRT_QUE_ELEM_CNT_AP_MIN ...           */
                        /*        TCPIP_SRT_QUE_ELEM_CNT_AP_MAX               */

  TLR_UINT32    ulPoolElemCnt;  /* Size of pool elements for indication packets */
                        /* to AP. One pool element allocates (approx.) 1524   */
                        /* bytes                                              */
                        /* Range: TCPIP_SRT_POOL_ELEM_CNT_MIN  ...            */
                        /*        TCPIP_SRT_POOL_ELEM_CNT_MAX                 */


  /*** TCP/IP Stacks configuration ***/
  TLR_UINT32    ulStartFlags;   /* Start flags (see TCPIP_SRT_FLAG_xx above)    */

  TLR_UINT32    ulTcpCycleEvent;  /* Cycletime of "TCP_UDP" task in ms - call   */
                        /* intervall of functions like IpTick(),              */
                        /* tcp_Retransmitter, UdpTick(), TimerTick(), ...     */
                        /* Must be greater or equal the OS cycletime          */
                        /* ptRsc->tLoc.ulTcpOsCycleTime!                      */
                        /* Range: TCPIP_SRT_TCP_CYCLE_EVENT_MIN ...           */
                        /*        TCPIP_SRT_TCP_CYCLE_EVENT_MAX               */

  TLR_UINT32    ulQueFreeElemCnt; /* Count of free queue elements (module       */
                        /* TcpipQueue.c) This is the list of free queue       */
                        /* elements to hold AP requests temporarily over all  */
                        /* sockets.                                           */
                        /* Range: TCPIP_SRT_QUE_FREE_ELEM_CNT_MIN ...         */
                        /*        TCPIP_SRT_QUE_FREE_ELEM_CNT_MAX             */

  TLR_UINT32    ulSocketMaxCnt; /* Count of sockets, the TCP/IP stack allocates */
                        /* fix while startup-sequence. This socket count can  */
                        /* be used simultaneous for TCP or UDP communication. */
                        /* Every socket allocates sizeof(tcp_Socket) (2236)   */
                        /* bytes. The most space needs the receive buffer     */
                        /* abRecvBuf[TCP_RECV_BUF_SIZE] with 2048 bytes       */
                        /* Range: TCPIP_SRT_SOCKET_MAX_CNT_MIN ...            */
                        /*        TCPIP_SRT_SOCKET_MAX_CNT_MAX                */
```

```
  TLR_UINT32    ulArpCacheSize; /* Number of entries in ARP cache, must be      */
                                /* multiple of 16                               */
                                /* Range: TCPIP_SRT_ARP_CACHE_SIZE_MIN ...      */
                                /*        TCPIP_SRT_ARP_CACHE_SIZE_MAX          */

  /*** EDD ***/
  TLR_STR FAR*  pszEddName;     /* EDD name                                     */

  TLR_UINT32    ulEddQuePoolElemCnt;  /* EDD: Sizes of:                         */
                          /*        - queue for received EDD packets           */
                          /*          and                                      */
                          /*        - resource pool for received EDD packets   */
                          /*       (both must have the same size)              */
                          /* Range: TCPIP_SRT_EDD_QUE_POOL_ELEM_CNT_MIN ...    */
                          /*        TCPIP_SRT_EDD_QUE_POOL_ELEM_CNT_MAX        */

  TLR_UINT32    ulEddOutBufMaxCnt;  /* Maximum count of outgoing EDD buffers,   */
                          /* the TCP/IP stack can use simultaneous             */
                          /* Range: TCPIP_SRT_EDD_OUT_BUF_MAX_CNT_MIN ...      */
                          /*        TCPIP_SRT_EDD_OUT_BUF_MAX_CNT_MAX          */

  /*** EIF (Ethernet Interface) ***/
  TCPIP_TCP_TASK_SRT_EIF_T FAR*  ptEif;

  /*** ARP ***/
  TLR_UINT32    ulArpTimeoutCache;  /* ARP cache timeout (seconds)              */

  /*** netX hardware name ***/
  TLR_STR FAR*  pszHwNameNetX;     /* netX hardware name                        */
                          /* NULL or "": The internal hardware names are used  */
                          /* otherwise : This hardware name is used ("netXXX") */
                          /* String length = 1 ... 63 characters              */

  /*** NetLoad Limiter ***/
  TLR_UINT32    ulNetLoadMaxFramesPerTick;      /*  Maximum number of frames   */
                              /* per systemtick the tcpip stack shall handle */

  TLR_UINT32    ulNetLoadMaxPendingARP;         /*  Maximum number of received */
                            /* and still pending (not yet processed) ARP frames */

  TLR_UINT32    ulNetLoadMaxPendingMCastARP;    /*  Maximum number of received */
                  /* and still pending (not yet processed) ARP multicast frames */

  TLR_UINT32    ulNetLoadMaxPendingIP;          /*  Maximum number of received */
                             /* and still pending (not yet processed) ip frames */

  TLR_UINT32    ulNetLoadMaxPendingMCastIP;     /*  Maximum number of received */
                   /* and still pending (not yet processed) multicast ip frames */

} TCPIP_TCP_TASK_STARTUPPARAMETER_T;
```

Note: The startup parameter **`ptEif`** is only for Hilscher internal use. Set it fix to **NULL**!

The range of the startup parameters is explained below (_MIN / _DEFAULT / _MAX). If there are no special requirements, we suggest the use of the default startup parameters named with TCPIP_SRT_xx_DEFAULT.

Startup parameter limits

```
#define TCPIP_STARTUPPARAMETER_VERSION_6  6
      /* ATTENTION: If you get here a                                          */
      /* compiler error because of new name of define (e.g. _3 --> _4), check  */
      /* startup parameters for changes/extensions!!                           */
      /* E.g, version 4 is because of incompatible changes in struct           */
      /* TCPIP_TCP_TASK_SRT_EIF_T (startup parameter ptEif)                    */


/*** Queue, pool element sizes ***/
/* Min/Default/Max TCP/IP stacks process queue size for AP packets */
/* (ulQueElemCntAp)                                                */
#define TCPIP_SRT_QUE_ELEM_CNT_AP_MIN           (4)
#define TCPIP_SRT_QUE_ELEM_CNT_AP_DEFAULT       (32)
#define TCPIP_SRT_QUE_ELEM_CNT_AP_MAX           (1024)  /* 8 per socket max     */
                                                        /* count (128 *8)       */

/* Min/Default/Max pool element count (ulPoolElemCnt) */
#define TCPIP_SRT_POOL_ELEM_CNT_MIN             (4)
#define TCPIP_SRT_POOL_ELEM_CNT_DEFAULT         (64)    /* Suggestion: 4  per   */
      /* (TCPIP_SRT_SOCKET_MAX_CNT_DEFAULT * 4)            socket               */
#define TCPIP_SRT_POOL_ELEM_CNT_MAX             (4096)  /* Maximum   : 32 per   */
      /* (TCPIP_SRT_SOCKET_MAX_CNT_MAX     * 8)            socket               */


/*** TCP/IP Stacks configuration ***/
/* Start flags (ulStartFlags) */
#define TCPIP_SRT_FLAG_DBM                      (0x00000001L) /* Use DBM/NXD    */
      /* database CONFIG.DBM/CONFIG.NXD. Otherwise, the stack must be          */
      /* configured via TCPIP_IP_CMD_SET_CONFIG command                        */

#define TCPIP_SRT_FLAG_DBM_ETHERNET_ADDR        (0x00000002L) /* Use MAC        */
      /* address from DBM/NXD database table ETHERNET. Otherwise, the MAC      */
      /* address comes from EDD. Conditon: TCPIP_SRT_FLAG_DBM must be set also */

      /* !!Add new flags also by startup-parameter check in module             */
      /* TcpipTcpTask_Resources.c, function                                    */
      /* TaskResource_TcpipTcpTask_InitLocal()!!                               */

#define TCPIP_SRT_FLAG_FAST_START               (0x00000004L) /* Activate fast  */
      /* start of TCP/IP stack (suppress the Gratuitous ARPs)                  */
      /* IMPORTANT!!!: Do not enable fast startup and ACD simultaneously       */
#define TCPIP_SRT_FLAG_KEEP_ALIVE_PATCH         (0x00000008L) /* Activate       */
      /* answer, if we receive wrong keep-alive packets without any flags set  */

#define TCPIP_SRT_FLAG_ACTIVATE_ACD             (0x00000010L) /* Activate       */
      /* address conflict detection */
      /* IMPORTANT!!!: Do not enable fast startup and ACD simultaneously       */
/* ACD defense options: when ACD is active, one of these options need to be set */
#define TCPIP_SRT_FLAG_ACD_DEFEND_DEF           (0x00000020L) /* Activates      */
      /* ACD option to defend ip address in any case                           */
#define TCPIP_SRT_FLAG_ACD_DEFEND_COND          (0x00000040L) /* Activates      */
      /* ACD option to defend the ip address with the condition, that there was */
      /* no ip address conflict within the last 10 seconds                     */
#define TCPIP_SRT_FLAG_ACD_NO_DEFENSE           (0x00000080L) /* Activates      */
      /* ACD option to drop the ip address in case of a conflict               */
#define TCPIP_SRT_FLAG_DISABLE_NET_IDENT        (0x00000100L)
      /* Disables NetIdent protocol */
#define TCPIP_SRT_FLAG_ENABLE_ARP_FILTER        (0x00000200L)
      /* enable an arp filter in receive callback. The Purpose is
       * to drop ARP frames in an early stage if they are not of interest for us.
       * Main usage is Profinet NetLoad for netX50*/

/* Min/Default/Max Cycletime of "TCP_UDP" task in ms (ulTcpCycleEvent) */
```

```
#define TCPIP_SRT_TCP_CYCLE_EVENT_MIN           (5)
#define TCPIP_SRT_TCP_CYCLE_EVENT_DEFAULT       (10)
#define TCPIP_SRT_TCP_CYCLE_EVENT_MAX           (20)

/* Min/Default/Max count of free queue elements (ulQueFreeElemCnt) */
#define TCPIP_SRT_QUE_FREE_ELEM_CNT_MIN         (4)
#define TCPIP_SRT_QUE_FREE_ELEM_CNT_DEFAULT     (128)   /* Default (Suggestion):*/
      /* 8 per socket                                                    */
#define TCPIP_SRT_QUE_FREE_ELEM_CNT_MAX         (4096)  /* Maximum: 32 per      */
      /* socket max count (128 * 32)                                     */

/* Min/Default/Max socket count (ulSocketMaxCnt) */
#define TCPIP_SRT_SOCKET_MAX_CNT_MIN            (1)
#define TCPIP_SRT_SOCKET_MAX_CNT_DEFAULT        (16)
#define TCPIP_SRT_SOCKET_MAX_CNT_MAX            (128)

/* Min/Default/Max size of Number of entries in ARP cache (ulArpCacheSize) */
#define TCPIP_SRT_ARP_CACHE_SIZE_MIN            (16)
#define TCPIP_SRT_ARP_CACHE_SIZE_DEFAULT        (64)    /* Default (Suggestion):*/
      /* 4 per socket, Depends on the host count                         */
#define TCPIP_SRT_ARP_CACHE_SIZE_MAX            (512)   /* Maximum: 4 per       */
      /* socket max count (128 * 4)                                      */


/*** EDD ***/
/* Min/Default/Max  queue/pool element count (ulEddQuePoolElemCnt) */
#define TCPIP_SRT_EDD_QUE_POOL_ELEM_CNT_MIN     (4)
#define TCPIP_SRT_EDD_QUE_POOL_ELEM_CNT_DEFAULT (32)
#define TCPIP_SRT_EDD_QUE_POOL_ELEM_CNT_MAX     (1024)  /* 8 per socket max     */
      /* count (128 *8)                                                  */

/* Min/Default/Max count of maximum outgoing EDD buffers (ulEddOutBufMaxCnt) */
#define TCPIP_SRT_EDD_OUT_BUF_MAX_CNT_MIN       (1)
#define TCPIP_SRT_EDD_OUT_BUF_MAX_CNT_DEFAULT   (10)    /* If this TCP/IP       */
      /* stacks instance is the only EDD user, we can use all 20 buffers of EDD */
      /* on netX (but these buffers are used for data send and receive - so we  */
      /* use per default only the half!)                                 */
      /* Otherwise, we must share the EDD buffers with other stack(s) of this   */
      /* instance                                                        */
#define TCPIP_SRT_EDD_OUT_BUF_MAX_CNT_MAX       (20)    /* Maximum buffer count */
      /* of netX HAL EDD                                                 */

/*** EIF (Ethernet interface) ***/
/* EDD instance (ulEddInstance)                                         */
#define TCPIP_SRT_EIF_EDD_INSTANCE_MIN          (0)
#define TCPIP_SRT_EIF_EDD_INSTANCE_MAX          (3)

/* Modes (ulEifMode)                                                    */
/*#define TCPIP_SRT_EIF_MODE_PORT_FILTER          (1) Only for V2.0.13.0!!     */
#define TCPIP_SRT_EIF_MODE_PORT_FILTER_2        (2) /* V2.0.14.0 and newer. The */
      /* new port filter mode is because of an incompatible change in struct    */
      /* TCPIP_TCP_TASK_SRT_EIF_T!!                                     */
#define TCPIP_SRT_EIF_MODE_PORT_FILTER_3        (3) /* Same as                  */
      /* TCPIP_SRT_EIF_MODE_PORT_FILTER_2, but netX stack process also ICMP and */
      /* IGMP packets to the own IP address                             */

/* Flags (ulEifFlags)                                                   */
#define TCPIP_SRT_EIF_FLAG_EIF_REGISTER_RUN_TIME  (0x00000001L) /* Activate     */
      /* run-time registration of Ethernet interface                    */

/* Min/Default/Max for port range (ulEifPortStart, ulEifPortEnd)        */
#define TCPIP_SRT_EIF_PORT_MIN                  (1024)
#define TCPIP_SRT_EIF_PORT_MAX                  (0xFFFF)
#define TCPIP_SRT_EIF_PORT_DEFAULT_START        (0xE000)
#define TCPIP_SRT_EIF_PORT_DEFAULT_END          (0xEFFF)

/* Min/Default/Max count of EIF ports to filter for netX (ulEifPortNmb) */
#define TCPIP_SRT_EIF_PORT_NMB_MIN              (0)
#define TCPIP_SRT_EIF_PORT_NMB_DEFAULT          (0)   /* 0 = Filter is off      */
#define TCPIP_SRT_EIF_PORT_NMB_MAX              (20)
```

```
/*** ARP ***/
/* Min/Default/Max of ARP cache timeout (ulArpTimeoutCache) in seconds          */
/* Remark: found in www: Dynamic ARP cache entries persist for 2-20 minutes,    */
/* depending on the system - so be carefull with too small/big values!          */
/* TCPIP_SRT_ARP_TIMEOUT_CACHE_DEFAULT is the suggestion!                       */
#define TCPIP_SRT_ARP_TIMEOUT_CACHE_MIN         (60)    /* 1 minute            */
#define TCPIP_SRT_ARP_TIMEOUT_CACHE_DEFAULT     (600)   /* 10 minutes          */
#define TCPIP_SRT_ARP_TIMEOUT_CACHE_MAX         (3600)  /* 1 hour              */

/*** NetLoad Limiter ***/
/* Maximum number of frames per system tick the tcp ip stack shall handle      */
#define TCPIP_SRT_NETLOAD_MAXFRAMESPERTICK_DEFAULT    (10) /* 6 Frames         */
/* Maximum number of unprocessed non-multicast arp frames in receive queue.
 * Usually responses to our own ARP requests                                    */
#define TCPIP_SRT_NETLOAD_MAXPENDING_ARP_DEFAULT      (6) /* 6 Frames          */

/* Maximum number of unprocessed multicast arp frames in receive queue. Usually
 * incoming ARP requests                                                        */
#define TCPIP_SRT_NETLOAD_MAXPENDING_MCASTARP_DEFAULT (4) /* 4 Frames          */

/* Maximum number of unprocessd non-multicast ip frames in receive queue.      */
#define TCPIP_SRT_NETLOAD_MAXPENDING_IP_DEFAULT       (10) /* 10 Frames        */

/* Maximum number of unprocessd multicast ip frames in receive queue.          */
#define TCPIP_SRT_NETLOAD_MAXPENDING_MCASTIP_DEFAULT  (5)  /* 5 Frames         */
```

The following source code example from rcX configuration file (for example `Config_netX.c`) show the use of the startup parameters.

Source code example

```
const TCPIP_TCP_TASK_STARTUPPARAMETER_T tTcpipTcpTaskParam =
{
.ulTaskIdentifier            = TLR_TASK_TCPUDP,
.ulParamVersion              = TCPIP_STARTUPPARAMETER_VERSION_6,
.ulQueElemCntAp              = TCPIP_SRT_QUE_ELEM_CNT_AP_DEFAULT,
.ulPoolElemCnt               = TCPIP_SRT_POOL_ELEM_CNT_DEFAULT,
.ulStartFlags                = TCPIP_SRT_FLAG_DBM, /* TCP Stack startup flags (see */
                                                   /* TCPIP_SRT_FLAG_xx in header  */
                                                   /* TcpipTcpTask_Functionlist.h) */
.ulTcpCycleEvent             = TCPIP_SRT_TCP_CYCLE_EVENT_DEFAULT,
.ulQueFreeElemCnt            = TCPIP_SRT_QUE_FREE_ELEM_CNT_DEFAULT,
.ulSocketMaxCnt              = TCPIP_SRT_SOCKET_MAX_CNT_DEFAULT,
.ulArpCacheSize              = TCPIP_SRT_ARP_CACHE_SIZE_DEFAULT,
.pszEddName                  = "ETHERNET",         /* EDD name (see name defined   */
                                                   /* in RX_EDD_SET_T parameters   */
.ulEddQuePoolElemCnt         = TCPIP_SRT_EDD_QUE_POOL_ELEM_CNT_DEFAULT,
.ulEddOutBufMaxCnt           = TCPIP_SRT_EDD_OUT_BUF_MAX_CNT_DEFAULT,
.ptEif                       = NULL,
.ulArpTimeoutCache           = TCPIP_SRT_ARP_TIMEOUT_CACHE_DEFAULT,
.pszHwNameNetX               = NULL,
.ulNetLoadMaxFramesPerTick   = TCPIP_SRT_NETLOAD_MAXFRAMESPERTICK_DEFAULT,
.ulNetLoadMaxPendingARP      = TCPIP_SRT_NETLOAD_MAXPENDING_ARP_DEFAULT,
.ulNetLoadMaxPendingMCastARP = TCPIP_SRT_NETLOAD_MAXPENDING_MCASTARP_DEFAULT,
.ulNetLoadMaxPendingIP       = TCPIP_SRT_NETLOAD_MAXPENDING_IP_DEFAULT,
.ulNetLoadMaxPendingMCastIP  = TCPIP_SRT_NETLOAD_MAXPENDING_MCASTIP_DEFAULT,
};
```

6 Status/error codes

Values of ulSta.

6.1 Status/error codes (general)

Hex value	Definition / description
0x00000000	TLR_S_OK Status ok
0xC0000004	TLR_E_UNKNOWN_COMMAND Unknown Command in Packet received.
0xC0000007	TLR_E_INVALID_PACKET_LEN Packet length is invalid.
0xC0000009	TLR_E_INVALID_PARAMETER Invalid Parameter in Packet found.
0xC000001A	TLR_E_REQUEST_RUNNING Request is already running.
0xC0000101	TLR_E_DATABASE_ACCESS_FAILED Database access failure.
0xC0000119	TLR_E_NOT_CONFIGURED Configuration not available Remark: This is only a temporary error, e.g. if no database access is configured (flag `TCPIP_SRT_FLAG_DBM` of startup-parameter `ulStartFlags` is not set).

Table 77: Status and error codes (general)

6.2 Status/error codes of TCP/IP (IP task)

Hex value	Definition / description
0xC0070034	TLR_E_IP_ERR_INIT_NO_ETHERNET_ADDR There is no Ethernet address (MAC address) available.
0xC0070036	TLR_E_IP_ERR_INIT_INVALID_FLAG The start parameters contains one or more unknown flags.
0xC0070037	TLR_E_IP_ERR_INIT_INVALID_IP_ADDR The start parameters contains an invalid IP address.
0xC0070038	TLR_E_IP_ERR_INIT_INVALID_NETMASK The start parameters contains an invalid subnet mask.
0xC0070039	TLR_E_IP_ERR_INIT_INVALID_GATEWAY The start parameters contains an invalid gateway IP address.
0xC007003B	TLR_E_IP_ERR_INIT_UNKNOWN_HARDWARE The device type is unknown.
0xC007003C	TLR_E_IP_ERR_INIT_NO_IP_ADDR Failed to obtain an IP address from the specified source(s).
0xC007003D	TLR_E_IP_ERR_INIT_DRIVER_FAILED The initialization of the driver layer (EDD) is failed.
0xC007003E	TLR_E_IP_ERR_INIT_NO_IP_ADDR_CFG There is no source for an IP address (BOOTP, DHCP, IP address parameter) specified.
0xC008007A	TLR_E_TCP_ERR_PROTOCOL_UNKNOWN_TCP_UDP_CMD_OPEN The protocol parameter ulProtocol in command TCPIP_TCP_UDP_CMD_OPEN_REQ is invalid.
0xC008007B	TLR_E_TCP_ERR_NO_SOCKETS_TCP_UDP_CMD_OPEN Command TCPIP_TCP_UDP_CMD_OPEN_REQ: There are no socket handles available.

Hex value	Definition / description
0xC007007C	TLR_E_IP_ERR_ETH_ADDR_INVALID_IP_CMD_SET_PARAM The Ethernet address (MAC address) abEthernetAddr in command TCPIP_IP_CMD_SET_PARAM_REQ is invalid. Invalid means, abEthernetAddr is equal to the broadcast address FF-FF-FF-FF-FF-FF.
0xC0070083	TLR_E_IP_ERR_ARP_CACHE_FULL_IP_CMD_SET_PARAM The command TCPIP_IP_CMD_SET_PARAM_REQ could not be executed, because the ARP cache is full. The ARP cache has per default configuration 64 entries.
0xC0070086	TLR_E_IP_ERR_ARP_ENTRY_NOT_FOUND_IP_CMD_SET_PARAM The specified ARP entry in command TCPIP_IP_CMD_SET_PARAM_REQ could not be deleted. The ARP entry was not found in ARP cache.
0xC0070087	TLR_E_IP_ERR_ARP_ENTRY_NOT_FOUND_IP_CMD_GET_PARAM The requested ARP information in command TCPIP_IP_CMD_GET_PARAM_REQ could not be delivered. The ARP entry was not found in ARP cache.
0xC00700FE	TLR_E_IP_ERR_DELAYED Special internal error code returned by IpStart function.
0xC00700FF	TLR_E_IP_ERR_GENERIC Special internal error code returned by IpStart function.
0xC0070100	TLR_E_IP_ERR_IP_ADDR_INVALID_IP_CMD_PING The IP address parameter `ulIpAddr` in command `TCPIP_IP_CMD_PING_REQ` is invalid. This means, the IP address `ulIpAddr` is equal to the TCP/IP stacks own IP address.
0xC0070120	TLR_E_IP_ERR_TIMEOUT_INVALID_IP_CMD_PING The timeout parameter `ulTimeout` in command `TCPIP_IP_CMD_PING_REQ` is invalid.
0xC0070130	TLR_E_IP_ERR_MODE_UNKNOWN_IP_CMD_SET_PARAM The mode parameter `ulMode` in command `TCPIP_IP_CMD_SET_PARAM_REQ` is invalid.
0xC0070131	TLR_E_IP_ERR_MODE_UNKNOWN_IP_CMD_GET_PARAM The mode parameter `ulMode` in command `TCPIP_IP_CMD_GET_PARAM_REQ` is invalid.
0xC0070150	TLR_E_IP_ERR_INIT_INVALID_FLAGS_IP_CONFIG The start parameters configures an invalid flag combination for the manual IP configuration (`IP_CFG_FLAG_IP_ADDR`, `IP_CFG_FLAG_NET_MASK`, `IP_CFG_FLAG_GATEWAY`). Valid flag combinations are: ▪ No flag set: No manual configuration - only DHCP and/or BOOTP ▪ `IP_CFG_FLAG_IP_ADDR` + `IP_CFG_FLAG_NET_MASK`: Local network without gateway ▪ `IP_CFG_FLAG_IP_ADDR` + `IP_CFG_FLAG_NET_MASK` + `IP_CFG_FLAG_GATEWAY`: Network with gateway.
0xC0070300	TLR_E_IP_ERR_DEST_UNREACHABLE_IP_CMD_PING The target IP address `ulIpAddr` in command `TCPIP_IP_CMD_PING_REQ` is not reachable.
0xC0070310	TLR_E_IP_ERR_TIMEOUT_IP_CMD_PING The specified timeout `ulTimeout` in command `TCPIP_IP_CMD_PING_REQ` has expired. The specified host is not reachable.

Table 78: Status/error codes TCP/IP (IP task)

6.3 Status/error codes of TCP/IP (TCP task)

Hex value	Definition / description
0x80080043	TLR_W_TCP_ERR_INIT_TPIF_INIT_REQ_PCKT Warning: A pending application packet has discarded (because of a new application packet).
0xC0080003	TLR_E_TCP_ERR_CODEDIAG_FATAL A fatal error is occured. Terminate the task.
0xC0080005	TLR_E_TCP_TASK_F_INITIALIZATION_FAILED Failed to initialize the task. Accept Init packets and Config packets only.
0xC0080006	TLR_E_IP_ERR_INIT_INVALID_SERIAL_NUMBER Invalid serial number.
0xC0080007	TLR_E_IP_ERR_INIT_IP_INIT_ERROR Failed to initialize the IP layer - see task status.
0xC0080009	TLR_E_TCPIP_TCP_TASK_PROCESS_CANCELED Cancel process is in progress, command can not be executed.
0xC008000A	TLR_E_TCPIP_EDD_IDENTIFY_FAILED Failed to identify the EDD (Ethernet Device Driver).
0xC008000B	TLR_E_TCPIP_APPLICATION_TIMER_CREATE_FAILED Failed to create an application timer (Timer task).
0xC008000C	TLR_E_TCPIP_APPLICATION_TIMER_INIT_PACKET_FAILED Failed to initialize a packet of application timer (Timer task).
0xC008000D	TLR_E_TCPIP_INVALID_STARTUP_PARAMETER_SOCKET_MAX_CNT Invalid Startup Parameter ulSocketMaxCnt.
0xC008000E	TLR_E_TCPIP_INVALID_STARTUP_PARAMETER_POOL_ELEM_CNT Invalid Startup Parameter ulPoolElemCnt.
0xC008000F	TLR_E_TCPIP_INVALID_STARTUP_PARAMETER_EDD_OUT_BUF_MAX_CNT Invalid Startup Parameter ulEddOutBufMaxCnt.
0xC0080010	TLR_E_TCPIP_INVALID_STARTUP_PARAMETER_ARP_CACHE_SIZE Invalid Startup Parameter ulArpCacheSize.
0xC0080011	TLR_E_TCPIP_INVALID_STARTUP_PARAMETER_QUE_FREE_ELEM_CNT Invalid Startup Parameter ulQueFreeElemCnt.
0xC0080012	TLR_E_TCPIP_INVALID_STARTUP_PARAMETER_TCP_CYCLE_EVENT Invalid Startup Parameter ulTcpCycleEvent.
0xC0080014	TLR_E_TCPIP_INVALID_STARTUP_PARAMETER_QUE_ELEM_CNT_AP Invalid Startup Parameter ulQueElemCnt.
0xC0080015	TLR_E_TCPIP_INVALID_STARTUP_PARAMETER_EDD_QUE_POOL_ELEM_CNT Invalid Startup Parameter ulEddQuePoolElemCnt.
0xC0080016	TLR_E_TCPIP_INVALID_STARTUP_PARAMETER_START_FLAGS Invalid Startup Parameter ulStartFlags. Unknown flags are set.
0xC0080017	TLR_E_TCPIP_INVALID_STARTUP_PARAMETER_EDD_NAME Invalid Startup Parameter pszEddName.
0xC0080018	TLR_E_TCPIP_INVALID_STARTUP_PARAMETER_EIF_EDD_NAME Invalid Startup Parameter EIF pszEddName.
0xC0080019	TLR_E_TCPIP_INVALID_STARTUP_PARAMETER_EIF_EDD_INSTANCE Invalid Startup Parameter EIF ulEddInstance.
0xC008001A	TLR_E_TCPIP_INVALID_STARTUP_PARAMETER_EIF_ETH_INTF_NAME Invalid Startup Parameter EIF pszEifEthIntfName.
0xC008001B	TLR_E_TCPIP_INVALID_STARTUP_PARAMETER_EIF_MODE Invalid Startup Parameter EIF ulEifMode.
0xC008001C	TLR_E_TCPIP_INVALID_STARTUP_PARAMETER_EIF_PORT_RANGE Invalid Startup Parameter EIFs ulEifPortStart, ulEifPortEnd.

Hex value	Definition / description
0xC008001D	TLR_E_TCPIP_INVALID_STARTUP_PARAMETER_EIF_PORT_NMB Invalid Startup Parameter EIF ulEifPortNmb.
0xC008001E	TLR_E_TCPIP_INVALID_STARTUP_PARAMETER_ARP_TIMEOUT_CACHE Invalid Startup Parameter ulArpTimeoutCache.
0xC008001F	TLR_E_TCPIP_INVALID_STARTUP_PARAMETER_EIF_FLAGS Invalid Startup Parameter EIF ulEifFlags.
0xC0080020	TLR_E_TCPIP_INVALID_STARTUP_PARAMETER_HW_NAME_NETX Invalid Startup Parameter pszHwNameNetX.
0xC0080032	TLR_E_TCP_ERR_INIT_IP_TASK_NOT_READY The IP layer is not ready.
0xC0080034	TLR_E_TCP_ERR_INIT_IP_TASK_FAILED The initialization of IP layer has failed.
0xC0080044	TLR_E_TCP_ERR_INIT_OS_CYCLETIME The configured operating system cycletime is out of range (0.1 ms ... 20 ms).
0xC0080045	TLR_E_TCP_ERR_INIT_OS_AND_TCPUDP_CYCLETIME The combination of configured operating system cycletime and TCP/IP stacks cycletime (startup-parameter ulTcpCycleEvent) is not possible. The operating system cycletime must be smaller or equal than the TCP/IP stacks cycletime.
0xC0080070	TLR_E_TCP_ERR_SOCKET_INVALID The socket handle ulDestId is invalid. A further reason for this error: The command (Mode) is not applicable on this socket type (TCP/UDP).
0xC0080074	TLR_E_TCP_ERR_OPTION_NOT_SUPPORTED_TCP_CMD_SEND The option parameter ulOptions in command TCPIP_TCP_CMD_SEND_REQ is invalid.
0xC0080075	TLR_E_TCP_ERR_PARAMETER_INVALID_TCP_UDP_CMD_SET_SOCK_OPTION The parameter in command TCPIP_TCP_UDP_CMD_SET_SOCK_OPTION_REQ is invalid.
0xC0080078	TLR_E_TCP_ERR_CONN_CLOSED The connection has closed (Graceful close).
0xC0080079	TLR_E_TCP_ERR_CONN_RESET The connection has closed by reset (Hard close).
0xC0080085	TLR_E_TCP_ERR_MAX_GROUP_EXCEEDED_TCP_UDP_CMD_SET_SOCK_OPTION Command TCPIP_TCP_UDP_CMD_SET_SOCK_OPTION_REQ, ulMode = TCP_SOCK_ADD_MEMBERSHIP: The maximum number of IP multicast groups has exceeded (Default configuration = 64).
0xC0080086	TLR_E_TCP_ERR_DISCARD_KEPT_REQ_CMD A kept request command has discarded. This confirmation has no further meaning for the application, unless the application must give back this packet to their resource pool!
0xC0080095	TLR_E_TCP_ERR_UNEXP_ANSWER An unexpected/unknown confirmation command has received.
0xC00800C8	TLR_E_TCP_TASK_F_NOT_INITIALIZED The task is not initialized.
0xC00800C9	TLR_E_TCP_TASK_F_BUSY The task is busy (intern).
0xC0080101	TLR_E_TCP_ERR_IP_ADDR_INVALID_TCP_UDP_CMD_OPEN The IP address parameter ulIpAddr in command TCPIP_TCP_UDP_CMD_OPEN_REQ is invalid. The parameter ulIpAddr must be zero (0.0.0.0) or equal to the TCP/IP stacks own IP address.
0xC0080102	TLR_E_TCP_ERR_IP_ADDR_INVALID_TCP_CMD_CONNECT The IP address parameter ulIpAddr in command TCPIP_TCP_CMD_CONNECT_REQ is invalid. The parameter ulIpAddr must be unequal to the TCP/IP stacks own IP address.
0xC0080103	TLR_E_TCP_ERR_IP_ADDR_INVALID_UDP_CMD_SEND The IP address parameter ulIpAddr in command TCPIP_UDP_CMD_SEND_REQ is invalid or doesn't match to the local sub network. This error occurs, if the IP address is zero (0.0.0.0) or equal to the address of the local subnet.

Hex value	Definition / description
0xC0080104	TLR_E_TCP_ERR_IP_ADDR_INVALID_TCP_UDP_CMD_SET_SOCK_OPTION The parameter ulMulticastGroup (ulMode = TCP_SOCK_ADD_MEMBERSHIP or TCP_SOCK_DROP_MEMBERSHIP) in command TCPIP_TCP_UDP_CMD_SET_SOCK_OPTION_REQ is invalid. The parameter ulMulticastGroup must be a valid Multicast address. Valid Multicast addresses are 224.0.0.1 ... 239.255.255.255 (224.0.0.0 is reserved as Base-multicast address).
0xC0080106	TLR_E_TCP_ERR_IP_ADDR_INVALID_NO_GATEWAY The IP address parameter ulIpAddr is invalid, because there is no gateway configured. The parameter ulIpAddr must be inside the local network.
0xC0080110	TLR_E_TCP_ERR_PORT_INVALID_TCP_UDP_CMD_OPEN The port parameter ulPort in command TCPIP_TCP_UDP_CMD_OPEN_REQ is invalid or not available. The parameter ulPort must be in range 0 ... 65535.
0xC0080111	TLR_E_TCP_ERR_PORT_INVALID_TCP_CMD_CONNECT The port parameter ulPort in command TCPIP_TCP_CMD_CONNECT_REQ is invalid or not available. The parameter ulPort must be in range 1 ... 65535.
0xC0080112	TLR_E_TCP_ERR_PORT_INVALID_UDP_CMD_SEND The port parameter ulPort in command TCPIP_UDP_CMD_SEND_REQ is invalid or not available. The parameter ulPort must be in range 0 ... 65535.
0xC0080121	TLR_E_TCP_ERR_TIMEOUT_INVALID_TCP_UDP_CMD_CLOSE The timeout parameter ulTimeout in command TCPIP_TCP_UDP_CMD_CLOSE_REQ is invalid. Consider the distinction between TCP and UDP sockets! For UDP sockets, ulTimeout must be zero.
0xC0080122	TLR_E_TCP_ERR_TIMEOUT_INVALID_TCP_UDP_CMD_CLOSE_ALL The timeout parameter ulTimeout in command TCPIP_TCP_UDP_CMD_CLOSE_ALL_REQ is invalid.
0xC0080123	TLR_E_TCP_ERR_TIMEOUT_INVALID_TCP_CMD_WAIT_CONNECT The timeout parameter ulTimeoutSend and/or ulTimeoutListen in command TCPIP_TCP_CMD_WAIT_CONNECT_REQ is invalid.
0xC0080124	TLR_E_TCP_ERR_TIMEOUT_INVALID_TCP_CMD_CONNECT The timeout parameter ulTimeoutSend and/or ulTimeoutConnect in command TCPIP_TCP_CMD_CONNECT_REQ is invalid.
0xC0080125	TLR_E_TCP_ERR_TIMEOUT_INVALID_TCP_UDP_CMD_SET_SOCK_OPTION The timeout parameter ulTimeoutSend (ulMode = TCP_SOCK_SEND_TIMEOUT) or ulTimeoutInactive (ulMode = TCP_SOCK_INACTIVE_TIMEOUT) or ulTimeoutKeepAlive (ulMode = TCP_SOCK_KEEPALIVE_TIMEOUT) in command TCPIP_TCP_UDP_CMD_SET_SOCK_OPTION_REQ is invalid.
0xC0080132	TLR_E_TCP_ERR_MODE_UNKNOWN_TCP_UDP_CMD_SET_SOCK_OPTION The mode parameter ulMode in command TCPIP_TCP_UDP_CMD_SET_SOCK_OPTION_REQ is invalid.
0xC0080133	TLR_E_TCP_ERR_MODE_UNKNOWN_TCP_UDP_CMD_GET_SOCK_OPTION The mode parameter ulMode in command TCPIP_TCP_UDP_CMD_GET_SOCK_OPTION_REQ is invalid.
0xC0080134	TLR_E_TCP_ERR_MODE_UNKNOWN_FATAL_DUMMY Internal fatal error in module TcpipTcphdlPckt.c!
0xC0080140	TLR_E_TCP_ERR_MAX_DATA_LEN_EXCEEDED_CP_CMD_SEND The maximum TCP data count *n* in command TCPIP_TCP_CMD_SEND_REQ has exceeded. See parameter ulLen. The maximum value for *n* is TCPIP_MAX_TCP_DATA_CNT (1460).
0xC0080141	TLR_E_TCP_ERR_MAX_DATA_LEN_EXCEEDED_UDP_CMD_SEND The maximum UDP data count *n* in command TCPIP_UDP_CMD_SEND_REQ has exceeded. See parameter ulLen. The maximum value for *n* is TCPIP_MAX_UDP_DATA_CNT (1472).
0xC0080200	TLR_E_TCP_ERR_SOCKET_STATE_TCP_CMD_WAIT_CONNECT The command TCPIP_TCP_CMD_WAIT_CONNECT_REQ cannot be executed, because the socket is in an inappropriate state.
0xC0080201	TLR_E_TCP_ERR_SOCKET_STATE_TCP_CMD_CONNECT The command TCPIP_TCP_CMD_CONNECT_REQ cannot be executed, because the socket is in an inappropriate state.

Hex value	Definition / description
0xC0080202	TLR_E_TCP_ERR_SOCKET_STATE_TCP_CMD_SEND The command TCPIP_TCP_CMD_SEND_REQ cannot be executed, because the socket is in an inappropriate state.
0xC0080210	TLR_E_TCP_ERR_NO_FREE_QUEUE_ELEMENT_TCP_CMD_SEND The TCP send command TCPIP_TCP_CMD_SEND_REQ must be rejected, because the list of free queue elements is empty. Remark: Per default configuration, the initial size of this list is 128. Every send command (TCPIP_TCP_CMD_SEND_REQ or TCPIP_UDP_CMD_SEND_REQ) occupy one queue element, until the confirmation command is given back to the application. To avoid this resource problem, the application must reduce the count of open send jobs over all sockets.
0xC0080211	TLR_E_TCP_ERR_NO_FREE_QUEUE_ELEMENT_UDP_CMD_SEND The UDP send command TCPIP_UDP_CMD_SEND_REQ must be rejected, because the list of free queue elements is empty. Remark: Per default configuration, the initial size of this list is 128. Every send command (TCPIP_TCP_CMD_SEND_REQ or TCPIP_UDP_CMD_SEND_REQ) occupy one queue element, until the confirmation command is given back to the application. To avoid this resource problem, the application must reduce the count of open send jobs over all sockets.
0xC0080212	TLR_E_TCP_ERR_NO_ETH_OUT_BUFFER_UDP_CMD_SEND The UDP send command TCPIP_UDP_CMD_SEND_REQ must be rejected, because all outgoing Ethernet buffers are occupied.
0xC0080213	TLR_E_TCP_ERR_NO_FREE_RESOURCE_FOR_ARP_REQ_INTF The command TCPIP_IP_CMD_SET_PARAM_REQ in mode IP_PRM_SEND_ARP_REQ/ IP_PRM_SEND_ARP_TMT_REQ must be rejected, because all free resources for this command are occupied. A maximum of 128 parallel jobs is possible.
0xC0080214	TLR_E_TCP_ERR_ETH_OUT_SEND_BUFFER The send of the outgoing Ethernet buffer has failed. The reason of this error is normally a resource problem - there is no EDD buffer available.
0xC0080220	TLR_E_TCP_ERR_MCAST_CREATE Failed to create an IP Multicast group.
0xC0080301	TLR_E_TCP_ERR_DEST_UNREACHABLE_TCP_UDP_CMD_CLOSE Command TCPIP_TCP_UDP_CMD_CLOSE_REQ: The destination (host, network, or port) is unreachable.
0xC0080302	TLR_E_TCP_ERR_DEST_UNREACHABLE_TCP_UDP_CMD_CLOSE_ALL Command TCPIP_TCP_UDP_CMD_CLOSE_ALL_REQ: The destination (host, network, or port) is unreachable.
0xC0080303	TLR_E_TCP_ERR_DEST_UNREACHABLE_TCP_CMD_WAIT_CONNECT Command TCPIP_TCP_CMD_WAIT_CONNECT_REQ: The destination (host, network, or port) is unreachable.
0xC0080304	TLR_E_TCP_ERR_DEST_UNREACHABLE_TCP_CMD_CONNECT Command TCPIP_TCP_CMD_CONNECT_REQ: The destination (host, network, or port) is unreachable.
0xC0080305	TLR_E_TCP_ERR_DEST_UNREACHABLE_UDP_CMD_SEND Command TCPIP_UDP_CMD_SEND_REQ: The destination (host, network, or port) is unreachable.
0xC0080311	TLR_E_TCP_ERR_TIMEOUT_TCP_UDP_CMD_CLOSE The TCP Close timeout has expired. A connection to the remote host could not be closed gracefully within this time. For this timeout, see command TCPIP_TCP_UDP_CMD_CLOSE_REQ, parameter ulTimeout.
0xC0080312	TLR_E_TCP_ERR_TIMEOUT_TCP_UDP_CMD_CLOSE_ALL The TCP Close timeout has expired. One or more connections to remote host(s) could not be closed gracefully within this time. For this timeout, see command TCPIP_TCP_UDP_CMD_CLOSE_ALL_REQ, parameter ulTimeout.
0xC0080313	TLR_E_TCP_ERR_TIMEOUT_TCP_CMD_WAIT_CONNECT The TCP Connect timeout has expired. No remote host has connected within this time. For this timeout, see command TCPIP_TCP_CMD_WAIT_CONNECT_REQ, parameter ulTimeoutListen.

Hex value	Definition / description
0xC0080314	TLR_E_TCP_ERR_TIMEOUT_TCP_CMD_CONNECT The TCP Connect timeout has expired. A connection to the specified remote host could not be established within this time. For this timeout, see command TCPIP_TCP_CMD_CONNECT_REQ, parameter ulTimeoutConnect.
0xC0080315	TLR_E_TCP_ERR_TIMEOUT_TCP_CMD_SEND The TCP Send timeout has expired by sending TCP data with command TCPIP_TCP_CMD_SEND_REQ. The remote host has not answered within the Send Timeout. The TCP Send timeout is set in command TCPIP_TCP_CMD_WAIT_CONNECT_REQ or TCPIP_TCP_CMD_CONNECT_REQ, parameter ulTimeoutSend (Default = 31 s).

Table 79: Status/error codes TCP/IP (TCP task)

7 Appendix

7.1 List of tables

7.2 List of figures

7.3 Contacts

Headquarters

Germany
Hilscher Gesellschaft für
Systemautomation mbH
Rheinstrasse 15
65795 Hattersheim
Phone: +49 (0) 6190 9907-0
Fax: +49 (0) 6190 9907-50
E-Mail: info@hilscher.com

Support
Phone: +49 (0) 6190 9907-99
E-Mail: de.support@hilscher.com

Subsidiaries

China
Hilscher Systemautomation (Shanghai) Co. Ltd.
200010 Shanghai
Phone: +86 (0) 21-6355-5161
E-Mail: info@hilscher.cn

Support
Phone: +86 (0) 21-6355-5161
E-Mail: cn.support@hilscher.com

France
Hilscher France S.a.r.l.
69500 Bron
Phone: +33 (0) 4 72 37 98 40
E-Mail: info@hilscher.fr

Support
Phone: +33 (0) 4 72 37 98 40
E-Mail: fr.support@hilscher.com

India
Hilscher India Pvt. Ltd.
Pune, Delhi, Mumbai
Phone: +91 8888 750 777
E-Mail: info@hilscher.in

Italy
Hilscher Italia S.r.l.
20090 Vimodrone (MI)
Phone: +39 02 25007068
E-Mail: info@hilscher.it

Support
Phone: +39 02 25007068
E-Mail: it.support@hilscher.com

Japan
Hilscher Japan KK
Tokyo, 160-0022
Phone: +81 (0) 3-5362-0521
E-Mail: info@hilscher.jp

Support
Phone: +81 (0) 3-5362-0521
E-Mail: jp.support@hilscher.com

Korea
Hilscher Korea Inc.
Seongnam, Gyeonggi, 463-400
Phone: +82 (0) 31-789-3715
E-Mail: info@hilscher.kr

Switzerland
Hilscher Swiss GmbH
4500 Solothurn
Phone: +41 (0) 32 623 6633
E-Mail: info@hilscher.ch

Support
Phone: +49 (0) 6190 9907-99
E-Mail: ch.support@hilscher.com

USA
Hilscher North America, Inc.
Lisle, IL 60532
Phone: +1 630-505-5301
E-Mail: info@hilscher.us

Support
Phone: +1 630-505-5301
E-Mail: us.support@hilscher.com