Ethernet/Web Adapter

Programming Guide

Version 1.04

1. Introduction

1.1 Revisions

- 1.01 - the first version of this manual.
- 1.02:
  - The description of the newly implemented protocols is added. See chapters: "UDP Sender/Reciever Interaction", "UDP Sender/Reciever Interaction using TBSockets", "Send and Receive Data (UDP)", "SNTP Time Service".
- 1.04:
  - The description of the newly implemented subroutines is added. See chapters: "System Reset", "Get Ethernet Link State".
  - The newly added variable *wActAcceptLocalPort* can be analyzed in the task OnAccept to know at which local port the connection was actually accepted.

1.2 About this document

- This document refers to the **Ethernet/Web Adapter** types **EM01-ETH-S**, **EM02-WEB-S, EM03-ETH-P**, **EM04-WEB-P**.
- The information in this document is relevant for the **Ethernet/Web Adapters version V1.3 or higher** (see imprint on the front side of the Ethernet/Web modules) and for the **Tiger Basic Sockets (TBSockets) implementation version 1.01** (see the bGetAdapterProgVers subroutine in the "Get Version of the Adapter Software" paragraph of this manual).
- If a function is available only for special types or versions of hardware or software, the appropriate notice can be read in the paragraph describing this function.
- In this document if the notation Ethernet/Web Adapter is used, then the corresponding text refers to all mentioned module types; the notation Ethernet Adapter is used for EM01-ETH-S and EM03-ETH-P modules only; the term Web Adapter describes only EM02-WEB-S and EM04-WEB-P modules.
- This document consists of two main chapters. The "**Demo Programs**" chapter introduces the client/server programming with Ethernet/Web Adapter and Tiger Basic and describes in details the sample programs:
  - *Client_Simple_Ethe.Tig*, *Client_Simple_Ppp.Tig* - implement simple tcp client that actively opens connection and communicates with an echo server.
  - *Server_Ethe.Tig* - implements simple tcp server that waits passively for someone to contact the Ethernet Adapter and sends in loop simple messages to the remote peer.
  - *Client_Dhcp_Ethe.Tig* - demonstrates how to get dynamic IP address (subnet mask and gateway) from DHCP server and starts a simple tcp client that actively opens connection and communicates with an echo server.
  - *Client_Dns_Ethe.Tig* - demonstrates how to get an IP address corresponding to a host name from DNS server and starts a simple tcp client that actively opens connection using the obtained IP address and communicates with an echo server.
  - *Smtp_Client_Ethe.Tig*, *Smtp_Client_Ppp.Tig* - demonstrate how to send an email using SMTP (RFC 821, RFC 1651) protocol. The protocol is implemented in Tiger Basic language, it is delivered as source code and can be changed by user to comply with the requirements of the particular SMTP server.
  - *Client_Simple_Ethe_Udp.Tig* - implements simple udp client (sender) that actively opens connection and sends a message to an udp server.

- *Server_Ethe_Udp.Tig* - implements simple udp server (receiver) that receives a number of bytes from a remote peer through the particular port.
- *Sntp_Get_Time.Tig* - demonstrates how to request the time from a Network Time Server using SNTP protocol.

The “**Programming with Tiger Basic Sockets (TBSockets)**” chapter explains how to use particular Tiger Basic subroutines for implementing the client/server applications by means of Ethernet/Web Adapter.

- To understand how to use the Ethernet/Web Adapter without Tiger Basic controller, read the “Ethernet_Adapter__Protocol_[Vers].pdf” document additionally.

1.3 Protocols and Services Provided

1.3.1 Ethernet Adapter

- Address Resolution Protocol (**ARP**; RFC 0826, RFC 1122)
- Internet Protocol (**IP**; RFC 0791)
- Internet Control Message Protocol (**ICMP**; RFC 0792)
- Domain Name Services (**DNS**; RFC 1034, RFC 1035) Client
- Transmission Control Protocol (**TCP**; RFC 0793, RFC 1122)
- User Datagram Protocol (**UDP**; RFC 768)
- Dynamic Host Configuration Protocol (**DHCP**; RFC 2131) Client
- Simple Network Time Protocol (**SNTP**; RFC 2030)

1.3.2 Web Adapter

- Point-to-Point Protocol (**PPP, LCP, IPCP, AHDLC**; RFC 1172, 1570, 1332)
- Internet Protocol (**IP**; RFC 0791)
- Internet Control Message Protocol (**ICMP**; RFC 0792)
- Domain Name Services (**DNS**; RFC 1034, RFC 1035) Client
- Transmission Control Protocol (**TCP**; RFC 0793)
- User Datagram Protocol (**UDP**; RFC 768)
- Dynamic Host Configuration Protocol (**DHCP**; RFC 2131) Client
- Simple Network Time Protocol (**SNTP**; RFC 2030)

1.3.3 Protocols implemented as software examples

- Simple Mail Transfer Protocol (**SMTP**, RFC 2821)
- Post Office Protocol - Version 3 (**POP3**, RFC 1939)

1.3 Software Terms and Requirements

- The Ethernet/Web Adapter is pre-programmed to be able to exchange the commands and data with Basic Tiger controller (or another controller) and to execute the commands. The user has no direct access to the program in the Ethernet/Web Adapter.
- The Basic Tiger controller communicates with the Ethernet/Web Adapter by using of a collection of tasks and subroutines named Tiger Basic Sockets (TBSockets). TBSockets are written in Tiger Basic programming language and delivered as source code. To compile TBSockets and appropriate demos the **Tiger Basic IDE version 5.01 or higher** is required.

- The include (.inc) files with names having the “ts_” prefix contain the implementation of TBSockets. All TBSockets include files must be copied to the place that is accessible for the Tiger Basic Compiler, by default the location of the include files is the “Include” directory of the Tiger Basic installation and this location is immediately visible for the Compiler.

2. Demo Programs

2.1 Terminology

IP (Internet Protocol): IP specifies the format of packets, also called *datagrams,* and the addressing scheme. Most networks combine IP with a higher-level protocol called TCP, which establishes a virtual connection between a destination and a source.
IP by itself is something like the postal system. It allows you to address a package and drop it in the system, but there's no direct link between you and the recipient. TCP/IP, on the other hand, establishes a connection between two hosts so that they can send messages back and forth for a period of time.

TCP (Transmission Control Protocol): TCP is one of the main protocols in TCP/IP networks. Whereas the IP protocol deals only with packets, TCP enables two hosts to establish a connection and exchange streams of data. TCP guarantees delivery of data and also guarantees that packets will be delivered in the same order in which they were sent.

UDP (User Datagram Protocol): a connectionless protocol that, like TCP, runs on top of IP networks. Unlike TCP/IP, UDP/IP provides very few error recovery services, offering instead a direct way to send and receive datagrams over an IP network. It's used primarily for broadcasting messages over a network.

IP address: An identifier for a computer or device on a TCP/IP network. Networks using the TCP/IP protocol route messages based on the IP address of the destination. The format of an IP address is a 32-bit numeric address written as four numbers separated by periods. Each number can be zero to 255. For example, 1.140.16.220 could be an IP address. Within an isolated network, you can assign IP addresses at random as long as each one is unique. However, connecting a private network to the Internet requires using registered IP addresses (called Internet addresses) to avoid duplicates.

Port: In TCP/IP and UDP networks, an endpoint to a logical connection. The port number identifies what type of port it is. For example, port 80 is used for HTTP traffic.

DNS (Domain Name System): An Internet service that translates domain names into IP addresses. Because domain names are alphabetic, they're easier to remember. The Internet however, is really based on IP addresses. Every time you use a domain name, therefore, a DNS service must translate the name into the corresponding IP address. For example, the domain name *www.example.com* might translate to *198.121.204.2*. The DNS system is, in fact, its own network. If one DNS server doesn't know how to translate a particular domain name, it asks another one, and so on, until the correct IP address is returned.

DHCP (Dynamic Host Configuration Protocol): A protocol for assigning dynamic IP addresses to devices on a network. With dynamic addressing, a device can have a different IP address every time it connects to the network. In some systems, the device's IP address can even change while it is still connected. DHCP also supports a mix of static and dynamic IP addresses. Dynamic addressing simplifies network administration because the software keeps track of IP addresses rather than requiring an administrator to manage the task. This means that a new computer can be added to a network without the hassle of manually assigning it a unique IP address. Many ISPs use dynamic IP addressing for dial-up users.

PAP (Password Authentication Protocol): The most basic form of authentication, in which a user's name and password are transmitted over a network and compared to a table of name-password pairs. Typically, the passwords stored in the table are encrypted. The main weakness of PAP is that both the username and password are transmitted "in the clear" -- that is, in an unencrypted form.

SNTP (Simple Network Time Protocol): a simplified version of NTP, an Internet standard protocol (built on top of TCP/IP) that assures accurate synchronization to the millisecond of computer clock times in a network of computers.

Client: An application that initiates the connection and relies on a server to perform some operations.

Server: An application that passively waits to respond to clients requests.

Socket: A software object that provides interface to a network protocol (i.e. TCP/IP or UDP). It is identified by protocol and local/remote address/port.

Tiger Basic Sockets (TBSockets): A collection of subroutines and tasks written in Tiger Basic programming language. It provides software interface between Basic Tiger and Ethernet/Web Adapter that actually implements the TCP/IP protocol. To use TBSockets an application must include the “ts_coinc.inc” file.

2.2 Network Configuration

- The Ethernet/Web Adapter is not pre-configured, so any free address on the net may be used either for the Ethernet/Web Adapter itself or for peer host (e.g. PC).
- If a crossover cable connected directly to peer host is used, static IP addresses and subnet masks are required.
- If a router is used in the net to which the PC (and the Ethernet/Web Adapter) will be attached, the default gateway should be set up additionally to the IP address and the subnet mask.
- For a DHCP client demo a DHCP server must be installed on the network (e.g. on the PC).
- Please ask your **network administrator** for valid specifications. Setting improper data may cause the applications to not work (correctly) or even disturb other network participants.

2.3 Ethernet/Web Adapter Settings

- The **EM01/EM02** Adapters communicate with the controlling device through the serial channel. Each Tiger Basic application intending to work with the EM01/EM02 Adapters must install the device driver for the particular **serial channel** with correct settings. The baud rate for the serial channel may be modified by using of the external selection mechanism (see “Datasheet_EM01_Eth_S_[Vers]” or “Datasheet_EM02_Ser_S_[Vers]” documents). The default setting for baud rate is **38400 Baud** for **EM01**, and **19200 Baud** for **EM02**. Other parameters are unchangeable: **8N1**. The Basic-Tiger controller uses the serial channel 0 (**SER0**, with hardware handshake) to communicate with the EM01/EM02 Adapters.

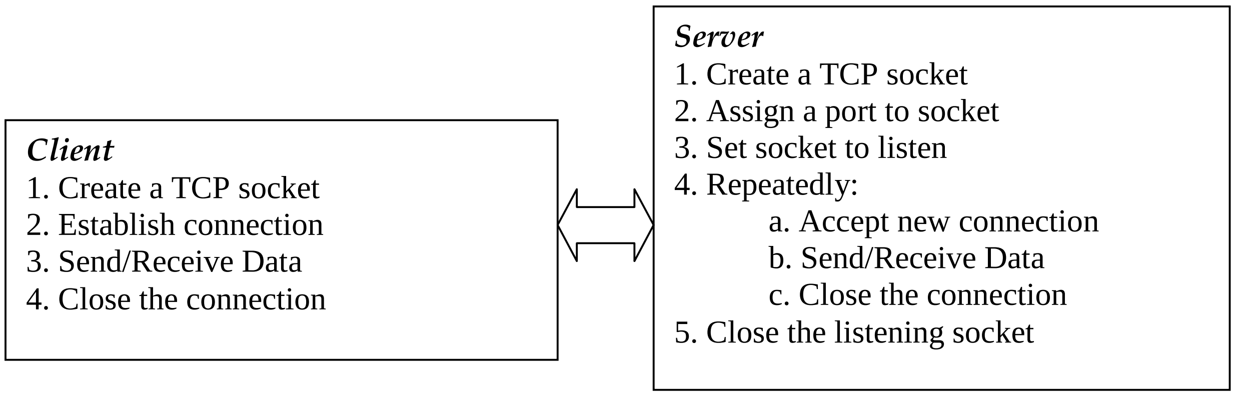
- Most of configuration constants useful for demo programs and applications can be found in the “ts_conf.inc” file. The file is subdivided into logical sections. E.g. all constants concerning DNS are put together and a comment line containing the word “DNS” marks the beginning of the section. The settings for the particular demo programs will be explained below, in the corresponding paragraphs.
- The “ts_conf.inc” file is included into the “ts_coinc.inc” file. So if an application includes the “ts_coinc.inc”, the “ts_conf.inc” is included automatically.
- To work with many copies of the “ts_conf.inc” file at the same time, please comment out the corresponding ‘#include’ line in the “ts_coinc.inc” file and include the proper copy of the “ts_conf.inc” into the particular application file (tig).
- The constants defined in the section “MODULE TYPE” select the software configuration for the particular module type. It is very important to choose the type constant properly. Only one constant (TS_EM_01 or TS_EM_02 or TS_EM_03 or TS_EM_04) must be activated at the moment.

2.4 Dialling Procedure for Web Adapters

Before the particular subroutines can be used to implement the client/socket functionality, the connection to an ISP (Internet Service Provider) should be established in the case of Web Adapter (EM02, EM04). Some of the demo programs below use the dialling procedure with the subsequent PAP authentication. The actual dialling is done with the *bDialIsp* subroutine call (or with the *bDialIspWithLogin* if the login and the password must be entered explicitly, not by using of PAP), but some settings must be performed before. The *bSetupPapSecret* subroutine sets the user name and the user password for PAP. The *bSetupIsp* subroutine defines the dialling number of the ISP (other parameters of this subroutine are used only if the *bDialIspWithLogin* mechanism must be activated).

2.5 Client/Server Modell

2.5.1 TCP Client/Server Interaction



2.5.2 TCP Client/Server Interaction using TBSockets

Client
0. vInitSockets, bSetupLocalIp
[bSetDefaultGateway]
1. lSocket
2. bConnect (sBuildSockAddrBlock)
3. lSend, OnData
4. bCloseSocket (OnRemoteClose)

Server
0. vInitSockets, bSetupLocalIp
[bSetDefaultGateway]
1. lSocket
2. bBind (sBuildSockAddrBlock)
3. bListen
4. Repeatedly:
    a. OnAccept
    b. lSend, OnData
    c. bCloseSocket (OnRemoteClose)
5. bCloseSocket

2.5.2.1 *How to build a very simple TCP Client using TBSockets*

- Initialize the internal variables (*vInitSockets*), set the IP address (*bSetupLocalIp*) and, possibly, the default gateway (*bSetDefaultGateway*) for the local host,
- Create a new socket (*lSocket*),
- Connect the remote peer (*bConnect*; one of the parameters of the subroutine is a timeout value defining how long to wait till success of connecting),
- Send the data (*lSend*),
- Receive the data serving a callback task (*OnData*) launched each time when the data are coming,
- Close the socket (*bCloseSocket*).

2.5.2.2 *How to build a very simple TCP Server using TBSockets*

- Initialize the internal variables (*vInitSockets*), set the IP address (*bSetupLocalIp*) and, possibly, the default gateway (*bSetDefaultGateway*) for the local host,
- Create a new socket (*lSocket*),
- Bind the socket to a particular port (*bBind*),
- Start listening to the bound port for incoming connections (*bListen*)
- Accept the connection from a remote peer serving a callback task (*OnAccept*) launched each time when a new socket for the accepted connection is created,
- Send the data using the new socket (*lSend*),
- Receive the data serving a callback task (*OnData*) launched each time when the data are coming,
- Close all open sockets (*bCloseSocket*).

2.5.3 UDP Sender/Reciever Interaction

UDP Sender
1. Create a UDP socket
2. Send Data
3. Close the connection

UDP Reciever
1. Create a UDP socket
2. Assign a port to socket
3. Send/Receive Data
4. Close the connection

2.5.4 UDP Sender/Reciever Interaction using TBSockets

UDP Sender
0. vInitSockets, bSetupLocalIp
[bSetDefaultGateway]
1. lSocket
2. lSendTo (sBuildSockAddrBlock)
3. bCloseSocket

⇔

UDP Reciever
0. vInitSockets, bSetupLocalIp
[bSetDefaultGateway]
1. lSocket
2. bBind (sBuildSockAddrBlock)
3. OnDataRecvFrom
4. bCloseSocket

2.5.4.1 How to build a very simple UDP Sender using TBSockets

- Initialize the internal variables (*vInitSockets*), set the IP address (*bSetupLocalIp*) and, possibly, the default gateway (*bSetDefaultGateway*) for the local host,
- Create a new socket (*lSocket*),
- Send the data (*lSendTo*),
- Close the socket (*bCloseSocket*).

2.5.4.2 How to build a very simple UDP Reciever using TBSockets

- Initialize the internal variables (*vInitSockets*), set the IP address (*bSetupLocalIp*) and, possibly, the default gateway (*bSetDefaultGateway*) for the local host,
- Create a new socket (*lSocket*),
- Bind the socket to a particular port (*bBind*),
- Receive the data serving a callback task (*OnDataRecvFrom*) launched when the data are coming,
- Close the socket (*bCloseSocket*).

All subroutines and tasks that may be of importance for implementing of client/server applications with Ethernet/Web Adapter are described in detail below in this "Programming Guide" manual.

2.6 Simple Client Demo

- Name :
  client_simple_ethe.tig (EM01, EM03), *client_simple_ppp.tig* (EM02, EM04)
- Include Files:
  1. *ts_coinc.inc* – includes all TBSockets files,
  2. *client_clbks.inc* – includes the user callback tasks for the demo; originally, these tasks are only placeholders for user-defined reactions on some asynchronous events like coming of data from remote peer etc.

- Purpose: This demo program is a simple client that actively opens connection and communicates with an **echo** server.

- Explanation:
  1. Dials the ISP (Web Adapters only) :

   - See “2.3 Dialling Procedure for Web Adapters”,
2. Creates a new socket (*lSocket*),
3. Establishes the connection to a remote host (*bConnect*),
4. If the remote host was connected the message loop is started; the message loop involves:
   - Sending (*lSend*) the TEXT_TO_SEND character sequence,
   - Waiting for an echo or for quit signal ("Q" or "QUIT"; *OnData* task in the "client_clbks.inc") or for closing of socket on the remote host (*OnRemoteClose* task in the "client_clbks.inc"); the timeout is not checked;
   - Repeating the loop either SEND_LOOPS times or quitting by quit or close-socket signal or running endless (timeout error),
5. Closes the socket (*bCloseSocket*) after leaving the message loop.

- Configuration Constants:
  All constants that must be correctly set by user to compile and run this demo are saved in the *ts_conf.inc* file in the following sections:
  - ➢ Module type in the “MODULE TYPE” section:
    - ❖ TS_EM_01 or TS_EM_02 or TS_EM_03 or TS_EM_04
  - ➢ Static IP address and subnet mask of the Ethernet Adapter in the “LOCAL HOST” section:
    - ❖ TS_LOCAL_IP_ADDRESS
    - ❖ TS_LOCAL_IP_SUBNET_MASK
  - ➢ IP address and port of the remote computer to which the connection must be established in the “CONNECTION TARGET” section:
    - ❖ TS_CONNECT_TO_PEER_IP
    - ❖ TS_CONNECT_TO_PEER_PORT
  - ➢ Default getaway address in the “GATEWAY” section (if a router is a part of the network)
    - ❖ TS_DEFAULT_GATEWAY

- How to test it:
  ✓ Use the “SServer.exe” program installed in the “..\Tools\SimpleServer” directory or another tcp server that is enabled to send an echo to the client.

2.7 Simple Server Demo

- Name :
  server_ethe.tig
- Include Files:
  1. *ts_coinc.inc* – includes all TBSockets files,
  2. *server_clbks.inc* – includes the user callback tasks for the demo; originally, these tasks are only placeholders for user-defined reactions on some asynchronous events like coming of data from remote peer etc.

- Purpose: This demo program is a simple server that waits passively for someone to contact the Ethernet Adapter and sends in loop simple messages to the remote peer.

- Explanation:
  1. Creates a new socket (*lSocket*),
  2. Binds the socket to a particular port (*bBind*),
  3. Listens to the bound port for incoming connections (*bListen*),
  4. Accepts the connection from a remote peer and stores a new socket for the accepted connection (*OnAccept* task in the "server_clbks.inc"),
  5. If a new connection was established the message loop is started; the message loop involves:
     - Sending (*lSend*) the TEXT_TO_SEND character sequence through the new socket,
     - Waiting for quit signal ("Q" or "QUIT"; *OnData* task in the "client_clbks.inc") or for closing of socket on the remote host (*OnRemoteClose* task in the "client_clbks.inc"); the timeout is not checked;
     - Repeating the loop either SEND_LOOPS times or quitting by quit or close-socket signal or running endless (timeout error),
  6. Closes all open sockets (*bCloseSocket*) after leaving the message loop.

- Configuration Constants:
  All constants that must be correctly set by user to compile and run this demo are saved in the *ts_conf.inc* file in the following section:
  - ➢ Module type in the “MODULE TYPE” section:
    - ❖ TS_EM_01 or TS_EM_02 or TS_EM_03 or TS_EM_04
  - ➢ Static IP address and subnet mask of the Ethernet Adapter in the “LOCAL HOST” section:
    - ❖ TS_LOCAL_IP_ADDRESS
    - ❖ TS_LOCAL_IP_SUBNET_MASK
  - ➢ Port and IP address to listen on (INADDR_ANY to accept every address) in the “SERVER SETTINGS” section:
    - ❖ TS_SERVER_LISTEN_TO_PORT
    - ❖ TS_SERVER_ACCEPT_IP

- How to test it:
  - ✓ Use **ping** program to check whether a device with the TS_LOCAL_IP_ADDRESS ip address is attached to the network.
    ping < TS_LOCAL_IP_ADDRESS >

  - ✓ Use **telnet** program to establish connection to the server, to receive messages from it and to force the server to quit the session by entering “q”.
    telnet < TS_LOCAL_IP_ADDRESS > < TS_SERVER_LISTEN_TO_PORT >

2.8 DHCP Client Demo

- Name :
  client_dhcp_ethe.tig
- Include Files:
  1. *ts_coinc.inc* – includes all TBSockets files,

2. *client_clbks.inc* – includes the user callback tasks for the demo; originally, these tasks are only placeholders for user-defined reactions on some asynchronous events like success of DHCP request, coming of data from remote peer etc.

- Purpose: This program demonstrates how to get dynamic IP address (subnet mask and gateway) from **DHCP** server and starts a simple client that actively opens connection and communicates with an **echo** server.

- Explanation:
  1. Enables DHCP request (*bSetupDhcp*),
  2. Creating of a new (first) socket forces (*lSocket)* the Ethernet Adapter to send request to the DHCP server and obtain dynamic IP address, subnet mask and gateway,
  3. Creates a new socket (*lSocket*),
  4. Waits for dynamic parameters sent by the DHCP server (*OnDhcpUp* task in the "client_clbks.inc") and checks the timeout value,
  5. Establishes the connection to a remote host (*bConnect*),
  6. If the remote host was connected the message loop is started; the message loop involves:
     - Sending (*lSend*) the TEXT_TO_SEND character sequence,
     - Waiting for an echo or for quit signal ("Q" or "QUIT"; *OnData* task in the "client_clbks.inc") or for closing of socket on the remote host (*OnRemoteClose* task in the "client_clbks.inc"); the timeout is not checked;
     - Repeating the loop either SEND_LOOPS times or quitting by quit or close-socket signal or running endless (timeout error),
  7. Closes the socket (*bCloseSocket*) after leaving the message loop.

- Configuration Constants:
  All constants that must be correctly set by user to compile and run this demo are saved in the *ts_conf.inc* file in the following sections:
  - Module type in the “MODULE TYPE” section:
    - TS_EM_01 or TS_EM_02 or TS_EM_03 or TS_EM_04
  - Flag to enable the compiling of source code parts dealing with DHCP in the section “DHCP”
    - TS_DHCP_ENABLED
  - How long to wait for a response from the DHCP server in the section “DHCP”
    - TS_DHCP_REQUEST_TIMEOUT
  - Static IP address and subnet mask of the Ethernet Adapter (for the case if the DHCP server is unreachable) in the “LOCAL HOST” section:
    - TS_LOCAL_IP_ADDRESS
    - TS_LOCAL_IP_SUBNET_MASK
  - IP address and port of the remote computer to which the connection must be established in the “CONNECTION TARGET” section:
    - TS_CONNECT_TO_PEER_IP
    - TS_CONNECT_TO_PEER_PORT
  - Default getaway address in the “GATEWAY” section (if a router is a part of the network)
    - TS_DEFAULT_GATEWAY

- How to test it:
  - ✓ Install and configure a DHCP server on the computer to which the Ethernet Adapter can connect.
    - o Some links to DHCP servers for Windows:
      - http://www.magikinfo.com/dhcp.htm (MagikDHCP)
      - http://www.billiter.com/ (ipLease)
    - o Some links to Internet sites describing how to configure a DHCP server on Linux:
      - http://www.tldp.org/HOWTO/Net-HOWTO/
  - ✓ Use the “SServer.exe” program installed in the “..\Tools\SimpleServer” directory or another tcp server that is enabled to send an echo to the client.

2.9 DNS Client Demo

- Name :
  client_dns_ethe.tig
- Include Files:
  1. *ts_coinc.inc* – includes all TBSockets files,
  2. *client_clbks.inc* – includes the user callback tasks for the demo; originally, these tasks are only placeholders for user-defined reactions on some asynchronous events like coming of data from remote peer etc.

- Purpose: This program demonstrates how to get an IP address corresponding to a host name from **DNS** server and starts a simple client that actively opens connection using the obtained IP address and communicates with an **echo** server.

- Explanation:
  1. Creates a new socket (*lSocket*),
  2. Enables DNS requests and sets the DNS server IP address (*bSetupDns*),
  3. Sets a default gateway if necessary (*bSetDefaultGateway*),
  4. Tries to obtain an IP address for a host name from the DNS Server (*lDnsGetIpByName*),
  5. Establishes the connection to a remote host using the obtained IP address (*bConnect*),
  6. If the remote host was connected the message loop is started; the message loop involves:
     - Sending (*lSend*) the TEXT_TO_SEND character sequence,
     - Waiting for an echo or for quit signal ("Q" or "QUIT"; *OnData* task in the "client_clbks.inc") or for closing of socket on the remote host (*OnRemoteClose* task in the "client_clbks.inc"); the timeout is not checked;
     - Repeating the loop either SEND_LOOPS times or quitting by quit or close-socket signal or running endless (timeout error),
  7. Closes the socket (*bCloseSocket*) after leaving the message loop.

- Configuration Constants:
  All constants that must be correctly set by user to compile and run this demo are saved in the *ts_conf.inc* file in the following sections:
  - ➢ Module type in the “MODULE TYPE” section:
    - ❖ TS_EM_01 or TS_EM_02 or TS_EM_03 or TS_EM_04
  - ➢ Flag to enable the compiling of source code parts dealing with DNS in the section “DNS”
    - ❖ TS_ DNS_ENABLED
  - ➢ IP address of a DNS server in the section “DNS”
    - ❖ TS_DNS_SERVER_IP
  - ➢ How long to wait for a response from the DNS server in the section “DNS”
    - ❖ TS_DNS_REQUEST_TIMEOUT
  - ➢ Static IP address and subnet mask of the Ethernet Adapter in the “LOCAL HOST” section:
    - ❖ TS_LOCAL_IP_ADDRESS

    - ❖ TS_LOCAL_IP_SUBNET_MASK
  - ➢ IP address and port of the remote computer to which the connection must be established in the “CONNECTION TARGET” section:
    - ❖ TS_CONNECT_TO_PEER_NAME
    - ❖ TS_CONNECT_TO_PEER_PORT
    - ❖ TS_CONNECT_TO_PEER_IP – for the case if the DNS server is unreachable
  - ➢ Default getaway address in the “GATEWAY” section (if a router is a part of the network)
    - ❖ TS_DEFAULT_GATEWAY

- How to test it:
  - ✓ Use DNS server of an ISP or DNS server on LAN.
  - ✓ Use the “SServer.exe” program installed in the “..\Tools\SimpleServer” directory or another tcp server that is enabled to send an echo to the client.

2.10 SMTP Client Demo

- Name :
  smtp_client_ethe.tig (EM01, EM03), *smtp_client_ppp.tig* (EM02, EM04)
- Include Files:
  1. *ts_coinc.inc* – includes all TBSockets files,
  2. *smtp_pop_clbks.inc* – includes the user callback tasks for the demo; originally, these tasks are only placeholders for user-defined reactions on some asynchronous events like coming of data from remote peer etc.,
  3. *smtp_pop_subs.inc* – includes the subroutines implementing SMTP and POP client protocols.

- Purpose: This program demonstrates how to send an email using **SMTP** (RFC 821, RFC 1651) protocol. The protocol is implemented in Tiger Basic language, it is delivered as source code and can be changed by user to comply with the requirements of the particular SMTP server,

- Explanation:
  1. Dials the ISP (Web Adapters only) :
     - o See “2.3 Dialling Procedure for Web Adapters”,
  2. Creates a new socket (*lSocket*),
  3. Establishes the connection to a SMTP server (*bConnect*), identified by IP address and port number (normally: 25)
  4. If the server was connected, an email will be prepared and sent; the sending of an email involves:
     - o Setting all values relevant to the authentication of the owner of the email account and of the sender (see below: *Configuration Constants*),
     - o Setting all values relevant to the header information and the contents of the particular email (see below: *Configuration Constants*),

   - o Sending the email (*mail_send*) by means of exchange of the SMTP requests and responses between this SMTP client program and the connected SMTP server;
5. Closes the socket (*bCloseSocket*) after leaving the message loop.

- Configuration Constants:
  The most of constants that must be correctly set by user to compile and run this demo are saved in the *ts_conf.inc* file in the following sections:
  - ➢ Module type in the "MODULE TYPE" section:
    - ❖ TS_EM_01 or TS_EM_02 or TS_EM_03 or TS_EM_04
  - ➢ Static IP address and subnet mask of the Ethernet Adapter in the "LOCAL HOST" section:
    - ❖ TS_LOCAL_IP_ADDRESS
    - ❖ TS_LOCAL_IP_SUBNET_MASK
  - ➢ IP address or name (if DNS is enabled), and port of the SMTP server to which the connection must be established in the "SMTP" section:
    - ❖ TS_SMTP_SERVER_IP or TS_SMTP_SERVER_NAME
    - ❖ TS_SMTP_PORT
  - ➢ Default getaway address in the "GATEWAY" section (if a router is a part of the network)
    - ❖ TS_DEFAULT_GATEWAY
  - ➢ Parameters of the user of the SMTP service, i.e. account name, optional login and password (if authentication is activated) in the "SMTP" section:
    - ❖ TS_SMTP_ACCOUNT_NAME
    - ❖ TS_SMTP_AUTH_ENABLED, TS_SMTP_LOGIN, TS_SMTP_PASSWORD
  - ➢ Parameters of the sender of an email, i.e. sender name and sender domain (in the form "domainname.com") in the "SMTP" section:
    - ❖ TS_SMTP_SENDER_NAME
    - ❖ TS_SMTP_SENDER_DOMAIN

  The parameters and the contents of the particular email are placed in the *smtp_client_ethe.tig* file directly:
  - ➢ The name of the email sender, the email address of the receiver, the subject of the email and the message itself:
    - ❖ EMAIL_DATA_FROM
    - ❖ EMAIL_DATA_TO
    - ❖ EMAIL_DATA_SUBJECT
    - ❖ EMAIL_DATA_INIT_TEXT

- How to test it:
  ✓ Use an extern SMTP server or a local one (e.g. from "QKsoft" http://www.qksoft.com/).

3. Programming with Tiger Basic Sockets (TBSockets)

3.1 Terms

3.1.1 What is TBSockets

TBSockets is a collection of tasks and subroutines written in the Tiger Basic programming language and implementing an interface to the network world for the Basic Tiger based applications using specific hardware (Ethernet/Web Adapter designed by Wilke Technology GmbH). TBSockets consists of user interface subroutines (plus user defined callback tasks) and of chain of the supporting tasks and subroutines. This document describes the user interface.

3.1.2 How to use TBSockets

To use the TBSockets features:

- Include the 'ts_coinc.inc' file into the projects main 'tig' file,
- For the EM01/EM02 Adapters: install the device driver for serial channel in the "Main" task. The default settings for serial channel 0 (**SER0**): **38400, 8N1** for **EM01**, and **19200, 8N1** for **EM02**. The serial channel 0 is preferred because of implemented hardware handshake,
- Write the callback tasks *OnData*, *OnAccept, OnRemoteClose, OnDhcpUp* etc (find the appropriate placeholder tasks in the "def_clbks.inc" file in the "Ethernet_Web_Examples" directory),
- Call the *vInitSockets* (sub vInitSockets()) subroutine before using any other TBSockets subroutine. The *vInitSockets* intializes some variables, and starts supporting tasks,
- Write a client/server application based on the subroutines described below.

3.2 General Setup Subroutines

3.2.1 Set Local Ip Address and Local Ip Subnet Mask

Purpose:
The *bSetupLocalIp* subroutine is used to set the local ip address to *lpLocalIpAddress* and the local ip subnet mask to *lpLocalIpSubnetMask*.

Signature:
sub bSetupLocalIp(long lpLocalIpAddress; long lpLocalIpSubnetMask; var byte bpvSuccess)

3.2.2 Set Default Gateway

Purpose:
The *bSetDefaultGateway* subroutine sets the default gateway (the IP address of the intranet interface for the incoming connection) to the *lpDefaultGateway* value.

Signature:
sub bSetDefaultGateway(long lpDefaultGateway; var byte bpvSuccess)

Comments:

- If this subroutine is not called, the default gateway for a Web Adapter will be initially set to 192.168.1.253 (hex: C0A801FD).
- If this subroutine is not called, the default gateway for a Ethernet Adapter will be not initialised at all.
- Once it was explicitly or implicitly set, the default gateway can not be deleted.

3.2.3 Set Local Port

Purpose:
The *bSetupLocalPort* subroutine sets the local port (the port of a client that is used by a server for addressing the client and sending data to it) for the *lpSocket* socket to the *lpLocalPort* value.

Signature:
sub bSetupLocalPort (long lpSocket; long lpLocalPort; var byte bpvSuccess)

Comments:
- The *lpSocket* argument is the socket, i.e. the value returned by the *lSocket* subroutine.
- The value of the *lpLocalPort* cannot be greater than 65535 (0FFFFh).
- If the local port was not explicitly set before the connection establishing, the local port is chosen automatically by the module by means of a pseudo-random generator.
- Attention: the pseudo-random generator creates the same first value for the local port after each restart of the module. Therefore in the case of the broken TCP connection the application can be forced to implicitly use the *bSetupLocalPort* subroutine, if the restart is executed quickly.
  E.g. the implementation of the TCP stack for Windows OS blocks an open socket for 240s if the connection was broken. To connect to the same ip address and the same remote port during the 240s a client application should use another local port than the local port of the previous, blocked connection. If the module is merely restarted and the new local port is not explicitly set, the pseudo-random generator can produce the same default local port as for the previous socket, and Windows will not accept the connection.

Availability:
Only for the Ethernet/Web Adapters version V1.6 or higher (see imprint on the front side of the Ethernet/Web modules) and for the Tiger Basic Sockets (TBSockets) implementation version 1.03 (see the bGetAdapterProgVers subroutine in the “Get Version of the Adapter Software” paragraph of this manual).

3.2.4 System Reset

Purpose:
The *bSystemReset* subroutine causes the immediate reset of the Ethernet/Web Adapter.

Signature:
sub bSystemReset(var byte bpvSuccess)

Return:
On error (the command could not be sent to the adapter): the *bpvSuccess* is set to FALSE.
On success (the command is sent to the adapter): the *bpvSuccess* is TRUE.

Availability:
Only for the Ethernet/Web Adapters version V1.7 or higher (see imprint on the front side of the Ethernet/Web modules) and for the Tiger Basic Sockets (TBSockets) implementation version 1.04 (see the bGetAdapterProgVers subroutine in the "Get Version of the Adapter Software" paragraph of this manual).

3.3 'Get Param' Subroutines

3.3.1 Get Local Ip Address

Purpose:
The *lGetLocalIp* subroutine gets the local ip address for the *lpSocket* socket.

Signature:
sub lGetLocalIp(long lpSocket; var long lpvLocalIp)

Return:
On error: the *lpvLocalIp* is set to DUMMY_IP (hex: FFFFFFFF) and the *lLastErrorCode* contains the error code.
On success: the *lpvLocalIp* contains the requested ip address.

3.3.2 Get Local Port Number

Purpose:
The *wGetLocalPort* subroutine gets the local port number for the *lpSocket* socket.

Signature:
sub wGetLocalPort(long lpSocket; var word wpvLocalPort)

Return:
On error: the *wpvLocalPort* is set to DUMMY_PORT (hex: FFFF) and the *lLastErrorCode* contains the error code.
On success: the *wpvLocalPort* contains the requested port number.

3.3.3 Get Remote Ip Address

Purpose:
The *lGetRemoteIp* subroutine gets the ip address of the remote host for the *lpSocket* socket.

Signature:
sub lGetRemoteIp(long lpSocket; var long lpvRemoteIp)

Return:
On error: the *lpvRemoteIp* is set to DUMMY_IP (hex: FFFFFFFF) and the *lLastErrorCode* contains the error code.
On success: the *lpvRemoteIp* contains the requested ip address.

3.3.4 Get Remote Port Number

Purpose:

The *wGetRemotePort* subroutine gets the port number of the remote host for the *lpSocket* socket.

Signature:
sub wGetRemotePort(long lpSocket; var word wpvRemotePort)

Return:
On error: the *wpvRemotePort* is set to DUMMY_PORT (hex: FFFF) and the *lLastErrorCode* contains the error code.
On success: the *wpvRemotePort* contains the requested port number.

3.3.5 Get Version of the Adapter Software

Purpose:
The *bGetAdapterProgVers* subroutine returns the version of the adapter program.

Signature:
sub bGetAdapterProgVers(var long lpvProgVers)

Return:
On error: the *lpvProgVers* is set to (-1) and the *lLastErrorCode* contains the error code.
On success: the *lpvProgVers* contains the requested version.

Comments:
- The program version is returned in a long variable and can be converted to the readable string form by using of the *sConvProgVersToString* subroutine.

  Signature:
  sub sConvProgVersToString(long lpProgVers; var string spvProgVers$)
- The ADAPTER_PROG_VERS_STR_SIZE (16) constant tells how long the spvProgVers$ string must minimally be.

3.3.6 Get Ethernet Link State

Purpose:
The *bGetEtheLinkState* subroutine gets the state of the Ethernet link.

Signature:
sub bGetEtheLinkState(var byte bpvLinkState)

Return:
On bad link: the *bpvLinkState* is 0.
On good link: the *bpvLinkState* is 1.

Availability:
Only for the Ethernet/Web Adapters version V1.7 or higher (see imprint on the front side of the Ethernet/Web modules) and for the Tiger Basic Sockets (TBSockets) implementation version 1.04 (see the bGetAdapterProgVers subroutine in the "Get Version of the Adapter Software" paragraph of this manual).

3.4 Ethernet MAC Address Subroutines (Ethernet Adapter)

Ethernet MAC (Media Access Control) address is a hardware address that uniquely identifies each node of an Ethernet network. The Ethernet MAC addresses are 48 bits, usually expressed as 12 hexadecimal digits. All Ethernet adapters (both EM01 and EM03 series) are delivered with an unique MAC address stored in the non-volatile memory. So, normally, the using of this subroutine must be avoided, except an area of unique MAC addresses was additionally reserved for produced series, otherwise the change of the MAC address can be dangerous.

3.4.1 Set MAC Address

Purpose:
The *bSetMacAddress* subroutine forces an Ethernet adapter to use the new *spMacAddress$* MAC address and to store this in the non-volatile memory.

Signature:
sub bSetMacAddress(string spMacAddress$; var byte bpvSuccess)

Comments:
- There are two methods to assign the 12 hexadecimal digits of an Ethernet MAC address to the *spMacAddress$* string in the Tiger Basic program:

spMacAddress$ = "FF FF FF FF FF FF"%
or
spMacAddress$ = "<0FFh> <0FFh> <0FFh> <0FFh> <0FFh> <0FFh>"

- The MAC_ADDR_SIZE (6) constant tells how long the *spMacAddress$* string must minimally be.

3.4.2 Get MAC Address

Purpose:
The *bGetMacAddress* subroutine gets the actually used MAC address of the Ethernet adapter and saves it in the spvMacAddress$ variable.

Signature:
sub bGetMacAddress(var string spvMacAddress$; var byte bpvSuccess)

Comments:
The MAC_ADDR_SIZE (6) constant tells how long the *spMacAddress$* string should be.

3.5 Tcp Settings Subroutines

3.5.1 Set Tcp Window Size

The tcp window size determines how many characters can be sent or received in one tcp package.

Purpose:
The *bSetTcpWinSize* subroutine sets the tcp window size to the *lpTcpWinSize* value. This value cannot be greater than 128.

Signature:
sub bSetTcpWinSize(long lpSocket; long lpTcpWinSize; var byte bpvSuccess)

Comments:
- The default tcp window size for any adapter is 128 bytes.

3.5.2 Set Tcp Keep-Alive Segments

Purpose:
The *bSetTcpKeepAlive* subroutine activates the keep-alive mechanism and sets its parameters to send a keep-alive probe every *lpTcpKATicks* milliseconds *bpTcpKAProbes* times.

Signature:
sub bSetTcpKeepAlive(long lpSocket; long lpTcpKATicks; byte bpTcpKAProbes; var byte bpvSuccess)

Comments:
- The *lpTcpKATicks* variable specifies the number of milliseconds between consecutive keep-alive probes.
- The *bpTcpKAProbes* variable specifies the number of probes that can be sent without response before declaring a socket to be dead.
- To deactivate the keep-alive mechanism, call the *bSetTcpKeepAlive* subroutine with the *lpTcpKATicks* and *bpTcpKAProbes* parameters set to 0 (zero).
- By default, the sending of the keep-alive segments is not activated.
- This function is not implemented for the EM02/EM04 at the moment.

3.5.3 Get Tcp Settings

Purpose:
The *lGetTcpSettings* subroutine gets the actually used tcp window size and tcp keep-alive parameters.

Signature:
```
sub lGetTcpSettings( long lpSocket; var long lpvTcpWinSize, lpvTcpKATicks;
        var byte bpvTcpKAProbes; var byte bpvSuccess )
```

Comments:
If the *lpTcpKATicks* and the *bpvTcpKAProbes* variables are set to 0 (zero), the keep-alive mechanism is deactivated.

3.6 Modem Subroutines (Web Adapter)

3.6.1 Communicate with Modem directly

The subroutines of this subsection enable the data exchange immediately with a modem, without involving any function of a Web Adapter that might influence the transferred data.

Purpose:

The *bModemSend* subroutine sends the *spDataToSend$* string to the attached modem.

Signature:
sub bModemSend(string spDataToSend$; long lpTimeOut; var byte bpvDataSent)

Purpose:
The *bModemReceive* subroutine receives in the *spvReceivedData$* string the data of the *lpRecvBufSize* maximal size from the modem.

Signature:
sub bModemReceive(var string spvReceivedData$; long lpRecvBufSize; long lpTimeOut; var byte bpvIsReceived)

Purpose:
The *wModemGetSendReady* subroutine returns in the *wpvSendSize* parameter the number of bytes of data that a modem can accept for sending.

Signature:
sub wModemGetSendReady(long lpTimeOut; var word wpvSendSize)

Purpose:
The *wModemGetRecvReady* subroutine returns in the *wpvRecvSize* parameter the number of bytes of unprocessed data the modem has in its receive buffer.

Signature:
sub wModemGetRecvReady(long lpTimeOut; var word wpvRecvSize)

Return:
All above subroutines try to get the particular work done during the *lpTimeOut* period of time. If the *lpTimeOut* was expired and the function was not executed the return value is FALSE for *bModemSend* and *bModemReceive* subroutines or 0 (zero) for *wModemGetSendReady* and *wModemGetRecvReady* subroutines, the corresponding error code is saved in the *lLastErrorCode* variable.

3.6.2 Send AT Commands

Purpose:
The *bSendATCommand* subroutine sends the *spATCommandToSend$* AT command to the modem and receives reply in the *spvATReply$*.

Signature:
sub bSendATCommand(string spATCommandToSend$; long lpATTimeOut; var string spvATReply$; var byte bpvSuccess)

Comments:

- The *bSendATCommand* subroutine appends the ending Carriage Return (0d hex) character to the *spATCommandToSend$* string.
- The maximal size of the reply that is set now to 128 (see: "AT_REPLY_MAX_SIZE") must not be changed by user. The *lpATTimeOut* is a timeout value measured in seconds to wait for reply from the modem.

3.6.3 Listen to Modem Data

The alternative mechanism to communicate with the modem listening to the modem until the certain number of data is come and a callback task is launched.

Purpose:
The *bModemStartListen* subroutine starts the listening procedure. The *lpMinRecvDataSize* parameter defines how many characters must be received before the callback task can be launched. The *lpRequestFreq* says how often (measured in ms) the listening procedure must be activated (e.g. if the *lpRequestFreq* value is 2000, the procedure will be called every 2 seconds).

Signature:
sub bModemStartListen(long lpMinRecvDataSize, lpRequestFreq; var byte bpvSuccess)

Purpose:
The bModemStopListen subroutine stops the listening procedure that was started by the *bModemStartListen* call.

Signature:
sub bModemStopListen(var byte bpvSuccess)

Purpose:
The callback task that is launched when the data is come.

Signature:
task OnModemData

Comments:

- Two global variables are set by the TBSockets Launcher at launching this task: *lActModemDataSize* and *sActModemDataBuffer*$.
- The *lActModemDataSize* variable stores the size of the coming data.
- The *sActModemDataBuffer*$ variable stores the coming data itself.
- The *OnModemData* task is normally written by user. In the delivered examples the implementation of this task can be found in the include files named corresponding to the following pattern: 'application_name_clbks.inc'.
- Internally, the listening procedure is a time-driven function that firstly reads all the data up to 128 characters from modem, then checks whether the number of the read data is equal or greater than *lpMinRecvDataSize*, and if yes, the *OnModemData* task is launched, and the timer for the listening procedure is started again using the *lpRequestFreq* value.

3.6.4 Dialling Procedure

The dialling procedure for modem consists of dialling an Internet Service Provider (phone number set by *bSetupIsp*) and authenticating of the user (user name and password set by *bSetupIsp* or *bSetPapSecrets* correspondingly to the chosen scheme of dialling).

3.6.4.1 Set Internet Service Provider Data

Purpose:
The *bSetupIsp* subroutine sets the data used while dialling Internet Service Provider to the *spModemDialString*$ phone number (without ATDT-prefix and without <Carriage Return>-suffix), the *spUserName*$ user name and the *spUserPassword*$ user password.

Signature:
sub bSetupIsp(string spModemDialString$; string spUserName$;
string spUserPassword$; var byte bpvSuccess)

Comments:
- In case of applying the *bDialIsp* (not *bDialIspWithLogin*) subroutine for dialling an ISP the *spUserName*$ and the *spUserPassword*$ parameters will be ignored and the proper user name and password should be set by means of the *bSetPapSecrets* subroutine.
- The *bSetupIsp* subroutine must be called before dialling the ISP by *one* of the dialling subroutines.

3.6.4.2 Set PAP Secrets (Authentication Parameters)

Purpose:
The *bSetPapSecrets* subroutine is used to define the authentication parameters transferred to ISP as a part of PPP protocol. The so named PAP secrets will be set to *spUserName*$ user login name and to *spUserPassword*$ user password. The parameter *bpIndex* declares to which set of authentication parameters the login name and password belong.

Signature:
sub bSetPapSecrets (string spUserName$; string spUserPassword$; byte bpIndex; var byte bpvSuccess)

Comments:
- The parameters of each set will be tested one after another until the connection to an ISP is established. Maximal number of the authentication parameter sets is 3.
- The call of the *bDialIsp* (not *bDialIspWithLogin*) subroutine causes the transferring of PAP secrets during PPP phase.
- The *bSetPapSecrets* subroutine must be called before dialling the ISP by the *bDialIsp* subroutine.

3.6.4.3 Dial with Login

Purpose:
According to this dialling scheme the login procedure is started immediately after dialling an ISP, and PPP is actively launched on the successful logging (user name and password are correct)

Signature:
sub bDialIspWithLogin(long lpDialTimeout; var long lpvAssignedIpAddr; var byte bpvSuccess)

Comments:
The phone number, user name and password must be set by means of the *bSetupIsp* subroutine.

3.6.4.4 Dial without Login

Purpose:
Some ISPs (for example ISPs for GPRS) require the authentication made by PAP during the PPP phase. The dialling procedure for such an ISP doesn't invoke the sending of user name and password, because the ISP itself establishes the PPP connection and asks the client about authentication.

Signature:
sub bDialIsp(long lpDialTimeout; var long lpvAssignedIpAddr; var byte bpvSuccess)

Comments:
The appropriate authentication parameters (login and password) must be set for this dialling scheme by means of the *bSetPapSecrets* subroutine.

Comments:
- Both dialling subroutines give back in the *lpvAssignedIpAddr* parameter an IP address assigned by the ISP to our node.
- The *lpDialTimeout* defines the timeout to wait for connection to the ISP.

3.6.5 Hanging Up Procedure

Purpose:
The *bHangUp* subroutine forces the adapter to finish the modem session sending the commands "+++" and "ath" (both commands wait for "OK"s) to the modem.

Signature:
sub bHangUp(var byte bpvSuccess)

3.6.6 Set the Modem Baudrate

Purpose:
The *bSetModemBaudrate* subroutine sets the speed of communication between the adapter and modem to *lpModemBaudRate*.

Signature:
sub bSetModemBaudrate(long lpModemBaudRate; var byte bpvSuccess)

3.6.7 Get the CTS Pin State

Purpose:

The *bGetCtsPinState* subroutine returns in the *bpvCtsPinState* the state of the CTS pin of the adapter connected to modem. The pin state is one of the following constants: PIN_STATE_HIGH (1), PIN_STATE_LOW (0), PIN_STATE_UNDEF (hex: FF).

Signature:
sub bGetCtsPinState(var byte bpvCtsPinState)

Return:
On error: the *bpvCtsPinState* is set to PIN_STATE_UNDEF (hex: FF) and the *lLastErrorCode* contains the error code.
On success: the *bpvCtsPinState* is either PIN_STATE_HIGH (1) or PIN_STATE_LOW (0).

3.7 DHCP Subroutines

An application using DHCP should enable dhcp (*bSetupDhcp* call) before creating the first socket (*lSocket* call), the very first call of the *lSocket* causes sending of dhcp request, and in this phase the application should check whether the dhcp request was successful while the *bDhcpIsUp* global variable is tested on “TRUE” (*bDhcpIsUp* must be previously set to “FALSE”); if the request was successful, the callback task *OnDhcpUp* is launched, the *bDhcpIsUp* variable is set to “TRUE” and the new ip address assigned by dhcp server is stored in the *lActDhcpUpIpAddr* global variable.

3.7.1 Enable DHCP

Purpose:
The *bSetupDhcp* subroutine activates or deactivates the request made by the client to the remote DHCP server to get the ip address and other dynamically assigned parameters of the connection.

Signature:
sub bSetupDhcp(byte bpDhcpFlag; var byte bpvSuccess)

Comments:
- The value of *bpDhcpFlag* is one of the following constants defined in ‘ts_com_d.inc’:
  USE_DHCP (1) – activates DHCP request,
  NO_USE_DHCP (2) – deactivates DHCP request,
  USE_STANDARD - NO_USE_DHCP
- The *bSetupDhcp* subroutine must be called before opening the first socket by the *lSocket* subroutine.

3.7.2 Get the IP address assigned by DHCP Server

Purpose:
If the request succeeds, the *OnDhcpUp* callback task is launched and the new ip address assigned by dhcp server is stored in the *lActDhcpUpIpAddr* global variable.

Signature:
task OnDhcpUp

Comments:
- Three global variables are set by the TBSockets Launcher at launching this task:

long lActDhcpUpIpAddr,
long lActDhcpUpNetMask,
long lActDhcpUpGateway.

- The *lActDhcpUpIpAddr* variable stores the ip address, the *lActDhcpUpNetMask* variable – the subnet mask, and the *lActDhcpUpGateway* variable – the default gateway assigned to the client by the remote DHCP server.
- The *OnDhcpUp* task is normally written by user. In the delivered examples the implementation of this task can be found in the include files named corresponding to the following pattern: 'application_name_clbks.inc'.

3.8 DNS Subroutines

An application using DNS should send the first dns request only after the first new socket was created (*lSocket* call), then the actual dns request is done by means of the following:

- the dns is enabled and the dns server ip address is set by *bSetupDns* call,
- the default gateway ip address is set by *bSetDefaultGateway* call,
- the dns request for a remote host name is done by *lDnsGetIpByName* call;

if the dns request was successful, the corresponding ip address is given back to the application as a parameter of the *lDnsGetIpByName* subroutine.

3.8.1 Enable DNS and set DNS Server IP address

Purpose:
The *bSetupDns* subroutine enables the DNS request made by the client to the remote DNS server to get the ip address corresponding to a particular host name. The *bSetupDns* subroutine sets also the IP address of the DNS server to the *lpDnsServerIpAddress* value.

Signature:
sub bSetupDns(byte bpDnsFlag; long lpDnsServerIpAddress; var byte bpvSuccess)

Comments:

- The value of *bpDnsFlag* is one of the following constants defined in 'ts_com_d.inc':
  USE_DNS (1) – enables DNS request,
  NO_USE_DNS (2) – disables DNS request,
  USE_STANDARD - NO_USE_DNS
- The *bSetupDns* subroutine must be called after opening the first socket by the *lSocket* subroutine.

3.8.2 Get IP address for a Host Name from DNS Server

Purpose:
The *lDnsGetIpByName* subroutine requests the IP address for the spHostName$ host name from the DNS Server.

Signature:
sub lDnsGetIpByName(string spHostName$; long lpTimeOut; var long lpvIpAddress)

Return:
On error: the *lpvIpAddress* is set to zero (0) and the *lLastErrorCode* contains the error code.

On success: the *lpvIpAddress* contains the requested ip address.

Comments:

- Before the DNS request can be accomplished, the DNS must be enabled and the IP address of the DNS Server must be configured (*bSetupDns* call), and the Default Gateway must be set (*bSetDefaultGateway* call).

3.9 Client-Server Subroutines

Typically, one of the ends of a socket-based data communication is a *server*, the other is a *client*.

3.9.1 The Common Elements

3.9.1.1 Open Socket

Purpose:
The subroutine used by both, clients and servers, is *lSocket*. The *lSocket* subroutine allocates a new socket with the required characteristics. The maximal number of the sockets running concurrently is limited to 6 (six).

Signature:
sub lSocket(byte bpAddrFormat, bpType; var long lpvSocket)

Return:
On error: the *lpvSocket* is set to (-1) and the *lLastErrorCode* contains the error code.
On success: the *lpvSocket* contains the numerical identifier of the allocated socket.

Comments:

- The *bpAddrFormat* is an address format specification. The only format currently supported is PF_INET (2), which is the ARPA Internet address format. The constant PF_INET (2) is defined in ‘ts_com_d.inc’.
- Two values are defined for the *bpType* argument, again, in ‘ts_com_d.inc’. Both start with ‘SOCK_’. The most common one is SOCK_STREAM (1), which tells the system you are asking for a *reliable stream delivery service* (which is TCP in this case).
- If you asked for SOCK_DGRAM (2), you would be requesting a *connectionless datagram delivery service* (in our case, UDP).
- The Unconnected Socket: Nowhere, in the *lSocket* subroutine have we specified to what other system we should be connected. Our newly created socket remains *unconnected*.

3.9.1.2 Close Socket

Purpose:
Each opened socket must be closed if it is no longer in use.

Signature:
sub bCloseSocket(long lpSocket; var byte bpvSuccess)

Comments:

- The *lpSocket* argument is the socket, i.e. the value returned by the *lSocket* subroutine or the value saved in the *lActAcceptSocket* variable on accepting a new client.

3.9.1.3 *Remote Socket Closed Notification*

Purpose:
If the remote peer closes the connection the '*OnRemoteClose'* task is started.

Signature:
Task OnRemoteClose

Comments:
- The *lActRemoteCloseSocket* global variable contains the socket that was immediately closed by the remote peer.
- The additional call of the bCloseSocket subroutine is not necessary. If called, it will return an error.

3.9.1.4 *Socket Address Block (SA Block)*

Various subroutines of the sockets family expect the string generally referenced as 'Socket Address Block' as one of the arguments. The data of different types and sizes are stored in the Socket Address Block string. The particular fields of this block can be accessed by means of the built-in functions (like nfroms, rfroms, mid$ etc) reading the definite number of bytes from the specific offset into a variable. The following offset and size values can be applied for accessing the information about the network addresses:

Offset	Size	Description
SABLK_LEN_OFFS (0)	SABLK_LEN_SIZE (1)	Size of SA Block
SABLK_FAMILY_OFFS (1)	SABLK_FAMILY_SIZE (1)	Address Family
SABLK_PORT_OFFS (2)	SABLK_PORT_SIZE (2)	Port
SABLK_ADDR_OFFS (4)	SABLK_ADDR_SIZE (4)	IP Address

The size of the Socket Address Block is fix now (use the constants SABLK_DEFAULT_LEN (16), SABLK_INET_LEN (16) defined in 'ea_conf.inc' and 'ts_com_d.inc'), but may be modified in the future.

The only *address family* currently supported is AF_INET (2) (defined in 'ts_com_d.inc').

The *port* is a value of type 'word' (16-bit unsigned integer) standing for the connection port number.

The *IP address* is of type 'long' (32-bit signed integer). Because the value of a numeric type can not directly represent an IP address in the more convenient 'dotted' notation, one should convert it into a 32-bit integer. For example the value 0C0A80102H (or 3232235778) is used to express the 192.168.1.2 IP address.

The exact meaning of the *port* and *IP address* fields depends on the subroutine of the sockets family using the Socket Address Block.

Two subroutines facilitate considerably the accessing of the particular values of the Socket Address Block:

Signature:
sub sBuildSockAddrBlock(var string spvSockAddrBlock$; byte bpSaLength, bpSaFamily; word wpSaPort; long lpSaAddress)
and

Signature:
sub vParseSockAddrBlock(string spSockAddrBlock$; var byte bpvSaLength, bpvSaFamily; var word wpvSaPort; var long lpvSaAddress)

The *sBuildSockAddrBlock* subroutine copies the *bpSaLength*, *bpSaFamily, wpSaPort* and *lpSaAddress* to the Socket Address Block string *spvSockAddrBlock$*.
On the contrary the *vParseSockAddrBlock* subroutine extracts the particular values of the Socket Address Block *spSockAddrBlock$* and saves them to the *bpvSaLength*, *bpvSaFamily*, *wpvSaPort*, *lpvSaAddress*.

3.9.2 Client

Typically, the client initiates the connection to the server. The client knows which server it is about to call: it knows its IP address, and it knows the *port* the server resides at.

3.9.2.1 Connect

Purpose:
Once a client has created a socket, it needs to connect it to a specific port on a remote system.

Signature:
sub bConnect(long lpSocket; long lpConnEstTimeOut; string spSockAddr$; word wpSockAddrLen; var byte bpvSuccess)

Return:
If *bConnect* is successful, it returns TRUE in the *bpvSuccess*. Otherwise it returns FALSE and stores the error code in the *lLastErrorCode* variable (global).

Comments:
- The *lpSocket* argument is the socket, i.e. the value returned by the *lSocket* subroutine.
- If the connection can not be established and the reason of the failure is not known, the *bConnect* gets back in *lpConnEstTimeOut* seconds and the CME_CONNECT_TIMED_OUT value is assigned to the *lLastErrorCode* variable.
- The *spSockAddr$* string is a Socket Address Block, the structure we have talked about extensively. In this case, the *port* and *ip address* identify the server that has to be connected.
- Finally, *wpSockAddrLen* informs the system how many bytes are in our Socket Address Block string.
- There are many reasons why *bConnect* may fail. For example, with an attempt to an Internet connection, the IP address may not exist, or it may be down, or just too busy, or it may not have a server listening at the specified port. Or it may outright *refuse* any request for specific code.

3.9.3 Server

The typical server does not initiate the connection. Instead, it waits for a client to call it and request services. It does not know when the client will call, nor how many clients will call. It may be just sitting there, waiting patiently, one moment, the next moment, it can find itself swamped with requests from a number of clients, all calling in at the same time.
The sockets interface offers two basic subroutines and one callback task to handle this.

3.9.3.1 Bind

Purpose:
There are 65535 IP ports, but a server usually processes requests that come in on only one of them. The *bBind* subroutine is used to tell sockets which port is to serve.

Signature:
sub bBind(long lpSocket; string spSockAddr$; word wpSockAddrLen; var byte bpvSuccess)

Return:
If *bBind* succeeds, it returns TRUE in the *bpvSuccess*. Otherwise it returns FALSE and stores the error code in the l*LastErrorCode* variable (global).

Comments:
- The *lpSocket* argument is the socket, i.e. the value returned by the *lSocket* subroutine.
- The *spSockAddr$* string is a Socket Address Block of the *wpSockAddrLen* length. Beside specifying the *port* in *spSockAddr$*, the server may include its *ip address*. However, it can just use the symbolic constant INADDR_ANY (0) to indicate it will serve all requests to the specified port regardless of what its IP address is. This symbol, is defined in 'ts_com_d.inc'.

3.9.3.2 Listen

Purpose:
The server waits for incoming connection with the *bListen* subroutine.

Signature:
sub bListen(long lpSocket, lpBackLog; var byte bpvSuccess)

Return:
The *bListen* subroutine returns in the *bpvSuccess* TRUE on success, otherwise the *bpvSuccess* is FALSE and the l*LastErrorCode* variable contains the error code.

Comments:
- The *lpSocket* argument is the socket, i.e. the value returned by the *lSocket* subroutine.
- The *lpBackLog* variable tells sockets how many incoming requests to accept while you are busy processing the last request. In other words, it determines the maximum size of the queue of pending connections.
- The number of back logs (*lpBackLog*) is limited to 5.
- If the *lpBackLog* is set to 0 (zero), the number of acceptable requests is meant to be maximal (now: 5).

3.9.3.3 Connection Accepted Notification

Purpose:
The server accepts the connection by starting the '*OnAccept*' task.

Signature:
task OnAccept

Comments:
- Three global variables are set by server and can be analysed in the *OnAccept* task, that is started by the TBSockets Launcher if the server accepts a new connection:
  long lActAcceptSocket,
  word wActAcceptSABlockLen,
  string sActAcceptSABlock$,
  word wActAcceptLocalPort
- The *lActAcceptSocket* value differs from the socket used in the *bBind* and *bListen* subroutines. Indeed, a new socket is created on accept. You will use this new socket to communicate with the client.
- What happens to the old socket? It continues to listen for more requests (remember the *lpBackLog* variable we passed to *bListen*?) until we close it.
- Now, the new socket is meant only for communications. It is fully connected. We cannot pass it to *bListen* again, trying to accept additional connections.
- The *sActAcceptSABlock$* string contains the port number and the ip address of the client.
- The *wActAcceptLocalPort* value says at which local port the connection was actually accepted.
- An established connection with a client remains active until either server or client hang up.
- The *OnAccept* task is normally written by user. In the delivered examples the implementation of this task can be found in the include files named correspondingly to the following pattern: 'application_name_clbks.inc'.

Availability:
The parameter *wActAcceptLocalPort* is only for the Ethernet/Web Adapters version V1.7 or higher (see imprint on the front side of the Ethernet/Web modules) and for the Tiger Basic Sockets (TBSockets) implementation version 1.04 (see the bGetAdapterProgVers subroutine in the "Get Version of the Adapter Software" paragraph of this manual) available.

3.9.4 Send and Receive Data (TCP)
Once the connection is established the data can be exchanged in both directions.

3.9.4.1 Send (TCP)

Purpose:
The *lSend* subroutine must be used to send the data to the remote host by TCP.

Signature:
sub lSend(long lpSocket; string spData$; long lpDataLen; word wpFlags; var long lpvSentBytesNum)

Return:

The *lpvSentBytesNum* argument returns the total number of bytes sent to the remote host by the *lSend* subroutine. The l*LastErrorCode* variable contains the error code if the sending fails.

Comments:

- The *lpSocket* argument is the socket, i.e. the value returned by the *lSocket* subroutine.
- The *lSend* subroutine sends the data of *lpDataLen* from the *spData$* string. If *lpDataLen* is longer than the size of a packet, the data will be split into many packets by *lSend*. A time interval between two packets can be defined by using of the TS_PARTIAL_SEND_PAUSE constant. By default, the TS_PARTIAL_SEND_PAUSE constant is set to 0 (zero).
- The *wpFlags* argument is reserved for the future extensions and is not used now.

3.9.4.2 Data Received Notification (TCP)

Purpose:
If the “TCP”-data are received, the TBSockets launches the *OnData* task:

Signature:

```
task OnData
```

Comments:

- Three variables are set by the TBSockets Launcher and must be used to access to the received data:
  long lActDataSocket,
  long lActDataSize,
  string sActDataBuffer$
- The *lActDataSocket* variable identifies the socket to which the received data belong.
- The *sActDataBuffer$* string contains the proper data of the *lActDataSize* length.
- The *OnData* task is normally written by user. In the delivered examples the implementation of this task can be found in the include files named corresponding to the following pattern: ‘application_name_clbks.inc’.

3.9.5 Send and Receive Data (UDP)

Once a “UDP”-socket is created the data can be sent to or received from a removed peer. To create an “UDP”-socket one should use the *lSocket* subroutine with the *bpType* argument set to SOCK_DGRAM (2) value.

3.9.5.1 Send (UDP)

Purpose:
The *lSendTo* subroutine must be used to send the data to the remote host by UDP.

Signature:

```
sub lSendTo( long lpSocket; string spSockAddr$; word wpSockAddrLen; &
  string spData$; long lpDataLen; word wpFlags; var long lpvSentBytesNum )
```

Return:
The *lpvSentBytesNum* argument returns the total number of bytes sent to the remote host by the *lSendTo* subroutine. The l*LastErrorCode* variable contains the error code if the sending fails.

Comments:

- The *lpSocket* argument is the socket, i.e. the value returned by the *lSocket* subroutine.
- The *spSockAddr$* string is a Socket Address Block of the *wpSockAddrLen* length. Beside specifying the *port* in *spSockAddr$*, the server may include its *IP address*. However, it can just use the symbolic constant INADDR_ANY (0) to indicate it will serve all requests to the specified port regardless of what its IP address is. This symbol, is defined in 'ts_com_d.inc'.
- The *lSendTo* subroutine sends the data of *lpDataLen* from the *spData$* string. If *lpDataLen* is longer than the size of a packet, the data will be split into many packets by *lSendTo*. A time interval between two packets can be defined by using of the TS_PARTIAL_SEND_PAUSE constant. By default, the TS_PARTIAL_SEND_PAUSE constant is set to 0 (zero).
- The *wpFlags* argument is reserved for the future extensions and is not used now.

Availability:
Only for the Ethernet/Web Adapters version V1.6 or higher (see imprint on the front side of the Ethernet/Web modules) and for the Tiger Basic Sockets (TBSockets) implementation version 1.03 (see the bGetAdapterProgVers subroutine in the "Get Version of the Adapter Software" paragraph of this manual).

3.9.5.2 Data Received Notification (UDP)

Purpose:
If the "UDP"-data are received, the TBSockets launches the *OnDataRecvFrom* task.

Signature:

```
task OnDataRecvFrom
```

Comments:

- Five variables are set by the TBSockets Launcher and must be used to access to the received data:
  long lActRecvFromAddress,
  word wActRecvFromPort,
  word wActRecvFromFlags,
  long lActRecvFromDataSize,
  string sActRecvFromDataBuffer$
- The *lActRecvFromAddress* and *wActRecvFromPort* variables identify (by IP address and port number) the remote peer from which the data were received.
- The *wActRecvFromFlags* is reserved for future extensions.
- The *sActRecvFromDataBuffer$* string contains the proper data of the *lActRecvFromDataSize* length.
- The *OnDataRecvFrom* task is normally written by user. In the delivered examples the implementation of this task can be found in the include files named corresponding to the following pattern: 'application_name_clbks.inc'.

Availability:
Only for the Ethernet/Web Adapters version V1.6 or higher (see imprint on the front side of the Ethernet/Web modules) and for the Tiger Basic Sockets (TBSockets) implementation version 1.03

(see the bGetAdapterProgVers subroutine in the “Get Version of the Adapter Software” paragraph of this manual).

3.10 SNTP Time Service

3.10.1 Time Request

Purpose:
The *lSntpGetTime* subroutine requests the time from a Network Time Server using SNTP protocol. The IP address of the NTP server is in the *lpSntpServerIpAddress* argument.

Signature:
```
sub lSntpGetTime( long lpSntpServerIpAddress; word wpSntpRequestTimeout; &
  var long lpvSeconds; var long lpvFraction )
```

Return:
The *lSntpGetTime* subroutine tries to get the particular work done during the *wpSntpRequestTimeout* period of time. If the *wpSntpRequestTimeout* was expired and the function was not executed both *lpvSeconds* and *lpvFraction* return values are 0 (zero), the corresponding error code is saved in the *lLastErrorCode* variable. If the request succeeded, the *lpvSeconds* argument returns the seconds since midnight, January 1, 1900 and the *lpvFraction* argument returns the fraction of a second not included in seconds.

Comments:
- The *lpSntpServerIpAddress* argument is the IP address of an NTP server.
- The *wpSntpRequestTimeout* argument is timeout for the SNTP request (measured in seconds).
- To use this subroutine the TS_SNTP_ENABLED constant must be activated (defined) (see: the “SNTP” section in the “ts_conf.inc” file).
- The TS_SNTP_SERVER_IP and TS_SNTP_REQUEST_TIMEOUT constants (see: the “SNTP” section in the “ts_conf.inc” file) are used in the “Sntp_Get_Time.Tig” sample and they should be correctly initialised if this sample is intended to be used as it is.

Availability:
Only for the Ethernet/Web Adapters version V1.6 or higher (see imprint on the front side of the Ethernet/Web modules) and for the Tiger Basic Sockets (TBSockets) implementation version 1.03 (see the bGetAdapterProgVers subroutine in the “Get Version of the Adapter Software” paragraph of this manual).

3.10.2 Time Conversion

The *lSntpConvertTime* converts SNTP time (seconds since 1900) to standard time (seconds since 1970).

Signature:
```
sub lSntpConvertTime( long lpSntpTime; var long lpvStandardTime )
```

Return:

The *lpvStandardTime* argument returns the time stamp in the standard format (seconds since 1970) that is converted from the SNTP time (seconds since 1900). If the *lpSntpTime* value is smaller than the number of seconds elapsed from 1900 up to 1970 (i.e. 2208988800), the *lpvStandardTime* argument returns (-1).

Availability:
Only for the Ethernet/Web Adapters version V1.6 or higher (see imprint on the front side of the Ethernet/Web modules) and for the Tiger Basic Sockets (TBSockets) implementation version 1.03 (see the bGetAdapterProgVers subroutine in the “Get Version of the Adapter Software” paragraph of this manual).

3.11 Error Handling and Error Codes

- If the return value is not explicitly described for the particular subroutine, the simple rule must be applied: if the subroutine succeeds the *bpvSuccess* variable is set to TRUE (1), if it fails the *bpvSuccess* is FALSE (0).

- Nearly all TBSockets subroutines specify the reason of failure in the *lLastErrorCode* variable in case of error. The following constants are used as error codes (the constants are defined in the ‘ts_com_d.inc’ file):

- *all subroutines*
  CME_TIMEOUT(254) - timeout error for any command

- *all SetUp subroutines*
  CME_SETUP_BAD_SUBCOMMAND (1) - not implemented subcommand

- *bSetupLocalPort*
  CME_SETUP_LP_INV_SOCK (16) - invalid socket for set local port
  CME_SETUP_LP_INV_PORT (17) - invalid port for set local port

- *bSetupDhcp*
  CME_SETUP_INV_DHCP_FLAG (3) - invalid DHCP flag value

- *bSetupDns*
  CME_SETUP_INV_DNS_FLAG (4) - invalid DNS flag value

- *bSetupIsp*
  CME_SETUP_MDM_DIAL_NO_MEM (6) - no memory to copy the modem dial string
  CME_SETUP_USER_NAME_NO_MEM (7) - no memory to copy the isp user name
  CME_SETUP_USER_PASSWORD_NO_MEM (8) - no memory to copy the isp user password

- *bSetPapSecrets*
  CME_SETUP_PAP_ID_TOO_LONG (9) - pap id too long
  CME_SETUP_PAP_PASSW_TOO_LONG (10) - pap password too long
  CME_SETUP_PAP_INV_INDEX (11) - invalid pap secrets index

- *bSetMacAddress*
  CME_SETUP_MAC_INV_SIZE (12) - size of mac address is not 6 bytes

- *bSetTcpWinSize*
  CME_SETUP_TCP_WIN_SIZE_INV_SOCK (13) - invalid socket for tcp window size
  CME_SETUP_TCP_WIN_SIZE_INV (14) - invalid size for tcp window

- *bSetTcpKeepAlive*
  CME_SETUP_TCP_KA_INV_SOCK (15) - invalid socket for tcp keep-alive

- *lGetLocalIp, wGetLocalPort, lGetRemoteIp, wGetRemotePort, lGetTcpSettings*
  CME_GET_PARAM_INV_SOCK (1) - invalid socket

- *bGetCtsPinState*
  CME_GET_PARAM_CTS_NOT_IN_USE (2) - cts pin is not used (handshake is off)

- *bGetMacAddress*
  CME_GET_PARAM_MAC_INV_BUF_SIZE (3) - not enough memory for mac string (not sent by Ethernet/Web Adapter)

- *bSendATCommand*
  CME_SEND_AT_NOT_AVAIL (1) - the command not available
  CME_SEND_AT_MDM_NOT_INIT (2) - the modem is not yet initialised
  CME_SEND_AT_MDM_SEND_NOT_READY (3) - the modem send buffer is full, and the timeout is expired
  CME_SEND_AT_MDM_RECV_NOT_READY (4) - the modem receive buffer is empty, and the timeout is expired

- *bModemSend, bModemReceive, wModemGetSendReady, wModemGetRecvReady*
  CME_MC_NOT_AVAIL (1) - the command not available
  CME_MC_BAD_SUBCOMMAND (2) - the subcommand not available
  CME_MC_MDM_NOT_INIT (3) - the modem is not yet initialised
  CME_MC_SEND_NOT_READY (4) - the modem send buffer is full,
  and the timeout is expired
  CME_MC_RECV_NOT_READY (5) - the modem receive buffer is empty,
  and the timeout is expired

- *bModemStartListen*
  CME_MC_NOT_ALL_SOCKS_FREE (6) - not all sockets are free,
  ergo cannot start listening to modem
  CME_MC_MIN_SIZE_TOO_BIG (7) - min size is bigger than max packet size
  CME_MC_ALREADY_LISTENING (8) - listening to modem already started

- *bModemStopListen*
  CME_MC_NO_LISTENING (9) - no procedure listening to modem

- *bDialIsp, bDialIspWithLogin*

CME_DIAL_NOT_AVAIL (1) - the command not available
CME_DIAL_BAD_SUBCOMMAND (2) - the subcommand not available
CME_DIAL_INV_TIMEOUT (3) - invalid timeout value
CME_DIAL_ALREADY_DIALED (4) - already connected
CME_DIAL_TIME_EXPIRED (5) - not connected: time expired
CME_DIAL_NO_DIAL_STRING (6) - no dial string entered
CME_DIAL_NO_USER_NAME (7) - user name not initialised
CME_DIAL_NO_USER_PASSWD (8) - user password not initialised

- *bHangUp*
  CME_HANGUP_NOT_AVAIL (1) - the command is not available

- *lDnsGetIpByName*
  CME_DNS_EFORMAT (1) - the server believes the request was improperly formated
  CME_DNS_ESERVER (2) - the DNS server encountered an internal failure
  CME_DNS_ENAME (3) - the requested name does not exist
  CME_DNS_ENOTIMP (4) - the name server does not support the requested kind of query
  CME_DNS_EREFUSED (5) - the name server refused the request
  CME_DNS_EYXDOMAIN (6) - some name that ought not to exist, does exist
  CME_DNS_EYXRRSET (7) - some RRset that ought not to exist, does exist
  CME_DNS_ENXRRSET (8) - some RRset that ought to exist, does not exist
  CME_DNS_ENOTAUTH (9) - the server is not authorative for the zone named in the Zone section
  CME_DNS_ENOTZONE (10) - a name used in the Prerequisites or Update Section is not withing the zone denoted by the Zone section
  CME_DNS_ETIMEOUT (33) - the request timed out
  CME_DNS_EBADANSWER (34) - the DNS subsystem was unable to understand the return request. The return request failed one of several internal validations
  CME_DNS_NOT_AVAIL (40) – dns command not available
  CME_DNS_SERV_NOT_KNOWN (41) - ip of dns server is not set
  CME_DNS_NO_MEM (42) - no memory to execute the command
  CME_DNS_NOT_READY (43) - timeout error
  CME_DNS_NAME_TOO_LONG (50) – the requested name too long

- *lSocket*
  CME_SOCKET_INV_AF (1) - invalid address format
  CME_SOCKET_INV_TYPE (2) - invalid type
  CME_SOCKET_NO_MEM (3) - no memory for new socket
  CME_SOCKET_INV_SOCK (4) - invalid socket returned by pair (not sent by Ethernet/Web Adapter)
  CME_SOCKET_SOCK_OCCUPIED (5) - socket is already occupied (not sent by Ethernet/Web Adapter)

- *bCloseSocket*
  CME_CLOSE_INV_SOCK (1) - invalid socket

- *bConnect*
  CME_CONNECT_INV_SOCK (1) - invalid socket
  CME_CONNECT_SOCK_BOUND (2) - socket is already bound
  CME_CONNECT_SIZE_FAULT (3)- incorrect size of SABlk
  CME_CONNECT_TIMED_OUT (9)- timed out
  CME_CONNECT_SOCK_BUSY (10)- socket is busy
  CME_CONNECT_NO_ROUTE (11)- source address is invalid

- *bBind*
  CME_BIND_INV_SOCK (1) - invalid socket
  CME_BIND_SOCK_BOUND (2) - socket is already bound
  CME_BIND_SIZE_FAULT (3) - incorrect size of SABlk
  CME_BIND_ADDR_IN_USE (4) - ip/port pair is in use
  CME_BIND_NET_DOWN (6) - net is down (udp only)

- *bListen*
  CME_LISTEN_INV_SOCK (1) - invalid socket
  CME_LISTEN_BACKLOG_OVER (2) - too many backlogs
  CME_LISTEN_SOCK_BUSY (4) - socket is busy

- lSend
  CME_SEND_INV_SOCK (1) - invalid socket
  CME_SEND_NOT_SENT (2) - send error
  CME_SEND_SOCK_NOT_CONNECTED (3) - socket is not connected
  CME_SEND_NO_MEM (4) - no memory to send data

- lSendTo
  CME_SEND_TO_INV_SOCK (1) - invalid socket
  CME_SEND_TO_NO_MEM (2) - no memory to send data
  CME_SEND_TO_SAB_SIZE_FAULT (3) - incorrect size of SABlk

- lSntpGetTime
  CME_SNTP_ERROR_BUSY (1) - there is already a time request in progress
  CME_SNTP_ERROR_NO_MEM (2) - couldn't allocate memory for the UDP connection
  CME_SNTP_ERROR_PORT_BUSY (3) - the NTP port number (123) is being used already
  CME_SNTP_ERROR_TIMEOUT (4) - timeout before server responded
  CME_SNTP_ERROR_ALARM (5) - the server returned an Alarm condition (the time is not available now)

Index