A foundation for a sound TCP/IP implementation under MS-DOS
Waucomont, M.J.P.H.

Published: 01/01/1993

Document Version
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the author’s version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain.
- You may freely distribute the URL identifying the publication in the public portal.

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 15. Dec. 2018
A Foundation for a Sound TCP/IP Implementation under MS-DOS

by M. J. P. H. Waucomont

Supervisor: Prof. dr. ir. C. J. Koomen
Coach: Drs. C. G. S. M. Braam

The Department of Electrical Engineering of the Eindhoven University of Technology does not accept any responsibility regarding the contents of student projects and graduation projects.
Abstract

This report provides a summary of TCP/IP and Ethernet. It also describes the implementation of a C interface to a given Data Link Layer. The research done and the code written form a solid basis for writing a complete TCP/IP library for MS-DOS machines.

The underlying report is short because much care has been taken to make it concise and clear.

"Je n'ai fait cette lettre-ci plus longue que parce que je n'ai pas eu le loisir de la faire plus courte"
— Blaise Pascal

Acknowledgements

First of all I would like to thank my mentors. Drs. Carel Braam has been a tremendous source of knowledge and inspiration. He has spent a lot of time and effort on helping me to produce the material presented in this document and managed to bear my quirks during the development period. Prof. dr. ir. C. J. Koomen made it all possible by allowing me to work on this project.

Harry Stox has shared his experiences with KA9Q with me. Bruce Watson provided some style hints. Numerous people in the Computer Centre of the Eindhoven University of Technology have helped me by giving me the documentation and equipment needed to develop the software.
# Table of Contents

1 Introduction
   1.1 The project ........................................... 1
   1.2 The Roots of TCP/IP .................................... 1
   1.3 TCP/IP and the ISO-OSI Reference Model ................. 2

2 The Ethernet Physical Layer and Data Link Layer
   2.1 An Introduction to Ethernet ............................ 4
   2.2 The Ethernet Physical Layer ............................ 4
   2.3 The Ethernet Data Link Layer ........................... 6
   2.4 The Clarkson Packet Drivers
       An Implementation of the Data Link Layer .......... 7
   2.5 Token Ring
       An Alternative Physical Layer ........................ 8

3 The Interface to the Packet Driver
   —Basis of the IP Network Layer
   3.1 Introduction ........................................... 10
   3.2 Assembly glue ......................................... 11
   3.3 C Interface to Packet Driver Functions ............... 11
   3.4 Buffering and Utilities ............................... 12
   3.5 Test Suites ........................................... 13

4 The C Code Revealed
   —Insight in the Inside
   4.1 Introduction ........................................... 14
   4.2 C Interface to Packet Driver Functions ............... 14
   4.3 Buffering .............................................. 16

5 Epilogue .................................................. 18

References ............................................... 19

Appendix C Sources ...................................... 20
1 Introduction

1.1 The Project

The original assignment was to write a compact implementation of a TCP/IP library for the PC. There are a few public domain implementations of which WAT-TCP from the University of Waterloo and KA9Q from Phil Karn are the most well known. WAT-TCP has been used in a previous project at the RC (Rekencentrum) of the EUT (Eindhoven University of Technology) and turned out to be too bulky and unreliable. KA9Q seems to be better, but is even larger and the proprietary memory manager—included to circumvent the cumbersome Intel 64K segmentation—does not work properly. This called for a new, more comprehensive library.

Writing a complete TCP/IP library in a 200 hour period turned out to be an impossible job. For a major part this was due to the lack of support MS-DOS offers programmers in interfacing with the hardware. Any decent operating system should map physical devices to abstract ones. MS-DOS only provides handles to the hardware.

Not knowing this I started doing a lot of research on TCP/IP and Ethernet. Then I found out that the C interface sample code that comes with the Clarkson Packet Driver was rather useless, so I had to master the Packet Driver specification, in order to write a decent C interface to the Packet Driver myself, thus completing the Data Link Layer. By now I understand why Phil Karn is working in KA9Q for five years already.

1.2 The Roots of TCP/IP

In the early 1980s a new family of protocols was specified for ARPANET, the computer network sponsored by the Advanced Research Projects Agency. The official name for this family of protocols is "DARPA Internet protocol suite", but it is commonly referred to as the TCP/IP protocol suite, or just TCP/IP. The TCP/IP protocols are not only used by Internet to connect more than 150,000 computers worldwide, but also on a much smaller scale, for example to let two personal computers communicate. In many cases the computers will be connected by an Ethernet.
1.3 TCP/IP and the ISO-OSI Reference Model

Because of the wide variety of problems that arise when machines communicate over a data network—hardware failure, network congestion, packet delay or loss, data corruption, data duplication and sequence errors—it is advisable to split the communication problem into subproblems small enough to deal with. The ISO-OSI seven layer reference model gives an idea of how to partition the problem into subproblems. Each layer handles a part of the communication problem.

The TCP/IP protocols do not comply with the reference model, but comparing the two gives insight in the structure of TCP/IP. That is what a reference model is meant for. Figure 1.1 shows the relationship between the TCP/IP protocols along with their approximate mapping into the OSI model. Note that there are more members in the family, but since most Internet applications use TCP and TCP uses IP, the entire Internet protocol suite is often called TCP/IP.

The Physical Layer and Data Link Layer are treated in the next chapter. The abbreviations for the members of the TCP/IP protocol family are best explained by giving a short explanation of what each protocol does. Details will be treated later, as the project evolves.

<table>
<thead>
<tr>
<th>7</th>
<th>Application</th>
<th>User Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Presentation</td>
<td>TCP</td>
</tr>
<tr>
<td>5</td>
<td>Session</td>
<td>UDP</td>
</tr>
<tr>
<td>4</td>
<td>Transport</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Network</td>
<td>RARP</td>
</tr>
<tr>
<td>2</td>
<td>Data Link</td>
<td>BOOTP</td>
</tr>
<tr>
<td>1</td>
<td>Physical</td>
<td>IP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ICMP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ARP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Clarkson Packet Drivers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ethernet Hardware</td>
</tr>
</tbody>
</table>

Fig. 1.1 Layering in the Internet protocol suite

TCP—Transmission Control Protocol—A connection is a communication link between two hosts. It is established when a request for it of one host is granted by the other. It can be terminated by each of the hosts or because of a
hardware failure. As long as the connection exists communication is reliable. Reliability is guaranteed by using an ACK-NAK protocol and time-outs. Thus TCP is a connection oriented protocol that provides a reliable, full duplex, byte stream for a user process.

UDP -User Datagram Protocol- is a connectionless protocol for user processes. In contrast with the TCP protocol there is no verification that UDP datagrams ever reach their destination. There are also no retries. The user process has to take care of this itself.

IP -Internet Protocol- IP is the protocol that provides the packet delivery service. There are three IP package types: IP Datagram -used for TCP, UDP, ICMP and BOOTP,- ARP and RARP.

ICMP -Internet Control Message Protocol- is the protocol to handle error and control information between gateways and hosts. Although ICMP messages are encapsulated in normal IP datagrams, these messages are generated and processed by the TCP/IP networking software itself.

ARP -Address Resolution Protocol- Each host on the Internet is assigned a unique 32 bit address that is used in all communication with that host. A host connected to a physical network -e.g. Ethernet- has a hardware address which in general has another format than an internet address. Let I be the collection of internet addresses and P be the collection of hardware addresses in the physical network of the host. ARP is the function f : I → P. ARP relies on the existence of dedicated hosts that maintain mapping tables realizing that function. It is used to avoid that each host has to maintain a complete table of mappings. When host A wants to resolve the Internet address of host B, it broadcasts a special packet requesting to supply a hardware address where packets for host B have to be sent to.

RARP -Reverse Address Resolution Protocol- the function f⁻¹: P → I. There do exist machines that only know their own hardware address at boot time. To find out their Internet address they broadcast RARP packets with an address map request. Hosts that can resolve the address will send a RARP packet to the requestor with the internet address inquired for encapsulated in it.

BOOTP -Boot Protocol- just like RARP BOOTP allows a machine to find out its own Internet address, but additionally it provides the facility to load software over the network.
2 The Ethernet Physical Layer and Data Link Layer

2.1 An Introduction to Ethernet

The Ethernet local area network was developed at Xerox PARC in 1973 as part of a research programme on personal computers and distributed systems. It provides a communication facility for high speed data exchange among stations located in a moderately sized geographical area. Stations can be either computers or special devices.

Physically Ethernet is a single 10 Mbit/sec bus, to which all stations in the network are connected. Data is transmitted in packets of at most 1500 bytes. Because the communication channel is shared all machines receive all packets. Due to the fact that the channel is shared transmission of packets by different hosts can interfere, which leads to data corruption. Such an interference is called a collision. To avoid collisions and to recover from inevitable collisions –collisions that occur when stations send a packet at about the same time– the CSMA/CD (Carrier Sense Multiple Access with Collision Detect) access scheme is used. Transmission of packets is only initiated if no carrier is present on the bus. Due to the traveling time over the bus collisions still can occur. This type of collision is detected by the sending stations; retransmission is tried after a random period of time.

There are two standards for Ethernet. One is defined in the document "The Ethernet, A Local Area Network, Data Link Layer and Physical Layer Specifications" by Digital, Intel and Xerox [4] and is often referred to as the oldstyle Ethernet or the DIX Ethernet Specification, DIX for short. The other one is IEEE 802.3, which has a few 'improvements' over DIX and is less popular. Therefore we will only deal with DIX.

2.2 The Ethernet Physical Layer

An Ethernet consists of one to three cable segments, interconnected with repeaters. A repeater is a bidirectional amplifier connecting two segments. A segment is a coaxial cable up to 500 meters in length. Thus the maximum cable length between two machines is 1500 meters. Ethernets can be
connected to each other by bridges to form larger networks. The path between two stations may contain at most seven bridges. If a bridge receives a packet from one Ethernet for a station that is not connected to that particular Ethernet, the packet is passed to the other side of the bridge and transmitted on the other Ethernet. The bridge keeps track of the connected stations on both Ethernets by inspecting the source of traversing packets and storing them in its memory. Stations that didn't generate traffic for a while are removed from its tables. This and the fact that each station has a unique Ethernet Address makes it possible to move stations from one Ethernet to another.

![Diagram of an Ethernet network](image)

Fig. 2.1 Example of an Ethernet network

The cable itself is completely passive. The electronic components that make the network function are associated with the stations that are attached to the network. As stated in DIX, any connection to Ethernet has two major components: the transceiver and the host interface. The transceiver can sense a carrier and it can translate analog signals to digital signals and vice versa. The transceiver is connected to the cable. In the early days of Ethernet connections to the cable were made by taps, small holes in the outer layers of the cable allowing small pins to touch the core and the shield. The host interface was connected to the transceiver via a cable.

On modern Ethernet controllers transceiver and host interface are integrated on a single printed circuit board, which is inserted into a slot in the station. Connections to the coaxial cable are made with BNC connectors.

The Physical Layer performs two main functions: Data Encoding and Channel Access. There are two Data Encoding functions. One is generation and removal of the preamble of the frame. The other is encoding as well as decoding between binary representation and the phase encoded form of bits.
The Channel Access functions are transmission and reception of the encoded bits, detecting traffic on the channel—generating a carrier sense signal for the Data Link Layer when there is traffic on the channel—and detecting contention on the channel—generating a collision detect signal when a collision is detected.

2.3 The Ethernet Data Link Layer

The Data Link Layer defines a link level communication facility that is independent of the medium, built on the medium dependent physical channel provided by the Physical Layer. The two main functions of the Data Link Layer are Data Encapsulation and Link Management.

Data packets are transmitted in frames. The Transmit Data Encapsulation component of the Data Link Layer constructs a frame from the data supplied by the Network Layer. The Receive Data Decapsulation disassembles an incoming frame and passes the information to the Network Layer.
A frame consists of a six byte destination address, a six byte source address, a two byte frame type field, a data field and a four byte CRC field. The Physical Layer adds an eight byte preamble of alternating 1's and 0's for synchronisation purposes. The data field has a maximum length of 1500 bytes. The minimum length is 46 bytes. If less than 46 bytes have to be transmitted, padding has to be applied. The minimum of 46 bytes data ensures that when two machines—separated by maximum cable length—start transmitting at the same instant, the frame is long enough—in terms of the propagation delay—for both machines to detect a collision.

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Destination Address</th>
<th>Source Address</th>
<th>Packet Type</th>
<th>Data</th>
<th>CRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bytes</td>
<td>6 bytes</td>
<td>6 bytes</td>
<td>2 bytes</td>
<td>46 ... 1500 bytes</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

Fig. 2.3 The format of an Ethernet frame

The Transmit Link Management portion attempts to avoid contention with other traffic on the channel by monitoring the carrier sense signal; if necessary, transmission is delayed. When the channel is clear, frame transmission is initiated after a brief interframe delay to provide recovery time for other data link controllers and for the physical channel. The Receive Link Management collects bits from the Physical Layer as long as the carrier sense signal remains on. When the carrier sense signal goes off, the CRC computed over the incoming data is compared with the received CRC. If they do not match, an error has occurred and is reported. In case of no error the frame is passed to the Receive Data Decapsulation portion for processing.

2.4 The Clarkson Packet Drivers
An Implementation of the Data Link Layer

As stated before MS-DOS does not provide an abstraction of physical peripherals. So individual programs have to deal with a great variety of interfaces, displays etcetera. No allocation scheme of peripherals to applications is provided.

This is especially true for Ethernet controllers. Applications that have their own drivers claim the hardware for themselves. The Clarkson Packet Drivers partly solve this problem. The driver owns the Ethernet controller.
An application can register to the driver for the kind of packet it expects. If the driver receives a packet, it checks its list with subscribers to see if an application is waiting for that packet. If there is a match, it passes the packet to the requestor and clears the controller for the next packet. If there is no match, it discards the packet and then clears the controller for the next packet. If an application wishes to send a packet, it offers it to the packet driver, which takes care of the transmission.

The Clarkson Packet Drivers include drivers for almost all Ethernet controllers currently available. The three most common types of Ethernet controllers at the EUT are 3Com EtherLink II, Western Digital EtherCard PLUS and DEPCA.

2.5 Token Ring
An Alternative Physical Layer

The Data Link Layer accomplishes an abstraction of the Physical Layer. This implies that it is not necessary to have the same Physical Layer in different networks to have hosts in these networks communicate with each other. The software described in this report will probably run in combination with Ethernet, but could just as well be used on other networks, for example a Token Ring.

Fig. 2.4 Example of a Token Ring
The Token Ring is treated here as an alternative because it plays a significant role in commercial products and thus it is worth the short digression. The Token Ring is described in the IEEE 802.5 Standard. The topology is a circle. Its basics can be understood by comparing data transport with a freight train riding on a cyclic railroad with a number of stations. The token corresponds to a locomotive that circulates on the track continuously. The locomotive is 'busy' or 'free'. When there are no wagons attached it is free, otherwise it is busy. Wagons containing data can be attached to the locomotive whenever it passes a station if it is not already busy. When wagons are attached the locomotive is busy and the destination address of the wagons is marked on the locomotive. The destination station must detach the wagons to free it.

One of the weaknesses of the Ethernet is that collisions can occur. Due to the necessity of detection of such collisions, the bandwidth of the medium and the cable length impose a minimum packet length. That is why these are all three specified in the standard. Token Ring on the other hand is contention and collision free. So bandwidth, maximum cable length and minimum packet size play no role in its specification. Token ring though has its deficiencies as well. For example, there is no guarantee that a station which has the token does not return it. Or a busy token can keep circulating forever. These problems can be overcome by assigning controlling hosts.
3 The Interface to the Packet Driver
—Basis of the IP Network Layer

3.1 Introduction

The EUT-TCP library uses the Clarkson Packet Drivers, as these are the most widespread and readily available Public Domain packet drivers. Virtually all brands and types of Ethernet cards are supported. A Clarkson Packet Driver does a number of things. First of all, it owns the Ethernet card. It allows an application using it to subscribe to a certain type of packets. This provides a mechanism to filter out unwanted packets in a very early stage, thus reducing overhead in higher layers. Secondly it services the hardware interrupt that an Ethernet card generates when it receives a packet. It also takes care of the CRC handling and, of course, it allows the application to send packets out.

In fact, a Clarkson Packet Driver accomplishes many tasks that should be taken care of by the Operating System. These tasks include abstraction from interrupts and managing the hardware.

A Clarkson Packet Driver is a resident program with its own address space, written in assembly language. The EUT-TCP library is written in C. The functions the Packet Driver provides can be accessed by issuing a software interrupt. This is a relatively straightforward procedure. There is a serious problem, though.

The Packet Driver is hooked to a hardware interrupt vector. When an Ethernet card receives a packet, it generates an interrupt on this vector. The driver associated with the Ethernet card—there can be more than one network interface in one machine—has to service this interrupt. To do so, it has to call a library routine to get an address of a buffer in the address space of the application. This requires a context switch. If there is no buffer space available, this routine returns NULL.

If there is not enough buffer space, the packet driver discards the packet and frees the Ethernet card to enable it to receive the next packet. If there is enough space, the Packet Driver will copy the packet into the buffer and call another routine to inform the application that the buffer has been filled. This also requires a context switch. An assembly interface is needed to
enable the Packet Driver to do the context switches and call the routines written in C.

In addition to providing the interfaces to be able to call Packet Driver routines from C and to call C routines from the Packet Driver, this part of the library also creates and maintains a packet buffer queue for incoming packets. It also provides some utility routines like a control-break handler and a function to convert Ethernet addresses to readable ASCII format.

3.2 Assembly Glue

The file receive.asm contains the assembly glue code. This code provides a number of entry points, one for each possible interface. The first thing the glue code does when it is called is to save the interrupt flags and to disable the interrupts. Next it switches to its own data segment. It saves the stack pointers and sets them to its own stack. The desired action - allocate buffer space and transfer data to the buffer or signal completion - is specified in the AX register. Depending on the value in this register a call to either claim_buf or receive_buf is made. After returning from a call it restores the stack, the data segment and the interrupt state.

When an interrupt from an Ethernet card occurs, the Packet Driver calls the entrypoint designated to that driver with AX = 0, which results in a call to claim_buf with parameters device_no, handle and length. These parameters are passed on the stack. A successful call to claim_buf returns the buffer address - an offset and segment pair - in ES:DI that is not equal to NULL. In this case the Packet Driver fills the buffer and calls the entrypoint again, this time with AX = 1, which results in a call to receive_buf with parameters device_no and handle. The parameters for receive_buf are also passed on the stack.

3.3 C Interface to Packet Driver Functions

The C interface provides access to the essential functions of the Clarkson Packet Driver as defined in the PC/TCP Version 1.09 Packet Driver Specification [5]. These functions and their associated type- and error-declarations are declared in the header file packet.h. This file is also included by pktint.c, the actual core of the interface code providing the basis for the Network Layer.
The files drivers.c and pktint.h contain declarations of driver and interface class specific things. The TUE-TCP library currently supports DIX Ethernet only, but can be easily adapted to other interface types.

By using the essential functions a number of facilities have been made available. Let us assume at least one packet driver to be present. The function find_driver searches the software interrupt vector table for packet driver interrupt vectors and returns the first one it finds. This function can be called repeatedly to find all packet drivers.

Information about the drivers found can be obtained by calling driver_info. Next, an application can subscribe to a certain packet type by calling subscribe_type or unsubscribe by calling either unsubscribe_type or unsubscribe_all. Packets can be sent to other machines by calling send_data.

3.4 Buffering and utilities

Buffering is intended to prevent data loss in situations where an application is too busy at the moment data comes in to handle the incoming data. It only works well if the average arrival rate is less than the average service rate.

As pointed out earlier, MS DOS has its shortcomings. Among the many things it fails to provide is a buffering mechanism for devices. Inventing the wheel over and over again is left to developers. That is also what has been done in EUT-TCP.

EUT-TCP uses a cyclic buffer on a FIFO basis. The packet buffer may contain at most 64 entries of the type Data_packet. See packet.h for the declaration of this type. Data_packet is a fixed-length description of a packet with a pointer to the data. This pointer points into an 8K stringpool.

A packet has to be stored in a consecutive area in the stringpool. As a result of this the last bytes of the stringpool will not be used very often, id est: it is very unlikely that a packet coming in at the moment data in the stringpool has nearly reached the end fits in exactly.

If there is enough space available when a call to claim_buf is made there is no problem. If there is not enough space, there are two possibilities: either
the application is busy reading a packet from the stringpool or it is not. In the former case claim_buf will return NULL. In the latter case it will remove as many packets as needed, oldest first.

Another routine that is worth mentioning is check_destination. Due to a 'bug' in the 3Com card used in one of the development machines, the application received all packets traversing the segment, even the ones not addressed to it. To reduce overhead in higher layers, this routine sieves out these packets. Optionally broadcast packets are also discarded.

3.5 Test Suites

In order to test the software described so far, two different testsuites have been used. In the first a computer with a single 3Com 3C501 Ethernet card was hooked up to a segment of the Ethernet in the RC of the EUT. A little test program, based on the EUT-TCP library, received broadcast packets and dumped them to the console.

For the second testsuite a second Ethernet card, a Western Digital WD8003, was installed in the test machine. Let us call this machine A. Another computer, say machine B, with a WD8003 was connected to the WD8003 in machine A by a single cable, thus forming a tiny segment. Machine A changed the destination address of all incoming packets to the address of machine B and then sent them out on the tiny segment, thus performing a bridge-like function. Machine B dumped received packets to the console, just like machine A. In addition to this, it also counted the number of packets received. Every ten packets or so it sent a packet to machine A. The data in this packet contained the text 'Cut it out!!!'. Of course, these packets were relayed back to machine B again.

These tests have proved the functions implemented so far to be very robust. The cyclic buffering turned out to work very well. Only in the last test with an exponential growth of the number of packets the buffer would fill up, leading to loss of packets. This is as expected, because buffering assumes a balance between producer and consumer. The probability that information will get lost is very low, because in higher layers some form of an ACK-NAK protocol will be used which will result in retransmission of lost packets.
4 The C Code Revealed —Insight in the Inside

4.1 Introduction

This chapter intends to give insight in the inner workings of the interface to the packet driver as described in the previous chapter. This is done by describing the most important functions one by one. In this description the phrase 'calls the packet driver' is used very often. This stands for setting up the registers for this call as specified in the PC/TCP Packet Driver Specification [5] and generating a software interrupt bij using the Turbo C++ function intr.

4.2 C Interface to Packet Driver Functions

```c
int init_pktint (void);
Attempts to allocate memory for the cyclic packet buffer and for a buffer for a packet to be sent. Then it initializes a device table with the low level receive procedures. It also installs the exit routine pkt_clean_up, which is to be executed upon normal termination of the program. This function should be called at the beginning of an application.
```

```c
void set_pkt_signals (void);
Installs user-specified handlers —in this case the normal exit routine which among other things calls the exit routine installed by init_pktint—and sets the control-break handler to pkt_ctrl_brk. Like the previous function, this one should be called at the beginning of an application. Note that set_pkt_signals and pkt_ctrl_brk allow for a user defined control-break handler.
```

```c
int find_driver (int restart);
Searches the software interrupt table starting with the lowest software interrupt number. To identify a packet driver interrupt, bytes 4 to 12 of the memory area an interrupt vector points to are compared with the packet driver signature. An interrupt signature is an eight byte string used to identify an interrupt handler. If a packet driver is found, its interrupt number will be returned, otherwise the search is continued at the next
```
interrupt vector. If the end of the table is reached without finding a packet driver signature, zero will be returned. Subsequent calls to find_driver with restart not equals to one will cause find_driver to continue its search at the successor of the packet driver interrupt vector found previously. This is a way to find all installed packet drivers.

Driver far *driver_info (int interruptno);
Initializes the driver. Returns a pointer to a struct describing the driver at the given interrupt number or NULL if the device driver table is full. Information about a driver is obtained by calling get_driver_info, a function that does a call to the packet driver by means of a software interrupt that returns its version, interface class, interface type, interface number, name and functionality –see [5] for more information– and by calling get_driver_atts, a function that does a call to get_address to copy the local network address of the interface into the struct.

int reset_interface (Driver far *device);
Does the actual call to the packet driver to reset the interface. Because there must be exactly one handle open when this call is issued, all existing handles have to be closed –by calling unsubscribe_all– and a new one has to be opened –by calling subscribe_type– before this packet driver function is called. After a call to reset_interface there will be no open handles anymore.

int clear_driver (Driver far *device);
This function calls reset_interface to reset the driver. It also updates the struct that describes the driver so other routines can see from the fact that the interruptno field equals zero that the driver has been cleared and has to be reinitialized in order to be used again.

int subscribe_type (Driver far *device, Word type);
With this function a subscription is made to packets of the type specified received on the device specified. Supported types are defined in packet.h.

void unsubscribe_all (Driver far *device);
Unsubscribes to all types of packets previously subscribed to.

int unsubscribe_type (Driver far *device, Word type);
Unsubscribes to a given packet type.
int send_data (Driver far *device, Byte far *dest_addr,
             Word type, Byte far *data_buf, Word len);
Sends a packet out over a given device to the destination address specified.
It transmits len bytes, starting at data_buf. The packet will be of type type.

int terminate_driver (Driver far *device);
Calls the packet driver function that terminates the driver associated with
the handle. If possible, the driver will exit and allow MS-DOS to reclaim the
memory it was using. Like with reset there must be exactly one handle
open when this call is issued, so all existing handles have to be closed and a
new one has to be opened before this packet driver function is called.

In addition to these functions there are also some variables that are
important to the user of this library. The variable accept_broadcast
determines whether broadcast packets are sieved out or not. Its initial value
is 0, which means that broadcast packets will be sieved out. A value of 1
turns sieving off.

For debugging and monitoring purposes there are a number of variables
available. The variable rl_count contains the number of receive calls
made. The variable cl_count contains the number of calls to claim_buf.
Finally, reject_count contains the number of packets rejected by the
function check_destination.

4.3 Buffering

Data_packet far *get_buffer (void);
If the packet buffer is empty, this function returns NULL. Otherwise a
pointer to the first packet in the queue is returned. This packet must be
considered to be read only. Changes to its contents can cause serious
damage. If the queue is empty, buf_lock is 0 on return, otherwise it is 1.

void flush_queue (void);
This function flushes all packets in the queue.

int delete_qhead (Byte respect_lock);
This function is called to delete the oldest entry in the queue. The oldest
one is removed because this is the one most likely to time out and to be
retransmitted. The function delete_qhead can be called by the application
as well as the driver -from claim_buf if there is not enough space
available. If it is called by the driver, respect_lock must be 1. If the application has locked the buffer because it is reading information from the buffer and the driver calls delete_qhead, it will return 1 immediately. If called by the application, respect_lock must be 0, because the application is the only one that can lock the queue. The lock can only be released by delete_qhead.
5 Epilogue

At the beginning of this project I thought that it was possible to write a complete TCP/IP library in a relatively short period of time. At that time I had underestimated TCP/IP. The general descriptions of TCP/IP found in many books tend to give one the impression that it is an extremely simple protocol suite. But in fact it is quite large and not that simple to implement at all.

Even though I took into account that MS-DOS was going to cause me some trouble, I had never thought that it would be so bad.

I also overestimated the services the Clarkson Packet Driver provides. The sample C interface code that comes with the drivers turned out to be of a very poor quality. In addition to that our demand for abstraction has made it even harder to complete the Data Link Layer, but the result is therefore a very good basis for a TCP/IP library.

The existing public domain TCP/IP libraries have their drawbacks—they are huge, often poorly documented and unreliable, and take a long period to learn to work with—so it is definitely worthwhile continuing the project, especially considering the demand for a comprehensive library.
References

[1] D. Comer
INTERNETWORKING WITH TCP/IP: PRINCIPLES, PROTOCOLS AND ARCHITECTURE
London: Prentice Hall, 1988
Section 2.4 (Ethernet), Chapters 5...7 (ARP, RARP, IP), Chapters 9...12 (ICMP, UDP, TCP)

OPERATING SYSTEM DESIGN — VOLUME II: INTERNETWORKING WITH XINU
London: Prentice Hall, 1987
Chapter 2 (Ethernet), Chapters 5...9 (RARP, ARP, IP, ICMP, UDP)

DISTRIBUTED SYSTEMS, CONCEPTS AND DESIGN
Wokingham, England: Addison-Wesley, 1988
Section 3.3.1


[5] PC/TCP Packet Driver Specification Revision 1.09

UNIX NETWORK PROGRAMMING
Section 5.2 (TCP/IP)

COMPUTERNETWERKEN
London: Prentice Hall, 1988
Sections 3.4.1 (Ethernet), 6.4.2 (TCP) and 5.5.2 (IP)

[8] RFCs (Request For Comments):
The TCP/IP protocols and related issues are defined in RFC's. The ones that provide a good introduction are: rfc768 User Datagram Protocol, rfc791 Internet Protocol, rfc792 Internet Control Message Protocol, rfc793 Transmission Control Protocol, rfc826 An Ethernet Address Resolution Protocol and rfc903 A Reverse Address Resolution Protocol.
Appendix  C sources
Receive.asm is the interface between the Clarkson packet drivers and the EUT-TCP library. It must be assembled by Borland's Turbo Assembler.

The interface has to provide each driver with its own entry point. This to enable the TCP library to identify the caller. The entry points are called receive0, receive1, etc. They are passed to the drivers by the TCP library. The number of the entry point is passed as device_no to the C-procedures claim_buf and receive_buf. The number of entry points is defined in Max_devices (currently 4). The macro assembler automatically generates Max_devices entry points.

The interface saves the interrupt state and disables interrupts. Then it saves ds and the stack pointers, sets ds to the data segment of the application and installs an own (small) stack. Depending on the value passed in al claim_buf (al = 0) or receive_buf (al = 1) is called.

Before returning to the driver ds, the stack pointers and the interrupt; are restored. If claim_buf was called es:di is set to its return value.

Warning: the programmer must keep in mind that claim_buf and receive_buf; are called with interrupts disabled and with a limited stack; as defined in Stack_size (currently 64 bytes).

IDEAL MODEL compact

; constant definitions
Stack_size = 64
Max_devices = 4

; externals
extrn C claim_buf: proc
extrn C receive_buf: proc

DATASEG
stackbottom db Stack_size dup (?)
stacktop db ?

; We have to switch to our own environment.
; In order to be able to switch back to the driver environment
; we need storage locations to save the settings of ss and sp.
; ds will be saved on the driver stack.
driver_ss dw ? ; location to save ss
driver_sp dw ? ; location to save sp

CODESEG

; Create Max_devices entry points (receive0, receive1, ... )
MACRO def_entry Dev_no

; define entry_point for device Dev_no

public C receive&Dev_no
label receive&Dev_no far

mov ah, Dev_no
jmp body
Device = 0
REPT Max_devices
    def_entry &Device
    Device = Device+1
ENDM

body:

; save interrupt flags and disable interrupts
pushf
ci

; Switch to our data segment.
push  ds
mov   dx, @Data
mov   ds, dx

; save stack pointers
mov   [ds:driver_ss], ss
mov   [ds:driver_sp], sp

; set up new stack
mov   dx, SEG stacktop
mov   ss, dx
mov   sp, OFFSET stacktop

; The desired action is specified in al:
;
al  = 0:     call claim_buf
al  = 1:     call receive_buf

dec   al
jz    receivebuf

claimbuf:

; We must call claim_buf in order to get space for a packet.
; C-declaration:
;   Byte far *claim_buf (int device_no, Word handle, Word length)
push   cx ; length
push   bx ; handle
mov    al, ah
mov    ah, 0
push   ax ; device_no
call   claim_buf
mov    di, ax ; buffer offset
mov    es, dx ; buffer segment
jmp    finish

receivebuf:

; Call receive_buf to tell that the packet is stored in the buffer.
; C-declaration:
; void receive_buf (int device_no, Word handle)
push bx ; handle
mov al, ah
mov ah, 0
push ax ; device_no
call receive_buf

finish:
; Don't touch es and di in the code below.
; If claim_buf was called, they contain the buffer address.

mov ss, [ds:driver_ss] ; restore stack
mov sp, [ds:driver_sp] ; restore data_segment
popf ; restore interrupt state

END
/* PKT_OUTSIZE is the size of the buffer used to send packets. */
/* It must be large enough to hold the destination address, */
/* the source address, the type, the data and in some cases */
/* also the size of the data. For DIX 1514 bytes is sufficient. */
#define PKT_OUTSIZE 1532
/* Maximum length of type field in packet. */
#define MAX_TYPELEN 16
/* Interface classes */
/* Class 1: DIX (Digital, Intel, Xerox) Ethernet */
/* DIX_type_table contains the types in network (normal) order. */
/* The offset in the array is equal to the internal type number. */
extern Word far DIX_type_table[];
#define set_DIX_type(p, t) (*((Word *)p) = DIX_type_table[t])
#define DIX_ADDR_LEN 6 /* hardware address length */
#define DIX_TYPE_LEN 2 /* length of type field */
#define DIX_HDR_LEN 14 /* start of type field */
#define MIN_DIXDATA 46 /* length of header preceding data */
#define MAX_DIXDATA 1500 /* minimum data size */
#define DIX_BROADCAST "\xff\xff\xff\xff\xff\xff"
extern int assemble_packet(int if_class, Byte far *dest, Byte far *src,
     Word type, Byte far *data, Word len,
     Byte far *packet);
Supported packet types

The types below are the ones recognized by the library.
The numbers only have meaning for this library. They are translated
to packet types with help of internal tables.
ANY_TYPE matches any type; its value must exceed that of all other
ones.

#define IP_DGRAM 0
#define ARP 1
#define RARP 2
#define ANY_TYPE 3

/* MAX_ADDR_LEN is used to allocate buffers for network and hardware
addresses. So it must have at least the length of the longest
address. */
#define MAX_ADDR_LEN 8 /* more than enough */

/* Interface classes */
#define ANY_IF_CLASS 0
#define DIX 1 /* Digital, Intel, Xerox ethernet. */

typedef unsigned char Byte;
typedef unsigned int Word;

typedef struct driver {
    Word interruptno;
    Byte installed;
    int version;
    int if_class;
    int if_type;
    int if_number;
    char far *name;
    int maxdatalen;
    int net_addrlen;
    Byte net_address [MAX_ADDR_LEN]; /* In network byte order. */
    Byte far *broadcast;
    int functionality;
    Byte types_in_use [ANY_TYPE+1];
    Word handles [ANY_TYPE+1];
    void (far *receive_proc) (void);
} Driver;

typedef struct data_packet {
    Word length; /* total packet size */
    Word data_len; /* data size */
    Driver far *device;
    Word handle;
    Word type;
    Byte far *dest; /* Destination address. */
    Byte far *src; /* Source address. */
    Byte far *data; /* Start of data */
    Byte far *pkt; /* Start of packet */
} Data_packet;

/* pktint return codes */
#define PKT_ERROR -1
#define PKT_NORMAL 0
#define PKT_SYSERR 1

#define Pkt_error(err) {pkt_errno = err; return (PKT_ERROR);}

#define NO_DRIVERS -1 /* No packet driver found. */
#define TOO_MANY_DRVRS -2 /* Too many packet drivers. */
#define UNSUPPORTED -3 /* Device is not supported. */
#define NO_DEVICE -4 /* Device does not exist. */
#define DATA_OVERFLOW -5 /* Too much data for driver class. */
#define INVALID_TYPE -6 /* Unknown type (out of range). */
#define DBL_SUBSCRIBE -7 /* Type already subscribed to. */
#define NO_SUBSCRIBE -8 /* Type not subscribed to. */
#define NO_DRIVER -9 /* Driver not installed (any more) */

/* Packet Driver error values. */

#define BAD_HANDLE 1 /* Invalid handle number. */
#define NO_CLASS 2 /* No interfaces of specified class. */
#define NO_TYPE 3 /* No interfaces of specified type. */
#define NO_NUMBER 4 /* No interfaces of specified number. */
#define BAD_TYPE 5 /* Bad packet type. */
#define NO_MULTICAST 6 /* No multicast on this interface. */
#define CANT_TERMINATE 7 /* Termination of packet driver failed. */
#define BAD_MODE 8 /* Invalid receiver mode. */
#define NO_SPACE 9 /* Insufficient space */
#define TYPE_INUSE 10 /* Type was used and not released. */
#define BAD_COMMAND 11 /* Invalid command. */
#define CANT_SEND 12 /* Packet not sent (hardware error). */
#define CANT_SET 13 /* Cannot change hardware address: */
/* more than 1 handle open. */
#define BAD_ADDRESS 14 /* Invalid hardware address. */
#define CANT_RESET 15 /* Cannot reset interface: */
/* more than 1 handle open. */

extern int accept_broadcast;
extern int pkt_errno;
extern int clear_driver (Driver far *device);
extern int delete_qhead (Byte respect_lock);
extern Driver far *driver_info (int interruptno);
extern int find_driver (int restart);
extern void flush_queue (void);
extern int get_address (Driver far *device);
extern Data_packet far *get_buffer (void);
extern int init_pktint (void);
extern int net_addr_to_hex (Byte far *addr, int 1);
extern int reset_interface (Driver far *device);
extern int send_data (Driver far *device, Byte far *data buf, Word len);
extern void set_pkt_signals (void);
extern int subscribe_type (Driver far *device, Word type);
extern int terminate_driver (Driver far *device);
extern void unsubscribe_all (Driver far *device);
extern int unsubscribe_type (Driver far *device, Word type);
/* stack and buffer sizes */

#define PACKET_QLEN 64  /* Maximum number of queued packets. */
#define Q_STRPOOLSIZE 8192 /* Maximum number of queued characters. */

/* halfword operations (useful for register manipulations) */

#define high_byte(x)    ((x >> 8))
#define low_byte(x)      ((x & 0xff))
#define set_high_byte(x, v) ((x & 0xff) | (v << 8))
#define set_low_byte(x, v) ((x & 0xff00) | (v & 0xff))

#define CARRY_FLAG 0x1

/* definitions for the packet driver interrupts. */

#define Pktint_sig "PKT DRVR"  /* signature of packet driver */
/* Packet driver interrupts are between Pktint_low and Pktint_high. */
/* So we have to search for drivers in that range. */
#define Pktint_low 0x60
#define Pktint_high 0x80

/* Packet Driver function call numbers. */

/* Supported functions: */

#define DRIVER_INFO 1  /* Essential functions. */
#define ACCESS_TYPE 2
#define RELEASE_TYPE 3
#define SEND_PKT 4
#define TERMINATE 5
#define GET_ADDRESS 6
#define RESET_INTERFACE 7

/* Unsupported functions: */

#define GET_PARAMETERS 10  /* High performance functions. */
#define AS_SEND_PKT 11
#define SET_RCV_MODE 20  /* Extended functions. */
#define GET_RCV_MODE 21
#define SET_MULTICAST_LIST 22
#define GET_MULTICAST_LIST 23
#define GET_STATISTICS 24
#define SET_ADDRESS 25
```
#include <errno.h>
#include <stdlib.h>
#include <dos.h>
#include <mem.h>
#include "packet.h"
#include "drivers.h"

/* DIX_type_table contains the types in network (normal) order. */
/* The offset in the array is equal to the internal type number as */
/* indicated in the comment. Types are specified with swapped bytes */
/* due to the stupid (!) little endian byte ordering. */

Word DIX_type_table[] = {
    /* Types in network (normal) byte order */

    /* little endian       normal (hex)       internal number */
    0x0008, 0x0800, IP_DGRAM, */
    0x0608, 0x0806, ARP, */
    0x2580, 0x8035, RARP, */
    0, 0x0000, ANY_TYPE, */
};

int assemble_Facket (int if_class, Byte far *dest, Byte far *src,
                     Word type, Byte far *data, Word len, Byte far *packet) {
    Byte far *pt;

    switch (if_class) {
    case DIX:
        if (len > MAX_DIXDATA) {
            pkt_errno = DATA_OVERFLOW;
            return (0);
        }
        memcpy (packet, dest, DIX_ADDR_LEN);
        memcpy (packet + DIX_ADDR_LEN, src, DIX_ADDR_LEN);
        pt = packet + DIX_TYPE;
        set_DIX_type (pt, type);
        pt += DIX_TYPE_LEN;
        memcpy (pt, data, len);
        if (len < MIN_DIXDATA) /* pad it with zeroes */
            memset (pt + len, 0, MIN_DIXDATA - len);
        return (MIN_DIXDATA + DIX_HDR_LEN);
    }
    default:
        if (len > MAX_DIXDATA) {
            pkt_errno = DATA_OVERFLOW;
            return (0);
        }
    }
    return (0);
}
```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <dos.h>
#include <mem.h>
#include <signal.h>
#include "pktint.h"
#include "packet.h"
#include "drivers.h"

/* Currently at most 4 interfaces can be supported simultaneously. */
/* If more are needed, MAX_DEVICES (and Max_devices in receivew.asm) */
/* must be set to the desired value, additional receive procedures */
/* must be declared below and the procedure init_pkt must changed to */
/* enter the new receive procedures in the device_table. */
#define MAX_DEVICES 4

/* low level interface routines, one for each device */
extern void far receive0 (void);
extern void far receive1 (void);
extern void far receive2 (void);
extern void far receive3 (void);

Driver device_tab [MAX_DEVICES];
int pkt_errno;
int accept_broadcast;
Word cl_count, /* number of calls of claim_buf */
   rl_count, /* number of calls of receive_buf */
   reject_count; /* number of rejected packets */
#define Hex_digits "0123456789abcdef"

static Byte initialized;
static Byte far *Pkt_outbuf;
static Data_packet far *packet_q;
static int next_packet, /* first free packet. */
   q_head, /* first queue element. */
   qpkt_count; /* number of packets in the queue. */
static Byte buf_lock, /* assure exclusive access to the queue. */
   buf_claimed; /* packet_driver may claim only one buffer. */
static Byte far *q_stringpool;
static int q_str_in,
   q_str_out;

static int check_destination (Data_packet far *packet);
static int get_driver_atts (Driver far *device);
static int get_driver_info (Driver far *device);
static int get_space (int length);
static Word get_type (Driver far *device, Word handle);
static void pkt_clean_up (void);
static int pkt_ctrl_break (void);
int init_pktint (void) {
    if ((packet_q = calloc (PACKET_QLEN, sizeof (Data_Packet))) == NULL)
        return (PKT_SYSERR);
    if ((q_stringpool = calloc (1, Q_STRINGPOOLSIZE)) == NULL)
        return (PKT_SYSERR);
    if ((Pkt_outbuf = calloc (1, PKT_OUTSIZE)) == NULL)
        return (PKT_SYSERR);

    /* Initialize device_tab with the low level procs */
    device_tab [0].receive_proc = receive0;
    device_tab [1].receive_proc = receive1;
    device_tab [2].receive_proc = receive2;
    device_tab [3].receive_proc = receive3;

    initialized = 1;
    if (atexit (pkt_clean_up))
        return (PKT_SYSERR);
    return (PKT_NORMAL);
}

void set_pkt_signals (void) {
    signal (SIGABRT, exit);
    signal (SIGFPE, exit);
    signal (SIGILL, exit);
    signal (SIGSEGV, exit);
    signal (SIGTERM, exit);
    ctrlbrk (pkt_ctrl_break);
}

int pkt_ctrl_break (void) {
    pkt_clean_up ()
    fprintf (stderr, "\nProgram aborted (^C)\n"
        return (0); /* Abort */
    }

void pkt_clean_up (void) {
    /* Release all types on all drivers */
    int i;
    for (i = 0; i < MAX_DEVICES; i++)
        clear_driver (device_tab+i);
}

char far *net_addr_to_hex (Byte far *addr, int 1) {
    /* net_addr_to_hex returns a pointer to a static buffer which */
    /* is reused at the next call. So the return value must be used */
    /* between two successive calls. Therefore net_addr_to_hex may */
    /* occur only once in a parameter list (e.g. to printf). */

    static char hex_buf [3*MAX_ADDR_LEN+1];
    char *ph = hex_buf;
    Byte byte;

    if (l > MAX_ADDR_LEN)
        return (NULL);
while (l-- > 0) {
    byte = *addr++;
    *ph++ = Hex_digits [byte >> 4];
    *ph++ = Hex_digits [byte & 0x0f];
    *ph++ = ':';
}
*(ph-1) = 0;
return (hex_buf);
int find_driver (int restart) {
    static int intr = Pktint_low-1;
    char far *vect;

    if (restart)
        intr = Pktint_low-1;
    while (++intr <= Pktint_high) {
        vect = (char far *) getvect (intr);
        if (strcmp (Pktint_sig, vect+3) == 0)
            return (intr);
    }
    return (0);
}

Driver far *driver_info (int interruptno) {
    Driver far *device = NULL;
    int i;

    if (! initialized)
        init_pktint ();
    for (i = 0; i < MAX_DEVICES; i++) {
        if ((device_tab+i)->interruptno == 0) {
            device = device_tab+i;
            device->interruptno = interruptno;
            break;
        }
    }
    if (device == NULL) {
        pkt_errno = TOO_MANY_DRVRS;
        return (NULL);
    }
    if (get_driver_info (device) != 0) {
        device->interruptno = 0;
        return (NULL);
    }
    if (get_driver_atts (device) != 0) {
        device->interruptno = 0;
        return (NULL);
    }
    return (device);
}

int get_driver_atts (Driver far *device) {
    int rslt;

    rslt = get_address (device);
    if (rslt != 0)
        return (rslt);
    switch (device->if_class) {
    case DIX:
        device->net_addrlen = DIX_ADDR_LEN;
        device->maxdatalen = MAX_DIXDATA;
        device->broadcast = DIX_BROADCAST;
        break;
    default:
        break;
    }
    return (0);
}

int get_driver_info (Driver far *device) {
    /* Returns information about the interface. */
struct REGPACK regs;

if (device == NULL)
    Pkt_error (NO_DEVICE); /* No such device */
if (device->interruptno == 0)
    Pkt_error (NO_DRIVER);

set_high_byte (regs.r_ax, DRIVER_INFO);
set_low_byte (regs.r_ax, 0xff);

intr (device->interruptno, &regs);

if (regs.r_flags & CARRY_FLAG)
    Pkt_error (high_byte (regs.r_dx));

device->version = high_byte (regs.r_bx);
device->if_class = high_byte (regs.r_cx);
device->if_type = regs.r_dx;
device->if_number = low_byte (regs.r_cx);
device->name = MK_FP (regs.r_ds, regs.r_si);
device->functionality = low_byte (regs.r_ax);
return (0);

int clear_driver (Driver far *device) {
    int rslt;

    rslt = reset_interface (device);
    if (rslt)
        return (rslt);
    device->interruptno = 0;
    return (0);
}

int delete_qhead (Byte respect_lock) {

    /* Respect_lock must be 1 if called by a driver. In that case */
    /* delete_qhead will immediately return 1, if the application */
    /* has locked the queue. */
    /* Respect_lock must be 0 if called by the application, because */
    /* the application is the only one that can lock the queue and */
    /* the lock can only be released by delete_q.head */

    Data_packet far *pkt;

    if (respect_lock & buf_lock)
        return (1);
    if (qpkt_count == 0)
        return (2);
    buf_lock = 1; /* Baat het niet, het schaadt ook niet */
    pkt = packet_q+q_head;
    q_str_out += pkt->length;
    q_head = (q_head+1) % PACKET_QLEN;
    qpkt_count--;
    if (qpkt_count == 0) {
        /* queue is empty, reset it for more optimal use */

        next_packet = 0;
        q_head = 0;
        q_str_in = 0;
        q_str_out = 0;
    }
    buf_lock = 0;
pktint.c  Thu Mar 26 09:02:04 1992  6

    return (0);
};

void flush_queue (void) {
    buf_lock = 1; /* lock the queue. */
    qpkt_count = 0;
    next_packet = 0;
    q_head = 0;
    q_str_in = 0;
    q_str_out = 0;
    buf_lock = 0;
}

Data_packet far *get_buffer (void) {
    /* This routine returns NULL, if the packet buffer is empty. */
    /* Otherwise a pointer to the first packet in the queue is */
    /* returned. This packet must be considered to be read only. */
    /* Changes to its contents can cause serious damage. */
    /* If the queue is empty, buflock is 0 on return, otherwise 1. */

    Data_packet far *head = NULL;
    Word handle;
    Word far *my_handle;
    Word type;

    if (buf_lock) /* We've seen the head of the queue, delete it. */
        delete_qhead (0);
    if (qpkt_count == 0) /* queue is empty */
        return (NULL);

    /* Assumption: the driver never sets qpkt_count to 0. */
    buf_lock = 1; /* Ne distubas circulas meas */
    head = packet_q+q_head;
    q_str_out = head->pkt->q_stringpool;

    handle = head->handle;
    my_handle = head->device->handles;
    for (type = 0; *my_handle != handle; type++) {
        if (type == ANY_TYPE)
            break; /* Unknown handle */
        my_handle++;
    }
    head->type = type;

    switch (head->device->if_class) {
    case DIX:
        head->dest = head->pkt;
        head->src = head->pkt+DIX_ADDR_LEN;
        head->data = head->pkt+DIX_HDR_LEN;
        head->data_len = head->length-DIX_HDR_LEN;
        break;
    }
    return (head);
}

int subscribe_type (Driver far *device, Word type) {
    /* Subscribe to packets of the specified type. */

    struct REGPACK regs;
    int typelen;

Byte type_str [MAX_TYPELEN];

if (device->interruptno == 0)
    Pkt_error (NO_DRIVER);
if (type > ANY_TYPE)
    Pkt_error (INVALID_TYPE);
if (device->types_in_use [type])
    Pkt_error (DBL_SUBSCRIBE);
if (type == ANY_TYPE) {
    typelen = 0;
    switch (device->if_class) {
      case DIX:
        set_DIX_type (type_str, type);
        typelen = DIX_TYPE_LEN;
        break;
      default:
        Pkt_error (UNSUPPORTED);
    }
}

set_high_byte (regs.r_ax, ACCESS_TYPE);
set_low_byte (regs.r_ax, device->if_class);
regs.r_bx = device->if_type;
set_low_byte (regs.r_dx, device->if_number);
regs.r_ds = FP_SEG (type_str);
regs.r_si = FP_OFF (type_str);
regs.r_cx = typelen;
regs.r_es = FP_SEG (device->receive_proc);
regs.r_di = FP_OFF (device->receive_proc);
intr (device->interruptno, &regs);

if (regs.r_flags & CARRY_FLAG)
    Pkt_error (high_byte (regs.r_dx)); /* error code */

device->handles [type] = regs.r_ax;
device->types_in_use [type] = 1;
return (0);
}

void unsubscribe_all (Driver far *device) {
    Word type;

    for (type = 0; type <= ANY_TYPE; type++) {
        if (device->types_in_use [type])
            (void) unsubscribe_type (device, type);
    }
}

int unsubscribe_type (Driver far *device, Word type) {
    /* Unsubscribe to packets of the specified type. */

    struct REGPACK regs;

    /* This type must be flagged as unused, also if the call fails */

    device->types_in_use [type] = 0;
    if (device->interruptno == 0)
        return (0);
    if (type > ANY_TYPE)
        Pkt_error (INVALID_TYPE);
    if (! device->types_in_use [type])
pktint.c

Pkt_error (NO_SUBSCRIBE);
set_high_byte (regs.r_ax, RELEASE_TYPE);
regs.r_bx = device->handles [type];
intr (device->interruptno, &regs);

if (regs.r_flags & CARRY_FLAG)
Pkt_error (high_byte (regs.r_dx));
return (0);

int send_data (Driver far *device, Byte far *dest_addr, Word type,
Byte far *data_buf, Word len) {

/* Transmits len bytes of data, starting at data_buf */

int length;
struct REGPACK regs;

if (device->interruptno == 0)
Pkt_error (NO_DRIVER);
if (type > ANY_TYPE)
Pkt_error (INVALID_TYPE);

length = assemble_packet (device->if_class, dest_addr, device->net_address,
type, data_buf, len, Pkt_outbuf);
if (length <= 0)
return (PKT_ERROR);

set_high_byte (regs.r_ax, SEND_PKT);
regs.r_ds = FP_SEG (Pkt_outbuf);
regs.r_si = FP_OFF (Pkt_outbuf);
regs.r_cx = length;
intr (device->interruptno, &regs);

if (regs.r_flags & CARRY_FLAG)
Pkt_error (high_byte (regs.r_dx));
return (0);
}

int reset_interface (Driver far *device) {

/* Resets the driver described by device */

struct REGPACK regs;
Word handle;

if (device->interruptno == 0)
Pkt_error (NO_DRIVER);

/* There must be exactly one handle open. */
/* So we close all handles and open a new one. */

unsubscribe_all (device);
subscribe_type (device, ARP);
handle = device->handles [ARP];

set_high_byte (regs.r_ax, RESET_INTERFACE);
regs.r_bx = handle;
intr (device->interruptno, &regs);
unsubscribe_type (device, ARP);
  if (regs.r_flags & CARRY_FLAG)
    Pkt_error (high_byte (regs.r_dx));
  return (0);
}

int terminate_driver (Driver far *device) {

  /* Terminates the driver described by device */

  struct REGPACK regs;
  Word handle;

  if (device->interruptno == 0)
    Pkt_error (NO_DRIVER);

  /* There must be exactly one handle open. */
  /* So we close all handles and open a new one. */

  unsubscribe_all (device);
  subscribe_type (device, ARP);
  handle = device->handles [ARP];

  set_high_byte (regs.r_ax, TERMINATE);
  regs.r_bx = handle;

  intr (device->interruptno, &regs);

  if (regs.r_flags & CARRY_FLAG)
    Pkt_error (high_byte (regs.r_dx));
  device->interruptno = 0;
  unsubscribe_type (device, ARP);

  return (0);
}

int get_address (Driver far *device) {

  /* Copies the current local network address of the interface into 
  * device->net_address. */

  struct REGPACK regs;

  if (device->interruptno == 0)
    Pkt_error (NO_DRIVER);
  set_high_byte (regs.r_ax, GET_ADDRESS);

  regs.r_es = FP_SEG(device->net_address);
  regs.r_di = FP_OFF(device->net_address);
  regs.r_cx = MAX_ADDR_LEN;

  intr (device->interruptno, &regs);

  if (regs.r_flags & CARRY_FLAG)
    Pkt_error (high_byte (regs.r_dx));

  return (0);
Byte far *claim_buf (int device_no, Word handle, Word length) {
    /* Claim_buf is called by the packet driver to reserve buffer space for an incoming packet. */

    Data_packet far *pkt;

    cl_count++;
    if (buf_claimed) /* We allow only one outstanding buffer */
        return (NULL);

    /* Try to allocate a packet buffer. If the queue is full or there is lack of space in the stringpool, as many packets as necessary will be deleted from the head of the queue to accommodate the request. This is done because the oldest are the first to time out and to be resent. We give up, if delete_qhead fails. */

    if (qpkt_count == PACKET_QLEN) { /* queue is full */
        if (delete_qhead (1))
            return (NULL);
    }

    while (get_space (length)) { /* Get space for the data */
        if (delete_qhead (1))
            return (NULL);
    }

    pkt = packet_q+next_packet;
    pkt->length = length;
    pkt->device = device_tab+device_no;
    pkt->handle = handle;
    pkt->pkt = q_stringpool+q_str_in;
    buf_claimed = 1;
    return (pkt->pkt);
}

int get_space (int length) {
    /* Get space for the data */

    if (q_str_out > q_str_in) {
        if (q_str_in+length > q_str_out) /* not enough space */
            return (1);
    }

    if (q_str_in+length > Q_STRPOOLSIZE) {
        /* not enough space, try to wrap around */

        if (q_str_out < length) /* bad luck */
            return (1);

        q_str_in = 0;
    }
    return (0);
}

void receive_buf (int device_no, Word handle) {
    /* Receive_buf is called by the packet driver to notify that the buffer allocated by claim_buf is filled. */

    Data_packet far *pkt;

    rl_count++;

buf_claimed = 0; /* We've got it */
pkt = packet_q+next_packet;
if (check_destination (pkt))
    return; /* we queue only packets directed to us */
q_str_in += (pkt)->length;
next_packet = (next_packet+1) % PACKET_QLEN;
qpkt_count++;
}

int check_destination (Data_packet far *packet) {

    /* Packets not directed to us are not always sieved out by the */
    /* driver. So we have to do it ourselves. */
    /* If we are the destination of the packet the return value is 0. */
    /* If accept_broadcast is set and the packet is broadcasted, the */
    /* return value is also 0. */
    /* Otherwise the return value is 1. */

#define REJECT {reject_count++; return (1); }

    Driver *device;

device = packet->device;
switch (device->if_class) {
    case DIX:
        /* Must use memcmp here, because strncmp halts at a 0 */
        if (memcmp (packet->pkt, device->net_address, DIX_ADDR_LEN) == 0)
            return (0);
        if (accept_broadcast &&
            memcmp (packet->pkt, DIX_BROADCAST, DIX_ADDR_LEN) == 0)
            return (0);
            REJECT;
    default:
        return (0); /* We don't know, so keep it save */
    }
}