



Device Driver Manual

TCP/UDP IP Driver

Access to Hilscher Devices via TCP/IP and UDP/IP

Hilscher Gesellschaft für Systemautomation mbH
Rheinstraße 15
D-65795 Hattersheim
Germany

Tel. +49 (6190) 9907 - 0
Fax. +49 (6190) 9907 - 50

Sales: +49 (6190) 9907 - 0
Hotline and Support: +49 (6190) 9907 - 99

Sales Email: sales@hilscher.com
Hotline and Support Email: hotline@hilscher.com

Web: <http://www.hilscher.com>

Index	Date	Version	Chapter	Revision
1	19.03.02	1.000	all	Drawn up

Although this program has been developed with great care and intensively tested, Hilscher Gesellschaft für Systemautomation mbH cannot guarantee the suitability of this program for any purpose not confirmed by us in writing.

Guarantee claims shall be limited to the right to require rectification. Liability for any damages which may have arisen from the use of this program or its documentation shall be limited to cases of intent.

We reserve the right to modify our products and their specifications at any time in as far as this contributes to technical progress. The version of the manual supplied with the program applies.

1	INTRODUCTION.....	4
1.1	Terms for this Manual	4
1.2	Overview	5
1.3	Message Structure.....	6
2	USING SYSTEM SPECIFIC TCP/UDP IP API.....	7
2.1	Parameters	7
2.2	Open Connection Endpoints	7
2.3	Establishing of a Connection (Device = Server).....	7
2.4	Waiting for Incoming Connection (Device = Client).....	7
2.5	Sending and Receiving Data	8
2.5.1	Examples	9
2.6	Avoiding TCP Send Delay.....	12
2.6.1	Acknowledge Message Format	12
3	USING TCP/UDP IP DRIVER	13
3.1	General	13
3.2	Operating systems	13
3.3	Function Overview.....	13
3.4	Contents for Windows 9x, Windows NT and Windows 2000	14
3.5	Installation of the Device Driver	15
3.5.1	Standard Registry Entries Windows 9x, Windows NT and Windows 2000.....	16
3.5.2	Driver File Installation	18
3.5.3	Driver Utilities.....	18
3.6	Configure the Windows 9x/2000/NT Driver	19
3.7	Programming Instructions.....	20
3.7.1	Include the Interface API in your Application.....	20
3.7.2	The Application Programming Interface	20
3.7.3	Hints.....	21
4	ERROR CODES.....	22
4.1	List of Error Numbers.....	22
4.1.1	Error Codes	22
4.1.2	Hints.....	23

1 Introduction

This manual describes the way of accessing Hilscher devices with IP interfaces via TCP/IP or UDP/IP and the application programming interface (API) to our devices. In the following DEVICE stands for communication interface, the communication module, NetNode, NetLink or any other device from Hilscher with IP interface.

The general mechanism of data transfer is protocol independent and for each hardware the same procedure and is described therefore in the Toolkit Manual 'General Definitions'

All parameter and data have basically the description LSB/MSB. This corresponds to the convention of the Microsoft C compiler. The storage format of word oriented send and receive process data of the handled I/O DEVICES is configurable.

Values with a following 'h' are in hexadecimal notation such as 1Eh = 30. Values without any following letter are in decimal notation.

Supplementary information is contained in the following Manuals:

- Toolkit Manual 'General Definitions' (Tk:TKIT),
- Protocol Interface Manuals of used protocols
- Device Driver Manual 'Device Driver' (Dd:DevDrv).

1.1 Terms for this Manual

DPM	D ual- P ort M emory this is the physical interface to all communication board (DPM is also used for PROFIBUS- DP Master).
CIF	C ommunication I nter F ace
COM	C ommunication M odule
HOST	Application on the PC or a similar device
DEVICE	Synonym for communication interfaces or communication modules
RCS	R ealtime C ommunicating S ystem, this is the name of the operating system that runs on the communication boards
DLL	D ynamic L ink L ibrary

1.2 Overview

There are two ways of accessing Hilscher devices via IP protocol.

- Using the IP Driver
- Using the system specific IP stack Application Programming Interface (like Windows or Berkley sockets) on the system directly

In both cases communication is made by sending and receiving messages over the TCP/IP or UDP/IP protocol. The format and the meaning of these messages are described in the Toolkit Manual 'General Definitions' (Tk:TKIT) and in Protocol Interface manuals of the given protocol on the device. For Example: Using a NetLink with PROFIBUS Master interface, the description is made in Protocol Interface manual of PROFIBUS-DP Master (Pi:DPM).

1.3 Message Structure

A message consists of an 8 byte message header, and optional 8 byte telegram header and up to 247 bytes of user data.

- **Message Header** Used by the DEVICE operating system for transporting and address the message. This structure is fixed and constant.
- **Telegram Header** Defines the action for the protocol task.
- **User data** Send/received data.

Parameter	Type	Meaning	
Msg.Rx	byte	Receiving Task	Message Header
Msg.Tx	byte	Sending Task	
Msg.Ln	byte	Data length	
Msg.Nr	byte	Identification Code	
Msg.A	byte	Response Code	
Msg.F	byte	Error Code	
Msg.B	byte	Command Code	
Msg.E	byte	Extension Code	
Msg.DeviceAdr	byte	Communication Reference	Telegram Header
Msg.DataArea	byte	Data Block	
Msg.DataAdr	word	Object Index	
Msg.DataIdx	byte	Object Subindex	
Msg.DataCnt	byte	Data Quantity	
Msg.DataType	byte	Data Type	
Msg.Fnc	byte	Service	
Msg.D[0-246]	byte ... byte	User Data	Telegram User Data

General structure of message

This is an example for a PROFIBUS-FMS command message. For other protocols the structure is the same, but the containing parameters must be changed when Modbus Plus is used for example, from communication reference to slave address, object index to register address, or service to function code.

2 Using System specific TCP/UDP IP API

It is very simple to access a device from own applications via TCP/IP or UDP/IP. The application just has to establish a IP connection and to send specified messages.

If the device works as TCP/IP or UDP/IP server, an application has to work as TCP/IP or UDP/IP client and has to open the connection. If the device works as TCP/IP or UDP/IP client, the device will establish the connection and the application has to wait for incoming connections.

The following section will describe the way of communication with the well known BSD socket or Winsock model. Most IP Application Programming Interfaces (API) are based on these model.

2.1 Parameters

Standard IP communication with Hilscher devices is handled over **TCP/IP Port 1099**, device is server. Other communication modes have to be configured on the devices, if possible.

Standard transmitter number (MESSAGE.tx) in messages is 255 (0xff).

2.2 Open Connection Endpoints

Open a socket with a the function call:

```
SOCKET socket( int af, int type, int protocol);
```

On Windows systems, call WSStartup() before opening a socket.

Bind the successfully opened socket to local IP address with function:

```
int bind( SOCKET s, const struct sockaddr FAR *name, int namelen);
```

2.3 Establishing of a Connection (Device = Server)

Connect the bound socket to the server with function call:

```
int connect( SOCKET s, const struct sockaddr FAR *name, int namelen);
```

2.4 Waiting for Incoming Connection (Device = Client)

Set socket in listen state

```
int listen( SOCKET s, int backlog);
```

Wait for incoming connection on socket in listen state:

```
SOCKET accept( SOCKET s, struct sockaddr FAR *addr, int FAR *addrlen);
```

2.5 Sending and Receiving Data

After a successful connect you can send and receive messages with the function calls:

```
int send( SOCKET s, const char FAR *buf, int len, int flags);
```

```
int recv( SOCKET s, char FAR *buf, int len, int flags);
```

Please see your system specific development documentation for further details.

2.5.1 Examples

2.5.1.1 Application in client mode

The following shows a simple example to establish a TCP/IP connection (device = server) and sending and receiving a message with Windows sockets.

```
.....
#include "winsock.h"
#define FIXED_PORT      1099
typedef struct MESSAGEGRAMtag {
    unsigned char rx;          /* receiver          */
    unsigned char tx;          /* transmitter      */
    unsigned char ln;          /* length          */
    unsigned char nr;          /* number          */
    unsigned char a;           /* answer          */
    unsigned char f;           /* fault           */
    unsigned char b;           /* command         */
    unsigned char e;           /* extension       */
    unsigned char device_adr;   /* device address  */
    unsigned char data_area;    /* data area       */
    unsigned short data_adr;     /* data address    */
    unsigned char data_idx;     /* data index      */
    unsigned char data_cnt;     /* data count      */
    unsigned char data_type;    /* data type       */
    unsigned char function;     /* function        */
    unsigned char d[ 247 ];
} MESSAGEGRAM;

.....
int      err;
WORD     wVersionRequired;
WSADATA  wsaData;
SOCKET   soc;
wVersionRequired = MAKEWORD(1,1);
// initialize WinSock library
err = WSASStartup(wVersionRequired, &wsaData);
if (err != 0)
    exit(1);

// create a TCP/IP socket
soc = socket ( AF_INET, SOCK_STREAM, 0);
if (soc != INVALID_SOCKET)
{
    struct sockaddr_in LocalAddr;
    LocalAddr.sin_family = AF_INET;
    LocalAddr.sin_addr.s_addr = htonl( INADDR_ANY );
    LocalAddr.sin_port = 0;
    // bind the socket to local address
    if( bind( soc, (struct sockaddr *)&LocalAddr, sizeof(LocalAddr)) != SOCKET_ERROR )
    {
        struct sockaddr_in RemoteAddr;
        char szAddress[] = {"192.168.10.161"};
        unsigned long IpAddress = inet_addr( szAddress );
        RemoteAddr.sin_family = AF_INET;
```

```

RemoteAddr.sin_addr.s_addr = IpAddress;
RemoteAddr.sin_port = htons(FIXED_PORT);

//connect to server with IP address 192.168.10.161 on port 1099
if( connect( soc, (struct sockaddr *)&RemoteAddr, sizeof(RemoteAddr)) == 0 )
{
    MESSAGETELEGRAM SendMessage;
    MESSAGETELEGRAM ReceiveMessage;
    int SendLen, ReceiveLen;
    // build message to read data block with MPI protocol, see Protocol Interface
    // manual of PROFIBUS-DP Master (Pi:DPM)
    SendMessage.rx = 3;
    SendMessage.tx = 255;
    SendMessage.ln = 8;
    SendMessage.nr = 0;
    SendMessage.a = 0;
    SendMessage.f = 0;
    SendMessage.b = 0x31;
    SendMessage.e = 0;
    SendMessage.device_adr = 0;
    SendMessage.data_area = 0;
    SendMessage.data_adr = 0;
    SendMessage.data_idx = 0;
    SendMessage.data_cnt = 1;
    SendMessage.data_type = 5;
    SendMessage.function = 1;
    // Send data over TCP/IP connection to device
    SendLen = send(soc, (char*)&SendMessage, sizeof(SendMessage), 0);
    if( SendLen == sizeof(SendMessage))
    {
        // receive answer message
        ReceiveLen = recv( soc, (char*)&ReceiveMessage, sizeof(ReceiveMessage), 0);
        // .. do something with answer
    }
}
}
// close socket
closesocket ( soc );
// Cleanup and return
WSACleanup();
.....

```

2.5.1.2 Application in server mode

```

.....
#include "winsock.h"
.....
int      err;
WORD     wVersionRequired;
WSADATA  wsaData;
SOCKET   soc, AcceptedSocket;
wVersionRequired = MAKEWORD(1,1);
// initialize WinSock library
err = WSASStartup(wVersionRequired, &wsaData);
if (err != 0)
    exit(1);

// create a TCP/IP socket
soc = socket ( AF_INET, SOCK_STREAM, 0);
if (soc != INVALID_SOCKET)
{
    struct sockaddr_in LocalAddr;
    LocalAddr.sin_family = AF_INET;
    LocalAddr.sin_addr.s_addr = htonl( INADDR_ANY );
    LocalAddr.sin_port = htons(FIXED_PORT);
    // bind the socket to local address
    if( bind( soc, (struct sockaddr *)&LocalAddr, sizeof(LocalAddr)) != SOCKET_ERROR )
    {
        char szAddress[MAX_PATH];
        int AddressLen = sizeof( szAddress );
        // set socket in listen state
        if( listen( soc, SOMAXCONN ) == NO_ERROR )
        {
            //accept incoming connection
            if( (AcceptedSocket = accept( soc, (struct sockaddr FAR*)&szAddress,
                                         &AddressLen)) != INVALID_SOCKET )
            {
                MESSAGETELEGRAM SendMessage;
                MESSAGETELEGRAM ReceiveMessage;
                int SendLen, ReceiveLen;

                ReceiveLen = recv(AcceptedSocket, (char*)&ReceiveMessage,
                                sizeof(ReceiveMessage), 0);

                .....
                // see example below
            }
        }
    }
}

```

Note:

The example shows the use of sockets in blocking mode. Every function call on a socket is blocking until the command was successfully done or if any error occurred. You can use the `ioctlsocket()` or the `select()` function to handle sockets in non-blocking mode. Another way to handle blocking sockets in a Windows environment is to use threads for sending and receiving data.

2.6 Avoiding TCP Send Delay

On some circumstances it is possible that TCP/IP messages send from Hilscher devices will block on the device until an answer from the TCP/IP stack of the host application is available, also if no answer from application is required. This is, because by default Windows TCP/IP stack is waiting about 200ms until a TCP/IP acknowledge is sent to the communication partner. The device is not able to send a new TCP/IP message until this acknowledge, so new messages will block on the device. To avoid this, the application can send an own acknowledge. This is done by sending an defined message (see below) over TCP/IP to the device. This message causes automatically an TCP/IP acknowledge and the device is able to send new messages to the application.

Example: A Hilscher device, like a NetNode or NetLink, has to send TCP/IP messages with user data to the remote application in high speed. If no answer from the application is required, the device can send max. 5 messages per second, because Windows TCP/IP stack is waiting about 200ms until an acknowledge is sent to the device. The device is not able to send new data until this acknowledge is received. If the application sends the defined acknowledge message right after receiving the user data, the device is able to send new user data right after this.

The defined acknowledge message has no further effects on the device.

2.6.1 Acknowledge Message Format

Command Message			
Parameter	Type	Value	Description
msg.rx	USIGN8	0	<u>Identification of Receiver</u> Operating system
msg.tx	USIGN8	255	<u>Identification of Transmitter</u> User application
msg.ln	USIGN8	0	<u>Message Length</u> Length
msg.nr	USIGN8	0 .. 255	<u>Message Identification</u> Unique number
msg.a	USIGN8	255	<u>Reply Identification</u> Acknowledge Reply
msg.f	USIGN8	0	<u>Error Number</u> No error
Msg.b	USIGN8	0	<u>Command Identification</u> no command
Msg.e	USIGN8	0	<u>Extension</u> No answer message

Message format of acknowledge message

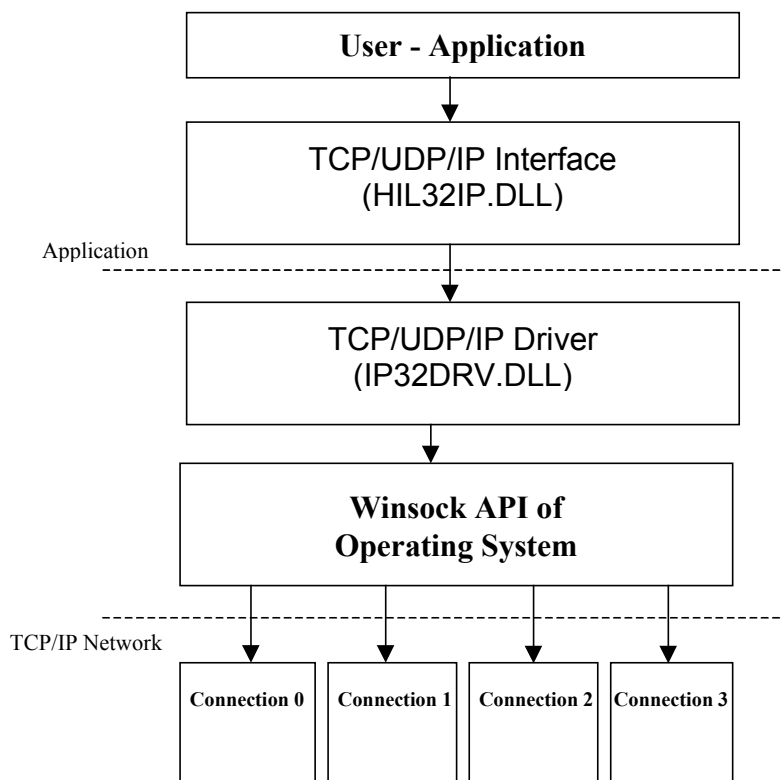
3 Using TCP/UDP IP Driver

3.1 General

The API of the TCP/UDP IP Driver is the same like the API of Device Driver for our CIF cards and COM modules. The driver provides the same functionality like Device Driver, please see Device Driver manual (Dd:DevDrv) for further information. If someone has already an application running with the Device Driver it is very easy to use TCP/UDP/IP Driver instead.

3.2 Operating systems

For **Windows 9x**, **Windows NT** and **Windows 2000** we are using IP driver. The communication between the application and the driver is done by a DLL. This DLL can be statically or dynamically linked to the application.



TCP/UDP IP Driver components

3.3 Function Overview

The IP drivers for Windows 9x, Windows NT and Windows 2000 can handle up to four connections.

On each connection only one command can be active at the same time, because there is no command queuing in the driver implemented.

3.4 Contents for Windows 9x, Windows NT and Windows 2000

Directory	Subdirectory	Description
<INSTALL>	API	Application Programming Interface, libraries and header files to access the 32 Bit driver DLL (the DLL is installed by the driver installation)
	Demo	Simple Message and IO data transfer source code example (IODemo.cpp)
		IpDrvTest: Complete TCP/UDP/IP driver test program written in C++, created with Microsoft Visual C/C++ 6.0
	MANUALS	TCP/UDP/IP driver manual

CD content

Windows 9x, Windows NT and Windows 2000 driver files:

HILIP32.DLL	Dynamic link library of the driver interface, created for use with Windows 9x, Windows NT and Windows 2000
HILIP32.LIB	Definition file with the exported function of the HILIP32.DLL.
IPDRVUSR.H	Definition header file for the user interface.
IP32DRV.DLL	TCP/UDP IP driver DLL

Applications:

IpDrvSetup.EXE	Driver Setup program for registry entries
IpDrvTest.EXE	Driver Test program to run the various device driver functions

Development platform:

Windows 9x	Microsoft Visual C++, V 6.x
Windows NT 4.0	Microsoft Visual C++, V 6.x
Windows 2000	Microsoft Visual C++, V 6.x

ATTENTION:

The TCP/UDP IP Interface DLL and the driver files are installed during the driver installation and not included in the development directories.

3.5 Installation of the Device Driver

The driver will be installed by an installation program. This will guide you to the installation process. The installation program will run the following steps:

- Creating the standard registry entries for the TCP/UDP IP Driver
- Copying the device driver / interface DLL files
- Copying the device driver setup and test program

3.5.1 Standard Registry Entries Windows 9x, Windows NT and Windows 2000

Registry Path:

\HKEY_LOCAL_MACHINE\Software\Hilscher GmbH\

TCP/UDP IP Driver Entry:

IP Driver	- Company	// Hilscher GmbH
	- CurrentFolder	// Installation folder of driver
	- CurrentVersion	// Current version of driver
	- Directory	// Installation directory of driver
	- Name	// Name of driver
	\Connection0	
	\Connection1	
	\Connection2	
	\Connection3	

The default entries are

Connection0	- IpAddress	0x00000000	// TCP/IP Address of connection 0
	- PortNumber	0x0000044B	// Port Number of TC/IP connection
	- Mode	Client	// Client Mode
	- Protocol	'TCP'	// TCP protocol is used
	- ConnectTimeout	0x00002710	// Timeout for connection in msec.
Connection1	- IpAddress	0x00000000	// TCP/IP Address of connection 1
	- PortNumber	0x0000044B	// Port Number of TC/IP connection
	- ClientMode	TRUE	// Client Mode = TRUE
	- Protocol	'TCP'	// TCP protocol is used
	- ConnectTimeout	0x00002710	// Timeout for connection in msec.
Connection2	- IpAddress	0x00000000	// TCP/IP Address of connection 2
	- PortNumber	0x0000044B	// Port Number of TC/IP connection
	- ClientMode	TRUE	// Client Mode = TRUE
	- Protocol	'TCP'	// TCP protocol is used
	- ConnectTimeout	0x00002710	// Timeout for connection in msec.
Connection3	- IpAddress	0x00000000	// TCP/IP Address of connection 3
	- PortNumber	0x0000044B	// Port Number of TC/IP connection
	- ClientMode	TRUE	// Client Mode = TRUE
	- Protocol	'TCP'	// TCP protocol is used
	- ConnectTimeout	0x00002710	// Timeout for connection in msec.

3.5.2 Driver File Installation

TCP/UDP IP Interface DLLs:

Windows 9x	The interface DLL HILIP32.DLL is copied to the %System Root%\System directory.
Windows NT	The interface DLL HILIP32.DLL is copied to the %System Root%\System32 directory.
Windows 2000	The interface DLL HILIP32.DLL is copied to the %System Root%\System32 directory.

TCP/UDP IP Driver DLLs:

Windows 9x	The driver DLL IP32DRV.DLL is copied to the %System Root%\System directory.
Windows NT	The driver DLL IP32DRV.DLL is copied to the %System Root%\System32 directory.
Windows 2000	The driver DLL IP32DRV.DLL is copied to the %System Root%\System32 directory.

Device Driver Utilities:

Installation path	<System>\Program Files\HILSCHER GmbH\IP Driver
IpDrvSetup	Driver setup programm
IpDrvTest	Driver test programm

3.5.3 Driver Utilities

The driver includes a driver setup (IPDRVSETUP.EXE) and a driver test (IPDRVTEST.EXE) program. These files are also installed during the installation procedure. Therefore, the installation program creates a HILSCHER GmbH\ IP Driver directory below the standard PROGRAM directory where the files are copied.

3.6 Configure the Windows 9x/2000/NT Driver

The user must configure the IP address, the port number, the kind of connection and the used protocol of each connection. All these informations are written to the registry data base of the operating system.

To get an easy access to this data the device driver gets its own setup program IPDRVSETUP.EXE. This program will help you to change the registry entries without using REGEDIT.EXE.

Parameter	Description
IP address	IP address of device to which the connection should be established, maybe 0 if application works in server mode
Port Number	Port Number on which the IP connection should be established
Client / Server Mode	Determines if application should work as client or server
Protocol	Determines which protocol (TCP or UDP) is used
Connect Timeout	Determines how long the driver should try to establish a connection on DevInitBoard() function call

Driver parameters

3.7 Programming Instructions

3.7.1 Include the Interface API in your Application

For the user API there is only one include file IPDRVUSR.H which contains all the necessary information like structure, constant and prototype definitions. A complete function description is given in the Device Driver manual (Dd:DevDrv). Link the device API-DLL (HILIP32.LIB) to your program. Make sure you have installed the driver if this one is used.

Further programming instructions can be found in the Device Driver Manual (Dd:DevDrv), since the API is same.

3.7.2 The Application Programming Interface

Since the TCP/UDP/IP driver has the same API as the Device Driver, please refer to Device Driver manual for further information.

3.7.2.1 Differences

- The meaning of the parameter usDevNumber is changed from board number to connection number
- DevGetBoardInfo delivers IRQ number always as 0, Physical Address is changed to IP address

```
typedef struct tagBOARD_INFO{
    unsigned char abDriverVersion[16]; // DRV version information
    struct {
        unsigned short usBoardNumber; // DRV connection number
        unsigned short usAvailable; // DRV board is available
        unsigned long ulIpAddress; // DRV IP address
        unsigned short usIrqNumber; // DRV irq number, always 0
    } tBoard [MAX_DEV_BOARDS];
} BOARD_INFO;
```

- DevGetBoardInfo may need a long time to return, if an configured device is not available (TCP/IP Timeout)
- DevGetBoardInfo returns always usBoardAvailable = TRUE on UDP connections
- Function DevInitBoard tries to establish an IP connection to the configured device, no further testing actions are done
- DevGetMBXState delivers the state, if sending or receiving data over the configured IP connection is possible.
- DevGetInfo(), Info Area GET_DRIVER_INFO: Parameters IRQCnt, bHostFlags, bMyDevFlags, bExIOFlag are not supported, set to zero
- DevGetInfo(); Info Area GET_RCS_INFO: Parameters: bRcsError, bHostWatchdogState, bDevWatchdogState, bSegmentCount, bDeviceAddress, bDriverType are not supported, set to zero
- DevGetInfo(); Info Area GET_DEV_INFO: not supported, set to zero
- DevGetInfo(); Info Area GET_IO_INFO: not supported, set to zero

-
- DevExchangeIOEx(); not supported
 - DevReadSendData(); not supported
 - DevReadWriteDPMRaw(); not supported
 - DevExtendedData(); not supported
 - DevGetMbxData(); not supported
 - DevSpecialControl(); not supported
 - DevDownload(); not supported
 - DevReadWriteDPMDData(); not supported

3.7.3 Hints

Timeouts with TCP/UDP IP driver may be much longer than under device driver.

4 Error Codes

4.1 List of Error Numbers

Error numbers are compatible with error numbers from device driver. Please see device driver manual for further details.

Some new error codes are defined for TCP/UDP IP use.

4.1.1 Error Codes

Error Number	Description
-1000	Function not implemented
-1001	Error by getting resources from Host PC
-1100	Internal NULL pointer exception
-1101	Error by accessing registry
-1102	Error reading key in registry
-1103	Unknown mode (not Client or Server)
-1104	Unknown protocol (not TCP or UDP)
-1105	Connection number is invalid
-1200	Wrong answer for sent command received
10000	Windows socket error number start

Error Codes

4.1.2 Hints

Error numbers 10000 and higher are Windows socket system specific error numbers. You will find detailed information in the documentation of your operating system.

Important Windows socket error numbers:

Error Number	Description
10036	Operation now in progress. A blocking operation is currently executing. Windows Sockets only allows a single blocking operation—per- task or thread—to be outstanding
10051	Network is unreachable. A socket operation was attempted to an unreachable network. This usually means the local software knows no route to reach the remote host
10054	Connection reset by peer. An existing connection was forcibly closed by the remote host.
10057	Socket is not connected. A request to send or receive data was disallowed because the socket is not connected
10058	Cannot send after socket shutdown. A request to send or receive data was disallowed because the socket had already been shut down
10060	Connection timed out. A connection attempt failed because the connected party did not properly respond after a period of time, or the established connection failed because the connected host has failed to respond
10061	Connection refused. No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the foreign host—that is, one with no server application running.
10091	Network subsystem is unavailable. This error is returned by WSASStartup if the Windows Sockets implementation cannot function at this time because the underlying system it uses to provide network services is currently unavailable
10092	Winsock.dll version out of range. The current Windows Sockets implementation does not support the Windows Sockets specification version requested by the application.