

sevenstax Transmission Control Protocol

Specification and User Manual

Revision No.: 4.20
State: Released
Author: sevenstax GmbH

Initial version: 21/08/02
Last change: 12/03/07
Last Review: 12/03/07
Publication: Public
Filename: sevenstaxTCP_UserManual_V42

Table of Contents

1 Preface.....	6
2 Supported Protocols	8
2.1 Protocols Overview.....	8
2.2 PPP support.....	8
2.3 Ethernet support.....	9
2.4 ICMP echoes.....	9
2.5 UDP support.....	9
2.6 DNS support.....	9
2.7 NTP support.....	9
3 Processing Received Packets.....	10
3.1 Processing received IP Packets.....	10
3.2 Processing Received TCP Packets.....	10
4 TCP Protocol Layer.....	12
4.1 Mode of Operation.....	12
4.2 TCP basics.....	12
4.3 Connection establishment.....	12
4.4 Receiving TCP/IP Packets.....	13
4.5 Transmitting TCP/IP Packets.....	13
4.6 Terminating a TCP connection.....	13
4.7 Server functionality.....	14
4.8 Single and Multiple Connections.....	14
5 Internal Data Structures.....	16
5.1 PACKET_DESC - the packet descriptor.....	16
5.2 IPV4 - the IP address.....	16
5.3 TCP_INFORMATION - the TCP information structure.....	17
6 Integration into User Application	18
6.1 Target Settings.....	18
6.1.1 <i>Packing and alignment - features.h</i>	19

6.1.2	<i>Sytem Time - features.h</i>	19
6.1.3	<i>Controlling sevenstaxTCP Features - features.h</i>	20
6.1.4	<i>Standard Lib Functions - runtimelib.c/h</i>	20
6.1.5	<i>Lower Layer API - tcpconnector.h</i>	20
6.2	NotifyHandler - Application Handler Procedure.....	22
6.2.1	<i>NotifyHandler Function Prototype</i>	22
6.2.2	<i>Notify Codes</i>	22
6.2.3	<i>Return Value</i>	22
6.2.4	<i>Additional lparam and wparam</i>	22
6.2.5	<i>Registering a NotifyHandler</i>	23
6.2.6	<i>Using the Notifyhandler</i>	23
6.3	Include sevenstax header files.....	24
7	IP Protocol API	25
7.1	Public Functions.....	25
7.1.1	<i>stxIP_Init()</i>	25
7.1.2	<i>stxIP_ProcessPacket()</i>	25
7.2	Notifycodes.....	25
7.3	Adjustable Parameters.....	25
8	TCP Protocol API	26
8.1	Public Functions.....	26
8.1.1	<i>stxTCP_Init (handler)</i>	26
8.1.2	<i>stxTCP_Connect (ip_address, port)</i>	26
8.1.3	<i>stxTCP_Disconnect (connection_id)</i>	27
8.1.4	<i>stxTCP_Listen (port)</i>	27
8.1.5	<i>stxTCP_WriteBulk (connection_id, data, len, bSendDynamic)</i>	28
8.1.6	<i>stxTCP_Tick (void)</i>	28
8.2	Notifycodes.....	29
8.2.1	<i>NC_TCP_CONNECTED (TCP-Info, ConnID)</i>	29
8.2.2	<i>NC_TCP_CONNTIMEOUT (TCP-Info, ConnID)</i>	29
8.2.3	<i>NC_TCP_FIN (AbortReason, ConnID)</i>	29
8.2.4	<i>NC_TCP_RECV (TCP-Info, ConnID)</i>	29
8.2.5	<i>NC_TCP_TXREADY (TCP-Info, ConnID)</i>	30
8.2.6	<i>NC_TCP_GET_TXDATA (TCP_TxData, ConnID)</i>	30
8.3	Adjustable Parameters.....	30
9	UDP Protocol API	33
9.1	Public Functions.....	33
9.1.1	<i>stxUDP_Init (handler)</i>	33
9.1.2	<i>stxUDP_Attach (local_port)</i>	33

9.1.3	<i>stxUDP_Detach (uConnId)</i>	34
9.1.4	<i>stxUDP_Write (uConnId, psRemotelp, uRemotePort, data, datalen)</i>	34
9.2	Notifycodes.....	35
9.2.1	<i>NC_UDP_RECV (UDP-Info, UDPConnID)</i>	35
9.3	Adjustable Parameters.....	35
10	DNS Protocol API	36
10.1	Public Functions.....	36
10.1.1	<i>stxDNS_Init (handler)</i>	36
10.1.2	<i>stxDNS_QueryStart (dns_ip, www_name)</i>	36
10.1.3	<i>stxDNS_QueryStop ()</i>	37
10.1.4	<i>stxDNS_Tick ()</i>	37
10.2	Notifycodes.....	37
10.2.1	<i>NC_DNS_IP</i>	37
10.2.2	<i>NC_DNS_TIMEOUT</i>	38
10.3	Adjustable Parameters.....	38
10.4	Using DNS (Domain Name Service).....	38
11	ICMP Protocol API	40
11.1	Public Functions.....	40
11.2	Notifycodes.....	40
11.3	Adjustable Parameters.....	40
12	NTP Protocol API	41
12.1	Public Functions.....	41
12.1.1	<i>stxNTP_Init (handler)</i>	41
12.1.2	<i>stxNTP_Query (IPV4 FPTR_stx ntp_server)</i>	41
12.1.3	<i>stxNTP_Transform (ntps, sec, min, hour, year, month, day, day_of_week)</i>	42
12.1.4	<i>stxNTP_Tick ()</i>	42
12.2	Notifycodes.....	42
12.2.1	<i>NC_NTP_SUCCESS</i>	42
12.2.2	<i>NC_NTP_FAILURE</i>	43
12.3	Adjustable Parameters.....	43
12.4	Using NTP.....	43
13	Debugging Support	44
14	Restrictions	45
15	Change History	46



1 Preface

The Internet is today's most disposable information source. On almost every place in the world (if there is a telephone jack), people using computers can access this vast reservoir of information. Normally, these computers are big, fast and powerful. They have large amount of memory, processors faster than light and bloated, full-featured operating systems. Because of those computing power, they easily deal with complex network protocols such as PPP and TCP/IP.

As a contrast to this, connecting smaller devices (such as embedded systems) to the Internet is somehow a tricky and difficult task. The biggest problem is to put such network protocol software into the limited memory space of small microcontrollers. But is there always a need to do so? The answer is: 'No, sometimes it's enough to implement only the basic parts of a protocol'. This is why sevenstaxTCP was made.

For the reason of smallness, the sevenstaxTCP has no additional protocols (by default) which usually belongs to a complete TCP/IP suite of protocols. For example: sevenstaxTCP cannot be 'pinged' because it doesn't know ICMP (but this can be enabled by conditional compilation). sevenstaxTCP relies on the other end's intelligent control algorithms to improve transmission speed (if it has). Therefore, receiving data with sevenstaxTCP is at an acceptable speed.

To show for what sevenstaxTCP is good for and when it's better to select another TCP, see the following comparisons:

The advantages of sevenstaxTCP are:

- No receive and transmit buffers. sevenstaxTCP shares buffers with the lower level network protocol (e.g. sevenstaxPPP or sevenstaxEthernet).
- Minimum ROM and RAM usage. Code size differs from 5K to 9K (CISC) depending on the processor where it is executed. This makes it an excellent choice for small embedded systems.
- Pure TCP/IP. No additional protocols such as ICMP, DHCP, UDP etc. are implemented to save memory (But some of them can be enabled by conditional compilation - see the chapter 'Currently Supported Add-ons').
- Sliding windows support on TCP transmission to maximise send performance.
- Easy integration with existing software. Only few interfaces.
- Asynchronous behaviour. Other software will not be delayed by sevenstaxTCP.
- Trouble-free operation under an RTOS environments like CMX and embOS.
- Shipped as well-documented source code (ANSI-C) which makes it easy to modify.

The disadvantages of sevenstaxTCP are:

- Does not follow RFC in all cases, but fulfils most MUST criteria.
- No dedicated reception buffer. So no TCP fragmentation is supported.

You should give sevenstaxTCP a trial when you...

- ...want to bring the Internet to small embedded devices which suffer from less memory, both RAM & ROM.
- ...want to create HTML browsers or other TCP clients for systems without TCP/IP stack.
- ...want to build devices which are configurable over a TCP/IP network (printers etc.).

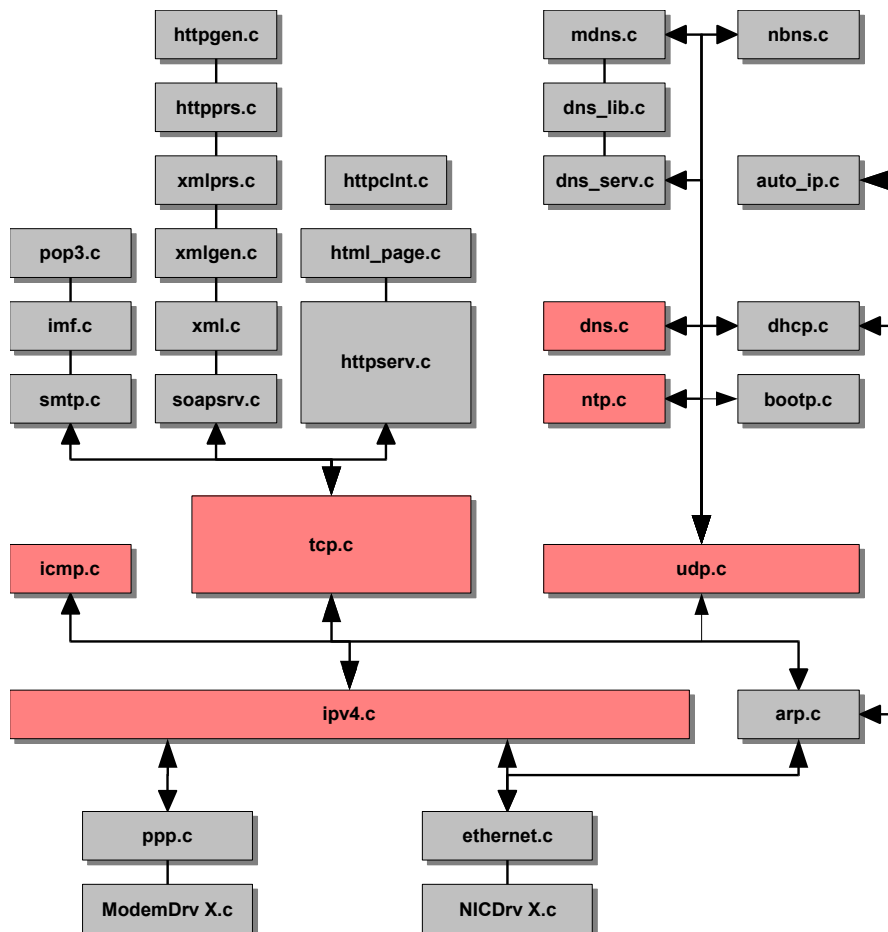
2 Supported Protocols

This manual deals with the main sevenstax internet suite product - sevenstaxTCP. sevenstaxTCP includes the protocols IP, TCP, ICMP, UDP, NTP and a simple DNS client. sevenstaxTCP in this context is used as a general protocol suite name.

Please note, that sevenstaxTCP is only able to communicate via exactly one physical layer, e.g. PPP via modem or Ethernet via a dedicated Controller. If your application has the need to handle multiple physical interface at the same time, like multiple modems, multiples EthernetControllers or both, please use the special edition of **sevenstaxTCP Multidevice**. This is separate sevenstax product.

2.1 Protocols Overview

The following graphic shows the integration of sevenstaxTCP (red marked) inside the other optional sevenstax internet libraries, available as separate products.



2.2 PPP support

sevenstaxPPP is available as a separate product.

sevenstaxPPP has been approved in many products and countries worldwide in reliable connections to common public Internet Service Providers (ISPs). sevenstaxTCP connects to PPP, if `PPP_SUPPORTED = 1` is defined in `features.h`.

2.3 Ethernet support

sevenstaxEthernet is available as a separate product.

sevenstaxEthernet offers a very tiny and effective protocol implementation for LAN connections, providing implicitly support for ARP and Network Interface Card (NIC) usage. sevenstaxTCP connects to Ethernet, if `ETHERNET_SUPPORTED = 1` is defined in `features.h`.

2.4 UDP support

UDP is included in sevenstaxTCP (file `udp.c`).

UDP is the abbreviation for User Datagram Protocol, an unreliable packet delivery service, which is part of the TCP/IP suite of protocols. If you need UDP please `#define UDP_SUPPORTED = 1` in `features.h`.

See section 9, UDP Protocol API for details.

2.5 DNS support

DNS is included in sevenstaxTCP (file `dns.c`).

A simple DNS client can be used by the application to send a query to a DNS server and extract the returned IP address of the response. If you need DNS, please `#define DNS_SUPPORTED = 1` in `features.h`.

See section 10, DNS Protocol API for details.

If you additional need DNS server functionality, please use a separate product called *sevenstaxZeroConf*, which includes a DNS server module and a separate DNS library.

2.6 ICMP echoes

ICMP is included in sevenstaxTCP (file `icmp.c`).

Sometimes it may be useful that a TCP responds to ICMP echo requests (also known as 'PING'). The ICMP support of sevenstaxTCP is very simple and small in code size therefore enabling it will not waste much memory. sevenstaxTCP can be extended to reply to such a request by defining `ICMP_SUPPORTED = 1` in `features.h`.

2.7 NTP support

NTP is included in sevenstaxTCP (file `ntp.c`).

A simple NTP client can be used by the application to send a query to a time server and extract the returned time of the response. If you need NTP, please `#define NTP_SUPPORTED = 1` in `features.h`. See section 12, NTP Protocol API for details.

3 Processing Received Packets

The nearly most important attribute of a TCP/IP stack is how it handles received packets. In fact, sevenstaxTCP handles received packets in a different way than most other TCPs. The sequence in which packets are processed is shown here:

3.1 Processing received IP Packets

The underlying protocol layer (PPP or Ethernet) calls *stxIP_ProcessPacket()*.

1. First protocol and port number are validated. If they don't match, the packet is rejected.
2. Then the IP address is checked in relation to the current network configuration of the local IP address and the subnet mask. Multicast and global and local Broadcast addresses areas are checked too.
3. If the packets protocol is UDP and `UDP_SUPPORTED` is #defined, the packet is forwarded to *stxUDP_ProcessPacket()*.
4. If the the packets protocol is ICMP and `ICMP_SUPPORTED` is #defined, the packet goes to *stxICMP_ProcessPacket()*.
5. If the protocol is TCP, the packet goes to *stxTCP_ProcessPacket()*.

In any other case, the packet is rejected.

3.2 Processing Received TCP Packets

1. **Please Note:** sevenstaxTCP's SYN_REVEIVED state has a slightly different meaning than those described in RFC 793.
2. Sequence- and acknowledge number from the packet are stored in global variables.
3. If there is a pending transmission, the received sequence number will be compared with the one expected in order to cancel retransmissions and to free the retransmission buffer so as new data can be send.
4. If a connection is established, a time-out value to cancel a dead connection (a connection where no data has been received for a long time) will be checked first.
5. If a connection is established, the received acknowledge number will be compared with the calculated one. This means we have received a packet in the right order which will then be delivered to the upper layer (if it carries data). Also the acknowledge value for the next packet will be calculated here.
6. If a connection is established, and the received acknowledge number does not match the expected one, the global acknowledge number will be set to the one awaited and an ACK segment with this number is send to the server to force a retransmission. At this point, the handling of this packet will be aborted.
7. If sevenstaxTCP receives a SYNACK (after a SYN has been sent), an ACK will be returned, the alive timer initialized, the first acknowledge number calculated and the application notified that we the connection is established and data may be transferred. At this point no further packet will be handled.

8. If sevenstaxTCP receives a FINACK (e.g. the server wants to close the connection) an ACK is returned and the connection aborted. Furthermore the application will be notified of the connection's termination. No further packet handling will be executed.
9. If sevenstaxTCP receives a RST (e.g. the server has detected a TCP error) the connection will be aborted and the application notified of this event. No further packet handling will be done.
10. In case an ACK is received while a connection is established, a data packet must be acknowledged. sevenstaxTCP does this and leaves the receive function.
11. In case an ACK is received while no connection is established this may indicate that a half-open connection exists. sevenstaxTCP will send an RST which will close the old connection enabling a new one to be established on the same port.

4 TCP Protocol Layer

4.1 Mode of Operation

This section describes the way sevenstaxTCP internally works with TCP packets.

4.2 TCP basics

TCP is a connection-oriented protocol. This means that TCP has to compensate these deranging side effects of the underlying network:

- Packet loss
- Packet duplication
- Out-of-order data delivery

To perform this tasks, some kind of acknowledging and confirmation must be used. For this purpose, TCP packets carry additional information such as sequence/acknowledge numbers and bit flags. Before two TCPs can exchange data, they must synchronize (connect) to each other. This is done by both stations, exchanging a special sequence of datagrams where they tell their initial sequence numbers. During operation, each TCP knows what the other end awaits and tries to deliver the packets that the other TCP has 'ordered'. Also it manages a buffer with already transmitted data to repeat transmission if the recipient hasn't acknowledged the packets. After both TCPs have finished, the connection must be closed. This can be done either by exchanging datagrams similar to the connection process or (faster) by sending a 'connection reset' from any station. Of course, this is a very much incomplete description of TCP/IP (an TCP/IP tutorial is beyond the scope of this document) - but it's nearly all what sevenstaxTCP does.

4.3 Connection establishment

For the purpose of connection establishment, sevenstaxTCP has two interfaces named *stxTCP_Connect()* and *stxTCP_Listen()* which take the remote IP address and port number (how a connection is established is shown in RFC 793, figure 7).

NOTE: For *stxTCP_Connect()*, the local port numbers of sevenstaxTCP are pseudo-random values.

sevenstaxTCP tries to send some SYNs to the other end until it receives with a SYNACK. If, for any reason, no SYNACK arrives, sevenstaxTCP notifies the application (through the handler procedure described below) that a connection isn't possible.

When sevenstaxTCP runs as a TCP server (like a Webserver), it is also able to respond to connection attempts issued from remote TCPs. Instead of calling *stxTCP_Connect()*, the application should call *stxTCP_Listen()* to bring sevenstaxTCP into the listen state. For further information refer to the chapter 'Currently Supported Add-ons' (Server functionality).

Also sevenstaxTCP has a built-in mechanism to reuse half-open connections. After a possibly system crash, the remote server believes the connection is already open and thus, connecting to the same port will fail. In this case, sevenstaxTCP sends a RST

(TCP reset) to the server, using data the server has sent as a reaction of sevenstaxTCP's connection attempt. The RST cancels the old connection and a subsequent connection request will succeed.

4.4 Receiving TCP/IP Packets

If the lower network layer (e.g. sevenstaxEthernet) has recognized a TCP packet, it calls sevenstaxTCP's receive function *stxIP_ProcessPacket()* and afterwards *stxTCP_ProcessPacket()*. Before calling this function, the lower network layer prepares a packet descriptor (an internal data structure described below) which holds information about the TCP packet.

Usually, the packet resides in the receive buffer of the lower network layer thus sevenstaxTCP doesn't need a buffer itself. The *stxTCP_ProcessPacket()* function does everything necessary on reception including data delivery to the application, handling connection requests and transmits response packets back to the other end.

4.5 Transmitting TCP/IP Packets

To transmit TCP packets, sevenstaxTCP needs to know two things from the lower network layer (e.g. sevenstaxEthernet). First it needs the address of the transmit buffer (a packet descriptor) - sevenstaxTCP must know where to assemble a TCP packet.

Next, sevenstaxTCP needs to know the transmit function of the lower layer. This function takes the packet descriptor and forwards the packet to the network. This function might be an Ethernet or PPP send function.

Immediately after sevenstaxTCP has built the packet in memory, it prepares the packet descriptor and calls the transmit function to send out the packet.

Inside sevenstaxTCP there's a function named *stxTCP_InternalSend()* which does all the above described functions.

4.6 Terminating a TCP connection

Usually, a web server terminates the TCP connection after it has transferred the last byte of the page. When the server wants to close the connection, it sends a FINACK and enters the FIN_WAIT_1 state (see RFC 793, figure 13).

Users of sevenstaxTCP must call the function *stxTCP_Disconnect()* which initiates the closing process. When the connection is closed, sevenstaxTCP calls the application's handler procedure by posting a NC_TCP_FIN event.

There are some other conditions, when sevenstaxTCP issues NC_TCP_FIN as well. For example: If sevenstaxTCP sends data (e.g. an HTTP request) but the packet wasn't acknowledged after a few retries (ACK time-out), the TCP connection will also be terminated. To separate regular terminations from error conditions, the handler procedure is called with different parameters. For a detailed description see the following table (the constants are defined in tcp.h, typedef enum ABORT_REASON):

Cause of termination	First parameter of handler procedure (ABORT_REASON)	Meaning
A successful call to <code>stxTCP_Disconnect()</code> or the remote TCP has closed the connection.	AR_CLOSED	Normal operation. All data has been successfully transferred.
1. <code>stxTCP_Disconnect()</code> was called and <code>sevenstaxTCP</code> was in transition state (neither in the connected nor in the closed state). 2. The other end has send an RST. 3. Timeout. The remote TCP does not respond to our FINACK's. or FIN's	AR_CLOSE_FORCED	Error. Some data might be received.
Transmitted data was not acknowledged after some retransmissions.	AR_RTM_EXCEEDED	Error. Too many retransmissions.
ARP wasn't able to resolve the other side's MAC address. (Ethernet only)	AR_NO_MACADDRESS	Error. TCP can't continue. Maybe peer does not exist on the LAN.
not yet supported	AR_NO_NETWORK	-
Application actively terminated the connection.	AR_USER_1 AR_USER_2 AR_USER_3	Normal operation. The TCP connection was cancelled.

4.7 Server functionality

`sevenstaxTCP` is capable to act as a server TCP. This means it can passively wait for connections from other TCP's and send requested data or error messages to them. To enable the server functionality the following steps are required:

- After initialization of `sevenstaxTCP` call `stxTCP_Listen()`. This causes `sevenstaxTCP` to listen on a specified port for incoming connections.
- After being connected, which will be indicated by the received `NOTIFY_CODE_NC_TCP_CONNECTED` call `stxTCP_WriteBulk()` to send data of nearly any size to the remote TCP.
- When the entire data block was successfully transmitted, more data may be sent or the connection may be closed by calling `stxTCP_Disconnect()` for a graceful disconnect.

4.8 Single and Multiple Connections

In cases, where memory is a very limited resource, `sevenstaxTCP` can in *SingleConnectionMode*. This is useful for very small embedded devices with the need of only one TCP connection at a time. Set `TCP_MULTI = 0` to enable this mode.

Typically more than one simultaneous connection must exist (e.g. HTTP browser request). `sevenstaxTCP` can be configured to work with up to 65535 connections at a time. Multi connection support can be switched on at compile time by #defining `TCP_MULTI` and `TCP_MULTI_TICKS`.

With `TCP_MULTI` one can specify the maximum number of connections which `sevenstaxTCP` can handle. This should be well adjusted to save memory because for every connection about 100 bytes RAM are reserved.

TCP_MULTI_TICKS is the number of active connections which are handled by a single call to *stxTCP_Tick()*. This value also depends a little how frequent *stxTCP_Tick()* is called. For example: If you configure sevenstaxTCP to work with 100 possible connections, *TCP_MULTI_TICKS* can be set to 10.

Both #defines can be found in '*tcpdefs.h*'.

Most of sevenstaxTCP's interfaces have an additional parameter called 'Connection-ID'. Connection ID's are unique values which help to distinguish between connections. They are unique during the lifetime of a connection. Connection IDs are automatically recycled (newly assigned) when a connection ends.

Please Note: *TCP_MULTI_TICKS* does not affect incoming TCP data. Received packets are handled immediately, whenever the underlying protocol layer (PPP or Ethernet) calls *stxIP_ProcessPacket()*.

5 Internal Data Structures

5.1 PACKET_DESC - the packet descriptor

```
typedef struct _tag_PACKET_DESC
{
    PACKET_STATUS      state;
    UINT16_stx        size;
    UINT8_stx FPTR_stx data;
    UINT8_stx FPTR_stx payload;
} PACKET_DESC;
```

The packet descriptor is used to hold information about a data packet and for easy passing of this information between internal functions by reference.

PACKET_STATUS state

indicates whether the packet is in use (i.e. under construction) or ready for further processing. It is for internal use only, must not be initialized or changed by the application or lower network components.

FPTR_stx data

points to the location of the first byte in memory. This is the place where the packet's data is stored including link layer headers.

UINT16_stx size

specifies the length in bytes of the entire range pointed to by 'FPTR_stx data'.

FPTR_stx payload

points to the first byte of the IP header. This is usually the first byte after the end of the link layer header.

Packet descriptors are also used by sevenstaxPPP and sevenstaxEthernet to deliver packets to higher network protocols (such as TCP).

5.2 IPV4 - the IP address

```
typedef union _tag_IPV4
{
    struct
    {
        UINT8_stx b1;
        UINT8_stx b2;
        UINT8_stx b3;
        UINT8_stx b4;
    } ipb;
    UINT32_stx ipl;
} IPV4;
```

An IP address is actually a 32 bit value which is often shown as 4 decimal values, separated by dots, for better readability (e.g. 192.168.0.51).

IP addresses are stored in network byte order where the first byte (ipb.b1) is the most significant byte. The IPV4 union allows either access to the entire value (ipl) or partly through ipb.b1 to ipb.b4.

Please Note: IP addresses are internally stored as they came over the network (network byte order), regardless of the processor architecture sevenstaxTCP runs on.

5.3 TCP_INFORMATION - the TCP information structure

This structure is mainly used to pass all important properties regarding to a TCP connection. E.g. when sevenstaxTCP has posted an *NC_TCP_CONNECTED* event to the application's handler routine, some members (remote_ip_address, remote_port and local_port) contain values which describe the connection.

```
typedef struct _tag_TCP_INFORMATION
{
    UINT8_stx    FPTR_stx data;        /* Points to received data */
    IPV4         remote_ip_address;    /* IP address of peer */
    IPV4         dest_ip_address;     /* our IP / or broadcast / multicast */
    UINT16_stx  datasize;             /* Length of received data */
    UINT16_stx  remote_port;         /* Port number of peer */
    UINT16_stx  local_port;          /* Our port number */
} TCP_INFORMATION;
```

UINT8_stx FPTR_stx data

Points to received data. The application must only use this pointer and the data referenced to as long as the called function (Notifyhandler) is active.

IPV4 remote_ip_address

Remote IP address (sender), might be Unicast, Multicast or Broadcast.

IPV4 dest_ip_address

Destination IP address (receiver), might be Unicast, Multicast or Broadcast.

UINT16_stx datasize;

Amount of data in bytes, *data* points to.

UINT16_stx remote_port;

Remotes IP-Port number (senders port).

UINT16_stx local_port

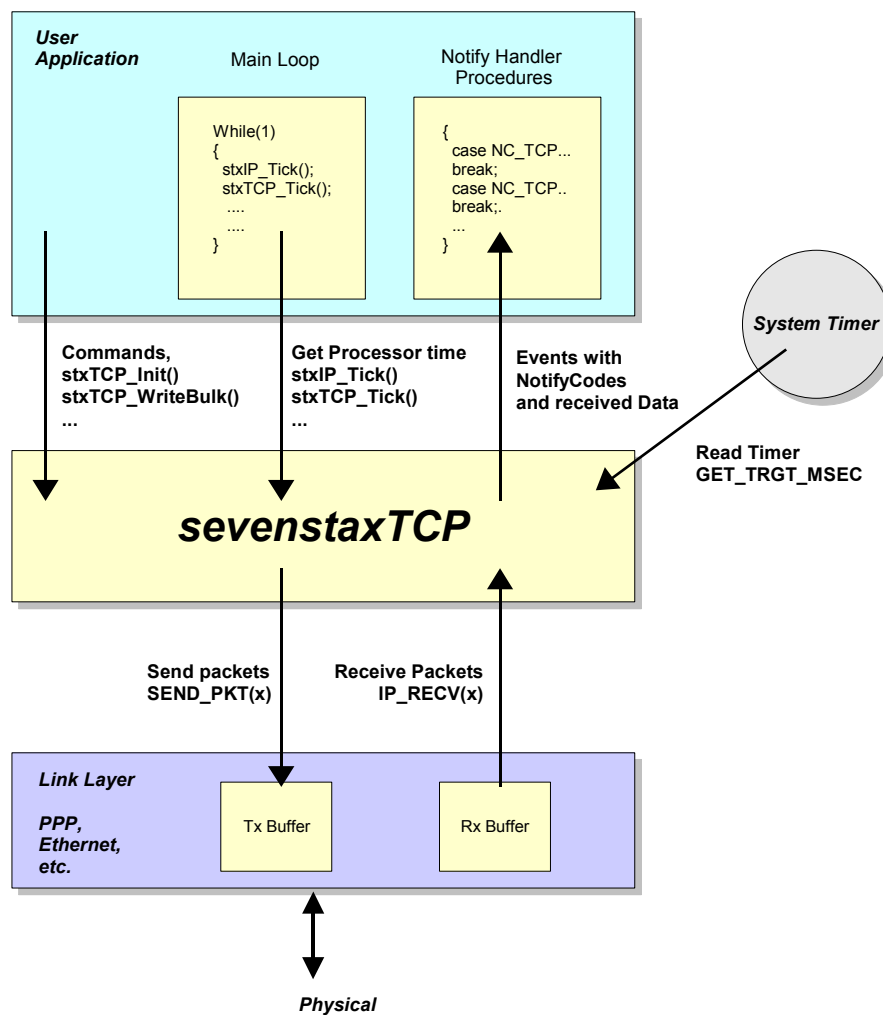
Destinations IP-Port number (receivers port)

If data arrives on a specific port, sevenstaxTCP issues a *NC_TCP_RECV* event with an additional pointer to a *TCP_INFORMATION* structure where (beside the other members) *FPTR_stx data* and *datasize* describe a newly received packet.

6 Integration into User Application

sevenstaxTCP is designed to work in conjunction with different data link layers such as PPP or Ethernet. There are some function prototypes and definitions which must be adjusted according to your application.

Please note: Various functions deal with packet descriptors (see section 5, Internal Data Structures for details). The link layer has to provide the required packet descriptors.



6.1 Target Settings

The main target configuration file is 'features.h'. This file is included into all sevenstax modules to control sevenstaxTCP's features.

sevenstaxTCP is implemented to run on every controller, independently of 8/16/32 bit, independently of endianness and compiler behaviour. Much effort has been done to support this universal behaviour on most target systems.

Per default, it is prepared and intended to generate a compiler error of this kind: *“features.h’ not yet adapted to your system! Please check settings in ‘features.h!’”*.

The following settings are to be checked or changed for your target system.

6.1.1 Packing and alignment - features.h

Packing and alignment are essential settings for a target CPU dealing with multibyte data like IP addresses. The following settings take effect in the “wordaccess.h”.

```
/* target specific settings for byte/struct handling */
/* Please adapt to your target system */
//#pragma BYTEALIGN      /* enable this, if compiler/processor supports byte alignment by
                        default */
#define BYTEALIGNED      0 /* set to '1', if your target cpu supports byte alignment */
#define LITTLE_ENDIAN    1 /* set to '1', if your target cpu stores multibyte values in
                        network order */
#define GCC_ALIGNED      /* set to '__attribute__((aligned))', if gcc compiler used */
#define GCC_PACK         /* set to '__attribute__((packed))', if gcc compiler used */
#define ARM_PACK         /* set to 'packed', if ARM compiler used */
#define PACK_SUPPORTED   1 /* set to '1', if compiler supports '#pragma pack()' */
```

Please note, that these packing settings are used in the files “stx_pack.h” and “stx_unpack.h”. These files are included before and after every structure, which needs special packing handling. You can adapt these files to your application, f.e. if packing should not be disabled for your application.

6.1.2 Sytem Time - features.h

sevenstaxTCP needs a system time to execute the protocols timing behaviour. A simple counter delivering milliseconds must be prepared by the user application.

Dedicated macros are prepared to easily integrate the user applications time, e.g. as direct variable access or by a function returning that time. The easiest way is to simply implement a target cpu timer, generating an interrupt every 1 ms and than incrementing the given counters variables below.

```

/* time base for stacks (target system specific) */
/* Please insert your application variables/functions here! */

extern unsigned long   gdwMilliSecCounter;
extern unsigned long   gdwSecCounter;
extern unsigned short  gwMilliSecCounter;
extern unsigned short  gwSecCounter;

#define GET_TRGT_MSEC      gdwMilliSecCounter /* 32-bit-value, supported e.g. by ISR */
#define GET_TRGT_MSEC_SHORT gwMilliSecCounter /* 16-bit-value */
#define GET_TRGT_SEC      gdwSecCounter /* 32-bit-value, supported ISR */
#define GET_TRGT_SEC_SHORT gwSecCounter /* 16-bit-value */

```

6.1.3 Controlling sevenstaxTCP Features - features.h

This file controls the features and therefore the resource requirements of sevenstaxTCP. You can enable/disable a complete protocol support by simply writing a 1 or 0 for the dedicated define. Disabled features wont be compiled and linked.

Please do not comment out unused features, use 0 or 1 only.

```

/* protocols */
#define TCP_SUPPORTED      1 /* Enables TCP */
#define UDP_SUPPORTED     1 /* Enables UDP */
#define DNS_SUPPORTED     0 /* Enables simple DNS client */
#define DNS_SRV_SUPPORTED 1 /* Enables enhanced DNS server and client */
#define NTP_SUPPORTED     1 /* Enables NTP (Network Time Protocol) */
#define ICMP_SUPPORTED    1 /* Enables ICMP-Ping responses */
#define HTTPSERV_SUPPORTED 1 /* Enables HTTP server */

/* serial */
#define PPP_SUPPORTED     0 /* Enables PPP */
#define MODEM_SUPPORTED  0 /* Enables Modem driver */
#define NULLMODEM_SUPPORTED 0 /* Enables Nullmodem driver */

/* ethernet */
#define ETHERNET_SUPPORTED 1 /* Enabled Ethernet */
#define MDNS_SUPPORTED    1 /* Enables Multicast-DNS */
#define BOOTP_SUPPORTED   0 /* Enables BOOTP */
#define DHCP_SUPPORTED    1 /* Enables DHCP */
#define AUTOIP_SUPPORTED  1 /* Enables Auto-IP */
#define NBNS_SUPPORTED    1 /* Enables NBNS (NetBIOS Name Service) */

```

Please note, that some of the features will only be available, if the equivalent sevenstax product was bought.

6.1.4 Standard Lib Functions - *runtime.lib.c/h*

sevenstaxTCP uses some standard lib functions for buffer or string manipulation. Several simple embedded compilers do not support a complete implementation of the ANSI-C standard libs. Or your application might use your own memory- or runtime-optimized implementation of the standard lib functions.

To support this, sevenstaxTCP comes with it's own simple runtime lib in "*runtime.lib.c*" and "*runtime.lib.h*". Please feel free, to adapt it to your requirements.

6.1.5 Lower Layer API - *tcpconnector.h*

sevenstaxTCP underlying protocols might be PPP or Ethernet. The essential interfaces to handle sending and receiving packets via the sub layer are already implemented in "*tcpconnector.h*". No changes are to be made to support this!

But feel free to adapt this interface to your application, if you use another sub layer protocol.

#define IP_RECV(x) stxIP_ProcessPacket(x)

Receive function of sevenstaxTCP.

Whenever the link layer (e.g. PPP or Ethernet) has detected a TCP packet, it must be delivered to sevenstaxTCP. Therefore, the link layer has to know the entry point into sevenstaxTCP where received packets are handled. For this reason sevenstaxTCPs internal function *stxIP_ProcessPacket()* is made public by the macro *IP_RECV*. The supplied packet descriptor must have the following settings when entering *TCP_RECV* (see section 5.1, *PACKET_DESC* - the packet descriptor for details):

- *state* Not used (any value)
- *size* Size of the entire packet (including link layer headers).
- *data* A pointer to the first byte of the link layer header.
- *payload* A pointer to the first byte of the IP header.

#define SEND_PKT(x) stxETH_SendIpFrame(x)
#define SEND_PKT(x) stxPPP_SendIpFrame(x)

One of these functions is the entry point of the link layer, which is responsible for handling data that should be sent off to the network. sevenstaxTCP requires a function which takes a pointer to a packet descriptor as a single parameter. When sevenstaxTCP calls *SEND_PKT* it has done these things before:

- It has build a complete TCP/IP packet beginning at the location where the packet descriptor's member '*FPTR_stx payload*' points to.
- It has set the packet descriptor's '*size*' member to the size of the TCP/IP packet.

The link layer might then do the following:

- Increase the '*size*' member to the length of its own headers.
- Build its headers at the location where '*FPTR_stx data*' points.
- Deliver the packet to the network hardware.

#define SENDBUFF gETH_sendpacket
#define SENDBUFF gPPP_sendpacket

sevenstaxTCP is designed to reuse the send buffer of the link layer (e.g. sevenstaxPPP) in order to save memory. Therefore *SENDBUFF* is by default set to one of the above buffers.

SENDBUFF is a packet descriptor structure (see section 5.1, *PACKET_DESC* - the packet descriptor for details) and must be initialised the following way:

- The '*FPTR_stx data*' member must point to first byte of the link layer header. This might be the first byte of the buffer which the link layer will send off to the network.
- The '*FPTR_stx payload*' member must point to the first byte of the IP header. This should be, for example, fourteen bytes behind '*FPTR_stx data*' when using sevenstaxTCP on Ethernet.
- '*size*' and '*state*' members are not used at this point and can have any value.

The buffer can be changed according to the projects needs. Therefor the *SENDBUFF* may be changed to a different packet descriptor but this is normally not required.

6.2 NotifyHandler - Application Handler Procedure

To signal events to the application, sevenstaxTCP needs a handler procedure provided by the application, a so called callback function. Whenever sevenstaxTCP has something to report, it calls this handler procedure.

6.2.1 NotifyHandler Function Prototype

The function prototype of this universal notify handler is defined in '*protdefs.h*' and looks like this:

```
typedef UINT32_stx FCTPTR_stx (PROTOCOL_NOTIFY_HANDLER)
    (NOTIFY_CODE code, UINT32_stx lparam, UINT16_stx wparam);
```

Please note: Check your compiler for declaration format of function pointers. The notify handler needs to be callable from everywhere within the stack!

6.2.2 Notify Codes

The first parameter of the Notifyhandler is the *NOTIFY_CODE*. This is actually an identifier of the message which sevenstaxTCP has sent. The sevenstaxTCP modules use different notifycodes areas, defined in '*protdefs.h*'.

```
...
#define NC_TCP_MIN                (NOTIFY_CODE) 400
#define NC_TCP_MAX                (NOTIFY_CODE) 419
...
#define NC_UDP_MIN                (NOTIFY_CODE) 420
#define NC_UDP_MAX                (NOTIFY_CODE) 439
...
```

The exact definition of a notifycode is defined in the dedicated modules export header, like "tcp.h":

```
...
#define NC_TCP_CONNECTED          (NOTIFY_CODE) NC_TCP_MIN+0
#define NC_TCP_CONNTIMEOUT        (NOTIFY_CODE) NC_TCP_MIN+1
#define NC_TCP_FIN                (NOTIFY_CODE) NC_TCP_MIN+2
...
```

You will find a separate list of notifycode definitions in the protocol API documentation (e.g. in section 8.2, TCP-Notifycodes).

6.2.3 Return Value

Per default, sevenstaxTCP does not use the return value of the applications notifyhandler function. But there are rare case, in which this value is essential, e.g. in conjunction with *NC_TCP_GET_TXDATA* and *NC_HTTP_DYNINFO*.

For details, please take a closer look into API documentation of the protocols.

6.2.4 Additional lparam and wparam

The next two parameters may carry additional information depending on the message. Please see the adequate header file for details.

6.2.5 Registering a NotifyHandler

By calling each `stxXXX_Init()` function with appropriate parameter, the application registers such a handler procedure at `sevenstaxTCP`.

The application might use always the same handler procedure for all the protocols. In such a case, the notifyhandler might be a very big switch/case construction. Another way is to register dedicated notifyhandler procedure, so that each handler already has a dedicated responsibility, which might reduce complexity of source code.

In reality, it has been an advantage to almost distinguish between a common notifyhandler for `sevenstaxTCP` and a special one for the `sevenstaxWebserver`.

6.2.6 Using the Notifyhandler

There are some general rules for executing code inside the notify handler procedure:

Leave as fast as possible

When `sevenstaxTCP` signals events, it cannot proceed before the handler procedure has exited. If you do lengthy tasks inside the handler procedure, the receive buffers may overrun, packets get lost and must be repeated. This will cause appreciable delays and might affect performance of the whole system. So be sure not to waste much time here.

Never hold given pointers

The optional given `w-` and `l-` parameters of the notify handler often are pointers to current connection information and/or user data. They are very connection and TCP process state specific, so they are valid only as long as the notify handler is being called and active.

After leaving/returning from the notify handler you **MUST NOT USE** these parameters (pointers) and the referenced data any more, because they will be changed in further TCP Stack internal action.

To use the data referenced by `w-` and `l-`parameter, you might process them immediately (but in very short time) or you might save the content (not the pointer!) into your application.

Never immediately call back into sevenstaxTCP!!!

Because the handler procedure is called from inside `sevenstaxTCP`, it is dangerous to immediately call back into `sevenstaxTCP`, by using `sevenstaxTCP` API functions. Doing so can cause deadlocks or even unlimited recursions which will result in stack overflows and thus crash the system or simply unpredictable reactions and data.

If you like to do some TCP reactions (write/open/close etc.) due to incoming notify codes, **execute them AFTER `stxXXX_Tick()` functions**. This will prevent any interference between you application and the TCP stack

Do not write to RX/TX buffers

sevenstaxTCP might be in a state where it waits for some data or builds a packet for transmission. Therefore, changing buffer contents can cause sevenstaxTCP to misbehave.

6.3 Include sevenstax header files

The sevenstax library code contains a lot of header files (*.h) to define e.g. stack usage, parameters and special prototypes. The sevenstax libraries internally handle all necessary include files for them self. To reduce complexity, sevenstax decided not to include header files into header files.

The user application might use some or all of the sevenstax internet suite features. To help the developer to find the correct header files and to include them in an order corresponding to the dependencies, sevenstax here gives you a standard dependency list, which might be used in the user application:

```
// please include these headers in given order,
// delete what you don't need
#include "features.h"
#include "stxtypes.h"
#include "xtoa.h"
#include "protdefs.h"
#include "ipv4.h"
#include "pktdesc.h"
#if (ETHERNET_SUPPORTED==1)
    #include "ethernet.h"
#endif
#if (PPP_SUPPORTED==1)
    #include "ppp.h"
#endif
#include "tcpconnector.h"
#include "debugstx.h"
#include "tcp.h"
#if (UDP_SUPPORTED==1)
    #include "udp.h"
#endif
#include "httpserv.h"
#include "runtimelib.h"
#include "arp.h"
#include "nic.h"
#if (DHCP_SUPPORTED==1)
    #include "dhcp.h"
#endif
#if (AUTOIP_SUPPORTED==1)
    #include "auto_ip.h"
#endif
#include "dns.h"
#if (DNS_CACHE>0)
    #include "dns_lib.h"
#endif
#if (MDNS_SUPPORTED==1)
    #include "mdns.h"
#endif
#if (NBNS_SUPPORTED==1)
    #include "nbns.h"
#endif
#if (NTP_SUPPORTED==1)
    #include "ntp.h"
#endif
```

7 IP Protocol API

Inside sevenstaxTCP the IP protocol layer is a very narrow layer, mainly to distinguish the protocol of received packets and to call higher layer protocols “*stxXXX_Processpacket()*” functions.

7.1 Public Functions

7.1.1 *stxIP_Init()*

Description

Initializes the local IP address. Must be called before using IP.

Parameter

PROTOCOL_NOTIFY_HANDLER handler
For future use, set to NULL.

Return Value

void

Comment

This must be called before IP can be used.

7.1.2 *stxIP_ProcessPacket()*

Description

Handles received packets according to their type etc. Must be called by sublayer protocol (PPP / Ethernet) for each received packet.

Parameter

CONST_stx PACKET_DESC FPTR_stx pkt
A pointer to the received packet descriptor

Return Value

BOOL_stx
TRUE_stx, if packet is processed, *FALSE_stx*, if packet is ignored

Comment

Does everything we need to satisfy TCP requests.

7.2 Notifycodes

Although, sevenstaxTCP supports a dedicated notifyhandler for the IPV4, currently no notifycodes are defined for the IP layer.

7.3 Adjustable Parameters

Currently no adjustable parameters defined.

8 TCP Protocol API

Usage and behaviour of sevenstaxTCP is discussed in the chapters above. Please see section 4, TCP Protocol Layer for details.

8.1 Public Functions

8.1.1 *stxTCP_Init (handler)*

Description

Before sevenstaxTCP can be used, this function must be called. It initializes all internal states of sevenstaxTCP and also registers the application-defined handler procedure.

Parameter

PROTOCOL_NOTIFY_HANDLER handler

Application handler function to receive notify codes from sevenstaxTCP.

Return Value

void

Comment

Please Note: It is advisable to initialize the link layer and set up the receive and transmit packet descriptors *before* calling *stxTCP_Init()*.

8.1.2 *stxTCP_Connect (ip_address, port)*

Description

Tries to establish a connection to another TCP.

Parameter

CONST_stx_IPV4_FPTR_stx ip_address

A pointer to an IPV4 structure which must be initialized to the IP address of the remote peer.

UINT16_stx port

The port number of the remote TCP on which the connection should be established. Ports are something like channels dedicated to particular services. E.g. HTTP resides on port 80 (decimal).

Return Value

UINT16_stx port

The sevenstaxTCP internal ConnectionID, or an error value (*INVALID_CONNECTION_ID*).

Subsequent calls into sevenstaxTCP must use this value to identify the connection. Also events, which the application receives by its handler procedure, carry connection IDs.

Comment

Upon a call to *stxTCP_Connect()*, successful or not, the application-defined handler procedure is called. *stxTCP_Connect()* is usually called when sevenstaxTCP acts as a client TCP. Server TCPs should call *stxTCP_Listen()* instead.

8.1.3 *stxTCP_Disconnect (connection_id)***Description**

This function can be called by the application to end a TCP connection. sevenstaxTCP does all necessary actions to close a TCP connection gracefully.

The application receives the result as a call to its handler procedure (*NC_TCP_FIN*).

Parameter

UINT16_stx uConnId

ConnectionID of TCP connection to be closed. ConnectionID was returned by *stxTCP_Connect()* or *stxTCP_Listen()*.

BOOL_stx bForced

Default *FALSE_stx* should be used, to support normal TCP disconnection states. *TRUE_stx* forces an immediate RST. Might be useful in case of any connection problems.

Return Value

BOOL_stx

TRUE_stx if a close can be performed, *FALSE_stx* otherwise

Comment

Usually *stxTCP_Disconnect()* only fails when the connection is already in its closed state.

8.1.4 *stxTCP_Listen (port)***Description**

Puts sevenstaxTCP into the listen state, which means that it waits for a connection attempt on the specified port. If another TCP sends a SYN, sevenstaxTCP handles it properly and enters the connected state.

Parameter

UINT16_stx port

The port number of the local TCP, on which the connection should be established. Ports are something like channels dedicated to particular services. E.g. HTTP resides on port 80 (decimal).

Return Value

UINT16_stx

The sevenstaxTCP internal ConnectionID, or an error value (*INVALID_CONNECTION_ID*). Subsequent calls into sevenstaxTCP must use this value to identify the connection. Also events, which the application receives by its handler procedure, carry connection IDs.

Comment

stxTCP_Listen() only succeeds, if a free TCP instance is found.

8.1.5 *stxTCP_WriteBulk (connection_id, data, len, bSendDynamic)*

Description

When a connection is established, *stxTCP_WriteBulk()* can be used to send data to the remote station. This function returns immediately and sends data asynchronously the next time *stxTCP_Tick()* is called.

Parameter

UINT16_stx uConnId

ConnectionId where to send. Can be obtained either by a call to *stxTCP_Connect()* or *stxTCP_Listen()*.

UINT8_stx FPTR_stx data

Pointer to the data to be send, points to the first byte of the data that should be send.

UINT32_stx len

Size of the data in bytes. If *bSendDynamic== TRUE_stx* is used, this length should be at least the number of bytes, which be assumed to be send (see comment below).

BOOL_stx bSendDynamic

Get data to send via notify handler. Set it to *FALSE_stx* per default. *TRUE_stx* might be used by an application, which dynamically generates send data instead of simply giving a const data pointer. sevenstaxTCP will repeatedly call the application notifyhandler with notifycode *NC_TCP_GET_TXDATA* to request for data to be send. See section 8.2.6 for details.

Return Value

BOOL_stx

TRUE_stx, if *stxTCP_WriteBulk()* has accepted a request to send. Otherwise *FALSE_stx*. Succeeds only if a connection is established and no transmission is currently in progress.

Comment

Because sevenstaxTCP is designed to run on small embedded systems, it doesn't copy the data but operates on the source to avoid wasting memory. This function is able to send huge amounts of data to the remote station. Data is fragmented into *RTM_MAXDATA* sized pieces and internally sent out using multiple calls. *NC_TCP_TXREADY* is issued only after the last fragment was successfully transmitted (and acknowledged).

Please Note: The data (and possibly a reference) must remain constant until *stxTCP_WriteBulk()* has finished (which is signalled by a *NC_TCP_TXREADY* event, see below).

8.1.6 *stxTCP_Tick (void)*

Description

This function keeps sevenstaxTCP alive. It must be called as often as possible in order to provide sevenstaxTCP with processor time.

It is designed to be very fast thus applications are not slowed down. The most time-consuming action might be to build an entire TCP packet (*TCP_MULTI* per call) and deliver it to the link layer. Therefore, the maximum time *stxTCP_Tick()* will use depends on the size of the biggest packet (defined by *RTM_MAXDATA*).

Parameter

void

Return Value

void

Comment

-

8.2 Notifycodes

With one of the NC_TCP_XXX codes, sevenstaxTCP will call the handler procedure to inform about occurred events. Below is a short description of their meanings.

8.2.1 NC_TCP_CONNECTED (TCP-Info, ConnID)

After calling *stxTCP_Connect()*, *NC_TCP_CONNECTED* might be issued in case of success. The application can now send and receive data to/from sevenstaxTCP. Both parameters contain port numbers copied directly from the received packet.

Parameter l points to a *TCP_INFORMATION* structure, which describes the newly created connection. The members 'remote_ip_address', 'remote_port' and 'local_port' are valid.

Parameter w contains a Connection-ID, which has been returned by *stxTCP_Connect()*.

ReturnValue not used.

8.2.2 NC_TCP_CONNTIMEOUT (TCP-Info, ConnID)

This event occurs if *stxTCP_Connect()* failed (i.e. timed out). sevenstaxTCP does a few retries to establish a connection and then gives up.

Parameter l points to a *TCP_INFORMATION* structure, which describes connection. The members 'remote_ip_address', 'remote_port' and 'local_port' are valid.

Parameter w is a Connection-ID, which identifies the connection that failed.

ReturnValue not used.

8.2.3 NC_TCP_FIN (AbortReason, ConnID)

This event is posted when the TCP connection is terminated. This may happen when a web server has finished sending data, when an error occurs, or after the application has called *stxTCP_Disconnect()*.

Parameter l is a *ABORT_REASON* (see section 4.6, Terminating a TCP connection)

Parameter w is a Connection-ID, which identifies the connection that has been closed.

ReturnValue not used.

8.2.4 NC_TCP_RECV (TCP-Info, ConnID)

When sevenstaxTCP has received a data packet, *TCP_RECV* is issued to tell the application, that data has arrived.

Parameter l points to a *TCP_INFORMATION* structure which describes the incoming data packet. All members are valid.

Parameter w is a Connection-ID, which identifies the connection.

ReturnValue not used.

8.2.5 **NC_TCP_TXREADY** (*TCP-Info, ConnID*)

This event is posted to signal the application, that sevenstaxTCP is ready to accept data for next transmission. It occurs after sevenstaxTCP has received an ACK as an answer to transmitted data.

After getting *NC_TCP_TXREADY*, the buffer allocated for *stxTCP_WriteBulk()* can be reused/freed. *NC_TCP_TXREADY* is issued after sevenstaxTCP has transmitted the entire range of data which was specified through a call to *stxTCP_WriteBulk()*.

Parameter l points to a *TCP_INFORMATION* structure, which describes the TCP connection. The members '*remote_ip_address*', '*remote_port*' and '*local_port*' are valid.

Parameter w is a Connection-ID, which identifies the connection.

ReturnValue not used.

8.2.6 **NC_TCP_GET_TXDATA** (*TCP_TxData, ConnID*)

If *stxTCP_WriteBulk()* was called with parameter *bSendDynamic== TRUE_stx* (see section 8.1.5 for details), sevenstaxTCP will call the notifyhandler with *NC_TCP_GET_TXDATA*, to get the next frame (PPP/Ethernet) to be send. The application now is responsible to copy the requested data into the given destination address. This will repeatedly be done until the application notifyhandler returns 0 for this call. Please see section for details.

Although this features might be used rarely, this gives the advantage, that the application dynamically might send a very large amount of data, without the need to keep the complete databuffer secure until *NC_TCP_TXREADY* is received. But, if there is any TCP retransmission necessary, the application must be able to generate this dynamic data again.

Parameter l is a pointer to a *TCP_TXDATA* structure, which contains sevenstaxTCP instructions for the application: *pSourceData* is the application source data pointer, which sevenstaxTCP wants to send next. Normally this pointer will increase with every call. But it will be set back, if e.g. a TCP retransmission is necessary. Please assure, that your application is able to set this source pointer back to old data!

pTargetBuffer is the target address, to which the application should copy the data. *uCount* is number of bytes, which sevenstaxTCP expects to receive.

Parameter w is a Connection-ID, which identifies the connection.

ReturnValue returns the number of bytes, the application has copied from *pSourceData* into *pTargetBuffer*. If the last data has been sent, the application should return 0 here.

8.3 Adjustable Parameters

There are some parameters defined in '*tcpdefs.h*' which can be customized. These parameters should properly be adjusted depending on the lower network layer and communication device. The following list describes them in detail.

according to its free buffer space. Because sevenstaxTCP does not have a separated buffer management, *TCP_WINDOW_SIZE* is a fixed value.

9 UDP Protocol API

9.1 Public Functions

UDP is part of sevenstaxTCP standard internet library. It can be enabled via `#define UDP_SUPPORTED = 1` in `features.h` (see section 2, Supported Protocols)

Please note: There is a little restriction on UDP packets. As fragmentation is not supported. sevenstaxTCP silently throws away all packets that are fragmented - but that is no real drawback because sevenstaxTCP receives unfragmented packets up to the maximum length of the link layer (PPP, Ethernet etc.).

Nevertheless care must be taken before building the TCP/IP stack. If the size of the IP packets (define `TCP_WINDOW_SIZE` in `tcpdefs.h`) is chosen larger than the size the link layer is able to transmit without fragmentation, it might be possible that no data will be received while the sending of data works without problems

9.1.1 `stxUDP_Init (handler)`

Description

Sets up some internal variables, initialises internal UDP instances and registers application callback function. Must be called before using any other UDP function.

Parameter

`PROTOCOL_NOTIFY_HANDLER handler`

Application handler function to receive notify codes from sevenstaxTCPs UDP module. The handler function will get called, when UDP data is received.

Return Value

void

Comment

-

9.1.2 `stxUDP_Attach (local_port)`

Description

Calling this function will enable one of the possible `UDP_MULTI` UDP instances, to be used for sending and/or listening UDP packets.

If any incoming UDP packet is detected, the application gets notified by `NC_UDP_RECV`.

Parameter

`UINT16_stx local_port`

The UDP port to listen on. Used too as local port while sending.

Return Value

`UINT16_stx`

UDP-ConnectionID, to be used for further UDP functions. If the same UDP port has

already been 'attached' by previous calls, this function will return the same ConnectionID as before. Returns *UDP_INVALID_ID*, if no more free UDP instances (#defined by *UDP_MULTI*).

Comment

Please note that it is possible to independently use a TCP port and a UDP port at the same time, on both either server (listening) or client (acting) mode.

9.1.3 *stxUDP_Detach (uConnId)***Description**

Calling this function disables listening or sending on the given UDP port. The UDP port gets closed.

Parameter

UINT16_stx uConnId
UDP-ConnectionID, returned by *stxUDP_Attach()*.

Return Value

void

Comment

-

9.1.4 *stxUDP_Write (uConnId, psRemotelp, uRemotePort, data, datalen)***Description**

Immediately Sends a UDP datagram to another computer.

Parameter

UINT16_stx uConnId
Valid UDP ConnectionID, returned by *stxUDP_Attach()*.

IPV4_FPTR_stx psRemotelp
The target's IP-Address.

UINT16_stx uRemotePort
The target's UDP port.

UINT8_stx FPTR_stx data
Pointer to the data to be inserted as UDP payload.

UINT16_stx datalen
Amount of payload bytes to be send.

Return Value

BOOL_stx
TRUE_stx if data has been send, *FALSE_stx* if any error.

Comment

stxUDP_Attach() must have been called before using this function.

10 DNS Protocol API

A simple DNS client is part of sevenstaxTCP standard internet library. The application can send a DNS query to get the IP address of server with a known name. DNS can be enabled via `#define DNS_SUPPORTED = 1` in `features.h`.

The DNS protocol works ahead UDP, so `UDP_SUPPORTED = 1` must be set too (see section 2, Supported Protocols).

10.1 Public Functions

10.1.1 `stxDNS_Init (handler)`

Description

Sets up some internal variables and registers the application callback function. Must be called before using any other DNS function.

Parameter

PROTOCOL_NOTIFY_HANDLER `handler`

Application handler function to receive notify codes from sevenstaxTCPs DNS module. The handler function will get called, when DNS data is received or timeout occurs

Return Value

void

Comment

-

10.1.2 `stxDNS_QueryStart (dns_ip, www_name)`

Description

Instructs sevenstaxTCP to transmit DNS queries on UDP port 53 to a known DNS server. Multiple DNS queries will be send by `stxDNS_Tick()`. The result is posted as a notification code to the handler procedure, which can either be `NC_DNS_IP` (success) or `NC_DNS_TIMEOUT` (failure).

Parameter

CONST_stx IPV4 FPTR_stx `dns_ip`

IP address of the DNS server to be used.

STRING_stx `www_name`

A pointer to the web-name, e.g. "www.yahoo.com".

Return Value

BOOL_stx

`TRUE_stx`, if the query packet was successfully delivered to UDP. `FALSE_stx` if a DNS query is already in progress, this means that neither the DNS answered nor a time-out occurred.

Comment

This function internally uses the UDP API of sevenstaxTCP. Although there are no additional function calls necessary, please assure, that at least one free UDP instance must be available.

10.1.3 stxDNS_QueryStop ()**Description**

Stops a pending DNS query and releases UDP port.

Parameter

void

Return Value

void

Comment

There is normally no need to call this function, because notifycodes will be delivered, if a time-out occurs. But if needed, DNS immediately will stop any query by calling this function, no notify code will be delivered.

10.1.4 stxDNS_Tick ()**Description**

This function keeps the DNS query alive, assuring multiple retries until a time-out occurs. It should repeatedly be called by the application, like all the other sevenstax "Tick" functions too, independently of a pending DNS query or not.

Parameter

void

Return Value

void

Comment

In case of timeout, this function will call the application notify handler with *NC_DNS_TIMEOUT*. If the DNS server responds, this will be synchronously be signaled by *NC_DNS_IP* by the responsible process-packet-functions.

10.2 Notifycodes**10.2.1 NC_DNS_IP**

After calling *stxDNS_QueryStart()*, this event is triggered when the DNS client has received a valid answer and a record is found, which contains the requested IP address.

Parameter l contains the IP address, passed in network byte order.

Parameter w is not used and always zero.

ReturnValue not used.

10.2.2 NC_DNS_TIMEOUT

This event is posted as a negative result when `stxDNS_QueryStart()` was called and the DNS resolver timed out.

There are various reasons for such a failure. The most likely reasons are, that the DNS server is not able to find the requested hostname (e.g. because it does not exist anymore) or the DNS server itself is temporarily not available.

Parameter l is not used and always zero.

Parameter w is not used and always zero.

ReturnValue not used.

10.3 Adjustable Parameters

DNS parameters are defined in '`dnsdefs.h`'. These parameters should properly be adjusted depending on the application requirements. The following list describes them in detail.

DNS_QNAME_MAX	default: 64	valid: 1..n
This is the buffer size in bytes, which is permanently allocated in RAM to handle the friendly name of a DNS request.		
DNS_SRV_USERFCT	default: 0	valid: 0..1
This is very application specific and only rarely used: If enabled (set to 1), <code>sevenstaxTCP</code> calls the application notifyhandler to support the DNS request, instead of automatically sending it.		
DNS_SRV_MASQUERADE_IP_ADDR	default: 0	valid: 0..1
This is very application specific and only rarely used: If enabled (set to 1) and <code>DNS_SRV_USERFCT</code> is set to 1 too, <code>sevenstaxTCP</code> expects the application notifyhandler to handle IP masquerading.		

10.4 Using DNS (Domain Name Service)

For humans it is rather convenient to work with readable names than with IP addresses. For example the URL 'www.yahoo.com' to access a website instead of using the IP addresses 64.58.79.230 is much more memorable. That is the reason why DNS was created. A DNS or domain name service, converts names to their corresponding IP addresses.

The modern DNS is organised a bit like a tree. There is a root called '.' with certain main branches called 'top level domains' (TLD). At the root are a few servers (e.g. `a.root-servers.net`, `b.root-servers.net` ...) which know all addresses of all the servers which handle individual TLDs (Top Level Domains) such as `.com`, `.edu`, `.mil` or `.net`.

Communication with the DNS is done by UDP on port 53. DNS client software (so called resolvers) must send special packets (see RFC 1035) to the DNS. The DNS, on his part, sends back a packet containing some records which contain e.g. the IP address.

To use `sevenstaxTCP`'s integrated DNS resolver within your application, the following steps are required:

- In '`features.h`' insert `#define DNS_SUPPORTED = 1`.

- Because the DNS protocol needs UDP as the transport layer, #define *UDP_SUPPORTED = 1* in 'features.h' too. Also assure at least 1 free UDP instance with *UDP_MULTI* in 'udp_defs.h'.
- The DNS-server's address can be retrieved via the InternetServiceProvider (if PPP used), via the DHCP-Server (if Ethernet used) or manually. The primary DNS should preferential be used. The secondary DNS-server should be used only if the primary DNS-server is down.
- Call *stxDNS_Init()* once to set-up sevenstax DNS internal state.
- After initialization of DNS call *stxDNS_QueryStart()* supplied with the IP address of the DNS server and the name of a website. The DNS local port is set automatically by the sevenstax software.
- In your main loop or operating system task call *stxDNS_Tick()* repeatedly in order to keep alive and finish DNS queries.
- On completion, sevenstaxTCP will notify the application either with *NC_DNS_IP* or *NC_DNS_TIMEOUT*.
- When receiving the event *NC_DNS_IP*, which means that the DNS has answered, *stxTCP_Connect()* may be called using this newly gained IP address.

11 ICMP Protocol API

A simple ICMP responder is part of sevenstaxTCP standard internet library. sevenstaxTCP automatically responds to an incoming “ping” with an ICM packet.

If an incoming packets is identified as an ICMP echo request, sevenstaxTCP does the following steps:

- The packet is copied to the TX buffer and the type is changed to an ICMP echo reply.
- A new checksum is calculated.
- Source and Destination addresses are swapped.
- The packet is send off to the network.

ICMP can be enabled via `#define ICMP_SUPPORTED = 1` in features.h. (see section ,).

11.1 Public Functions

Currently there is no public NTP API function available in sevenstaxTCP, nor a function to send an ICMP message by application.

11.2 Notifycodes

-

11.3 Adjustable Parameters

-

12 NTP Protocol API

A simple NTP client is part of sevenstaxTCP standard internet library. NTP can be used to retrieve the current time.

NTP can be enabled via `#define NTP_SUPPORTED = 1` in `features.h`.

The NTP protocol works ahead UDP, so `UDP_SUPPORTED = 1` must be set too (see section 2, Supported Protocols).

12.1 Public Functions

12.1.1 *stxNTP_Init (handler)*

Description

Sets up some internal variables and registers the application callback function. Must be called before using any other NTP function.

Parameter

PROTOCOL_NOTIFY_HANDLER handler

Application handler function to receive notify codes from sevenstaxTCPs NTP module. The handler function will get called, when NTP data is received or time-out occurs.

Return Value

void

Comment

-

12.1.2 *stxNTP_Query (IPV4 FPTR_stx ntp_server)*

Description

NTP queries will be started by calling this function. Sets some internal variables and returns immediately. The actual query will be done the next time *stxTCP_Tick()* is called. After a call to *stxNTP_Query()*, the application defined handler procedure will either receive an *NC_NTP_SUCCESS* (got the time) or *NC_NTP_FAILURE* (no response from the time server).

Parameter

CONST_stx IPV4 FPTR_stx ntp_server

A pointer to an IPV4 structure representing the IP address of the time server.

Return Value

void

Comment

Please note: Calls made to *stxNTP_Query()* while being in progress will re-trigger the query process.

12.1.3 *stxNTP_Transform (ntps, sec, min, hour, year, month, day, day_of_week)*

Description

Takes the NTP seconds (*ntp*s) delivered by the time server and writes the results into the locations for application time variables.

Parameter

UINT8_stx FPTR_stx sec, min, hour, year, month, day, day_of_week point
Address of application variables to receive the delivered time values.

Return Value

void

Comment

Because the result of an NTP query is a 32-bit value of seconds since 1.1.1900 this function is for convenient conversion of this infamous seconds into time and date values.

The time returned by the servers is UTC (Coordinated Universal Time) which means, it has to be corrected if the local time zone differs from UTC. This correction must be done before calling *stxNTP_Transform()*. The difference between the local time and UTC has to be added to the seconds returned by the server. After calling *NTP-Transform()* the right values can be read from the parameters as described above.

12.1.4 *stxNTP_Tick ()*

Description

This function keeps NTP alive, assuring multiple retries until a time-out occurs. It should repeatedly be called by the application, like all the other sevenstax "Tick" functions too, independently of a pending NTP query or not.

Parameter

void

Return Value

void

Comment

In case of timeout, this function will call the application notify handler with *NC_NTP_FAILURE*. If the NTP server responds, this will be synchronously be signaled by *NC_NTP_SUCCESS* by the responsible process-packet-functions.

12.2 Notifycodes

12.2.1 *NC_NTP_SUCCESS*

This event indicates an successful reception of a time server response.

Parameter l contains the time in seconds since 01.01.1900. This value might be converted by the application with *stxNTP_Transform()*.

Parameter w is not used and always zero.

ReturnValue not used.

12.2.2 NC_NTP_FAILURE

This is the opposite to NC_NTP_SUCCESS. When an NTP query was made, NC_NTP_FAILURE is posted to the handler procedure, if the time server does not respond properly.

Parameter l is not used and always zero.

Parameter w is not used and always zero.

ReturnValue not used.

12.3 Adjustable Parameters

12.4 Using NTP

The Network Time Protocol (NTP) is used to synchronize the time of a computer client or server to another server or reference time source, such as a radio or satellite receiver or modem. It provides accuracies typically within a millisecond on LANs and up to a few tens of milliseconds on WANs relative to Coordinated Universal Time (UTC) via a Global Positioning Service (GPS) receiver, for example. Typical NTP configurations utilize multiple redundant servers and diverse network paths in order to achieve high accuracy and reliability.

NTP uses UDP on port 123 for both, transmission and reception.

To access NTP servers with sevenstaxTCP you should proceed the following way:

1. #define *NTP_SUPPORTED* =1 in 'features.h'
2. Call *stxNTP_Init()* once to set-up sevenstaxNTP's internal state.
3. Call *stxNTP_Query()* once to start querying the NTP server.
4. In your main loop or operating system task call *stxNTP_Tick()* repeatedly in order to keep alive and finish NTP queries.
5. After a while, the handler procedure receives either an *NC_NTP_SUCCESS* or *NC_NTP_FAILURE* event.
6. On *NC_NTP_SUCCESS* call *stxNTP_Transform()* with the 32-bits value in order to convert NTP output into human suitable time values.
7. On *NC_NTP_FAILURE* the query can, for example, be repeated with another time server's IP address.

13 Debugging Support

At source level, sevenstaxTCP contains a simple debugging support through macros mapped to printf() (DebugOut-Strings).

The main switch to enable DebugOuts for all modules is DEBUG_STX = 1 in features.h.

```
/* application */
#define DEBUG_STX          1          /* Main switch: sevenstax debugging enabled */
```

Each module has a separate switch to enable debug information at compilation time. This enables a fine control about You can control it via settings in 'features.h':

```
...
/* protocols */
#define IP_DEBUG          1          /* ipv4 module */
#define TCP_DEBUG        1          /* tcp module */
#define TCPPACKET_DEBUG  1          /* tcp module, packet details */
#define TCPPACKET_DUMP   1          /* tcp module, packet content */
#define UDP_DEBUG        1          /* udp module */
#define DNS_DEBUG        1          /* dns & dns_srv module */
#define DNS_DETAILS      1          /* dns & dns_srv & mdns packet details */
#define ICMP_DEBUG       1          /* icmp module */
#define HTTPSERV_DEBUG   1          /* httpserv & html_page modules */
#define NTP_DEBUG        1          /* ntp module */
...
```

Each module has a separate macro to map DebugOuts to a common printf() function: (e.g. tcpdefs.h):

```
...
/* control debug out */
#if (TCP_DEBUG==1 && DEBUG_STX==1)
#include "debugstx.h"
#define TCPPRINTLN          STRINGOUT_CMPLT("TCP");V_STRINGOUT
#define TCPPRINT(a)        TCPPRINTLN(a);
#define TCPPRINT1(a,b)     TCPPRINTLN(a,b)
...
```

All DebugOuts are mapped to the central STRINGOUT_CMPLT macro in 'debugstx.h'. This is the point, where the developer might map to an alternative to printf(), e.g. to another serial out function or save it in memory.

Details of DebugOut might be controlled via define switches in features.h:

```
/* configure format */
#define DBGFMT_TIMESTAMP  1          /* enables preceding timestamp */
#define DBGFMT_FILELINE   1          /* enables preceding filename/line number */
#define DBGFMT_MODULE     1          /* enables preceding module name like */
```

Please Note: If you want to use some of the xxxPRINTx() macros for yourself and it follows an if/else statement etc., be sure to enclose it in braces. Not doing so might cause your code to work while debugging but misbehave in release versions.

14 Restrictions

sevenstaxTCP is focused on systems with very small resources, to give them relevant TCP/IP functionality. Therefore – in comparison with a full featured stack implementation of a PC software f.e. - it has some restrictions in usage and capabilities, the user should know:

#	Restriction	Details
1	Not aware of interrupts	Per default, sevenstaxTCP functions are called inside a "tick" function, which normally is located inside a endless while() loop or OS task. If parts of sevenstax embedded protocols are called from within an ISR, synchronization must be done by the user.
2	Not full RFC compliant	To give maximum capabilities within small system resources, sevenstax embedded protocols fulfil only basic and most relevant parts of the RFC specifications.
3	No buffering of any kind	Because sevenstaxTCP has no data buffers, when sending data over a TCP connection using stxTCP_WriteBulk(), the content must exist and must not be changed, until the application receives a feedback (NC_TCP_TXREADY).
4	IP fragmentation	sevenstaxTCP does not support IP fragmentation. This is no disadvantage for TCP/IP, but for UDP packets: Only the first part of fragmented UDP packets are received and used. sevenstaxTCP also never generates fragmented IP packets by itself.
5	Replies from name servers (DNS)	Because DNS packets are send over UDP and sevenstaxTCP throws away UDP fragments, DNS only works if either the reply (from the name server) is not fragmented or the wanted IP address is in the first fragment. For very large sites, such as www.google.com, name server replies doesn't fit into a single UDP packet. But In the majority of cases, an IP address is included and found in the first fragment.
6	IP address from name servers (DNS)	sevenstaxTCP only uses the first of (potential multiple) IP address found in a name server's reply.
7	ICMP	The only part of ICMP, which is currently implemented into sevenstaxTCP, are ICMP echo replies (PING).
8	Use of Ethernet and PPP	This version of sevenstaxTCP does not support use of PPP mode and Ethernet mode <u>at the same time</u> . Please use the separate product 'sevenstaxTCP MultiDevice', if you need several physical interfacesat the same time (e.g. router functionality).

sevenstaxTCP is object of permanent improvement. Please contact info@sevenstax.de to get more information about new TCP/IP features given in next release version.

15 Change History

Ver.	Date	by	Change description
2.88	21. Aug. 2002	pe	Base version
2.91	12. Dec. 2002	pe	Improved API documentation
3.01	16. Jan. 2003	pe	Additional features added
3.02	13. Feb. 2003	krz	Completely reviewed
3.03	21. Feb. 2003	krz	Some small corrections
3.11	26. Feb. 2003	pe	Restrictions reviewed
3.12	22. July 2003	ck	New graphics
3.13	24. May 2004	krz	Completely reviewed
3.13a	27. May 2004	krz	ARP and NotifyCodes updated
3.14	01. June 2004	krf	Released
3.15	29. June 2004	krz	Format review
3.16	30. July 2004	krz	Additional TCP Information in notify codes, ARP restrictions removed, types extended by '_stx' prefix, corrections
3.17	13. Aug. 2004	ck	Proof-Read
3.18	20. Aug. 2004	krz	Added header dependencies
3.30	04. May 2006	krz	Reviewed for sevenstaxTCP V3.3
4.20	12. Mar 2007	krz	Completely reviewed for sevenstaxTCP V4.2 - several new and changed API functions - new graphics

The information furnished in this document is believed to be accurate and reliable. However, no responsibility is assumed by sevenstax for its use, nor for any infringements of patents or other rights of third parties resulting from its use. No license is granted under any patents or patent rights of sevenstax.

This document is an intellectual property of sevenstax GmbH. Unauthorized copying and distribution is prohibited.

Copyright (c) 2007 by sevenstax GmbH