TCP/IP Stack

for DWP

Version 1.3.2, May 2003

(2000/2003), Diego De Marco (ddemarco@dpsautomation.com)

Table of Contents

1 Introduction

1.1 About this document

This document covers many issues related to my TCP/IP stack for DWP. It’s intended for people with at least some basic knowledge of Delphi and WDOSX, specially when it comes to understanding the basics of developing networked applications with it.

1.2 Description of the IP stack

After looking everywhere for a TCP/IP stack for DOS, I found there's none. After thinking carefully about embarking on such a task, and having nothing better to do, I decided to create one of my own with the aid of lots of books and RFC's. After a couple of months, I came up with the basic stuff, which kept growing over time. This is what there is so far:

- **ARP** (for resolving IP addresses in an Ethernet environment)
- **IP** (It pretty much complies with all the RFC's)
- **ICMP** (basic things like sending and responding PINGs)
- **UDP** (Complete, as far as I know)
- **TCP** (Complete, but with a few inefficiencies that I'll improve some day) ;)
- **DNS** (Seems to work fine, but I recommend more testing)
- **DHCP** (Complete, as far as I know)
- **Sockets API** for all of the above, which allows you to write your networked applications in a BSD / Linux / Win32 fashion.

In any case, working in non-blocking mode is **NOT SUPPORTED**. After all, DOS is not a multitasking environment. If you don't know what "non-blocking mode" is, then you surely don't have to worry about this issue.

Initially, the stack only worked on top of packet drivers. This release has also been tested with ODI drivers (with the aid of ODIPKT), making it possible to choose the most appropriate technology.

In addition, as of version 1.2.3, multiple IP addresses can be used on the same Ethernet card. This makes it possible, for instance, to create multiple Web or FTP sites using the same port (i.e. port 80, or port 21) without the need to install separate network cards.

1.3 Other modules built on top of this stack (included)

Once the stack was more or less completed, some interesting modules were added:

- A Web server, quite compatible with Delphi's concept for Web Server applications.
- An FTP server, originally developed by Xon Electronics (http://www.xonelectronics.com) and put together by me to make it compatible with this stack.

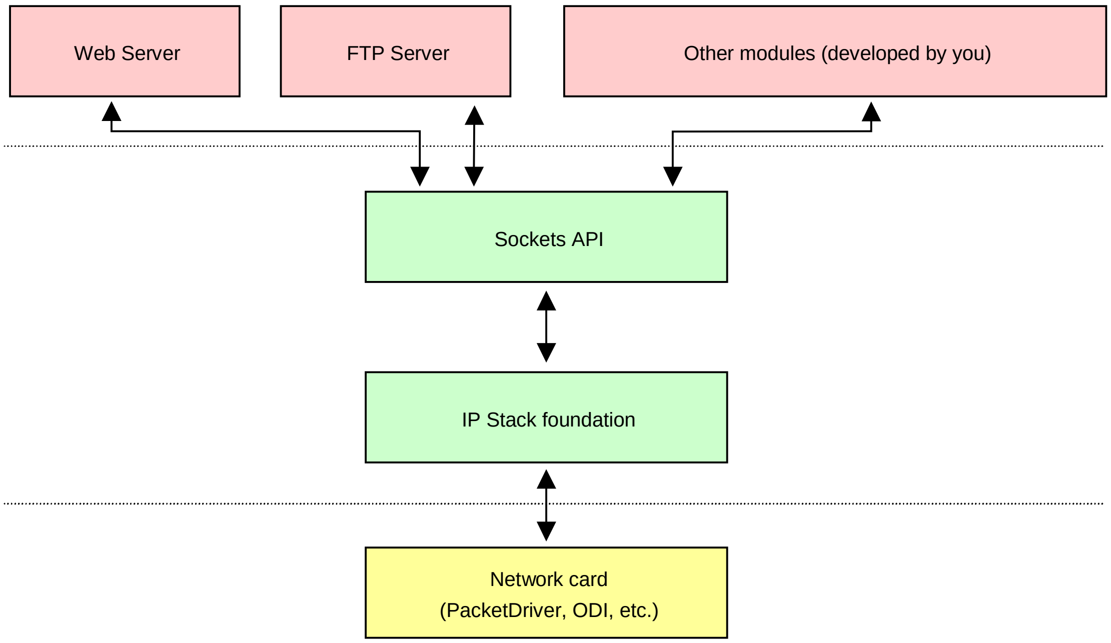
Other features are under development, such as:

- FTP Client
- Mail (SMTP) client.

2 Understanding the IP Stack

The IP stack allows the programmer to develop networking applications using a standard API, which in turn is independent from the physical media used to transmit the information. This means you won't have to worry about using a particular network card, because the stack itself is independent of them.

The following (rather simplified) diagram shows the interaction between the different layers that build a networked application:



- **<u>The lowest layer</u>** represents the network card drivers. These make it possible to communicate with the network card in a fashion that is independent of the card itself. They communicate with the middle-layer in a standard way allowing you to actually remove one network driver and replace it with another one. Currently network card drivers come in three flavors:
    1) <u>Packet drivers</u>: they give the better performance with the smallest footprint. However they usually can't coexist with other network drivers. If you have a Novell network, for instance, you won't be able to run a packet driver and Novell's drivers at the same time. This is the option I recommend when you don't need to use other stacks along with this.
    2) <u>ODI drivers</u>: ODI is an open standard carried out by several companies (Novell among them) which allows different protocol stacks to coexists. The major disadvantage of ODI is that it requires you to install four different executables in order

to make your network card work. Also, a text configuration file is required to set up the various configuration options. Don't worry, al these steps are described in detail in the installation section. I recommend using ODI drivers only if you are already using them for another application. If you don't plan to share the network card with other protocol stacks, Packet Drivers are a far better option.

3) NDIS drivers: NDIS is a standard developed by many companies (Microsoft among them) with the same goals as ODI. There's a lot of documentation pointing out the advantages and disadvantages of one over the other, but ODI has been far more popular. NDIS drivers are **NOT SUPPORTED** for the moment.

- **The middle layer** (this stack). The middle layer talks to the network card drivers and buffers information to be exchanged with the applications. It communicates with applications through a standard API known as BSD sockets. This API was developed long time ago (starting at UNIX systems) and has become a standard for programming networked applications in UNIX, Windows, Linux, and so on. The API found in this stack is almost completely compatible with the standard BSD, except for a few things that have changed as a result of "non-blocking mode" not existing.

- **The upper layer** (applications). This layer consists of applications or modules developed by you. Of course you can also use the existing standard modules like FTP and Web Servers, but the most important thing is you can build up your own applications.

3 Installation

For the time being, this stack can be used with two kinds of network drivers:

a) Packet drivers
b) ODI drivers

I'm doing some research in order to enable this stack to work with NDIS drivers, but note that this stack **hasn't been tested** with DIS_PKT or any other shim. The following chapters explain how to use the stack with the solutions named above.

3.1 Using a packet driver

Important: This chapter (2.3) describes "direct" work with packet drivers. However, this is the only feature that is not supported yet (see "what's missing" at the end of this document) Instead of this, please read chapter 3.3, which describes the use of PKTPM, which is completely

The first option for using this stack is to use a packet driver. Packet drivers are usually shipped with network cards, and lots of them are available. They are usually located in a "Packet" or "PCTCP" directory in the card's installation disk.
Packet drivers act as a link between the network card and the stack, and they're usually very effective. Also, they don't consume much memory. This isn't very important for DWP applications since the run in protected-mode and can use all the available memory, but it's a handy feature when you need to interact with lots of real-mode programs.

Almost all packet drivers allow the user to indicate an interrupt to hook to when they are loaded, which usually range from 0x60 to 0x69 (the default is 0x60). The stack searches automatically for a packet driver in the range 0x60..0x80. In other words, if you install your packet driver in an interrupt outside this range, the stack simply won't detect it.

That's all you need to do in order to use this stack when using packet drivers. Here's an example of an AUTOEXEC.BAT configured for this.

```
@ECHO OFF
REM *** First load the packet driver for my 3Com 3C509C network card
LH C:\NET\3C509PD.COM
```

That's it. If you don't find a packet driver for your network card, try here:

a) The DRIVERS\PACKET directory file included within this package
b) http://www.crynwr.com (home of the packet drivers)
c) http://netlab1.usu.edu/pub/pktdrvr/ (mirror for Crynwr's archive)
d) http://www.driverguide.com (a great place with all kinds of drivers)

3.2 Using ODI drivers

ODI drivers became very popular when Novell Networks ruled the DOS environment. There are lots of people using ODI drivers and they continue to be shipped with network cards nowadays. Usually they are located in a “Netware” directory in the card’s installation disk.

The procedure for installing ODI drivers consists of these steps:

1) Loading LSL, which is part of the ODI interface.
2) Loading the ODI driver for your network card. Under the ODI model, network drivers are called MLID’s.
3) Loading ODIPKT which makes the ODI model “packet-driver compatible”. As with packet drivers, ODIPKT must be hooked to an interrupt in the 0x60..0x80 range, otherwise, the stack **<u>WILL NOT WORK</u>**.

All of these files (except the MLID for your network card) are included in this package.

Here’s how AUTOEXEC.BAT would look when using ODI drivers:

```
@ECHO OFF
REM *** First load LSL
LH C:\NET\LSL
REM *** Next, load your MLID (this is the MLID for my NE2000 card)
LH C:\NET\NE2000
REM *** Next, load ODIPKT at int. 0x60 (96 decimal)
LH C:\NWCLIENT\ODIPKT 1 96
```

When using ODI drivers, there’s also a configuration file named NET.CFG used by LSL and other components. If you already have one, chances are you won’t need to change it. However, if any of the entries listed in this example are missing, just add them.

```
Link Driver NE2000
    Frame Ethernet_802.3
    Frame Ethernet_II
    PORT 300
    INT 5

Link support
    Buffers 5 1600
```

3.3 Using “anti-crash” debug mode (PKTPM)

Important: This chapter (3.3) describes “indirect” work with packet drivers. Please note that the method described in chapter 3.1 is NOT supported yet (see “what’s missing” at the end of this document) so you should use the method described here.

The two installation approaches mentioned before (using packet drivers or ODI drivers) have one major disadvantage when debugging: if your application crashes, chances are you will have to reboot your PC in order to run your application again.

This is because when a crash occurs, the application cannot close the connection to the packet or ODI driver properly. When trying to re-run the application after a crash, you will almost surely get an error when initializing your network adapter object.

Taking into account that crashes are quite common while coding and debugging, a solution is provided to avoid these side effects.

To use the “anti-crash” mode while debugging you should:

1. Load your packet driver or ODI driver as usual, except that they **must be** hooked to int 0x60 (the default).
2. Load PKTPM.EXE. This executable gives the “anti-crash” capabilities.
3. Instead of creating a TPacketDriverAdapter object or a TODIAdapter, create a **TPKTPMAdapter**.

VERY IMPORTANT: You cannot use PKTPM with any adapter object other than TPKTPMAdapter. If you use PKTPM with a TPacketDriverAdapter you won’t be able to initialize the stack.

You don’t have to change anything else. Here’s an example of how the code would look when you prepare it for regular and debug mode:

```
Uses
    Network, {$IFDEF DEBUG} PKTPM {$ELSE} PktDrv {$ENDIF};
Var
    Adapter: TNetworkAdapter;
Begin
    {$IFDEF DEBUG}
    Adapter := TPKTPMAdapter.Create;
    {$ELSE}
    Adapter := TPacketDriverAdapter.Create;
    {$ENDIF}
    if Adapter.Status = asOk then begin
        ...
    end;
    Adapter.Free;
End.
```

4 Compilation

No special precautions need to be taken during compilation other than updating the library path so that Delphi can find all these files.

5 Programming interface

5.1 Basic considerations

The stack has a programming interface which is based on WinSock. However, it uses a “cooperative multitasking” approach which allows many different things to work “concurrently-like”. In order to make you program work correctly, you should yield execution in your program’s main loop. This is done by calling the Yield() function. Please read the samples and the Threads.pas units if you need further information regarding this.

Please note that most sockets functions will yield automatically. As a result, your program MAY work correctly even if you don’t call Yield() explicitly. However, it is a good practice to include a call to Yield() in your program’s main loop.

5.2 Starting and managing the network

5.2.1 Starting the network

In order to start the network, you must follow the following steps:

a) Create a network adapter

The first step needed to start the Network is to start the network adapters. Adapters are created by instantiating a network adapter object of the required type, which represents a physical network adapter on your computer. In particular:

- If You’re using a packet driver or ODIPKT WITHOUT using PKTPM, you must create a network object of class “TPacketDriverAdapter”.
- If you ARE USING PKTPM, you must first

Example of creating a network adapter:

```
Var
    Adapter: TNetworkAdapter;
Begin
    Adapter := TpacketDriverAdapter.Create;
    If Adapter.Status <> asOk then begin
        WriteLn(‘Couldn’’t start network adapter’);
        Halt
    End;
    ...
```

b) Start your protocols by binding them to the adapter(s) you have created in the previous step.

As of today, you can only bind IP and its related protocols. In order to start the IP protocol on a particular adapter, you must use the “IP_Init” function, as in the example below:

```
Var
    IPHandle: Integer;
    IPConf: TIPAdapterConfig;
begin
    ...
    // Configure IP parameters
    IPConf.DHCP      := False;
    IPConf.IPAddress := IP_StringToAddr(’192.168.0.20’);
    IPConf.NetMask   := IP_StringToAddr('255.255.255.0');
    IPConf.Gateway   := IP_StringToAddr(‘192.168.0.1’);
```

```
SetDNSServer(IP_StringToAddr('192.168.0.1', DNS_DEFAULT_PORT));
IP_SetHostName(‘TESTHOST’);

// Start the IP layer on the newly created adapter
IPHandle := IP_Init(Adapter, IPConf);
if IPHandle = 0 then begin
    WriteLn('Could not initialize the IP protocol.');
    Adapter.Free;
    Halt;
end;
...
```

The example is pretty straightforward, but anyway:

- The “DHCP” field of the configuration record indicates whether you want to use a dynamic IP address or a fixed one. Setting it to True will force the stack to allocate a new IP address from any DHCP server available. In that case, all other fields will be ignored.
- The “IPAddress” field indicates the IP address you want to use. If you have set the “DHCP” member to true, this field is ignored.
- The “NetMask” field indicated the net mask you want to use. If you have set the “DHCP” member to true, this field is ignored.
- The “Gateway” field indicates the IP address of a default gateway or router. If you have set the “DHCP” member to true, this field is ignored.
- The “SetDNSServer” function can be used to indicate what DNS server must be used to resolve host names. Calling this function is optional.
- The IP_SetHostName function can be used to name your machine. The name must follow the host naming convention. If you don’t call this function, your machine will have no host name.

Special considerations:

- Even though the design of this networking software is not limited to one network adapter, the current implementation doesn’t allow you to use more than one adapter at a time. Next releases will hopefully overcome this limitation.
- On the other hand, you can start a protocol more than once on the same adapter. As a result, you can have multiple IP addresses bound to the same network card. This is specially useful if you plan to run more many web sites at the same time and you want them all to be started at port 80.

5.2.2 Stopping the network

Before your program ends, it is necessary to stop the network. Basically, you have to revert the steps you've followed before. Example:

```
    ...
    IP_DeInit(IPHandle);
    Adapter.Free;
end.
```

It is VERY advisable to close any open socket or other kind of network connection prior to executing the lines above. In particular, you must first terminate any Web Server or FTP Server you could have started.

5.2.3 Managing IP routes

Even though you're not likely to have to change IP routes, you can do it by using:

```
Function IP_AddRoute(Destination, Mask, Gateway, LocalIP: TIPAddress;
  Cost: Integer): Boolean;
```

Adds a new route to the routing table.
Destination indicates the IP address of the destination network
Mask indicates the mask to get the network part of the previous Destination
Gateway indicates the IP address of the gateway to use to reach the destination.
It can be the IP address of a local adapter, or a real router / gateway.
LocalIP indicates the local IP from which datagrams will be sent to the destination
Cost indicates "how hard" it is to reach the destination by using this route. 0 means "very easy", while 9999 means "very difficult or expensive".

```
Function IP_DeleteRoute(Destination, Mask, Gateway, LocalIP: TIPAddress): Boolean;
```

Deletes a route previously created with IP_AddRoute;

```
Function IP_GetRoute(Index: Integer; var Entry: TRouteEntry): Boolean;
```

Returns one route from the routing table. Index starts with 0. IP_GetRoute returns False if the provided index is out of range.

Special considerations:

Please note that the routing table may change automatically if you use DHCP, since your IP address may change without your noticing it. Even more you can loose your IP address in which case the routing table will usually contain only the “loopback” entry.

5.2.4 Using DHCP services

Apart from using DHCP to get an IP address, mask and so on, you can also use a DHCP server to retrieve parameters you use in your application. This is quite useful when you need to change configuration parameters dynamically. Even more, most DHCP servers allow you to define your own parameters, which later on can be retrieved by these applications. In order to retrieve parameters from a DHCP server, you can use:

```
Function P_GetDHCPOption(Option: BYTE; Var Buffer; Var BuffLen: LongInt): Boolean;
```

Option indicates the option number (from 0 to 255)
Buffer references a buffer in which the received option will be placed.
BufferLen indicated the length of the supplied buffer.
The function returns True if the option has been successfully retrieved, or False otherwise.

5.2.5 Using DNS services

This implementation allows only the “Name Lookup”. No “reverse lookup” or other fancy thing has been implemented.

```
Function GetHostByName(HostName: AnsiString; var IPAddress: TIPAddress): Boolean;
```

HostName indicated the name of the host for which an IP address is searched.
IPAddress is the variable where the retrieved IP address will be placed.
The function returns true if the IP address was resolved, or False otherwise.

5.3 Standard sockets API

The standard API allows the use of “BSD like” sockets to accomplish UDP and TCP networking. In the following pages, you’ll find the reference to the routines available for this purpose. If you don’t have any previous knowledge about the sockets standard, you can also read about this in the Linux documentation or in the help files available in various Windows SDKs (the latter are usually more friendly).

5.3.1 Socket

Prototype

```
Function socket(Domain, SocketType, Protocol: Longint): Longint;
```

Description

The Socket function is the first one to call when trying to use the network. The returned value acts as a handle to be used in subsequent calls to connect(), send(), recv(), etc.

Parameters

Domain	Indicates the type if domain to use. Must be **PF_INET**
SocketType	Indicates the type of sub-protocol to use: **SOCK_STREAM** means “Use TCP” **SOCK_DGRAM** means “Use UDP”
Protocol	Indicates protocol family. Must be set to **IPPROTO_IP**

Return value

Upon successful execution, Socket() returns a handle to the newly created socket. Note that the socket is inactive (closed). If the socket cannot be created (i.e. there’s not enough memory), Socket() returns 0.

5.3.2 CloseSocket

Prototype

```
Function CloseSocket(Socket: LongInt): Longint;
```

Description

CloseSocket closes a socket create with the Socket() function, and releases any resources associated with it.

Parameters

Socket	Is the socket handle returned by a previous call to Socket()

Return value

CloseSocket returns 0 upon successful execution, or nonzero if an error occurs.

5.3.3 GetLastError

Prototype

```
Function GetLastError: LongInt;
```

Description

Returns the error code for the last error related to sockets.

Return value

Error code of the last failed operation.

5.3.4 Send

Prototype

```
Function Send(Socket: Longint; Var Addr; AddrLen, Flags: Longint): Longint;
```

Description

Send is used to write outgoing data on a connected socket. For message-oriented sockets (that is, sockets created with SOCK_DGRAM socket type), care must be taken not to exceed a packet size of 64K.

For sockets created with SOCK_STREAM socket type, any data length is acceptable, including zero.

Parameters

Socket	Is the socket handle returned by a previous call to Socket()
Addr	Block of data to be transmitted.
AddrLen	Size (in bytes) of the block of data to be transmitted.
Flags	Currently not supported. Must be zero.

Return value

Send returns the number of bytes transmitted, which should be the same as AddrLen if the operation completed successfully.

Remarks

The socket must already be connected. This means you have to call Connect() prior to using Send().

Please note that Send blocks the caller until all bytes have been sent. This is very important if you plan to send large blocks of information over a slow network.

5.3.5 Recv

Prototype

```
Function Recv(Socket: Longint; Var Addr; AddrLen, Flags: Longint): Longint;
```

Description

This function is used to read incoming data from a socket.

Parameters

Socket	Is the socket handle returned by a previous call to Socket()
Addr	Buffer to receive the incoming data.
AddrLen	Size (in bytes) of the block of data to be received.
Flags	Indicates special operations to perform when receiving **MSG_PEEK** indicates that the received information should not be removed from the queue, and therefore could be read again.

Return value

Recv() returns the number of bytes actually received. If the operation completed successfully, the returned value should be equal to AddrLen.

Remarks

Recv() doesn’t block if there’s nothing to read. It just returns zero.

5.3.6 Bind

Prototype

```
Function Bind(Socket: Longint; Var Addr; AddrLen: Longint): Boolean;
```

Description

Binds the specified socket to the indicated local address or addresses. This operation is required when:

- Using SOCK_DGRAM sockets and you need to receive at a particular IP and/or port.
- Creating SOCK_STREAM server sockets.

Parameters

Socket	Is the socket handle returned by a previous call to Socket()
Addr	TInetSockAddr structure indicating the desired IP and/or port. If you plan to bind the socket to ALL available IP addresses, you can use INADDR_ANY instead of a particular address.
AddrLen	Size (in bytes) of the TinetSockAddr structure.

Return value

Bind returns True upon successful execution, or False otherwise.

5.3.7 Listen

Prototype

```
Function Listen(Socket, MaxConnect: Longint): Boolean;
```

Description

Puts a socket in “Listen” mode, also known as “Server” mode. The socket must be bound to a local address using the Bind() function prior to using Listen().

Parameters

Socket	Is the socket handle returned by a previous call to Socket()
MaxConnect	indicates the size of the incoming connections queue. This parameter is ignored, and internally set to 10. However, it is highly recommended to set this parameter to a nonzero value (such as ten) so that future implementations using this parameter don’t break existing applications.

Return value

Listen returns True upon successful execution, or False otherwise.

Remarks

MaxConnect doesn’t limit the number of concurrent connections on a particular server socket, but the size of the “pending connections queue”. This queue holds the incoming connections that haven’t been accepted yet through the Accept() function. In other words, you won’t be able to hold more than MaxConnect connections in the pending connections queue, but you can accept as many connections as you want, limited only by available memory.

5.4 Using the Web Server

This section is under construction.

5.5 Using the Web File System

This section is under construction.

5.6 Using the FTP Server

This section is under construction.

6 What’s missing

Here’s a list of features that are missing, or could be improved in the future:

a) I've tested this stack as much as I could. In fact I have more than 400 industrial PCs running applications that use it. However, I would really like someone to improve it in whatever way they find. The ideal would be to include this in the standard DWP libraries, so that more people can test it, improve it, and keep it up to date.

b) Support for ODI drivers is a bit odd. In fact, they are internally used as packet drivers, because ODIPKT is used to emulate a regular packet driver. It would be great to create a new ODI adapter object (something like “TODIAdapter”) capable of comunicating directly with ODI’s LSL.

c) Direct support for packet drivers (that is, without the need to use PKTPM) is a real “must”. However, I could not get it t work, because real-mode callbacks must be used, and I really couldn’t, after trying hard enough. If anyone knows how to do this, please let me know. In fact, I have the whole thing working under FreePascal. It just doesn’t work under DWP.

d) Any comments are welcome. I really did a hard job on developing this stack. I hope you find it useful. As with many other contributors, I don't have much time, but I'll try to answer any questions you have.

7 Revision history

Rev. 1.0 (1-Nov 2000)

First Release. Created the core-components:

- ARP
- IP
- ICMP
- UDP
- TCP
- DNS
- DHCP

Rev. 1.1 (21-Aug-2001)

Second release. I made this release available free to the FreePascal community.

- Minor bugs fixed.
- Added basic support for Web Servers.
- Added basic support for the Web File System.

Rev. 1.2 (18-Sep-2001)

Third release. Mostly bug fixes and improvements:

- Fixed memory leak in the DOSSOCK unit.
- Fixed bug when sending TCP packets, which affected performance.
- Fixed bug when receiving TCP packets, which affected performance.
- Moved all objects to use the “Delphi” convention (classes instead of objects).
- Renewed Web Server interface to make it compatible with Delphi.
- Added a new FTP server (based on Xon electronics’ FTP Server) with standard functionality.
- Added native support for Packet drivers. PKTPM is no longer needed, except for debug purposes.

Rev 1.2.1 (4-Dec-2001)

- Added the “Compilation” Chapter, indicating the switches needed for a successful compilation.

Rev 1.2.2 (19-Feb-2002)

- Changed some record definitions to “packed” to avoid compiler-settings conflicts, as suggested by “Skybuck” (skybuck2000@hotmail.com). These records include mostly packet headers for UDP, TCP, etc.

Rev 1.2.3 (24-Feb-2002)

- Added support for multiple IP addresses on the same network card. In other words, IP_Init() can be called several times on the same network adapter to create additional IP addresses bound to the same card. As usual, IP_Deinit() must be called as many times as IP_Init() in order to release all the allocated IP addresses.

Rev 1.2.4 (27-Aug-2002)

Added more documentation. Chapter 5.1 (new) explains how to start and manage the network step by step.

Rev 1.3.0 (27-Jan-2003)

Ported this stack to DWP. Updated the documentation. Ported many objects to use the “Delphi” syntax. A major breakthrough: cooperative multitasking was added.

Rev 1.3.1 (31-Jan-2003)

- Added more compatibility with WinSock. Many type definitions and prototypes have changed in accordance with WinSock 1.1 (and most are compatible with WinSock 2)
- Removed unit DosSock.pas (which simply acted as an intermediary between Sockets.pas and the protocol units).
- Updated the samples.

Rev 1.3.2 (12-May-2003)

- New Unit name Threads.pas updated. The library is now full incorporated to the DWPL library. All demos are tested and updated to the new unit names, compatible to DWPL naming conventions.

8 Legal considerations

I don't like doing this, but here comes the "legal" stuff, which is:

"I take no responsibility on any harm that could result from using this software, or any other software that uses any algorithm or piece of code derived from the source code or executables here enclosed.

Use at your own risk, blah, blah, blah....”