TCP LIBRARY FOR CAMERA AND PC

User's Guide

22 March 2005

Thank you for your interest in our TCP library for TI cameras and PC. In this manual you will find out how to use it.

Before going on reading the manual, we kindly ask you to read the following

DISCLAIMER

THIS DOCUMENTATION IS PROVIDED FOR REFERENCE PURPOSES ONLY. WHILE EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS DOCUMENTATION, THIS DOCUMENTATION IS PROVIDED “AS IS” WITHOUT ANY WARRANTY WHATSOEVER AND TO THE MAXIMUM EXTENT PERMITTED, ATTO-SYSTEMS LTD. DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY, NONINFRINGEMENT AND FITNESS FOR A PARTICULAR PURPOSE, WITH RESPECT TO THE SAME. ATTO-SYSTEMS LTD. SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, DIRECT, INDIRECT, CONSEQUENTIAL OR INCIDENTAL DAMAGES, ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS DOCUMENTATION OR ANY OTHER DOCUMENTATION. NOTWITHSTANDING ANYTHING TO THE CONTRARY, NOTHING CONTAINED IN THIS DOCUMENTATION OR ANY OTHER DOCUMENTATION IS INTENDED TO, NOR SHALL HAVE THE EFFECT OF, CREATING ANY WARRANTIES OR REPRESENTATIONS FROM ATTO-SYSTEMS LTD. (OR ITS SUPPLIERS OR LICENSORS), OR ALTERING THE TERMS AND CONDITIONS OF THE APPLICABLE LICENSE AGREEMENT GOVERNING THE USE OF THIS SOFTWARE.

THE DESCRIBED SOFTWARE IS PROVIDED 'AS IS', WITHOUT ANY WARRANTY EXPRESSED OR IMPLIED. NO GUARANTY IS GIVEN THAT THE SOFTWARE IS SUITABLE FOR ANY GIVEN PURPOSE.

COPYRIGHT

TRADEMARKS

CONVENTIONS USED IN THIS MANUAL

***INFORMATION**. This sign marks section in the manual, which is for information only.*

***ATTENTION**. This sign marks section of the manual, which is particularly important for the general understanding of the document. Please, make sure to read this section before proceeding with reading the manual.*

***TIPS & TRICKS**. This sign marks a Tips & Tricks section. Here you can find some practical advises on using the system or get a more detailed explanation of some features. Reading this section may help you in solving a particular problem or give you some ideas but is not vital for understanding the document.*

***PREMISE**. This sign marks a section, which requires you to do something before proceeding with reading the manual. Usually this is a demo program, you have to run or something similar.*

File	*Menu item*
File > Open	*Sub-menu item or dialog control*
"1.1 About"	*Section reference. If the section is within the current manual no manual name is specified. When the section is within external manual the name of the respective manual is also included.*
`Ctrl+E`	*Hot-key combination. The first part of the combination specifies which system key to use. Possible values are: Ctrl, Alt, Shift. The second part specifies the normal key to be used in the combination. The plus sign means that you should press these keys simultaneously.*

CONTENTS

1. Introduction

The TCP library is a set of I/O functions for Ethernet cameras VC20xxE and PC. The VC20xxE cameras are manufactured by Vision Components GmbH, Germany. They are based on a Texas Instruments (TI) processor and have an Ethernet port for LAN connection.

The library functions are divided in two groups:

- **Basic TCP library**. Basic TCP/IP functions for server/client connection and data transfer between remote peers. Here are functions for library initialization, connection and disconnection. The basic library contains byte and block oriented functions for data transfer.
- **VIMOS TCP library**. High-level I/O functions for communication with VIMOS I/O tools. VIMOS is a machine vision system created by atto-Systems Ltd, which runs on VC20xxE cameras.

The TCP library is available on two platforms – PC and VC20xxE camera. Using the library you may create programs for data transfer between PC/camera and a remote TCP/IP server or client.

***ATTENTION**. Unauthorized usage of the library is disabled on the camera. You need a registration key, which is received by the program vendor.*

1.1. Why a new library?

Both the PC and the camera have standard low-level libraries, which support TCP/IP communication. On the PC you can find such functions in Visual C/C++. On camera you can use the TCP/IP functions supplied by Vision Components.

Our library is based on the respective low-level libraries, but it simplifies and facilitates the creation of communication software. It frees your from the necessity to read dozens of pages with function descriptions and numerous options and to guess which option(s) should be used in your code. We have already done it instead of you.

Using the library you are able to:

- Create fully portable I/O code, which is identical on camera and PC with minor exceptions (different general-purpose headers).
- Easily create, connect, disconnect and close sockets (abstractions used to identify TCP/IP end-points).
- Use reliable stream sockets in two connection modes: server and client.
- Use simultaneously multiple data sockets and one listen socket.
- Use default socket options (send/receive buffer size, connection timeouts, etc.), which usually are not enabled, or may have unsuitable default values.
- Use stable and reliable byte and block transfer functions. These functions check the socket state and its readiness before each send/receive operation. They handle properly non-fatal (busy) errors, which should not abort the transfer. All “receive” function support timeouts.
- Keep track of connected remote peer parameters.

***ATTENTION**. Socket disconnection is done in hard (abort) mode.*

In addition, the library provides functions for communication with VIMOS I/O tools in TCP/IP mode. You can create stand-alone programs on the PC or the camera, which interchange data with VIMOS. Usually your stand-alone program and VIMOS are running on opposite platforms – one on the PC and the other on the camera, but it is possible to make connection between two cameras or two PC’s.

Initialization and connection is common for the basic library and the VIMOS library (see the examples below). The VIMOS library contains data transfer functions, which work with more complex data structures. You can't use the VIMOS library isolated from the basic library; therefore both libraries are contained in one library file:

- **TCP_PC.LIB** – PC library (basic + VIMOS);
- **TCP_CAM.LIB** – camera library (basic + VIMOS).

1.2. Simple examples

Using the library is not a complex task. You work in C (C files). Your program should have the following general structure:

- Declare sockets.
- Initialize the library by `TCP_Init()` before any other I/O code.
- Reset sockets.
- Connect socket(s) in ***server*** mode (see the example **SERVER.C**) or in ***client*** mode (see the example **CLIENT.C**).
- Perform I/O transfer between connected sockets after successful connection.
- Disconnect (close) sockets.
- Close the library by `TCP_Close()`.

The two examples below are fully functional programs, which transfer data between PC and camera. The examples use functions from the basic library. The server program is intended to run on camera and the client program - on PC as a console application. You can run the ready executable modules supplied with the library. Load your registration key file **TCPKEY.MSF** to the camera flash (the **FD:** drive). The registration file is not needed for the ready MSF demo files. Load **SERVER.MSF** to camera flash and start ***server*** from a TCP/IP terminal. The server program will wait 9 seconds for connection. Start immediately ***client.exe*** in a command prompt window. You should receive a data dump **`0,1,…,7`** and return code **`RC=0`**.

You may interchange the platforms by recompilation of the source code (see "*3.8. Compiling and linking programs with TCP library*"). Remember that you should set a PC IP address in the client example where currently the camera's IP address is used.

1.2.1. Server example

The server example **SERVER.C** waits for a connection request from a remote client on port 2000. After successful connection the program receives a block of 8 bytes and sends the block back to the client. Note that in a server program you don't need to specify IP address and port of a connecting remote peer.

```
/*-------------------------------------------------------------------*/
/*
* File: server.c
*
* Demo program for TCP server. The program waits for connection request
* from a remote client on port 2000. After successful connection the
* program receives a block of 8 bytes and sends back the block to the
* client.
*
* _TI_CAMERA defined     : Compile for camera
* _TI_CAMERA not defined : Compile for PC
*/
/*-------------------------------------------------------------------*/
#ifdef _TI_CAMERA
  #include <vcrt.h>
#else
  #include <conio.h>
```

```
#endif
#include "tcp_lib.h"

#define CONN_TIME_OUT   9000        /* connection wait time in ms   */
#define RECV_TIME_OUT    500        /* receive timeout in ms        */
#define BUF_SIZE           8        /* I/O buffer size              */
#define SERVER_PORT     2000        /* server listen port           */

int main()
{
    int  rc;
    int  iret;
    unsigned long time;
    unsigned char buf[BUF_SIZE];    /* I/O buffer               */
    TCP_SOCK ListenSock;            /* listen socket            */
    TCP_SOCK DataSock;              /* data socket              */

/*............. Init TCP */
    rc = TCP_Init();
    if(rc != TCP_OK)
    {
        printf("TCP_Init error = %d\n",rc);
        return rc;
    }

/*............. Reset sockets */
    TCP_DataSockReset(&ListenSock,1);
    TCP_DataSockReset(&DataSock,1);

/*............. Create listen socket and start listening on port 2000 */
    rc = TCP_Listen(&ListenSock,SERVER_PORT);
    if(rc != TCP_OK) goto done;

/*............. Wait for connection with a remote client */
    time = get_time();
    for(;;)
    {
        rc = TCP_AcceptConnection(&ListenSock,&DataSock,1);
        if(rc != TCP_OK) goto done;

        if(TCP_GetConnectionState(&DataSock)) break;    /* connection OK */
        if(get_time() - time >= CONN_TIME_OUT)
        {
           rc = TCP_NO_CONNECTION_ERROR;
           goto done;
        }
    } /* for(;;) */

/*............. Receive block */
    iret = TCP_RecvBlock(&DataSock,buf,BUF_SIZE,RECV_TIME_OUT);
    if(iret != BUF_SIZE)
    {
        printf("TCP_RecvBlock error: iret=%d buf_size=%d\n",iret,BUF_SIZE);
        rc = TCP_RECV_BLOCK_ERROR;
        goto done;
    }

/*............. Send block */
    iret = TCP_SendBlock(&DataSock,buf,BUF_SIZE);
    if(iret != BUF_SIZE)
    {
```

```
        printf("TCP_SendBlock error: iret=%d buf_size=%d\n",iret,BUF_SIZE);
        rc = TCP_SEND_BLOCK_ERROR;
        goto done;
    }

/*............. Exit */
done:
    time = get_time();
    while(get_time() - time < 300);     /* wait before disconnect */

    TCP_Disconnect(&DataSock);
    TCP_Disconnect(&ListenSock);
    TCP_Close();

    printf("RC = %d\n",rc);
    return rc;
}
```

1.2.2. Client example

The client example **CLIENT.C** connects to a server, specified by IP address and port. The program sends a block of 8 bytes to the server, receives back a block of 8 bytes and dumps the received data. Note that in a client program you have to specify IP address and port of a server you want to connect to.

```
/*-------------------------------------------------------------------*/
/*
* File: client.c
*
* Demo program for TCP client. The program attempts to connect to a server,
* specified by IP address and port. After successful connection the
* program sends a block of 8 bytes to the server and receives back a block
* of 8 bytes.
*
* Note: The program uses the default IP address of the camera 192.168.0.65.
*       Change if necessary !
*
* _TI_CAMERA defined     : Compile for camera
* _TI_CAMERA not defined : Compile for PC
*/
/*-------------------------------------------------------------------*/

#ifdef _TI_CAMERA
  #include <vcrt.h>
#else
  #include <stdio.h>
#endif
#include "tcp_lib.h"

#define CONN_TIME_OUT   2000           /* connection wait time in ms   */
#define RECV_TIME_OUT    500           /* receive timeout in ms        */
#define BUF_SIZE           8           /* I/O buffer size              */
#define SERVER_IP_ADDR  "192.168.0.65" /* server IP address            */
#define SERVER_PORT     2000           /* server listen port           */

int main(int argc, char *argv[])
{
    int  rc;
    int  iret;
    unsigned char buf[BUF_SIZE]  = { 0,1,2,3,4,5,6,7 }; /* send buffer  */
    unsigned char buf1[BUF_SIZE] = { 0,0,0,0,0,0,0,0 }; /* recv buffer  */
```

```
    char ip_str[32] = SERVER_IP_ADDR;

    TCP_SOCK DataSock;              /* data socket              */

/*.............. Get new IP address */
    if(argc >= 2)
    {
      strcpy(ip_str,argv[1]);      // save IP address
    }

/*.............. Init TCP */
    rc = TCP_Init();
    if(rc != TCP_OK)
    {
        printf("TCP_Init error = %d\n",rc);
        return rc;
    }

/*.............. Reset socket(s) */
    TCP_DataSockReset(&DataSock,1);

/*.............. Connect to a remote server */
    rc = TCP_Connect(&DataSock,ip_str,SERVER_PORT,CONN_TIME_OUT);
    if(rc != TCP_OK) goto done;

/*.............. Send block */
    iret = TCP_SendBlock(&DataSock,buf,BUF_SIZE);
    if(iret != BUF_SIZE)
    {
        printf("TCP_SendBlock error: iret=%d buf_size=%d\n",iret,BUF_SIZE);
        rc = TCP_SEND_BLOCK_ERROR;
        goto done;
    }

/*.............. Receive block */
    iret = TCP_RecvBlock(&DataSock,buf1,BUF_SIZE,RECV_TIME_OUT);
    if(iret != BUF_SIZE)
    {
        printf("TCP_RecvBlock error: iret=%d buf_size=%d\n",iret,BUF_SIZE);
        rc = TCP_RECV_BLOCK_ERROR;
        goto done;
    }
    else
    {
        printf("recv: %d %d %d %d %d %d %d %d\n",
            buf1[0],buf1[1],buf1[2],buf1[3],
            buf1[4],buf1[5],buf1[6],buf1[7]);
    }

/*.............. Exit */
done:
    TCP_Disconnect(&DataSock);
    TCP_Close();

    printf("RC = %d\n",rc);
    return rc;
}
```

2. Resources

This chapter presents in details the library resources.

2.1. Sockets

The library uses stream sockets. Stream sockets provide reliable peer to peer transfer by the TCP protocol. When you receive an OK transfer code, it is guaranteed that all sent data is received by the remote peer in the same byte order. Stream sockets are byte oriented (as indicated by the name). Data blocks are sent and received as sequences of bytes. For example you can send two blocks of 50 bytes and the receiver can read one block of 100 bytes or 4 blocks of 25 bytes..

The socket type **TCP_SOCK** used in our library represents an upgrade (extension) of the low-level I/O socket (look at **TCP_LIB.H**). It is a structure, which includes the traditional socket – the **socket** member, and contains some other members:

- Remote peer parameters, which are set when a connection is done successfully.
- The flag **connection_exists** shows current connection state – 0=off, 1=on.
- Other parameters, needed by the high-level VIMOS I/O tools.

You can set custom-defined socket options using the **socket** member using the respective (PC or camera) low-level TCP/IP library. Note that some socket options can't be changed at any time.

2.2. Socket options

When creating sockets, the functions `TCP_Listen` and `TCP_Connect` set some default socket options. Note that options of the listen socket become attributes of the server's data socket when a client's connection request is accepted by `TCP_AcceptConnection`.

Socket options for PC library:

- Hard (abort) socket disconnection – all I/O is stopped immediately.
- Size of receive buffer = 1 MByte.
- Re-use addresses before bind.

Socket options for camera library:

- Hard (abort) socket disconnection.
- Size of send buffer = size of sensor image (`getvar(VPITCH)*getvar(VWIDTH) + 256`). This option enables maximum transfer speed when sending images from camera to PC.
- Size of receive buffer = 80000 bytes.

Internally the library utilizes low-level functions `send` and `recv` in "no wait" mode, i.e. they send or receive the maximum possible bytes and return control immediately. High-level functions like `TCP_SendBlock` and `TCP_RecvBlock` wait for the end of the transfer (the receive function waits with timeout).

2.3. Connection modes

The library supports connection in two modes – server or client. The server mode requires creation of a listen socket by `TCP_Listen`, which is used to start listening for connection requests from remote client(s). `TCP_AcceptConnection` accepts requests and creates respective data sockets for each successful connection. The client mode is more simple – `TCP_Connect` makes connection with remote server and creates a data socket, further used in the send/receive operations.

Once you have made a connection, you can use equivalent code to transfer data between the server and the client. Remember to start the server first.

2.4. Multiple data sockets

You can use several data sockets for data transfer. Each socket should be created and connected in one of the two modes – client or server. Each connected socked can be passed as argument to the data transfer functions. It is possible (but not recommended) to use simultaneously client and server sockets in one program.

2.5. Memory requirements

When creating sockets, the library functions allocate memory for send/receive buffers (see "*2.2. Socket options*"). Keep in mind that creating of too many sockets can cause memory problems especially on the cameras with 16 Mbytes DRAM. The memory is released when the socket is disconnected by `TCP_Disconnect`.

***ATTENTION**. Don't forget to disconnect all created sockets when you finish an I/O transfer.*

3. Usage

This chapter describes how to use the TCP library. You can create programs with server or client sockets. A pair of a client and a server socket can exchange data. You can't make connection between two client or two server sockets. The server and the client programs usually work on different platforms – PC's and/or cameras. You can make connection between PC and camera, PC and PC or camera and camera.

To realize a successful connection, the server and the client should have matching parameters. If the server platform has IP address **`server_ip_addr`** and listens on port **`server_listen_port`**, then the client should make connection by:

```
TCP_Connect(&DataSock, server_ip_addr, server_listen_port, time_out);
```

If you are creating a program for communication with VIMOS, sockets in your program and VIMOS I/O tools should be configured respectively:

- Server socket – connects to VIMOS I/O tools with a client device.
- Client socket – connects to VIMOS I/O tools with a server device.

***ATTENTION**. The recommended execution sequence is to start the server program first.*

3.1. Header files

The header file **TCP_LIB.H** contains all definitions and prototypes, necessary to call functions from the basic library. Here are the prototypes of all initialization and connection functions, which are needed when you use also the VIMOS library. Include the header file in your C/CPP source files.

The header file **TCP_IO.H** contains definitions and prototypes, necessary to call functions from the VIMOS library. Include this file in your code when if you are creating a program for communication with VIMOS. When working with VIMOS library, you need prototypes of initialization and connection functions from the basic library, which are contained in **TCP_LIB.H**, so **TCP_LIB.H** is included in **TCP_IO.H**.

***ATTENTION**. Do not use macros and type definitions from the header files, which are not documented in this manual. They are for internal use only and may be changed without a notice*

3.2. Initialization

You should initialize the library by `TCP_Init` before calling any other I/O function. Reset all used sockets immediately after `TCP_Init`. Before exit (or when I/O is no longer needed), disconnect all sockets and close the library by `TCP_Close`. This initialization is obligatory for both the basic library and the VIMOS library. In general, a main I/O function should have the following structure (here you may use several data sockets):

```
#include "tcp_lib.h"
#include "tcp_io.h"          /* optional – for VIMOS library only */

void main()
{
    int rc;
    TCP_SOCK ListenSock;     /* listen socket (server mode only) */
```

```
    TCP_SOCK DataSock1;       /* data socket 1                    */
    TCP_SOCK DataSock2;       /* data socket 2                    */
    . . . . . .
/* Note: You can use a data socket buffer: TCP_SOCK SockBuf[N]; */

/* Initialize library */
    rc = TCP_init();
    if(rc != TCP_OK)
    {
       printf("TCP init error = %d\n", rc):
       return;
    }

/* Reset sockets */
    TCP_DataSockReset(&ListenSock,1);    /* server mode only */
    TCP_DataSockReset(&DataSock1,1);
    TCP_DataSockReset(&DataSock2,1);
    . . . . . .
/* Note: You can reset a whole data socket buffer by:
         TCP_DataSockReset(SockBuf,N); */

/* Connect server and/or client sockets */
    . . . . . .

/* Call send and receive functions from the basic library and/or
   I/O functions from the VIMOS library. */
    . . . . . .

/* Disconnect and close all sockets */
    TCP_Disconnect(&ListenSock);        /* server mode only */
    TCP_Disconnect(&DataSock1);
    TCP_Disconnect(&DataSock2);
    . . . . . .

/* Close library */
    TCP_Close();
    return;
}
```

3.3. Connecting server sockets

Declare a listen socket and one or more data sockets. Call `TCP_Listen` to create a listen socket and to start listening on the specified port for connection requests from remote servers. You can check for and accept connection requests by `TCP_AcceptConnection`. In case of successful connection the function creates a data socket, which further can be passed to the data transfer functions. The function `TCP_GetConnectionState` returns non-zero value if the data socket is connected or 0 when not connected.

The next piece of code demonstrates the connection of one data socket as a server. The code contains a wait connection loop, but you may skip the wait loop and put the connection code:

```
TCP_AcceptConnection(...);

TCP_GetConnectionState(...);
```

in a custom loop with other code. Library and socket initialization is omitted for clarity (see "*3.2. Initialization*").

```
#include "tcp_lib.h"
#define CONN_TIME_OUT 9000    /* connection timeout in ms */
    int rc;
```

```
    TCP_SOCK ListenSock;      /* listen socket  */
    TCP_SOCK DataSock;        /* data socket    */
    unsigned long time;       /* time in ms     */
    . . . . . .

/* Create a listen socket and start listening on port 2000 */
    rc = TCP_Listen(&ListenSock,2000);
    if(rc != TCP_OK) goto error_label;

/* Wait connection loop */
    time = get_time();
    for(;;)
    {
        rc = TCP_AcceptConnection(&ListenSock,&DataSock,1);
        if(rc != TCP_OK) goto error_label;

        if(TCP_GetConnectionState(&DataSock)) break;    /* connection OK */
        if(get_time() - time >= CONN_TIME_OUT)
        {
           rc = TCP_NO_CONNECTION_ERROR;
           goto error_label;
        }
    } /* for(;;) */


/* Call data-transfer functions with socket argument &DataSock */
    . . . . . .
```

3.4. Connecting client sockets

Declare one or more data sockets. Call `TCP_Connect` to create client data socket(s) and connect to specified server(s). You should specify valid server IP addresses and ports. After successful connection you can pass data sockets as arguments to data transfer functions.

The next code demonstrates connection of one client data socket. Library and socket initialization is omitted for clarity (see "*3.2. Initialization*").

```
#include "tcp_lib.h"
#define CONN_TIME_OUT   2000            /* connection wait time in ms   */
#define SERVER_IP_ADDR  "192.168.0.65"  /* server IP address (camera)   */
#define SERVER_PORT     2000            /* server listen port           */
    int rc;
    TCP_SOCK DataSock;        /* data socket    */
    . . . . . .

/* Connect to a remote server */
    rc = TCP_Connect(&DataSock,SERVER_IP_ADDR,SERVER_PORT,CONN_TIME_OUT);
    if(rc != TCP_OK) goto error_label;


/* Call data-transfer functions with socket argument &DataSock */
    . . . . . .
```

***ATTENTION**. To achieve robust connection(s), start server applications before client applications – i.e. the server programs should wait for connection requests from clients. **This is especially important for camera client programs connecting to PCs**. On some systems (Windows XP with LAN firewall enabled), the `TCP_Connect` function may hang on waiting for connection if the server is not started.*

3.5. Using multiple sockets

You can use multiple sockets for data transfer in server or client mode. You may use for example different sockets for different types of data. Thus by receiving data on a given socket you may determine what data is coming in.

Follow the connection scheme for each socket as described in the previous sections. Execute `TCP_Connect` for each client socket or the couple of functions `TCP_AcceptConnection` and `TCP_GetConnectionState` for each server socket. After successful connection, pass the sockets as arguments to the data-transfer functions. The example programs **SERVERM.C** and **CLIENTM.C** demonstrate how to work with multiple sockets. Please look at the included file **CONN_MUL.C**, which actually connects multiple server and client sockets.

***ATTENTION**. Sequential client connection requests, issued from PC with identical IP addresses and ports, may be received on camera in different order.*

The functions `TCP_AcceptMul` and `TCP_ConnectMul` in the file **CONN_MUL.C** offer a solution of the problem. After each acceptation, `TCP_AcceptMul` sends one byte via the connected server socket. On each successful connection, `TCP_ConnectMul` waits for a byte via the connected client socket before going to next connection request. Another method is to wait some time (no less than 400-500 ms) after each client request.

3.6. Transferring data

Once you have connected data socket(s) in client or server mode, you can call data transfer functions from the basic library and/or the VIMOS library. Pointers to connected sockets should be passed as arguments to the data transfer functions.

3.6.1. Basic data transfer functions

The data transfer functions from the basic library are byte or block oriented. Most of the “receive” functions support timeouts. Below is given a brief description of the functions. Refer to the function reference for more details:

Function	Description
`TCP_SendByte`	Send byte
`TCP_RecvByte`	Receive byte, no wait
`TCP_WaitByte`	Wait for byte with timeout
`TCP_SendBlock`	Send block
`TCP_RecvBlock`	Receive block with timeout

The code below demonstrates usage of basic data transfer functions with one socket. Library initialization and socket connection are omitted for clarity (see previous sections).

```
#include "tcp_lib.h"
#define RECV_TIME_OUT   300       /* receive timeout in ms  */
#define BUF_SIZE          8       /* I/O buffer size        */
    int iret;
    unsigned char buf[BUF_SIZE];  /* I/O buffer             */
    int byt;                      /* I/O byte               */
    TCP_SOCK DataSock;            /* data socket            */
    . . . . . .
```

```
/* Send byte */
    iret = TCP_SendByte(&DataSock,byt);
    if(iret != 0) goto error_label;

/* Receive byte if available, no wait */
    byt = TCP_RecvByte(&DataSock);
    if(byt < 0) goto error_label;   /* byte N/A or socket error */

/* Wait for byte with timeout */
    byt = TCP_WaitByte(&DataSock,RECV_TIME_OUT);
    if(byt < 0) goto error_label;   /* byte N/A or socket error */

/* Send block */
    iret = TCP_SendBlock(&DataSock,buf,BUF_SIZE);
    if(iret != BUF_SIZE) goto error_label;

/* Receive block */
    iret = TCP_RecvBlock(&DataSock,buf,BUF_SIZE,RECV_TIME_OUT);
    if(iret != BUF_SIZE) goto error_label;
```

3.6.2. VIMOS I/O functions

The VIMOS I/O functions transfer data in special format, which complies with the VIMOS I/O tools. Library initialization and socket connection are described in the preceding sections. The functions work in couples – each send function transfers data to respective VIMOS receive tool or respective receive function. All receive functions support timeouts. Refer to the function reference chapter for more details.

Function	Description
`TCP_SendPtlist`	Send point-list
`TCP_SendImage`	Send image
`TCP_SendResult`	Send result
`TCP_SendString`	Send string
`TCP_SendBinary`	Send binary (no protocol, same as `TCP_SendBlock`)
`TCP_RecvPtlist`	Receive point-list
`TCP_RecvImage`	Receive image
`TCP_RecvResult`	Receive result
`TCP_RecvString`	Receive string
`TCP_RecvBinary`	Receive binary (no protocol, same as `TCP_RecvBlock`)

See chapter "*4. Examples*" for example(s) with VIMOS I/O functions.

3.7. Copy protection

A registration file protects the TCP library on camera. Send the serial number of your VC20xxE camera to the program vendor and you will receive a registration file with the name **TCPKEY.MSF**. Load the file to camera flash. Now you will be able to execute camera programs, linked with the TCP library **TCP_CAM.LIB**.

3.8. Compiling and linking programs with TCP library

This section describes how to compile and link demo and custom programs on camera and PC. The PC demo programs are simple Windows console applications, executed in the command prompt window (the DOS window).

***ATTENTION**. When compiling your code for camera, you should define the macro `_TI_CAMERA` (see the batch file **TCL.BAT**). Camera programs need a registration key.*

3.8.1. Compiling and linking camera programs

Compile your code by the C/C++ compiler of the TI Code Composer Studio. Link the object modules with the TCP library **TCP_CAM.LIB**. We recommend using the following batch file **TCL.BAT** supposing the TI compiler and the VCRT development software are installed in **`c:\ti`**:

```
@echo off
cl6x %1.c -o3 -mi100000 -pdr -pdse225 -qq -d_TI_CAMERA
if ERRORLEVEL 1 goto end
lnk6x -q -s -u _c_int01 %1.obj -m %1.map -o %1.out -l tcp_cam.lib c:\ti\work\cc.cmd
if ERRORLEVEL 1 goto end

copy %1.out exec.out
c:\ti\util\econv %1
c:\ti\util\scvt
copy adsp.msf %1.msf
:end
```

***ATTENTION**. This batch file requires strict prototyping of all called functions.*

Examples for compilation and linking of demo programs:

```
tcl server
tcl serverm
```

The batch file produces MSF files (**SERVER.MSF**, **SERVERM.MSF**), which should be loaded to camera flash by a TCP terminal or by the VIMOS Simulator. Remember that TCP camera programs need registration file (see "*3.7. Copy protection*").

3.8.2. Compiling and linking PC programs

The suggested batch files create demo console applications by the C/C++ compiler of Microsoft® Visual Studio 6.00 or higher in batch mode. Of course you may create Visual Studio project(s) for this purpose and/or insert TCP code in arbitrary Win32 application, which uses graphical interface and MFC. Note that the PC library **TCP_PC.LIB** does not use MFC.

To setup the C/C++ compiler in batch mode it may be necessary to run the batch file **VCVARS32.BAT**, which is usually located in:

`C:\Program Files\Microsoft Visual Studio\Vc98\Bin`

If you don't have the necessary settings by default, you should run this file once after each opening of a new command prompt window. This could be done by the batch file **SET_VC.BAT** supplied with the library:

```
echo Setup Visual C/C++ compiler in batch mode
call c:\PROGRA~1\MICROS~3\Vc98\Bin\vcvars32.bat
```

You can compile and link the PC demo programs by the following batch file (see **VCL.BAT**):

```
echo Visual C/C++ compile and link
cl.exe /nologo /W3 /GX %1.c tcp_pc.lib wsock32.lib
```

If you create your own Visual Studio project(s), remember to link your code with the libraries **TCP_PC.LIB** and **WSOCK32.LIB**.

4. Examples

This chapter describes the examples supplied with the library. You may compile and link the example source files or use ready executable modules for camera (MSF files) and PC (EXE files).

***ATTENTION**. Remember that if you are going to rebuild and execute the camera examples, you will need a registration file (see "*3.7. Copy protection*"). Upload the registration file TCPKEY.MSF to camera before running your TCP camera program.*

If you are going to use the ready camera MSF examples, you don't need a registration file.

The default camera's IP address is 192.168.0.65. This is the default address, used in the respective PC client executables. If you have camera with different IP address, you should specify IP address when you start the PC demo programs, for example:

```
client 192.168.0.66
```

4.1. Basic library example – client and server

These are the examples, described in the introduction chapter. The client program (both the source file and the executable module) define camera's IP address and therefore is designed to run on PC. The server program should run on camera. If you want to switch the target platforms, you should modify the source code of the client program (server IP address) and recompile the examples (the source code is portable).

File	Description
SERVER.C	Server demo - source code for camera
CLIENT.C	Client demo - source code for PC
SERVER.MSF	Server demo - executable module for camera
CLIENT.EXE	Client demo - executable module for PC

Test execution:

Load the registration file **TCPKEY.MSF** to camera (not needed for the ready MSF demo files). Load **SERVER.MSF** to camera and start the program ***server*** from a TCP/IP terminal. The server program will wait maximum 9 seconds for connection. Start no later than 9 seconds the PC program ***client*** in a command prompt window. If you have changed the default IP address of the camera, you should start the client program by "`client` ***new_ip_address***", where the new IP address is in string format `xxx.xxx.xxx.xxx`.

You should receive a data dump of **`0,1,…,7`** and return code **`RC=0`**.

4.2. Basic library example – client and server with multiple sockets

The following examples demonstrate the simultaneous usage of several sockets. Here the code is not portable – the server program is created explicitly for camera. The client is a PC program – it is a simple TCP terminal, which may be used to enter camera shell commands like `dir`. The terminal program is used to start the camera server program and to get camera response (if any) from the Telnet port 23.

File	Description
SERVERM.C	Server demo with multiple sockets - source code for camera

CLIENTM.C	Client demo with multiple sockets – source code for PC
CONN_MUL.C	Server/client connection functions for multiple sockets – portable code. This file is included in the files SERVERM.C and CLIENTM.C.
SERVERM.MSF	Server demo with multiple sockets - executable module for camera
CLIENTM.EXE	Client demo with multiple sockets - executable module for PC

Test execution:

Load the registration file **TCPKEY.MSF** to camera (not needed for the ready MSF demo files). Load **SERVERM.MSF** to camera (don't start it, this is done by the PC program) and exit the terminal. Start the program ***clientm*** in a command prompt window. If you have changed the default IP address of the camera, you should start the client program by "`clientm` ***new_ip_address***", where the new IP address is in string format `xxx.xxx.xxx.xxx`.

Press Enter several times to see the camera shell prompt '$'. Press F1 to start the test. This command starts the camera program ***serverm*** and then executes PC client connection code. The camera and PC will connect exchange data through 4 sockets. You must receive the following messages in an endless loop:

```
PC: --- Iter = xxx ---
PC: isock=0: Send/recv block OK
PC: isock=1: Send/recv block OK
PC: isock=2: Send/recv block OK
PC: isock=3: Send/recv block OK
```

Press Q to disconnect camera or any other key to disconnect PC and return to the camera shell. Depending on the disconnection type, the PC or the camera will display an error message in the following format:

```
PC: . . . . .
Cam: . . . . .
```

Press Esc or F10 to exit the PC program or press F1 to restart the test.

***ATTENTION**. You can press F1 to restart the test when there are no pending shell commands like $xx…x - you should see the prompt character only. We recommend pressing Enter before pressing F1.*

4.3. VIMOS library example

The following examples demonstrate how to transfer data between a stand alone program and the VIMOS system. VDEMO.C contains portable code for camera and PC, which executes functions from the VIMOS TCP library. It connects in client mode to the VIMOS user-program (server).

You can run a ready PC executable module VDEMO.EXE, compiled from VDEMO.C. VIMOS should be started with VDEMO.AEF before you start VTDEMO on PC.

***INFORMATION**. You can swap the platforms by recompilation of VDEMO.C for camera. The VIMOS user-program should be started in Simulator. Remember that you should change the IP address in VDEMO.AEF (the create server tool).*

It is also possible to connect two PCs or two cameras – one should be running VIMOS and the other - VDEMO.

File	Description
VDEMO.C	Source code of client stand-alone program (portable)
VDEMO.AEF	VIMOS server user-program with I/O tools.
VDEMO.EXE	PC executable module created from VDEMO.C, which executes I/O

	functions from the VIMOS TCP library.

Test execution:

Follow the next instructions to run the stand-alone program VDEMO on PC and to communicate with VIMOS on the camera. You can use the ready test VDEMO.EXE:

- Open VDEMO.AEF in the Editor. Set your computer's IP address in the "Create server device" tool. Export the program to the Simulator and save it as file `up0.vm`.
- Load `up0.vm` to camera.
- Start VIMOS on camera with the program `up0.vm`.
- Open a command-prompt window and start VDEMO on PC. If you have changed the default camera IP address 192.168.0.65, you should start the program by "`vdemo` ***`new_ip_address`***", where the new IP address is in string format `xxx.xxx.xxx.xxx`.

You should receive in endless loop OK messages for the tested I/O tools. Press a key to disconnect and terminate the VDEMO program. You can restart VDEMO after a little wait time - when the "Create server device" tool result, shown on the camera monitor, receives a nonzero error value.

***INFORMATION**. This example was created using VIMOS version 2.61. Since VIMOS is constantly evolving it is possible that this example requires changes to be made in order to run it under newer VIMOS versions. If you have downloaded a newer VIMOS version and this example is not running, please send us and e-mail.*

Alternatively, you can download VIMOS v2.61 here:

5. Basic library function reference

This chapter presents detailed information about the functions in the basic TCP library.

TCP_Init (Initialize TCP library)

Prototype:

```
int TCP_Init ()
```

Description:

The function initializes the TCP library before connecting in server or client mode. This function should be called before any other library function.

Parameters:

None

Return code:

`TCP_OK`	Success
`TCP_AFXSOCKETINIT_ERROR`	AfxSocketInit failed (PC only)

TCP_Close (Close TCP library)

Prototype:

```
void TCP_Close ()
```

Description:

The function closes the TCP library. This function should be called when the I/O transfer is done and all sockets are disconnected.

Parameters:

None

Return code:

None

TCP_DataSockReset (Reset sockets)

Prototype:

```
void ( TCP_SOCK *data_sock, int data_sock_cnt )
```

Description:

The function resets a socket buffer. It should be called once after `TCP_Init` and before any connection and I/O transfer functions. Note that `TCP_Disconnect` resets the disconnected socket and you may use the socket for new connection without calling `TCP_DataSockReset`.

Parameters:

`data_sock`	Input/output socket buffer
`data_sock_cnt`	Input number of sockets in the `data_sock` buffer

Return code:

None

TCP_Listen (Start listening for client requests)

Prototype:

```
int TCP_Listen ( TCP_SOCK *list_sock, int port )
```

Description:

The function creates a listen socket and starts listening on the specified port for a connection request from a TCP client. Connection requests are accepted by the function `TCP_AcceptConnection`.

***ATTENTION**. Disconnect the created listen socket by `TCP_Disconnect` when the I/O transfer is done.*

Parameters:

`list_sock`	Input/output listen socket
`port`	Input listen port (0 = default listen port 2000)

Return code:

`TCP_OK`	Success
`TCP_SOCK_CREATE_ERROR`	Listen socket create failed
`TCP_STREAM_BIND_ERROR`	Stream bind failed
`TCP_LISTEN_FAIL_ERROR`	Listen failed

TCP_AcceptConnection (Accept connection requests)

Prototype:

```
int TCP_AcceptConnection ( TCP_SOCK *listen_sock, TCP_SOCK *data_sock,
                           int data_sock_cnt )
```

Description:

The function accepts connection requests from remote clients on the input socket `listen_sock`, which should be previously initialized by `TCP_Listen`. It setups the input/output data socket buffer `data_sock` by storing parameters of connected sockets into unconnected (non-initialized) buffer items. The function must be called regularly to check for new connections and/or to restore broken connections. To achieve exact correspondence between connected server and client sockets, we recommend calling the function multiple times with `data_sock_cnt = 1`.

***ATTENTION**. Several client connection requests, issued from PC to camera with identical IP addresses and ports, may be received on the camera in different order – see the example with multiple sockets.*

The connected data socket(s) can be passed as arguments to the data transfer functions. Sockets connected by this function are called ***server*** sockets.

***ATTENTION**. All connected sockets should be disconnected by `TCP_Disconnect` when the I/O transfer is done. Sockets are disconnected automatically if the connection is broken.*

Parameters:

`listen_sock`	Input listen socket
`data_sock`	Input/output data socket buffer
`data_sock_cnt`	Input number of sockets in the `data_sock` buffer

Return code:

`TCP_OK`	Success
Other	Accept connection failed

TCP_GetConnectionState (Get socket connection state)

Prototype:

```
int TCP_GetConnectionState ( TCP_SOCK *data_sock )
```

Description:

The function returns the connection state of the input data socket. You may check regularly the connection state of a given socket. If the connection is broken, you can reconnect the socket in server mode by `TCP_AcceptConnection` or in client mode by `TCP_Connect`.

Parameters:

`data_sock`	Input data socket

Return code:

`0`	No connection
`1`	Connection OK – you

TCP_Connect (Connect client socket)

Prototype:

```
int TCP_Connect ( TCP_SOCK *sock, char *ip_addr, int port, int wait_time )
```

Description:

The function creates and connects a to a remote server, specified by `ip_addr` and `port`. The parameter `wait_time` specifies connection timeout.

***ATTENTION**. If a respective server has not been started on some systems (Windows XP with LAN firewall enabled), this function may hang on camera by waiting for connection instead of returning control after the timeout interval. We recommend starting a server application on a remote platform before connecting a client socket.*

The connected data socket can be passed as argument to the data transfer functions. Sockets connected by this function are called ***client*** sockets.

***ATTENTION**. All connected sockets should be disconnected by `TCP_Disconnect` when the I/O transfer is done. Sockets are disconnected automatically if the connection is broken.*

Parameters:

`sock`	Input/output socket
`ip_addr`	Input string with IP address of remote server (NULL specifies default VC20xxE IP address "192.168.0.65")
`port`	Input port (0 specifies default camera shell port 23)

`wait_time`	Input connection timeout in ms

Return code:

`TCP_OK`	Success
`TCP_SOCKET_CREATE_ERROR`	Socket create failed
`TCP_CONNECT_TIMEOUT_ERROR`	Connection timeout error
`TCP_INVALID_IP_ERROR`	Invalid IP address
`TCP_CONNECT_FAIL_ERROR`	Connection failed – probably no remote server

TCP_Disconnect (Disconnect socket)

Prototype:

```
void TCP_Disconnect ( TCP_SOCK *sock )
```

Description:

The function disconnects and closes the specified socket.

***ATTENTION**. Use this function to close all connected listen and data sockets. Remember that functions which create sockets (`TCP_Listen`, `TCP_AccepConnection` and `TCP_Connect`) allocate memory. The closing of a socket frees this memory.*

Parameters:

`sock`	Input/output socket

Return code:

None

TCP_SendByte (Send byte)

Prototype:

```
int TCP_SendByte ( TCP_SOCK *sock, int byt )
```

Description:

The function sends a byte on the specified socket. The socket must be previously connected in server or client mode.

Parameters:

`sock`	Input socket
`byt`	Input byte to send

Return code:

`0`	Success
`TCP_SEND_BYTE_ERROR`	Send failed (socket error)

TCP_RecvByte (Receive byte)

Prototype:

```
int TCP_RecvByte ( TCP_SOCK *sock )
```

Description:

The function receives a byte (if available) from the specified socket. The socket must be previously connected in server or client mode.

Parameters:

`sock`	Input socket

Return code:

`>= 0`	Received byte (success)
`-1`	Byte not available or socket error

TCP_WaitByte (Wait for byte)

Prototype:

```
int TCP_RecvByte ( TCP_SOCK *sock )
```

Description:

The function waits to receive a byte from the specified socket. The socket must be previously connected in server or client mode.

Parameters:

`sock`	Input socket
`wait`	Input wait time in ms: `0` : wait forever `>0` : wait time in milliseconds

Return code:

`>= 0`	Received byte (success)
`-1`	Byte not available (timeout or socket error)

TCP_SendBlock (Send block)

Prototype:

```
int TCP_SendBlock ( TCP_SOCK *sock, unsigned char *buf, int cnt )
```

Description:

The function sends a block of bytes on the specified socket and waits until all block bytes are sent. The socket must be previously connected in server or client mode.

Parameters:

`sock`	Input socket
`buf`	Input data buffer to send
`cnt`	Input number of bytes to send (size of `buf`)

Return code:

`>=0`	Number of sent bytes (success)
`TCP_SEND_ERROR`	Send error
`TCP_NO_CONNECTION_ERROR`	No connection error

TCP_RecvBlock (Receive block)

Prototype:

```
int TCP_RecvBlock ( TCP_SOCK *sock, unsigned char *buf, int cnt, int wait )
```

Description:

The function receives a block of bytes from the specified socket and stores the received data into `buf`. In case of timeout, the function returns the number of bytes received so far, which is less than `cnt`. The socket must be previously connected in server or client mode.

Parameters:

`sock`	Input socket
`buf`	Output data buffer
`cnt`	Input number of bytes to receive (size of `buf`)
`wait`	Input wait time in ms: `0` : wait forever `>0` : wait time in milliseconds

Return code:

`>=0`	Number of received bytes (`RC < cnt` : timeout error)
`TCP_RECV_ERROR`	Receive error
`TCP_NO_CONNECTION_ERROR`	No connection error

TCP_SendCmd (Send command)

Prototype:

```
int TCP_SendCmd ( TCP_SOCK *sock, char *cmd, int mode )
```

Description:

The function sends a 0-terminated command string on specified socket and optionally reads echo bytes from the socket. The 0-terminating char is not sent – it is replaced by a new line character 0x0A. Usually this function is used to send commands to the camera shell via the Telnet port 23.

Parameters:

`sock`	Input socket
`cmd`	Input command string
`mode`	Input mode of operation: `0` : don't read command echo `1` : read command echo

Return code:

`0`	Success
Other	Socket error

TCP_ReadEcho (Read camera echo)

Prototype:

```
void TCP_ReadEcho ( TCP_SOCK *sock, int dump, unsigned int wait )
```

Description:

The function reads (and optionally dumps) all bytes received from the specified socket, until a timeout of `wait` ms occurs. Usually this function is used to read camera shell response on the Telnet port 23.

Parameters:

`sock`	Input socket
`dump`	Input dump flag: `0` : no dump `1` : dump received chars
`wait`	Input read timeout in ms

Return code:

None

get_time (Get system time)

Prototype:

```
unsigned long get_time ()
```

Description:

The function returns the system time in milliseconds. The time grows from 0 upwards.

Parameters:

None

Return code:

System time in ms

6. VIMOS library function reference

This chapter presents detailed information about the functions in the VIMOS TCP library. Here the functions transfer data in a format compatible with the VIMOS I/O tools when working in TCP mode (with TCP I/O device). The formats of the various send/receive data blocks are described in **TCP_IO.H**.

TCP_SendPtlist (Send point-list)

Prototype:

```
int TCP_SendPtlist ( TCP_SOCK *sock, unsigned char *ptl_buf,
                     int ptl_len, int wait )
```

Description:

The function sends a point-list on the specified socket. The receiver of the point-list should be a "Receive point-list" tool. The format of the point-list buffer `ptl_buf` is described in TCP_IO.H. The calling function is responsible to fill correct buffer data before calling this function.

Parameters:

`sock`	Input socket
`ptl_buf`	Input buffer with point-list items in format: `item_0, item_1, ...` Each item is a **`PTL_ITEM`** structure (see TCP_IO.H)
`ptl_len`	Input length of the point-list buffer in number of items. The actual size of `ptl_buf` in bytes is: `ptl_len * sizeof(PTL_ITEM)`
`wait`	Input wait time: `0` = don't wait for end of transfer `>0` = wait **`wait`** ms for reply, sent back from receiver on the end of the transfer

Return code:

`TCP_OK`	Success
`TCP_SEND_PTLIST_HDR_ERROR`	Send header error
`TCP_SEND_PTLIST_DATA_ERROR`	Send data error
`TCP_SEND_PTLIST_REPLY_ERROR`	Invalid reply byte received (`wait > 0`)
`TCP_SEND_PTLIST_TIMEOUT`	Timeout error when waiting for reply from the receiver tool (`wait > 0`)

TCP_RecvPtlist (Receive point-list)

Prototype:

```
int TCP_RecvPtlist ( TCP_SOCK *sock, unsigned char *ptl_buf,
                     int *ptl_len, int wait )
```

Description:

The function receives a point-list from specified socket. The sender of the point-list should be a "Send point-list" tool. The format of the point-list block is described in TCP_IO.H.

The function saves the received point-list into the output buffer `ptl_buf`. The buffer should be allocated by the calling function and must be large enough to hold the longest point-list, which may be received. If not, the function returns with error code `TCP_RECV_PTLIST_BUF_OVF`.

Parameters:

`sock`	Input socket
`ptl_buf`	Output buffer with point-list items in format: `item_0, item_1, ...` Each item is a **`PTL_ITEM`** structure (see TCP_IO.H)
`ptl_len`	Input/output length of the point-list buffer in number of items: On input : max number of items in `ptl_buf` On output : actual number of items received in `ptl_buf` Note: The actual size of `ptl_buf` in bytes is: `Ptl_len * sizeof(PTL_ITEM)`
`wait`	Input wait time: `0` = wait forever until a point-list is received `>0` = wait maximum **`wait`** ms for point-list before a timeout error

Return code:

`TCP_OK`	Success
`TCP_RECV_PTLIST_NO_DATA`	No data present from a send point-list tool
`TCP_RECV_PTLIST_BUF_OVF`	Point-list buffer overflow
`TCP_RECV_PTLIST_TIMEOUT`	Timeout when receiving point-list
Other	Socket error(s)

TCP_SendImage (Send image)

Prototype:

```
int TCP_SendImage ( TCP_SOCK  *sock, IMAGE *img, int type, int qual,
                    int wait )
```

Description:

The function sends an image on the specified socket. The receiver of the image should be a "Receive image" tool. The format of the image block is described in TCP_IO.H.

Parameters:

`sock`	Input socket
`img`	Input image in IMAGE format (defined in TCP_IO.H)
`type`	Input type of image block to send: `0` = Bitmap image `1` = Convert `img` into JPEG file and send JPEG file.

Note: Currently the JPEG type is not supported.

`qual`	Input JPEG quality in the range [1,100], used when `type=1`. Lower `qual` value means lower JPEG quality and smaller JPEG file.
`wait`	Input wait time: `0` = don't wait for end of transfer `>0` = wait **`wait`** ms for reply, sent back from receiver on the end of the transfer

***ATTENTION**. The **`wait`** parameter can be used for synchronization and test purposes. Remember that on camera the "Send image" tool returns when data is copied into a system VCRT buffer but before the data is received by the "Receive image" tool. When a next "Send image" tool is executed, it will wait until the previous transfer is over. Thus you may receive quite different tool execution times.*

Return code:

`TCP_OK`	Success
`TCP_SEND_IMAGE_HDR_ERROR`	Send header error
`TCP_SEND_IMAGE_DATA_ERROR`	Send data error
`TCP_SEND_IMAGE_TIMEOUT`	Timeout when waiting for reply from the receiver tool (`wait>0`)
`TCP_SEND_IMAGE_REPLY_ERROR`	Invalid reply byte received (`wait>0`)

TCP_RecvImage (Receive image)

Prototype:

```
int TCP_RecvImage ( TCP_SOCK  *sock, IMAGE *img, int img_buf_size,
                    int wait, int *type, int *qual )
```

Description:

The function receives an image from the specified socket. The sender of the image should be a "Send image" tool. The format of the image block is described in TCP_IO.H.

The function saves the received image and its dimensions into the output image `img` (an `IMAGE` structure). If the received image is a JPEG file, it is stored into the image memory buffer `img_buf` again, but the other `IMAGE` members are not used. The image buffer should be allocated by the calling function and must be large enough to hold the received image. The maximum size of the image buffer is specified by the `img_buf_size` argument. In case of buffer overflow, the function returns with error `TCP_RECV_IMAGE_BUF_OVF`.

ATTENTION**. Remember that on input **`img`** must have valid buffer pointer, stored by the calling function in **`img_buf`

Currently the "Send image" and the "Receive image" tools do not support image transfer in JPEG format.

Parameters:

`sock`	Input socket
`img`	Input/output image in IMAGE format (defined in TCP_IO.H). **On input:** The image buffer `img_buf` must be allocated by the calling function.

	On output: The image data is stored into `img_buf`. The image parameters are stored into `dx`, `dy` and `pitch`. If `type=1` the image buffer receives a JPEG file (**currently not supported**).
`img_buf_size`	Input size of the image buffer `img_buf`. The function checks this size to decide whether the received image can be stored in the image buffer.
`wait`	Input wait time: `0` = wait forever until image is received `>0` = wait maximum **`wait`** ms for image before a timeout error
`type`	Output type of received image: `0` = Bitmap image `1` = JPEG file (**currently not supported**)
`qual`	Output JPEG quality in the range [1,100], received when `type=1`. Lower `qual` value means lower JPEG quality and smaller JPEG file (**currently not supported**).

Return code:

`TCP_OK`	Success
`TCP_RECV_IMAGE_NO_DATA`	No data present from a send image tool
`TCP_RECV_IMAGE_BUF_OVF`	Image buffer overflow
`TCP_RECV_IMAGE_TIMEOUT`	Timeout when receiving image
Other	Socket error(s)

TCP_SendResult (Send result)

Prototype:

```
int TCP_SendResult ( TCP_SOCK *sock, void *res_buf, int type,
                     int data_fmt, int wait )
```

Description:

The function sends a result on the specified socket. The receiver of the result should be a "Receive result" tool. The format of the result block is described in TCP_IO.H.

Parameters:

`sock`	Input socket
`res_buf`	Input result buffer in format specified by `type`.

`type`	Input result type (see TCP_IO.H):
	`0 : float (size = 4)`
	`1 : point PTL_POINT (size = 8)`
	`2 : 16-bit integer TCP_INT16 (size = 2)`
	`3 : 32-bit integer TCP_INT32 (size = 4)`

`data_fmt`	Input format of sent data bytes:
	`0 : hex`
	`1 : binary (default)`
	`2 : ASCII`

3 : ASCII prompt

Note: Currently `data_fmt==1`(binary) is supported only !

`wait` Input wait time:

`0` = don't wait for end of transfer

`>0` = wait **`wait`** ms for reply, sent back from receiver on the end of the transfer

Return code:

`TCP_OK`	Success
`TCP_SEND_RESULT_HDR_ERROR`	Send header error
`TCP_SEND_RESULT_DATA_ERROR`	Send data error
`TCP_SEND_RESULT_REPLY_ERROR`	Invalid reply byte received (`wait > 0`)
`TCP_SEND_RESULT_TIMEOUT`	Timeout error when waiting for reply from the receiver tool (`wait > 0`)

TCP_RecvResult (Receive result)

Prototype:

```
int TCP_RecvResult ( TCP_SOCK *sock, void *res_buf, int *data_size,
                     int *type, int wait )
```

Description:

The function receives a result from the specified socket. The sender of the result should be a "Send result" tool. The format of the result block is described in TCP_IO.H.

The function saves the received result in the output buffer `res_buf`. The buffer should be allocated by the calling function and must be large enough to hold the longest result, which may be received. If not, the function returns with error code `TCP_RECV_RESULT_BUF_OVF`.

Parameters:

`sock` Input socket

`res_buf` Output result buffer in format specified by `type`. Should have minimum size of 8 bytes to receive all result types.

`data_size` Input/output size of result buffer:

On input : maximum `res_buf` size

On output : actual number of bytes stored in `res_buf`

`type` Output result type (see TCP_IO.H):

```
0 : float                    (data_size = 4)
1 : point PTL_POINT          (data_size = 8)
2 : 16-bit integer TCP_INT16 (data_size = 2)
3 : 32-bit integer TCP_INT32 (data_size = 4)
```

`wait` Input wait time:

`0` = wait forever until a result is received

`>0` = wait maximum **`wait`** ms for result before a timeout error

Return code:

TCP_OK	Success
TCP_RECV_RESULT_NO_DATA	No data present from a send result tool
TCP_RECV_RESULT_BUF_OVF	Result buffer overflow
TCP_RECV_RESULT_TIMEOUT	Timeout when receiving result
Other	Socket error(s)

TCP_SendString (Send string)

Prototype:

```
int TCP_SendString ( TCP_SOCK *sock, char *str_buf, int str_size,
                     int wait )
```

Description:

The function sends a string on the specified socket. The receiver of the string should be a "Receive string" tool. The format of the string block is described in TCP_IO.H.

Parameters:

sock	Input socket
str_buf	Input string buffer.
str_size	Input str_buf size (# of bytes to send)
wait	Input wait time: 0 = don't wait for end of transfer >0 = wait **wait** ms for reply, sent back from receiver on end of transfer

Return code:

TCP_OK	Success
TCP_SEND_STRING_HDR_ERROR	Send header error
TCP_SEND_STRING_DATA_ERROR	Send data error
TCP_SEND_STRING_REPLY_ERROR	Invalid reply byte received (wait > 0)
TCP_SEND_STRING_TIMEOUT	Timeout error when waiting for reply from the receiver tool (wait > 0)

TCP_RecvString (Receive string)

Prototype:

```
int TCP_RecvString ( TCP_SOCK *sock, char *str_buf, int *str_size,
                     int wait )
```

Description:

The function receives a string from the specified socket. The sender of the string should be a "Send string" tool. The format of the string block is described in TCP_IO.H.

The function saves the received string in the output buffer str_buf. The buffer should be allocated by the calling function and must be large enough to hold the longest string, which may be received. If not, the function returns with error code TCP_RECV_STRING_BUF_OVF.

Parameters:

`sock`	Input socket
`str_buf`	Output string buffer.
`str_size`	Input/output `str_buf` size: On input : maximum `str_buf` size On output : actual number of bytes stored in `str_buf` (a terminating byte is not stored in the string buffer)
`wait`	Input wait time: `0` = wait forever until a string is received `>0` = wait maximum **`wait`** ms for string before a timeout error

Return code:

`TCP_OK`	Success
`TCP_RECV_STRING_NO_DATA`	No data present from a send string tool
`TCP_RECV_STRING_BUF_OVF`	String buffer overflow
`TCP_RECV_STRING_TIMEOUT`	Timeout when receiving string
Other	Socket error(s)

TCP_SendBinary (Send binary)

Prototype:

```
int TCP_SendBinary ( TCP_SOCK *sock, unsigned char *buf, int cnt )
```

Description:

The function sends a block of bytes on the specified socket and waits until all block bytes are sent. The receiver of the data should be a "Receive binary" tool. The function does not implement any handshake or I/O protocol.

Parameters:

`sock`	Input socket
`buf`	Input data buffer to send
`cnt`	Input number of bytes to send (size of `buf`)

Return code:

`>=0`	Number of sent bytes (success)
`TCP_SEND_ERROR`	Send error
`TCP_NO_CONNECTION_ERROR`	No connection error

TCP_RecvBinary (Receive binary)

Prototype:

```
int TCP_RecvBinary ( TCP_SOCK *sock, unsigned char *buf, int cnt,
                     int wait )
```

Description:

The function receives a block of bytes from the specified socket and stores the received data into `buf`. In case of timeout, the function returns the number of bytes received so far, which is less than `cnt`.

The sender of the data should be a "Send binary" tool. The function does not implement any handshake or I/O protocol.

Parameters:

`sock`	Input socket
`buf`	Output data buffer
`cnt`	Input number of bytes to receive (size of `buf`)
`wait`	Input wait time in ms: `0` : wait forever `>0` : wait time in milliseconds

Return code:

`>=0`	Number of received bytes (`RC < cnt` : timeout error)
`TCP_RECV_ERROR`	Receive error
`TCP_NO_CONNECTION_ERROR`	No connection error

7. Error codes

The basic TCP functions return the following error codes, defined by macros in TCP_LIB.H:

Code	Macro	Description
0	TCP_OK	Success
-2101	TCP_STREAM_BIND_ERROR	Stream bind error
-2104	TCP_SOCKET_CREATE_ERROR	Socket create error
-2105	TCP_LISTEN_FAIL_ERROR	Listen fail error
-2106	TCP_ACCEPT_FAIL_ERROR	Accept error
-2107	TCP_NO_CONNECTION_ERROR	No connection error
-2108	TCP_INVALID_IP_ERROR	Invalid IP address
-2109	TCP_CONNECT_FAIL_ERROR	Connect failed
-2110	TCP_NO_DATA_SOCKETS_ERROR	No free data sockets
-2111	TCP_SEND_BYTE_ERROR	Send byte error
-2112	TCP_RECV_BYTE_ERROR	Receive byte error
-2113	TCP_SEND_ERROR	Send error
-2114	TCP_RECV_ERROR	Receive error
-2115	TCP_SEND_BLOCK_ERROR	Send block error
-2116	TCP_RECV_BLOCK_ERROR	Receive block error
-2117	TCP_GETSOCKNAME_ERROR	getsockname() error
-2118	TCP_GETPEERNAME_ERROR	getpeername() error
-2130	TCP_COPY_PROT_ERROR	Copy protection error

The VIMOS TCP functions return the following error codes, defined by macros in TCP_IO.H:

Code	Macro	Description
2200	TCP_NO_MEMORY	Memory allocation error
2210	TCP_SEND_IMAGE_HDR_ERROR	Send image - header error
2211	TCP_SEND_IMAGE_DATA_ERROR	Send image - data error
2212	TCP_SEND_IMAGE_REPLY_ERROR	Send image - reply error
2213	TCP_SEND_IMAGE_TIMEOUT	Send image - wait timeout
2214	TCP_SEND_PTLIST_HDR_ERROR	Send point-list - header error
2215	TCP_SEND_PTLIST_DATA_ERROR	Send point-list - data error
2216	TCP_SEND_PTLIST_REPLY_ERROR	Send point-list - reply error
2217	TCP_SEND_PTLIST_TIMEOUT	Send point-list - wait timeout
2218	TCP_SEND_RESULT_HDR_ERROR	Send result - header error
2219	TCP_SEND_RESULT_DATA_ERROR	Send result - data error

2220	TCP_SEND_RESULT_REPLY_ERROR	Send result - reply error
2221	TCP_SEND_RESULT_TIMEOUT	Send result - wait timeout
2222	TCP_SEND_STRING_HDR_ERROR	Send string - header error
2223	TCP_SEND_STRING_DATA_ERROR	Send string - data error
2224	TCP_SEND_STRING_REPLY_ERROR	Send string - reply error
2225	TCP_SEND_STRING_TIMEOUT	Send string - wait timeout
2310	TCP_RECV_IMAGE_NO_DATA	Receive image - no data
2311	TCP_RECV_IMAGE_BUF_OVF	Receive image - buffer overflow
2312	TCP_RECV_IMAGE_TIMEOUT	Receive image - timeout
2313	TCP_RECV_PTLIST_NO_DATA	Receive point-list - no data
2314	TCP_RECV_PTLIST_BUF_OVF	Receive point-list - buffer overflow
2315	TCP_RECV_PTLIST_TIMEOUT	Receive point-list - timeout
2316	TCP_RECV_RESULT_NO_DATA	Receive result - no data
2317	TCP_RECV_RESULT_BUF_OVF	Receive result - buffer overflow
2318	TCP_RECV_RESULT_TIMEOUT	Receive result - timeout
2319	TCP_RECV_STRING_NO_DATA	Receive string - no data
2320	TCP_RECV_STRING_BUF_OVF	Receive string - buffer overflow
2321	TCP_RECV_STRING_TIMEOUT	Receive string - timeout