



sevenstax Embedded Protocols

Ethernet Drivers, ARP, DHCP and BOOTP

Ethernet User Manual

Document-Rev.: 3.3
State: Release
Author: sevenstax GmbH

Initial version: 26/05/03
Last change: 08/05/06
Changed by: Ralf Krizsan, sevenstax GmbH

Publication: Public
Filename: sevenstaxEthernet_UserManual_V3_3.sxw

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

Copyright (c) 2006 by sevenstax GmbH

Table of Contents

1	Introduction.....	4
2	Ethernet Basics and Driver.....	5
2.1	Introduction.....	5
2.2	Ethernet Basics.....	5
2.3	Ethernet Addressing.....	5
2.4	Ethernet Framing.....	5
2.5	Ethernet Controller Chips.....	7
2.6	sevenstax' approach to interface Ethernet Controller Chips.....	7
2.7	The Network Interface Controller (NIC).....	9
2.8	Interrupts and the NIC.....	12
3	Ethernet Receive Ring Buffer (RXBuff).....	12
3.1	Path of received data.....	12
3.2	The Intermediate Ethernet Driver (IED).....	14
3.3	The Packet Descriptor.....	15
3.4	Conclusion.....	16
4	Address Resolution Protocol (ARP).....	17
4.1	Why ARP?.....	17
4.2	How does ARP work?.....	17
4.3	How does sevenstaxARP work?.....	18
4.4	sevenstaxARP Function Reference.....	19
4.5	sevenstaxARP Notification Codes Reference.....	22
4.6	sevenstaxARP Externals Reference.....	23
5	Bootstrap Protocol (BOOTP).....	26
5.1	Why BOOTP?.....	26
5.2	How does BOOTP work?.....	26
5.3	How does the sevenstaxBOOTP client work?.....	27
5.4	sevenstaxBOOTP Function Reference.....	28
5.5	sevenstaxBOOTP Notification Codes Reference.....	29
5.6	sevenstaxBOOTP Externals Reference.....	30
6	Dynamic Host Configuration Protocol (DHCP).....	33
6.1	Why DHCP?.....	33
6.2	How does DHCP work?.....	33



6.3 How does the sevenstaxDHCP client work.....33

6.4 sevenstaxDHCP Function Reference.....34

6.5 sevenstaxDHCP Notification Codes Reference.....36

6.6 sevenstaxDHCP Externals Reference.....38

1 Introduction

This is the User Manual of the Ethernet Part of the sevenstax Embedded Internet Protocol Suite.

It combines and obsoletes further versions of the documentation for the Ethernet Driver, the ARP module, the BOOTP module and DHCP module in one document.

Chapter 2 “Ethernet Basics and Driver“ will give you advices on how to use sevenstax Internet protocols together with an Ethernet controller.

Chapter 4 “Address Resolution Protocol (ARP)” might only give you the background for the sevenstax ARP implementation, but ARP is completely resolved inside the TCP stack and no API has to be supported by the user application to enable ARP functionality.

Chapter 5 “Bootstrap Protocol (BOOTP)” provides you with all information necessary to integrate sevenstaxBOOTP into your application.

Chapter 6 “Dynamic Host Configuration Protocol (DHCP)” explains how to use the sevenstaxDHCP API.

2 Ethernet Basics and Driver

2.1 Introduction

Ethernet is an extremely flexible and low cost LAN (local area network) technology which is capable to connect various computers and electronic devices together. Almost every computer manufacturer supports Ethernet and there is a huge assortment of Ethernet controller chips and network interface cards on the market.

This document describes how sevenstaxTCP can be used on Ethernet-enabled devices in order to communicate to other TCPs on the LAN or the Internet.

2.2 Ethernet Basics

Based on the OSI reference model, the Ethernet layer occupies the lowest parts (layer 2 and below) which are the lowest data link (Media Access Control – MAC) and physical signalling sub-layers.

The Ethernet system includes four building blocks that, when combined, make a working Ethernet:

- **Ethernet frame**, which is a standardized set of bits used to carry data over the system.
- **Media access control protocol**, which consists of a set of rules embedded in each Ethernet interface that allow multiple computers to access the shared Ethernet channel in a fair manner.
- **Signalling components**, which consists of standardized electronic devices that send and receive signals over an Ethernet channel.
- **Physical medium**, which consists of the cables and other hardware used to carry the digital Ethernet signals between computers attached to the network.

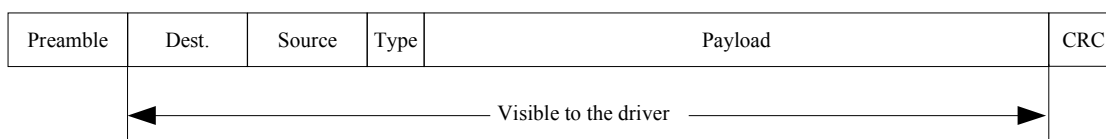
2.3 Ethernet Addressing

Like IP, Ethernet also uses unique numbers to identify its nodes. An Ethernet address is a 48-bit number which must be unique across the LAN. Two of them, the sender's and receiver's addresses can be found near the top of every Ethernet frame.

Also like IP, Ethernet knows broadcast (all bits are '1') and specific multicast addresses (the first bit is '1').

2.4 Ethernet Framing

Ethernet frames are the vehicles by which data packets travel through the cables. An Ethernet frame consists of some fields which carry certain information. Please see the drawing below.



Preamble (64 bits)

The preamble field is used by 10Mbps Ethernet systems to synchronize themselves with the incoming frame. It consists of alternating ones and zeros. Newer Ethernet technologies such as Fast- or Gigabit Ethernet do not use a preamble but send one out to preserve backwards compatibility with the original Ethernet frame.

Destination (48 bits)

This is the MAC address of the station which should receive the frame. A destination MAC address, where all bits are set, is called a 'broadcast' address. In this case all stations receive the frame and deliver it to the device driver software.

Source (48 bits)

This is the MAC address of the station which has originated the frame. Recipients use this address to send replies back to the sender.

Type (16 bits)

Depending on the standard used (DIX or IEEE), the type field contains either a type or length information. If the value of this field is greater or equal to 1536, then the field indicates the type of higher level protocol which is in the frame. For example: An IP packet always has the type field set to 0x0800. ARP packets are recognized through a value of 0x0806 in the type field.

Payload (368....12000 bits)

This is the only field of variable size. When using TCP/IP atop Ethernet, the payload field carries information which is all part of TCP/IP including headers and user data. This field must contain a minimum of 46 bytes and may range up to a maximum of 1500 bytes.

CRC (32 bits)

This is the last field which contains a CRC (cyclic redundancy check) value to ensure frame integrity. As the frame arrives, the CRC value is calculated again by the receiving station. If both, the received and calculated values are identical, the packet is accepted and delivered to the upper layers.

2.5 Ethernet Controller Chips

The purpose of an Ethernet Controller Chip is to assemble data into Ethernet frames and compute the cyclic redundancy check used for error detection. On reception, it does a frame check using the CRC value, then strips off preamble and CRC value and delivers the packet to the device driver software.

Most Ethernet Controller Chips are much versatile and fit into different hardware architectures. They can be connected to the processor's bus or to its I/O system. Often they have eligible bus width either 8 or 16 bits. Some of them have on-chip memory which is used as TX/RX FIFO. Others share memory through DMA. Most Ethernet Controller Chips for 10Mbps can either generate interrupts or operate in polling mode. Most of them can access EEPROMs directly to configure themselves on start up.

2.6 sevenstax' approach to interface Ethernet Controller Chips

sevenstax embedded protocols use a two-layered manner when used on top of Ethernet.

One early and important driver design issue is whether to implement the driver as a series of layers, or whether it should be structured as a single, monolithic unit. The following provides the trade-offs of using a layered approach.

Advantages of layered architecture

Depending on the goals, multiple driver layers can provide a number of benefits. Using layers allows the separation of higher-level protocol issues from management of the specific underlying hardware. This makes it possible to support a wider variety of hardware without having to rewrite large amounts of code. It also promotes flexibility by allowing the same protocol driver to plug into different hardware drivers at runtime.

Layering also makes it possible to hide hardware limitations from users of the device, or to add features not supported by the hardware itself. For example, if a given piece of hardware can handle transfers only of a certain size, another driver that would break oversized transfers into smaller pieces might be stacked on top. Users of the device would be unaware of the device's shortcomings.

Inserting driver layers provides a transparent way to add or remove features from a product without having to maintain multiple code bases for the same product.

Drawbacks of layered architecture

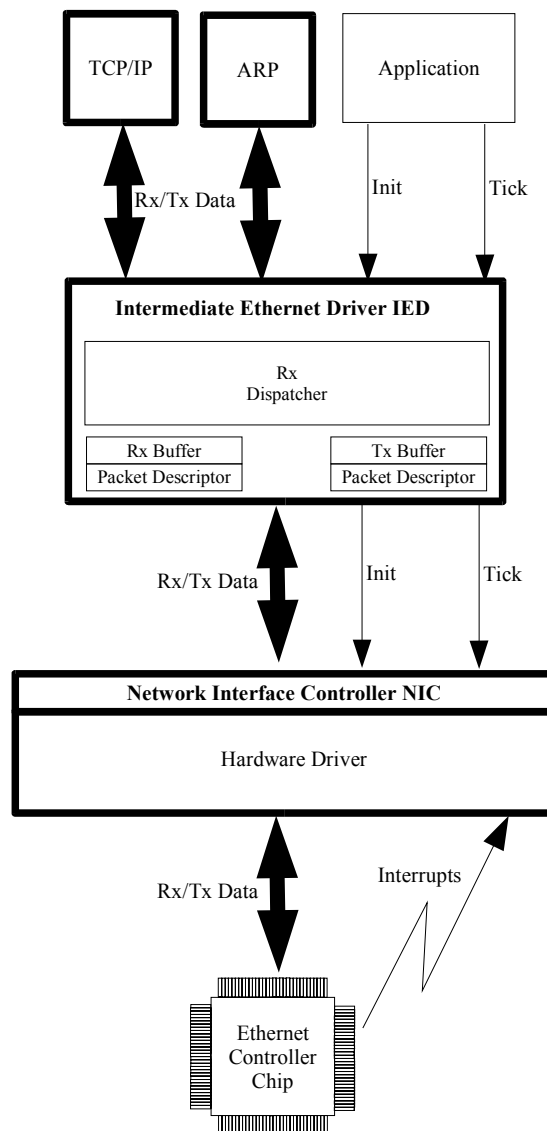
Of course there is a downside to the use of a layered architecture. First, I/O requests incur extra overhead because many requests has to take a trip through the layers. It also takes somewhat more design effort to make sure that the separate driver components fit together seamlessly.

Since the overall functionality is no longer contained in a single driver, there is somewhat more bookkeeping involved in managing the drivers. This also has some impact on maintaining version compatibility between various members of a hierarchy.

Now back to the real world: The lowest layer's interface is named **NIC - Network Interface Controller**. It directly accesses the hardware through a set of specialized routines. The NIC driver is responsible for handling for instance interrupts, processing errors and for transferring raw data from and to the Ethernet Chip.

Please note: In general there are a lot of facts which make it impossible to write generic drivers for all the purposes. Some arguments have been stated before for e.g. the variety of possibilities to connect the Ethernet controller. A design connecting the Ethernet controller memory mapped, using DMA or the I/O- interface of the controller will surely require adaptations of the driver software. This also effects the usage of interrupts to signal external events like the reception of data – which is not necessarily the most effective way to embed an Ethernet controller into a system

The layer above is called **IED - Intermediate Ethernet Driver**. It manages memory (e.g. TX/RX buffers and packet descriptors), performs synchronisation between TX and RX requests, provides appropriate packets for transmission, adjusts received packets and interacts with TCP/IP and ARP. This might be useful for cases, where the application polls the Ethernet reception. For extensive usage of receive interrupts, the developer might feel free to allocate the reception buffer in the low level Ethernet driver instead to give an immediate access to it.



2.7 The Network Interface Controller (NIC)

As already mentioned, the lowest layer must export a few functions which are, as a whole, the NIC. The NIC is a kind of hardware abstraction which simplifies the functionality of an Ethernet Controller down to a minimum.

The layer above (called IED, Intermediate Ethernet Driver) always uses an unified way to talk to the network through a device driver which exports the NIC and, on its part, works closely with an Ethernet Controller Chip.

In order to map the NIC to real function calls at compile/link time, it must be declared. Here is a little code snippet to point this up:

```

/* Prototypes of the Ethernet chip driver */
    
```

```

extern int          stxNIC_Init (unsigned char FPTR_stx macaddr, void
(FCTPTR_stx rx_notify)(void));
extern int          stxNIC_Tick (void);
extern int          stxNIC_Send (unsigned char FPTR_stx data, unsigned short
size);
extern void         stxNIC_RxBuff_Release (PACKET_DESC FPTR_stx packet);
extern void         stxNIC_RxBuff_Receive (PACKET_DESC FPTR_stx packet);

```

The higher layer calls the NIC using these functions. An Ethernet driver writer just implements the driver and declares its interfaces by using these function names. That's all - Now the default IED (please see the chapter 'Intermediate Ethernet Driver') can access them.

For successful operation, the lowest driver's interfaces must follow the specifications which are listed below:

NIC_ERRCODE_stx stxNIC_Init (UINT8_stx FPTR_stx macaddr, PROTOCOL_NOTIFY_HANDLER rx_notify)

This function is responsible for initializing the ReceiveBuffer and the Ethernet Controller Chip in all aspects. It usually establishes hardware I/O, initialises the Ethernet Controller, gives him a MAC address and enables immediate reception.

The first parameter is a pointer to an array of 6 bytes which is the new individual address (or MAC address) of the Ethernet Controller.

The second parameter is function pointer to a dedicated application notify handler, which might be used e.g. to notify the application, if an RX interrupt arrived.

The stxNIC_Init() should return 0 on failure or any other user definable value on success.

SINT16_stx stxNIC_Tick (void);

To do periodic tasks, such as error checking and recovery etc., this function is called by the higher layer in a frequent manner.

For a driver, which is not aware of interrupts, this function can poll the Ethernet Chip's status register(s) and perform relevant actions.

The stxNIC_Tick() should return 0 on failure. If desired the return value can be defined by the user. Please note that in these cases it might make sense to adapt the IED to evaluate the user defined return values

void stxNIC_RxBuff_Receive (PACKET_DESC FPTR_stx packet)

This function gives access to the Ethernet drivers receive ringbuffer. It will be called from inside the stxETH_Tick() to periodically check for exactly new incoming data and to process it by the TCP stack.

It returns the given packet descriptor structure parameter, which contains information of a received Ethernet frame (if any). If currently no RX

frames are available, it will return zero in the packet descriptors size member. If multiple frames are available, it will return the information of the oldest no yet processed received Ethernet frame.

Note: After processing that received frame, `stxNIC_RxBuff_Receive()` must be called to release that ring buffer memory.

void stxNIC_RxBuff_Release (PACKET_DESC FPTR_stx packet)

This function releases the RxBuffer memory used by that frame given in the packet descriptor. Additionally it increments the read index to the next available frame.

The given parameter must be the same as returned in `stxNIC_RxBuff_Receive()`.

Note: It is not allowed to call the `stxNIC_RxBuff_Receive()` function before releasing the memory of the preceding frame with `stxNIC_RxBuff_Release()`!

SINT16_stx stxNIC_Send (UINT8_stx FPTR_stx pcData, UINT16_stx wSize)

This function executes an immediate send on the Ethernet Controller Chip.

The supplied parameters are a pointer to the location in memory and a size indication. Both information together describe a packet which is send out as the payload field of an Ethernet frame.

`stxNIC_Send()` must return 0 on failure and any other value on success.

Also the software mapped to `stxNIC_Send()` should make some effort to ensure that a packet can be transmitted. This includes repeated attempts if the Ethernet Controller Chip is busy.

UINT16_stx stxNIC_Receive (UINT8_stx FPTR_stx pcDest)

Note: This function is currently disabled and should not be called! But it is prepared for users which prefer a polling mode instead of Interrupt Service Routines and like to manipulate the source code given by sevenstax.

This function **does not support the Receive Ring Buffer**. Instead, it gives direct access to the Ethernet Controllers physical receive buffer.

If the system is prepared to use a polling mode, the higher layer calls `stxNIC_Receive()` frequently to check if has data arrived.

The supplied pointer (as the one and only parameter) is used by `stxNIC_Receive()` to store the data. This memory area must be big enough to hold at least a full-length Ethernet payload field (1500 bytes).

`stxNIC_Receive()` must return zero if there's no data to process. Otherwise, the return value must be the number of bytes which `stxNIC_Receive()` has written into memory.

2.8 Interrupts and the NIC

Most Ethernet Controller Chips have interrupt outputs to notify a processor or microcontroller of events which need immediate attention. This can be caused by e.g. incoming Ethernet frames or fault situations.

The philosophy behind the NIC is maximum encapsulation and abstraction of the hardware. This means that any hardware-related interaction with the Ethernet Chip, from other places than the hardware driver, is prohibited. Thus, interrupts are handled completely inside the hardware driver and are not exported to the upper layers.

In some cases it might be useful to signal interrupts to the application. Although this is not supported by the NIC, a driver implementer may integrate it by himself.

Regardless of the operation mode of the Internet Controller Chip and the corresponding driver, from TCP/IP's view, the Ethernet Interface must always be polled to receive packets.

Note: The actual Ethernet driver template uses the ISR-mode instead of the polling mode.

3 Ethernet Receive Ring Buffer (RXBuff)

In order to provide the NIC driver with a high data throughput, the sevenstax NIC driver template contains a receive buffer as a ringbuffer with FiFo behaviour.

The buffer handles frames with dynamic length. This enables the application f.e. to save multiple small Ethernet frames (like ARP requests) in the ringbuffer, although the buffer size e.g. equals the maximum size of only one typical Ethernet frame (1500 bytes).

sevenstaxTCP and the application processes these saved frames later asynchronously inside it's Tick-functions.

Note: There currently is no adequate TX(!) Ring Buffer available for the ethernet driver. But the need to install such a TX buffer is not really given, as sending ethernet frames is not a time critical matter.

3.1 Path of received data

To give the user an overview, here is the path of received Ethernet data on it's way to the application:

1. The Ethernet frame is received by the Ethernet Controller, which will assemble that serial bitstream in its own physical receive buffer.
2. An RX interrupt is generated by the Ethernet Controller. This will be served by the host microcontroller's *stxNIC_ISR()*. The ISR function transfers the Ethernet frame into the hosts Ethernet drivers RXBuff, if there is enough memory available.
3. Some more Ethernet frames might be received and handled by step1 and 2 inside the same ISR call, until all RX frames are delivered.
4. In between, the user application spends some calculation time to *stxTCP_Tick()* and inside that one to *stxETH_Tick()*, which will get a handle to the next Ethernet frame from the RXBuff via *stxNIC_RXBuff_Receive()* and will process it by TCP or ARP functions without copying these data.

5. Inside this processing, sevenstaxTCP will provide the application with the payload via its *NotifyHandler-function* (please refer to *sevenstaxTCP User Manual*). The application has to process or save that given data immediately, in order to return execution time as fast as possible to the *stxETH_Tick()*.
6. *stxETH_Tick()* then will release that Ethernet frame in the RXBuff ringbuffer, to free memory for the next received data and to enable use of the next waiting frame in RXBuff.

RXBuff location and size

As an Ethernet Receive Buffer, it physically is located in the IED (Ethernet.c). But because of the close Ethernet controller interaction inside the RX Interrupt Service Routine, all RXBuff functions are implemented in the NIC-Driver module.

The RXBuff size can be determined in `#define ETH_RX_BUFFSIZE` (Ethernet.h) and should under normal circumstances be bigger than one Ethernet frame (>1514 bytes). For your application, it might be increased for more performance, if enough RAM resources are available. Or it might be reduced to save resources, if there is no need to receive Ethernet broadcast traffic or bigger TCP/IP packets.

RXBuff internal behaviour

Note: This part is to complete the Ethernet documentation with internals only. The user of sevenstaxTCP does not have to know of this internal behaviour to get the benefits of it. But if the user intends to create his own ethernet driver for another ethernet controller chip, this might help.

Internally the RXBuff is organised as a linear word buffer, supported by a Write-Index (WRidx) and Read-Index (RDidx) and a frame length in front of any saved Ethernet frame. The WRidx will be incremented by the receive operation inside the *stxNIC_ISR()*. The RDidx will be incremented by the *stxNIC_RxBuff_Release()* inside the *stxETH_Tick()*. The WRidx will never pass the RDidx, to prevent overwriting unprocessed data. And – of course – the RDidx will never pass the WRidx.

An **empty RXBuff** is indicated by WRidx and RDidx == 0! This is the initial state. It will additionally be re-set, whenever the RDidx reaches the WRidx. A **complete full RXBuff** (no single byte anymore available) is indicated by WRidx == RDidx and both != 0. A **partially filled RXBuff** is indicated by any difference between WRidx and RDidx.

Any frame received by the *stxNIC_ISR()* will be saved inside the RXBuff behind the last saved frame (indicated by WRidx). If there is not enough memory to save the complete frame from WRidx up to buffer end, it will try to save that frame before the oldest received frame (WRidx wrap around). If there isn't enough memory for the complete frame too, the frame will be discarded and therefore is lost for the application and must be re-send by upper layer protocols (like TCP). Any frame saved in RXBuff is preceded by one word indicating the size of that Ethernet frame.

Note: Received frames will never be fragmented by RXBuff wrap arounds! This is essential, because sevenstaxTCP always assumes TCP packets in a linear and unfragmented manner.

Reading out frames from RXBuff is internally done by *stxNIC_RxBuff_Receive()* and *stxNIC_RxBuff_Release()*. *stxNIC_RxBuff_Receive()* checks the RXBuff content at RDidx, which always contains the size of the next frame in RXBuff. If the size is >0, the subsequent data will be passed as an handle (packet descriptor) to the application. If the size is 0, the RDidx will be re-set to RXBuff start.

After reading out that frame, the RXBuff must subsequently be supported by calling *stxNIC_RxBuff_Release()*, which releases that frame in RXBuff and increments RDidx to the next frame. This will automatically be done inside the *stxETH_Tick()*, the user does not have to care about that.

3.2 The Intermediate Ethernet Driver (IED)

The IED is the layer which resides above the NIC. sevenstax provides a default IED that is suitable for most use cases. It contains one receive and one transmit buffer, the corresponding packet descriptors, MAC addresses and so on.

The main task of an IED is to serve as an RX/TX interface for higher level protocols such as IP and ARP. It accepts packets for transmission (through packet descriptors) and dispatches incoming Ethernet frames to the appropriate protocol.

The IED has a few functions which will be discussed here:

BOOL_stx stxETH_Init (PROTOCOL_NOTIFY_HANDLER intfunc)

stxETH_Init() must be called by the application, at an early state, before sevenstax Embedded Protocols can use the LAN. It initializes all internal variables, packet descriptors and so on. Then, *stxETH_Init()* calls the lower layers initialization function through *stxNIC_Init()* (please see the previous chapter), which starts the Ethernet Controller Chip.

The parameter is function pointer to a dedicated application notify handler, which might be used e.g. to notify the application, if an RX interrupt arrived.

The *stxETH_Init()* returns 0 on failure and any other value on success.

void stxETH_Tick(void)

The Ethernet driver performs its periodic tasks here. sevenstaxARP calls it while idling in *stxARP_QueryAndWait()*. Also the user application (your software) should call *stxETH_Tick()* right before calling *stxTCP_Tick()*. Because this function is needed by sevenstaxARP, it must exist – possibly empty. *stxETH_Tick()* calls the lower layer's *stxNIC_Tick()*.

void stxETH_SendIPFrame(PACKET_DESC_FPTR_stx pkt)

This function is called by sevenstaxTCP, when it has build a packet for transmission and wants to shoot it out to the LAN. The packet descriptor for TX packets is allocated and initialized by the IED.

When sevenstaxTCP calls *stxETH_SendIPFrame()*, the packet descriptor's size member contains the size of the IP frame which sevenstaxTCP has build recently. The parameter (pkt) is normally a pointer to the default TX packet descriptor (This is the one already allocated). To perform the send operation, *stxETH_SendIPFrame()* calls the internal *stxETH_SendRawFrame()*.

stxETH_SendRawFrame(PACKET_DESC FPTR_stx pkt);

sevenstaxARP needs this function to send out raw Ethernet packets. The behaviour of *stxETH_SendRawFrame()* is similar to *stxETH_SendIPFrame()*, except that the packet descriptor's size member must match the length of a full Ethernet frame. *stxETH_SendRawFrame()* calls the lower layer's *stxNIC_Send()*.

3.3 The Packet Descriptor

The packet descriptor (see `pktdesc.h`) is used to hold information about a data packet and for easy passing of this information between internal functions (by reference).

```

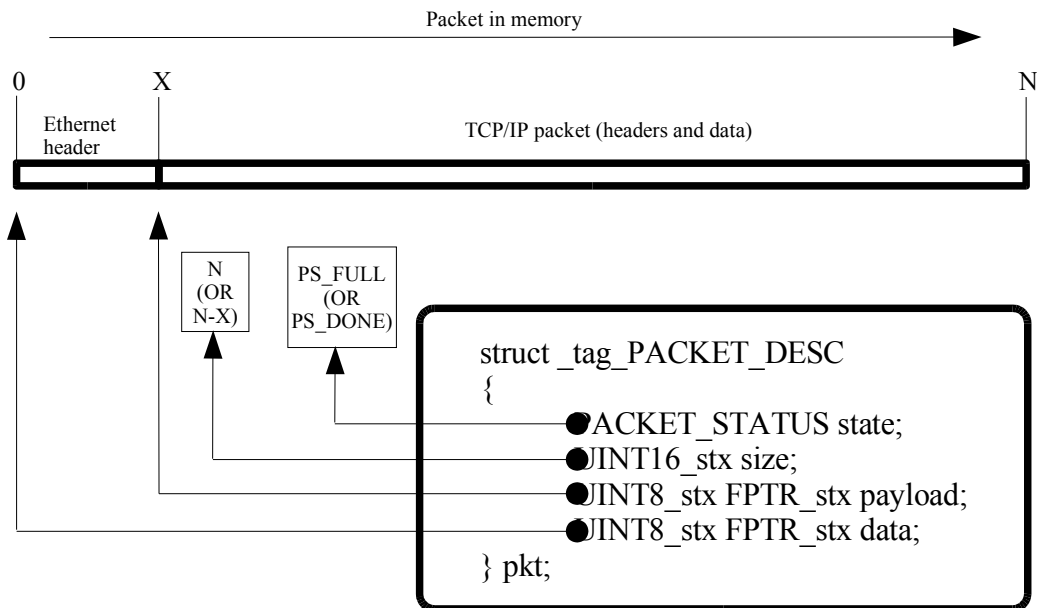
/* Status of a packet descriptor */
typedef enum
{
    PS_FULL,           /* New packet entirely received */
    PS_DONE           /* Packet worked up and ready to reuse */
} PACKET_STATUS;

/* 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 status member (`PACKET_STATUS state`) indicates whether the packet is in use (under construction) or ready for further processing. The member 'data' points to the location of the first byte in memory. The member 'size' specifies the length (in bytes) of the packet. The '*payload' member points to the first location after the PPP/Ethernet header. Packet descriptors are also used by sevenstaxPPP to deliver packets to higher network protocols (such as TCP).

Sometimes, only textual explanations may be hard to understand. Hence we show a drawing here which should remove contradictions on that.



As you can see the thin box represents an Ethernet frame containing an IP packet which resides in physical memory. The Ethernet frame packet begins at offset 0, the IP packet starts at offset X and the whole Ethernet frame ends at offset N.

The fat box is a Packet Descriptor which has been set up in order to reflect that packet.

As shown above, 'data' and 'payload' point to different locations where 'data' is always a pointer to offset 0 (which is the first address of the Ethernet header) and 'payload' always points to the beginning of data which is embedded into the Ethernet frame.

The 'size' member can be either the full length (N) or the length of the payload (N-X) depending on the use of the Packet Descriptor. When passing a TCP/IP packet (by a call to stxETH_SendIPFrame()), 'size' is N-X which is the full length of the TCP/IP packet but without any Ethernet header information.

Because Packet Descriptors can either be owned by a producer or consumer, the member 'state' denotes the owner. A producer owns a Packet Descriptor when 'state' is PS_DONE. PS_DONE means that the Packet Descriptor is free and its buffer (data) can be filled. When the producer has build a complete packet, it sets 'state' to PS_FULL which immediately transfers ownership to a consumer. The consumer now has access to the Packet Descriptor and can work on it. After the consumer has finished, it changes 'state' to PS_DONE which transfers control back again to the producer.

3.4 Conclusion

Dividing the network Interface into two parts, a hardware abstraction layer and an intermediate driver enables developers to change the underlying Ethernet hardware quickly and easily. This decoupling is an effective way to keep software as reusable as possible.

The author hopes that this document provides a little help for people who want to use sevenstax Embedded Protocols atop Ethernet and wishes always Good Speed!

4 Address Resolution Protocol (ARP)

4.1 Why ARP?

Ethernet uses another kind of addressing which is completely different from those of the Internet protocol. Packets on Ethernet start with so called 'MAC' addresses. These are values, six bytes in size, which uniquely identify network interface cards (NICs). The main difference is that MAC addresses are hardware addresses which are (usually but not always) assigned by the NIC's manufacturer, where IP addresses are virtual and can be, in most cases, arbitrarily selected by the user.

Please note: As MAC addresses have to be unique they are administrated by the IEEE where you can register and get your own pool of MAC-addresses. These addresses can be assigned to your products and there will never arise a conflict with hardware by other vendors. See <http://standards.ieee.org/oui/index.shtml> for details.

In order to run sevenstaxTCP on Ethernet, the local and remote MAC address must be kept somewhere during the connection. Usually it is no problem to query the local MAC address from the NIC but there is no way to get the remote MAC address without an additional protocol.

If another station connects to sevenstaxTCP, the remote MAC address will be extracted from the packet which carries the initial SYN sent by this station (because every packet on Ethernet contains the source and destination MAC addresses). On the other hand, when sevenstaxTCP acts as a client TCP (actively opens a connection), it must first call the ARP to get the MAC address of the remote station.

sevenstaxARP by default uses only one pair of IP/MAC address to reduce system requirements on small applications. This might be useful on systems, which in general do not have the need to contact multiple hosts or do not have to communicate across a gateway. This is configured in 'features.h' with a #define ARPCACHE value of '0'. In all other cases, the developer should use ARPCACHE values >1 to tailor the sevenstaxARP to his application requirements.

Please note: If your applications ethernet traffic has to pass a router/gateway, you MUST enable the ARP cache.

4.2 How does ARP work?

1. Suppose computer A (which IP address is 192.168.130.2) wants to talk to Computer B (which IP address is 192.168.130.155) using TCP/IP.
2. If computer A already knows the MAC address of computer B (because of a former ARP query) - Go to step 6.
3. If Computer A does not know the MAC address of computer B, it sends an ARP request to all computers (as an Ethernet broadcast): „Who has the MAC address of 192.163.130.155?“
4. Computer B receives the request and sends the answer back to computer A (as a directed Ethernet frame): I am 192.163.130.155 and my MAC address is 00-06-f3-72-aa-0d.

5. Computer A stores this information in a timed cache (ARPCACHE, associating a timestamp with it) where it lives approximately 20 minutes. This is done to reduce the number of ARP queries because too many broadcasts cause a heavy network load.
6. Computer A now knows the MAC address of computer B. It is now able to build Ethernet frames, put IP packets into it and send them to Computer B.

4.3 How does sevenstaxARP work?

Please note: sevenstaxARP, as a part of sevenstaxTCP, is seamlessly integrated and does not need any interaction with a user application.

In fact, `stxARP_Init()` is already included in `stxTCP_Init()` and is associated to the same callback procedure. And `stxARP_Tick()` is included in `stxTCP_Tick()`, to repeatedly hand over processor cycles. All the ARP handling is done automatically and is fully transparent to the application.

But: If an application wants to issue a manual ARP Query (which is normally not necessary), it may call the function '`stxARP_Query()`'. For a detailed discussion of sevenstaxARP's API functions, please look at *chapter 4.4 "sevenstaxARP Function Reference"*.

Although a user application does not have to take care of ARP (unless it has called `stxARP_Query()`), it receives notification events when sevenstaxARP gets active. For a detailed discussion of these events, please look at *chapter 4.5 "sevenstaxARP Notification Codes Reference"*.

The following shows an example of how to manually do an ARP query and connect to that IP address using sevenstaxTCP. The sample also shows how to initialize sevenstaxARP and the layout of a callback procedure:

```
// Flag to detect ARP response
BOOL_stx bARP_response = FALSE;

// This function receives all events from ARP and PPP
UINT32_stx handlerfunc (NOTIFY_CODE a, UINT32_stx b, UINT16_stx c)
{
    switch (a)
    {
        case NC_ARP_HOSTMESSAGE:
            if (b == ARP_OP_REPLY)
            {
                // Got ARP response
                // REMOTE_ETHERNET_ADDRESS is valid now
                bARP_response = TRUE;
            }
            break;

        case NC_ARP_TIMEOUT:
            // We got no response, TCP cannot connect
            // Do something to handle this error
            break;

        // Handle TCP related messages
        case NC_TCP_...
            ...
            break;
            ...
    } // switch(a)
    return 0;
}

// Here we go...
void main (void)
{
    IPV4 ip; // The IP address we want to connect to
```

```

ip.ipb.b1 = 192;           // This should be 192.168.130.68
ip.ipb.b2 = 168;           //      ''
ip.ipb.b3 = 130;           //      ''
ip.ipb.b4 = 68;           //      ''
// Set up our own IP address to 192.168.130.155
LOCAL_IP_ADDRESS.ipb.b1 = 192;
LOCAL_IP_ADDRESS.ipb.b2 = 168;
LOCAL_IP_ADDRESS.ipb.b3 = 130;
LOCAL_IP_ADDRESS.ipb.b4 = 155;
stxEth_Init();           // Start the NIC
stxTCP_Init(handlerfunc); // Init TCP + ARP, using the same handler

stxARP_Query(ip);       // Instruct ARP to ask for the remote MAC
while (1)
{
    stxEth_Tick();           // Do NIC related tasks
    stxTCP_Tick();           // Keep TCP + ARP alive
    // Wait for an ARP response ....
    if (bARP_response == TRUE)
    {
        // We're able to connect - now do it
        bARP_response = FALSE;
        TCP_Connect(ip, 80);
    }
    // Call routines elsewhere in your application
    .....
}
}

```

4.4 sevenstaxARP Function Reference

The following is a list of functions which are included in sevenstaxARP.

void stxARP_Init (PROTOCOL_NOTIFY_HANDLER h);

Description:

This function is already be called inside *stxTCP_Init()*. It sets up the internal state of sevenstaxARP and registers a callback procedure which is used to receive events from sevenstaxARP.

The handler procedure may be the same which is already used by sevenstaxTCP.

Arguments:

A function pointer (of type PROTOCOL_NOTIFY_HANDLER) to the callback procedure.

Return value:

None.

Notes:

Before calling this function, The Ethernet driver should already be up and running.

void stxARP_Tick (void);

Description:

stxARP_Tick() provides sevenstaxARP with processor time. It is already called inside the *stxTCP_Tick()* and keeps sevenstaxARP alive. If necessary,

it retries ARP queries and supports timeouts. If ARP_CACHE is enabled, it supports this too by calling the internal `stxARPC_Tick()`.

Arguments:

None.

Return value:

None.

Notes:

This function is mainly used to complete local ARP queries.

QWFLAG stxARP_Query (IPV4 FPTR_stx p);**Description:**

A call to this function instructs sevenstaxARP to query the MAC address of the corresponding IP address.

If ARP_CACHE is enabled, then sevenstaxARP first tries to find the IP in the cache. If it is not found/enabled, sevenstaxARP will send some ARP query packets over the network until the remote ARP answers. If no answer was detected during a short time (default is 2 seconds), sevenstaxARP will give up and notify the application of an unsuccessful attempt.

`stx_ARPQuery()` returns immediately.

Arguments:

A pointer to an IPV4 structure containing the IP address to query.

Return value:

Two of three possible values of an QWFLAG enumerated type.

QWF_START (0) ARP query in progress.

QWF_FOUND (1) MAC address already known. No queries needed.

Notes:

Regardless of the return value, the result is always posted to the application-defined callback procedure. Therefore, an application should rely on the notification messages instead of `stxARP_Query()`'s return value.

BOOL_stx stxARP_QueryAndWait (IPV4 FPTR_stx p);**Description:**

This is the blocking version of `stxARP_Query()`. It waits for an answer of the remote station and does not return until an answer arrives or the request timed out.

Arguments:

A pointer to an IPV4 structure containing the IP address to query.

Return value:

TRUE: The query succeeded.
FALSE: The query timed out.

Notes:

Like *stxARP_Query()*, the result is posted to the callback procedure.

BOOL_stx stxARP_ProcessPacket (PACKET_DESC FPTR_stx pkt);**Description:**

This function must be called from an Ethernet driver to deliver received ARP packets to sevenstaxARP.

Because sevenstaxARP does not look at the type of packet, the NIC driver must call *stxARP_ProcessPacket()* only when it has recognised the type of packet (which must be 0x0806 for ARP).

stxARP_ProcessPacket() returns TRUE_stx either if the packet was a valid ARP query or it was an answer to a query sevenstaxARP has sent before. Otherwise, *stxARP_ProcessPacket()* returns FALSE_stx either if the the IP address of the packet does not match the local one or the ARP opcode is an unknown value.

Arguments:

A pointer to a packet descriptor which describes an Ethernet frame.

Return value:

TRUE_stx: The packet was recognized an processed.
FALSE_stx: The packet was not accepted by sevenstaxARP.

Notes:

Usually, *stxARP_ProcessPacket()* should not be called from an application. Under normal situations, this function was called automatically by the sevenstax Ethernet Indermediate Driver.

void stxARP_FromEthPacket (IPV4 FPTR_stx ip, ETH_HEADER FPTR_stx ethh);**Description:**

To avoid needless ARP queries, this function reads the source MAC address from an already received Ethernet frame.

Arguments:

A pointer to an IPV4 structure containing an IP address and another pointer which points to an Ethernet frame which carries the relevant IP packet .

Return value:

none.

Notes:

`stxARP_FromEthPacket()` is for internal use only. It is called from `sevenstaxTCP` when a TCP/IP packet arrives which passes some reviews. A user application must never call it.

4.5 sevenstaxARP Notification Codes Reference

As a result of a call to `stxARP_Query()` or `stxARP_QueryAndWait()` and also on incoming ARP queries, `sevenstaxARP` posts one of these events to the application defined callback procedure:

NC_ARP_HOSTMESSAGE

Description:

Whenever `sevenstaxARP` receives a valid ARP message, it post an `NC_ARP_HOSTMESSAGE` event to the callback procedure.

Additional event parameters:

Depending on the kind of message, the 32bit value posted to the callback procedure can be one of:

ARP_OP_QUERY (1):

`sevenstaxARP` has received (and answered) a query from another station.

ARP_OP_REPLY (2):

`sevenstaxARP` has received an answer to a previously self-originated query .

The 16bit value is always zero (not used).

Return value from callback procedure:

The return value of the callback procedure is ignored.

Notes:

If the additional 32bit value is `ARP_OP_REPLY`, the resulting MAC address is stored in `REMOTE_MAC_ADDRESS`. This is the result of a successful call to `stxARP_Query()` or `stxARP_QueryAndWait()`.

NC_ARP_TIMEOUT

Description:

This event indicates a failure as a result to a call to `stxARP_Query()` or `stxARP_QueryAndWait()`.

`NC_ARP_TIMEOUT` is posted to the callback procedure when no answer arrives after several retries.

Additional event parameters:

Both values are zero and have no meanings.

Return value from callback procedure:

The return value of the callback procedure is ignored.

Notes:

If sevenstaxARP issues an NC_ARP_TIMEOUT, the remote MAC address could not be retrieved. Subsequent communication with the target (IP address) is impossible. This may occur because the remote station is not available on the network.

4.6 sevenstaxARP Externals Reference

Under normal configurations, this means when sevenstaxARP is used with sevenstaxTCP and atop of the Ethernet Intermediate Driver, all externals are automatically mapped to existing resources.

Of course, it is possible to unlink sevenstaxARP from its habitual environment. This section is for people who want to do so. The following is a list of external requirements which sevenstaxARP needs to function properly.

REMOTE_MAC_ADDRESS

Description:

This is a variable of type MAC_ADDRESS which receives the result of successful calls to stxARP_Query() and stxARP_QueryAndWait(). Whenever sevenstaxARP receives a valid ARP message, it posts an NC_ARP_HOSTMESSAGE event to the callback procedure.

Default location:

REMOTE_MAC_ADDRESS is #defined in the file 'tcpconnector.h' and is a substitution for gETH_remote_macaddress which lives in the Intermediate Ethernet Driver.

Notes:

This variable is set by sevenstaxARP. Write access from other modules is prohibited and may result in faulty behaviour of the entire system. The type MAC_ADDRESS is defined in the file 'ethernet.h' which belongs to the Intermediate Ethernet Driver.

LOCAL_MAC_ADDRESS

Description:

This is a variable of type MAC_ADDRESS which contains the individual address of the local Ethernet Controller Chip or network interface card (NIC).

Default location:

LOCAL_MAC_ADDRESS is #defined in the file 'tcpconnector.h' and is a substitution for gETH_local_macaddress which lives in the Intermediate Ethernet Driver.

Notes:

Under a normal configuration, this variable is set by the Intermediate Ethernet Driver. Write access from other modules is prohibited and may result in faulty behaviour of the entire system. The type `MAC_ADDRESS` is defined in the file 'ethernet.h' which belongs to the Intermediate Ethernet Driver.

LOCAL_IP_ADDRESS**Description:**

This is a variable of type `IPV4` which contains the local IP address. It is stored in Big Endian order, in other words, the same format as it is transmitted over the network.

Default location:

`LOCAL_IP_ADDRESS` is #defined in the file 'tcpconnector.h' and is a substitution for `gTCP_my_ipaddress` which lives in `sevenstaxTCP` (tcp.h).

Notes:

`LOCAL_IP_ADDRESS` is considered read only. Modules **must not change** `LOCAL_IP_ADDRESS` unless they are network control protocols such as ARP or BOOTP. Exception: An application may set `LOCAL_IP_ADDRESS` at startup to a static value.

stxETH_SendRawFrame(PACKET_DESC FPTR_stx pkt)**Description:**

This is the send function by what `sevenstaxARP` sends of its messages to the LAN. Because `sevenstaxARP` always constructs full Ethernet frames beginning with the destination MAC address, a send function must accept such packets.

Default location:

`stxETH_SendRawFrame()` lives in the default Intermediate Ethernet Driver 'ethernet.c'. It directly transfers the packets without any changes to the Ethernet Hardware Driver for immediate transmission.

Notes:

There is currently no connector defined in 'tcpconnector.h'.

GET_COUNTER_VALUE**Description:**

Using `GET_COUNTER_VALUE`, `sevenstaxARP` reads a 32bit free running millisecond counter for time measurement. `GET_COUNTER_VALUE` is an alias (c-macro) which may substitute either a function call, a global variable or a timer register.

Default location:

GET_COUNTER_VALUE is implemented as a macro in 'features.h'. In the PC version, it is an alias for the Win32 API function 'GetTickCount()'. In most Infineon C16x ports, it is a variable which is incremented by a timer interrupt.

Notes:

It is not necessary for the timer to run with an accuracy of 1ms. An accuracy of 1/10s should be enough. The only requirement is that GET_COUNTER_VALUE must produce the difference between two successive queries in milliseconds.

5 Bootstrap Protocol (BOOTP)

5.1 Why BOOTP?

LANs make it possible to use diskless hosts as workstations, routers, terminal concentrators and so on. Diskless hosts require a mechanism to boot remotely over a network. The BOOTP protocol is used for remote booting over IP networks. It allows a minimum IP protocol stack with no configuration information, typically stored in ROM, to obtain enough information to begin the process of downloading the necessary boot code.

5.2 How does BOOTP work?

The BOOTP process involves the following steps:

1. The client determines its own hardware address; this is normally in a ROM on the hardware.
2. A BOOTP client sends its hardware address in a UDP datagram to the server. If the client knows its IP address and/or the address of the server, it should use them, but in general BOOTP clients have no IP configuration data at all. If the client does not know its own IP address, it will use 0.0.0.0. If the client does not know the server's IP address, it will use the limited broadcast address (255.255.255.255). The UDP port number is 67.
3. The server receives the datagram and looks up the hardware address of the client in its configuration file, which contains the client's IP address. The server fills in the remaining fields in the UDP datagram and returns it to the client using UDP port 68. One of three methods may be used to do this:
 - If the client knew its own IP address (it was included in the BOOTP request), then the server will return the datagram directly to this address. It is likely that the ARP cache in the server's protocol stack will not know the hardware address matching the IP address. ARP will be used to determine it .
 - If the client did not know its own IP address (it sends 0.0.0.0 in the BOOTP request) the server will have to concern itself with its own ARP cache. ARP on the server cannot be used to find the hardware address of the client because the client does not know its IP address and so cannot reply to an ARP request. This is called the "chicken and egg" problem. There are two possible solutions:
 - a) If the server has a mechanism for directly updating its own ARP cache without using ARP itself, it will send the datagram directly.
 - b) If the server cannot update its own ARP cache, it will send a broadcast reply.
4. When it receives the reply, the BOOTP client will record its own IP address (allowing it to respond to ARP requests) and begin the bootstrap process

5.3 How does the sevenstaxBOOTP client work?

The sevenstaxBOOTP client is a special embedded version of BOOTP which leaves out any advanced features to keep it small and simple. In fact, sevenstaxBOOTP can only be used to gain an IP address from a BOOTP server.

An application using sevenstaxBOOTP must do these steps to get an IP address from any BOOTP server.

1. Initialize and run TCP/IP.
 stxBOOTP_Init() is already called inside stxTCP_Init() to initialize sevenstaxBOOTP and supplies some memory and the address of a callback procedure.
2. Call stxBOOTP_Start() to let sevenstaxBOOTP look for an IP address.
3. stxBOOTP_Tick() is already included inside stxTCP_Tick(), which keeps alive sevenstaxBOOTP processing.
4. Wait for an NC_BOOTP_XXX event, which was posted to the callback procedure. If this is NC_BOOTP_SUCCESS then an IP address was assigned.

The following is a simple example of how sevenstaxBOOTP can be used:

```

BOOL_stx bootp_success;
BOOL_stx bootp_error;

// This function receives all events from BOOTP and other protocols
UINT32_stx handlerfunc (NOTIFY_CODE a, UIN32_stx b, UINT16_stx c)
{
    switch (a)
    {
        case NC_BOOTP_SUCCESS:
            bootp_flag = TRUE;
            break;

        case NC_BOOTP_FAILURE:
            bootp_error = TRUE;
            break;

        // Handle TCP related messages
        case NC_TCP_...
            break;
    } // switch(a)
    return 0;
}

// Here we go...
void main (void)
{
    data BOOTP_PACKET packet; // This the memory which holds our BOOTP TX

    stxETH_Init(); // Start the NIC
    stxTCP_Init(handlerfunc); // Init TCP, ARP and BOOTP, using the same handler

    bootp_success = FALSE;
    bootp_error = FALSE;

    stxBOOTP_Start() // Try to get an IP address
    while(1)
    {
        stxETH_Tick(); // Do NIC related tasks
        stxTCP_Tick(); // Keep TCP, ARP and BOOTP alive
        if (bootp_success == TRUE) // OK, we have an IP address
        {
            // Call routines elsewhere in your application
            ....
        }
        if (bootp_error == TRUE)
        {
            // Call error handler

```

```
        }  
        . . . . .  
    }  
}
```

5.4 sevenstaxBOOTP Function Reference

The following is a list of functions which are included in sevenstaxBOOTP.

void stxBOOTP_Init (PROTOCOL_NOTIFY_HANDLER handler, void FPTR_stx memory)

Description:

This function is already called inside stxTCP_Init() and must be called before calling any other function of sevenstaxBOOTP. It sets up the internal state of sevenstaxBOOTP and registers a callback procedure which is used to receive events from sevenstaxBOOTP.

Because notify codes are unique values, the handler procedure may be the same used by sevenstaxTCP and other protocols.

Arguments:

A function pointer (of type PROTOCOL_NOTIFY_HANDLER) to the callback procedure.

A pointer to the memory location where sevenstaxBOOTP can build its BOOTP packet for transmission. There must be at least 300 bytes available at this address.

Return value:

None.

Notes:

After a call to stxBOOTP_Init(), the local IP address (referenced through the alias LOCAL_IP_ADDRESS) is set to 0.0.0.0

void stxBOOTP_Tick (void)

Description:

stxBOOTP_Tick() provides sevenstaxBOOTP with processor time.

It must be called fast and repeatedly to keep sevenstaxBOOTP alive. Because it does the same as stxTCP_Tick() does for sevenstaxTCP, this function can be called from the same location of your application from which stxTCP_Tick() is called.

Arguments:

None.

Return value:

None.

Notes:

stxBOOTP_Tick also does all the background processing for sevenstaxBOOTP.

void stxBOOTP_Start (void)**Description:**

This instructs sevenstaxBOOTP to ask for an IP address.

stxBOOTP_Start() should be called once shortly after sevenstaxTCP is running. The application-defined callback procedure receives NC_BOOTP_SUCCESS if an IP address was assigned or NC_BOOTP_FAILURE if nothing or an error happens after some retries.

Arguments:

None.

Return value:

None.

Notes:

This function returns immediately. Subsequent processing is performed by stxBOOTP_Tick(). Results are posted to the callback procedure.

void stxBOOTP_Receive (UINT8_stx FPTR_stx data);**Description:**

In order to transfer received data to sevenstaxBOOTP, this function must be called from the UDP receiver, when data arrives on UDP port 68.

Arguments:

A pointer to the received data. A length information is not needed because sevenstaxBOOTP expects a full-length BOOTP message.

Return value:

None.

Notes:

When sevenstaxBOOTP is used in conjunction with sevenstaxTCP, then stxBOOTP_Receive() is called automatically from within TCP.

5.5 sevenstaxBOOTP Notification Codes Reference

As a result of a call to stxBOOTP_Start(), sevenstaxBOOTP posts one of these events to the application defined callback procedure:

NC_BOOTP_SUCCESS

Description:

If sevenstaxBOOTP has received an IP address from a BOOTP server, it posts NC_BOOTP_SUCCESS to the application-defined callback procedure.

Additional event parameters:

The 32bit value contains the new IP address in Big Endian order.

The 16bit value is not used and always zero.

Return value from callback procedure:

The return value is ignored.

Notes:

sevenstaxBOOTP automatically sets the global variable LOCAL_IP_ADDRESS to the newly assigned one.

NC_BOOTP_FAILURE

Description:

When stxBOOTP_Start() is called, sevenstaxBOOTP repeatedly sends queries for an IP address. If none was answered, an NC_BOOTP_FAILURE event is posted to the application-defined callback procedure indicating an error.

Additional event parameters:

The 32bits value is not used and contains always zero.

The 16bits value is not used and always zero.

Return value from callback procedure:

The return value of the callback procedure is ignored.

Notes:

An error occurs most likely if there's no BOOTP server on the net or the network connection is broken.

BOOTP is a good cause if the server is online. An embedded system with no user interface should have some backup capacities when BOOTP does not work e.g. a default static IP address or something like that.

5.6 sevenstaxBOOTP Externals Reference

Under normal configurations, this means when sevenstaxBOOTP is used with sevenstaxTCP, all externals are automatically mapped to existing resources.

Of course, it is possible to unlink sevenstaxBOOTP from its habitual environment. This section is for people who want to do so. The following is a list of external requirements which sevenstaxBOOTP needs to function properly.

LOCAL_MAC_ADDRESS

Description:

This is a variable of type `MAC_ADDRESS` which contains the individual address of the local Ethernet Controller Chip or network interface card (NIC). `sevenstaxBOOTP` needs the local MAC address to identify itself when it sends requests to a BOOTP server.

Default location:

`LOCAL_MAC_ADDRESS` is `#defined` in the file `'tcpconnector.h'` and is a substitution for `gETH_local_macaddress` which lives in the Intermediate Ethernet Driver.

Notes:

Under a normal configuration, this variable is set by the Intermediate Ethernet Driver. Write access from other modules is prohibited and may result in faulty behaviour of the entire system. The type `MAC_ADDRESS` is defined in the file `'ethernet.h'` which belongs to the Intermediate Ethernet Driver.

LOCAL_IP_ADDRESS

Description:

This is a variable of type `IPV4` which contains the local IP address. It is stored in Big Endian order, in other words, the same format as it is transmitted over the network.

Default location:

`LOCAL_IP_ADDRESS` is `#defined` in the file `'tcpconnector.h'` and is a substitution for `gTCP_my_ipaddress` which lives in `sevenstaxTCP`.

Notes:

This variable is cleared to all zeros by a call to `stxBOOTP_Init()`. If a BOOTP server responds, `LOCAL_IP_ADDRESS` is set to the new value offered by that server. Other modules **must not change** `LOCAL_IP_ADDRESS` unless they are network control protocols such as DHCP or BOOTP. Exception: An application may set `LOCAL_IP_ADDRESS` at start-up to a static value.

```
UDP_SEND (  
    UINT16_stx my_port,  
    IPV4_FPTR_stx dest_ip,  
    UINT16_stx remote_port,  
    UINT8_stx FPTR_stx data,  
    UINT16_stx datasize);
```

Description:

This is the send function by what `sevenstaxBOOTP` sends UDP datagrams to BOOTP servers. `UDP_SEND` is actually an alias to `stxTCP_InternalSendUDP()`.

UDP_SEND needs five parameters which are:

- my_port, this is the local port number.
- dest_ip, a pointer to a variable (IPv4 structure) containing the server's IP address.
- remote_port, the server's port number.
- data, a pointer to the data which should be send as UDP datagram.
- datasize, the length of the UDP datagram where 'data' points to.

Default location:

stxTCP_InternalSendUDP() lives in sevenstaxTCP (tcp.c). It builds and sends UDP datagrams out to the net.

Notes:

To make UDP_SEND an alias to your own function, you must modify the appropriate entries in 'tcpconnector.h'. These are #define UDP_SEND... and the line below (the function prototype), which must reflect your own UDP send function.

GET_COUNTER_VALUE

Description:

Using GET_COUNTER_VALUE, sevenstaxBOOTP reads a 32bit free running millisecond counter for time measurement. GET_COINTER_VALUE is an alias (c-macro) which may substitute either a function call, a global variable or a timer register.

Default location:

GET_COUNTER_VALUE is implemented as a macro in 'features.h'. In the PC version, it is an alias for the Win32 API function 'GetTickCount()'. In most Infineon C16x ports, it is a variable which is incremented by a timer interrupt.

Notes:

It is not necessary for the timer to run with an accuracy of 1ms. An accuracy of 1/10s should be enough. The only requirement is that GET_COUNTER_VALUE must produce the difference between two successive queries in milliseconds.

6 Dynamic Host Configuration Protocol (DHCP)

6.1 Why DHCP?

LAN configuration needs permanent support to prevent conflicts while adding or removing workstations. To reduce the maintenance costs (and to make the administrators life easier), DHCP was created. DHCP f.e. automatically provides clients connected to the LAN with leased IP addresses from an IP address pool. It enables a simpler and central network configuration.

6.2 How does DHCP work?

When a DHCP client is first switched on, it sends a broadcast packet on the network with a DHCP request. This is picked up by a DHCP server, which allocates an IP address to the PC, from one of the scopes (the pools of addresses) available. Each DHCP scope is used for a different TCP/ IP network segment.

On networks with routers that support DHCP, extra information is added to the request by the router to tell the server which network the request came from. The DHCP server uses this information to pick an address from the correct scope. The server replies to the client, allocating the TCP/ IP address and required settings.

However, DHCP doesn't allocate the address permanently. It tells the client that it has "leased" the address for a specific time period which you as administrator can control. By default DHCP is installed with a three day lease period. When the lease expires, the client can ask the server to renew the lease. If the DHCP server doesn't hear from the client beyond the expiry of the lease period, it will put that address back in the pool ready to be re-used.

6.3 How does the sevenstaxDHCP client work

The sevenstaxDHCP client is a special embedded version of DHCP which leaves out advanced features to keep it small and simple.

An application using sevenstaxDHCP must do these steps to get an IP address from any DHCP server.

1. Initialise and run TCP/IP.
2. *stxDHCP_Init()* is called inside *stxTCP_Init()* to initialise sevenstaxDHCP and supply some memory and the address of a callback procedure.
3. Call *stxDHCP_Start()* to let sevenstaxDHCP look for an IP address.
4. *stxDHCP_Tick()* is repeatedly called inside *stxTCP_Tick()* which keeps alive sevenstaxDHCP processing.
5. Wait for an NC_DHCP_XXX event, which was posted to the callback procedure. If this is NC_DHCP_SUCCESS an IP address has been assigned.

The application does not have to take care of the lease period. sevenstaxDHCP automatically asks the server to renew the lease before it expires.

The following is a simple example of how sevenstaxDHCP can be used:

```

BOOL_stx dhcp_success;
BOOL_stx dhcp_error;

// This function receives all events from DHCP and other protocols
UINT32_stx handlerfunc (NOTIFY_CODE a, UINT32_stx b, UINT16_stx c)
{
    switch (a)
    {
        case NC_DHCP_SUCCESS:
            dhcp_flag = TRUE;
            break;

        case NC_DHCP_FAILURE:
            dhcp_error = TRUE;
            break;

        // Handle TCP related messages
        case NC_TCP_...
            ....
            break;
            ....
    } // switch(a)
    return 0;
}

// Here we go....
void main (void)
{
    stxEth_Init(); // Start the NIC
    stxTCP_Init(handlerfunc); // Init TCP, ARP, DHCP using the same handler
    procedure
    dhcp_success = FALSE;
    dhcp_error = FALSE;
    stxDCHP_Start() // Try to get an IP address
    while(1)
    {
        stxEth_Tick(); // Do NIC related tasks
        stxTCP_Tick(); // Keep TCP, ARP, DHCP alive
        if ( dhcp_success == TRUE) // OK, we have an IP address
        {
            // Call routines elsewhere in your application
            ....
        }
        if (dhcp_error == TRUE)
        {
            // Call error handler
            ....
        }
    }
}

```

6.4 sevenstaxDHCP Function Reference

The following is a list of functions which are included in sevenstaxDHCP.

***void stxDHCP_Init (PROTOCOL_NOTIFY_HANDLER handler,
void FPTR_stx memory);***

Description:

This function is called inside stxTCP_Init(), which must be done before calling any other function of sevenstaxDHCP. It sets up the internal state of sevenstaxDHCP and registers a callback procedure which is used to receive events from sevenstaxDHCP.

Because notify codes are unique values, the handler procedure may be the same used by sevenstaxTCP.

Arguments:

A function pointer (of type `PROTOCOL_NOTIFY_HANDLER`) to the callback procedure.

A pointer to the memory location where sevenstaxDHCP can build its DHCP packet for transmission. There must be at least 548 bytes available at this address.

Return value:

None.

Notes:

After a call to `stxDHCP_Init`, the local IP address (referenced through the alias `LOCAL_IP_ADDRESS`) is set to 0.0.0.0

void stxDHCP_Tick (void);**Description:**

`stxDHCP_Tick()` provides sevenstaxDHCP with processor time.

It is already be called inside `stxTCP_Tick()` to keep sevenstaxDHCP alive.

Arguments:

None.

Return value:

None.

Notes:

`stxDHCP_Tick` also does all the background processing for sevenstaxDHCP.

void stxDHCP_Start (void);**Description:**

This instructs sevenstaxDHCP to ask for an IP address.

`stxDHCP_Start()` should be called once shortly after sevenstaxTCP is running. The application-defined callback procedure receives `NC_DHCP_SUCCESS` if an IP address was assigned or `NC_DHCP_FAILURE` if nothing happens after some retries.

Arguments:

None.

Return value:

None.

Notes:

This function returns immediately. Subsequent processing is performed by `stxDHCP_Tick()`. Results are posted to the callback procedure.

In order to pass back the IP address to the server, `stxDHCP_Release()` must be called.

void stxDHCP_Release (void);

Description:

This function gives back an IP address to the DHCP server when it is no longer needed.

Arguments:

None.

Return value:

None.

Notes:

`stxDHCP_Release` should be called, before the device disconnects from the network. This ensures that the DHCP server can add the IP address to its free pool again.

void stxDHCP_Receive (UINT8_stx FPTR_stx data);

Description:

In order to transfer received data to `sevenstaxDHCP`, this function must be called from the UDP receiver, when data arrives on UDP port 68.

Arguments:

A pointer to the received data. A length information is not needed because `sevenstaxDHCP` expects a full-length DHCP packet.

Return value:

None.

Notes:

When `sevenstaxDHCP` is used in conjunction with `sevenstaxTCP`, then `stxDHCP_Receive` is called automatically from within TCP.

6.5 sevenstaxDHCP Notification Codes Reference

As a result of a call to `stxDHCP_Start()`, `sevenstaxDHCP` posts one of these events to the application defined callback procedure:

NC_DHCP_SUCCESS

Description:

If sevenstaxDHCP has received an IP address from a DHCP server, it posts NC_DHCP_SUCCESS to the application-defined callback procedure.

Additional event parameters:

The 32bit value contains the new IP address in Big Endian order.

The 16bit value is not used and always zero.

Return value from callback procedure:

The return value is ignored.

Notes:

sevenstaxDHCP automatically sets the global variable LOCAL_IP_ADDRESS to the newly assigned one.

NC_DHCP_FAILURE

Description:

This event means a serious problem. If any error occurs, sevenstaxDHCP will post NC_DHCP_FAILURE to the application-defined callback procedure.

Additional event parameters:

The 32bits value contains one of these values:

- DHCPE_NOT_RESPONDING (0)

There was no response from the DHCP server. This likely happens if there is no DHCP server on the network or if there is something wrong with the network connection (hardware/cables etc.).

- DHCPE_LEASETIME_TOO_LOW (1)

sevenstaxDHCP does not accept lease times smaller than 50 seconds. This mostly occurs if the DHCP server is misconfigured. To correct this error, configure the DHCP server to submit lease times \geq 50 seconds.

- DHCPE_NACK (2)

If the DHCP server responds with a DHCP NACK, sevenstaxDHCP will post this event to the application-defined callback procedure. A DHCP NACK occurs if the IP address cannot be reassigned for any reason.

The 16bits value is not used and always zero.

Return value from callback procedure:

The return value of the callback procedure is ignored.

Notes:

If an error occurs, especially DHCPE_NACK or DHCPE_LEASE-TIME_TOO_LOW, the client should restart sevenstaxDHCP by calling stxDHCP_Init() and stxDHCP_Start() again to get an IP address from another DHCP server.

6.6 sevenstaxDHCP Externals Reference

Under normal configurations, this means when sevenstaxDHCP is used with sevenstaxTCP, all externals are automatically mapped to existing resources.

Of course, it is possible to unlink sevenstaxDHCP from its habitual environment. This section is for people who want to do so. The following is a list of external requirements which sevenstaxDHCP needs to function properly.

LOCAL_MAC_ADDRESS

Description:

This is a variable of type MAC_ADDRESS which contains the individual address of the local Ethernet Controller Chip or network interface card (NIC). sevenstaxDHCP needs the local MAC address to identify itself when it sends requests to a DHCP server.

Default location:

LOCAL_MAC_ADDRESS is #defined in the file 'tcpconnector.h' and is a substitution for gETH_local_mac-address placed in the Intermediate Ethernet Driver.

Notes:

Under a normal configuration, this variable is set by the Intermediate Ethernet Driver. Write access from other modules is prohibited and may result in faulty behaviour of the entire system. The type MAC_ADDRESS is defined in the file 'ethernet.h' which belongs to the Intermediate Ethernet Driver.

LOCAL_IP_ADDRESS

Description:

This is a variable of type IPV4 which contains the local IP address. It is stored in Big Endian order, in other words, the same format as it is transmitted over the network.

Default location:

LOCAL_IP_ADDRESS is #defined in the file 'tcpconnector.h' and is a substitution for gTCP_my_ipaddress which lives in sevenstaxTCP (tcp.h).

Notes:

This variable is cleared to all zeros by a call to stxDHCP_Init(). If a DHCP server responds, LOCAL_IP_ADDRESS is set to the new value offered by

that server. Other modules **must not change** LOCAL_IP_ADDRESS unless they are network control protocols such as DHCP or BOOTP. Exception: An application may set LOCAL_IP_ADDRESS at start-up to a static value.

```
UDP_SEND (  
    UINT16_stx my_port,  
    IPV4_FPTR_stx dest_ip,  
    UINT16_stx remote_port,  
    UINT8_stx FPTR_stx data,  
    UINT16_stx datasize);
```

Description:

This is the send function by what sevenstaxDHCP sends UDP datagrams to DHCP servers. UDP_SEND is actually an alias to stxTCP_InternalSendUDP ().

UDP_SEND needs five parameters which are:

- my_port, this is the local port number.
- dest_ip, a pointer to a variable (IPV4 structure) containing the server's IP address.
- remote_port, the server's port number.
- data, a pointer to the data which should be send as UDP datagram.
- datasize, the length of the UDP datagram where 'data' points to.

Default location:

stxTCP_InternalSendUDP() lives in sevenstaxTCP (tcp.c). It builds and sends UDP datagrams out to the net.

Notes:

To make UDP_SEND an alias to your own function, you must modify the appropriate entries in 'tcpconnector.h'. These are #define UDP_SEND... and the line below (the function prototype), which must reflect your own UDP send function.

GET_COUNTER_VALUE

Description:

Using GET_COUNTER_VALUE, sevenstaxDHCP reads a 32bit free running millisecond counter for time measurement. GET_COINTER_VALUE is an alias (c-macro) which may substitute either a function call, a global variable or a timer register.

Default location:

GET_COUNTER_VALUE is implemented as a macro in 'features.h'. In the PC version, it is an alias for the Win32 API function 'GetTickCount()'. In most Infineon C16x ports, it is a variable which is incremented by a timer interrupt.

Notes:

It is not necessary for the timer to run with an accuracy of 1ms. An accuracy of 1/10s should be enough. The only requirement is that GET_COUNTER_VALUE must produce the difference between two successive queries in milliseconds.

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) 2006 by sevenstax GmbH

Change History

<i>Ver.</i>	<i>Date</i>	<i>by</i>	<i>Change description</i>
0.1	02/06/03	pe	Base version
1.0	19/06/03	krf	Review
2.0	27/05/04	krz	New Revision including BOOTP, DHCP and ARP
2.4	29/06/04	krz	Format Review
2.5	30/06/04	krf	Proof-read
2.6	15/08/04	krz	Ethernet RX buffering, Stack Version 3 reviewing
2.7	19/08/04	ck	Proof-read
2.8	19/08/04	krz	Released
3.3	07/05/06	krz	Reviewed for Ethernet Version 3.3