PC/TCP Packet Driver Specification

Revision 1.09
September-14-1989
Developed by:

FTP Software, Inc.
26 Princess St.
Wakefield, MA 01880-3004
(617) 246-0900

This specification was originally developed at FTP Software by John Romkey.

Support of a hardware interface, or mention of an interface manufacturer, by the Packet Driver specification does not necessarily indicate that the manufacturer endorses this specification.

1. Document Conventions

All numbers in this document are given in C -style representation. Decimal is expressed as 11, hexadecimal is expressed as 0x0B, octal is expressed as 013. All reference to network hardware addresses (source, destination and multicast) and demultiplexing information for the packet headers assumes they are represented as they would be in a MAC-level packet header being passed to the *send_pkt()* function.

2. Introduction and Motivation

This document describes the programming interface to FTP Software Packet Drivers. Packet drivers provide a simple, common programming interface that allows multiple applications to share a network interface at the data link level. The packet drivers demultiplex incoming packets among the applications by using the network media's standard packet type or service access point field(s).

The intent of this specification is to allow protocol stack implementations to be independent of the actual brand or model of the network interface in use on a particular machine. Different versions of various protocol stacks still must exist for different network media (Ethernet, 802.5 ring, serial lines), because of differences in

protocol-to-physical address mapping, header formats, maximum transmission units (MTUs) and so forth.

The packet driver provides calls to initiate access to a specific packet type, to end access to it, to send a packet, to get statistics on the network interface and to get information about the interface.

Protocol implementations that use the packet driver can completely coexist on a PC and can make use of one another's services, whereas multiple applications which do not use the driver do not coexist on one machine properly. Through use of the packet driver, a user could run TCP/IP, XNS, and a proprietary protocol implementation such as DECnet, Banyan's, LifeNet's, Novell's or 3Com's without the difficulties associated with pre-empting the network interface.

Applications which use the packet driver can also run on new network hardware of the same class without being modified; only a new packet driver need be supplied.

Several levels of packet drivers are described in this specification. The first is the **basic** packet driver, which provides minimal functionality but should be simple to implement and which uses very few host resources. The **basic** driver provides operations to broadcast and receive packets. The second driver is the **extended** packet driver, which is a superset of the **basic** driver. The **extended** driver supports less commonly used functions of the network interface such as multicast, and also gathers statistics on use of the interface and makes these available to the application. The third level, the **high-performance** functions, support performance improvements and tuning.

Functions which are available in only the **extended** packet driver are noted as such in their descriptions. All **basic** packet driver functions are available in the **extended** driver. The **high-performance** functions may be available with either **basic** or **extended** drivers.

3. Identifying network interfaces

Network interfaces are named by a triplet of integers, . The first number is the *class* of interface. The *class* tells what kind of media the interface supports: DEC/Intel/Xerox (DIX or Bluebook) Ethernet, IEEE 802.3 Ethernet, IEEE 802.5 Token Ring, ProNET-10, Appletalk, serial line, etc.

The second number is the *type* of interface: this specifies a particular instance of an interface supporting a class of network medium. Interface

types for Ethernet might name these interfaces: 3Com 3C503 or 3C505, Interlan NI5210, Univation, BICC Data Networks ISOLAN, Ungermann-Bass NIC, etc. Interface types for IEEE 802.5 might name these interfaces: IBM Token Ring adapter, Proteon p1340, etc.

The last number is the interface *number*. If a machine is equipped with more than one interface of a class and type, the interfaces must be numbered to distinguish between them.

An appendix details constants for *classes* and *types*. The *class* of an interface is an 8-bit integer, and its *type* is a 16 bit integer. *Class* and *type* constants are managed by FTP Software. Contact FTP to register a new *class* or *type* number.

The *type* 0xFFFF is a wildcard *type* which matches any interface in the specified *class*. It is unnecessary to wildcard interface *numbers*, as 0 will always correspond to the first interface of the specified *class* and *type*.

This specification has no provision for the support of multiple network interfaces (with similar or different characteristics) via a single Packet Driver and associated interrupt. We feel that this issue is best addressed by loading several Packet Drivers, one per interface, with different interrupts (although all may be included in a single TSR software module). Applications software must check the class and type returned from a *driver_info()* call in any case, to make sure that the Packet Driver is for the correct media and packet format. This can easily be generalized by searching for another Packet Driver if the first is not of the right kind.

4. Initiating driver operations

The packet driver is invoked via a software interrupt in the range 0x60 through 0x80. This document does not specify a particular interrupt, but describes a mechanism for locating which interrupt the driver uses. The interrupt must be configurable to avoid conflicts with other pieces of software that also use software interrupts. The program which installs the packet driver should provide some mechanism for the user to specify the interrupt.

The handler for the interrupt is assumed to start with 3 bytes of executable code; this can either be a 3-byte jump instruction, or a 2-byte jump followed by a NOP (do not specify "jmp short" unless you also specify an explicit NOP). This must be followed by the null-terminated ASCII text string "PKT DRVR". To find the interrupt being used by the driver, an

application should scan through the handlers for vectors 0x60 through 0x80 until it finds one with the text string "PKT DRVR" in the 12 bytes immediately following the entry point.

5. Link-layer demultiplexing

If a network media standard is to support simultaneous use by different transport protocols (e.g. TCP/IP, XNS, OSI), it must define some link-level mechanism which allows a host to decide which protocol a packet is intended for. In DIX Ethernet, this is the 16-bit "ethertype" field immediately following the 6-byte destination and source addresses. In IEEE 802.3 where 802.2 headers are used, this information is in the variable-length 802.2 header. In Proteon's ProNET-10, this is done via the 32-bit "type" field. Other media standards may demultiplex via a method of their own, or 802.2 headers as in 802.3.

Our *access_type()* function provides access to this link-layer demultiplexing. Each call establishes a destination for a particular type of link-layer packet, which remains in effect until *release_type()* is called with the *handle* returned by that particular *access_type()*. The link-layer demultiplexing information is passed via the *type* and *typelen* fields, but values and interpretation depend on the *class* of packet driver (and thus on the media in use).

A *class* 1 driver (DIX Ethernet) should expect *type* to point at an "ethertype" value (in network byte order, and greater than 0x05EE), and might reasonably require *typelen* to equal 2. A *class* 2 driver could require 4 bytes. However, a *class* 3 (802.5) or 11 (Ethernet with 802.2 headers) driver should be prepared for *typelen* values between 2 (for the DSAP/SSAP fields only) and 8 (3 byte 802.2 header plus 3-byte Sub-Network Access Protocol extension header plus 2-byte "ethertype" as defined in RFC 1042).

6. Programming interface

All functions are accessed via the software interrupt determined to be the driver's via the mechanism described earlier. On entry, register AH contains the code of the function desired.

The *handle* is an arbitrary integer value associated with each MAC-level demultiplexing *type* that has been established via the *access_type* call. Internally to the packet driver, it will probably be a pointer, a port table offset. The application calling the packet driver cannot depend on it assuming any particular range, or any other characteristics. In particular, if an application uses two or more packet drivers, handles

returned by different drivers for the same or different *type*s may have the same value.

Note that some of the functions defined below are labelled as **extended driver functions** and **high-performance functions**. Because these are not required for basic network operations, their implementation may be considered optional. Programs wishing to use these functions should use the *driver_info()* function to determine if they are available in a given packet driver.

6.1. Entry conditions

FTP Software applications which call the packet driver are coded in Microsoft C and assembly language. All necessary registers are saved by FTP's routines before the INT instruction to call the packet driver is executed. Our current *receiver()* functions behave as follows: DS, SS, SP and the flags are saved and restored. All other registers may be modified, and should be saved by the packet driver, if necessary. Processor interrupts may be enabled while in the upcall, but the upcall doesn't assume interrupts are disabled on entry. On entry, *receiver()* switches to a local stack, and switches back before returning.

Note that some older versions of PC/TCP may enable interrupts during the upcall, and leave them enabled on return to the Packet Driver.

When using a *class* 1 driver, PC/TCP will normally make 5 *access_type()* calls for IP, ARP and 3 kinds of Berkeley Trailer encapsulation packets. On other media, the number of handles we open will vary, but it is usually at least two (IP and ARP). Implementors should make their tables large enough to allow two protocol stacks to coexist. We recommend support for at least 10 open handles simultaneously.

6.2. Byte and Bit ordering

Developers should note that, on many networks and protocol families, the byte-ordering of 16-bit quantities on the network is opposite to the native byte-order of the PC. (802.5 Token Ring is an exception). This means that DEC-Intel-Xerox ethertype values passed to *access_type()* must be swapped (passed in network order). The IEE 802.3 *length* field needs similar handling, and care should be taken with packets passed to *send_pkt()*, so all fields are in the proper order. Developers working with MSB LANs (802.5 and FDDI) should be aware that a MAC address changes bit order depending on whether it appears in the header or as data.

6.3. *driver_info()*

```
        driver_info(handle)         AH == 1, AL == 255
                int     handle;     BX              /* Optional */

error return:
        carry flag set
        error code                  DH
        possible errors:
                BAD_HANDLE                          /* older drivers only */

non-error return:
        carry flag clear
        version                     BX
        class                       CH
        type                        DX
        number                      CL
        name                        DS:SI
        functionality               AL
                                        1 == basic functions present.
                                        2 == basic and extended present.
                                        5 == basic and high-performance.
                                        6 == basic, high-performance, extended.
                                        255 == not installed.
```

This function returns information about the interface. The version is assumed to be an internal hardware driver identifier. In earlier versions of this spec, the *handle* argument (which must have been obtained via *access_type()*) was required. It is now optional, but drivers developed according to versions of this spec previous to 1.07 may require it, so implementers should take care.

6.4. *access_type()*

```
        int access_type(if_class, if_type, if_number, type, typelen, receiver)
        AH == 2
                int     if_class;       AL
                int     if_type;        BX
                int     if_number;              DL
                char far *type;                 DS:SI
                unsigned typelen;       CX
                int     (far *receiver)();ES:DI

error return:
        carry flag set
```

```
        error code                          DH
        possible errors:
                NO_CLASS
                NO_TYPE
                NO_NUMBER
                BAD_TYPE
                NO_SPACE
                TYPE_INUSE

non-error return:
        carry flag clear
        handle                              AX

receiver call:
        (*receiver)(handle, flag, len [, buffer])
                int      handle;            BX
                int      flag;              AX
                unsigned len;               CX
        if AX == 1,
                char far *buffer; DS:SI
```

Initiates access to packets of the specified type. The argument*type* is a pointer to a packet type specification. The argument*typelen* is the length in bytes of the type field. The argument*receiver* is a pointer to a subroutine which is called when a packet is received. If the*typelen* argument is 0, this indicates that the caller wants to match all packets (match all requests may be refused by packet drivers developed to conform to versions of this spec previous to 1.07).

When a packet is received,*receiver* is called twice by the packet driver. The first time it is called to request abuffer from the application to copy the packet into. AX == 0 on this call. The application should return a pointer to the buffer in ES:DI. If the application has no buffers, it may return 0:0 in ES:DI, and the driver should throw away the packet and not perform the second call.

It is important that the packet length (CX) be valid on the AX == 0 call, so that the*receiver* can allocate a buffer of the proper size. This length (as well as the copy performed prior to the AX == 1 call) must include the MAC header and all received data, but not the trailing Frame Check Sequence (if any).

On the second call, AX == 1. This call indicates that the copy has been completed, and the application may do as it wishes with the buffer. The buffer that the packet was copied into is pointed to by DS:SI.

6.5. *release_type()*

```
        int release_type(handle)    AH == 3
                int     handle;         BX

error return:
        carry flag set
        error code                      DH
        possible errors:
                BAD_HANDLE

non-error return:
        carry flag clear
```

This function ends access to packets associated with a handle returned by *access_type()*. The handle is no longer valid.

6.6. *send_pkt()*

```
        int send_pkt(buffer, length)        AH == 4
                char far *buffer; DS:SI
                unsigned length;  CX

error return:
        carry flag set
        error code                      DH
        possible errors:
                CANT_SEND

non-error return:
        carry flag clear
```

Transmits *length* bytes of data, starting at *buffer*. The application must supply the entire packet, including local network headers. Any MAC or LLC information in use for packet demultiplexing (e.g. the DEC-Intel-Xerox Ethertype) must be filled in by the application as well. This cannot be performed by the driver, as no handle is specified in a call to the *send_packet()* function.

6.7. *terminate()*

```
        terminate(handle)           AH == 5
                int     handle;         BX
```

```
error return:
        carry flag set
        error code                          DH
        possible errors:
                BAD_HANDLE
                CANT_TERMINATE

non-error return:
        carry flag clear
```

Terminates the driver associated with *handle*. If possible, the driver will exit and allow MS-DOS to reclaim the memory it was using.

6.8. *get_address()*

```
        get_address(handle, buf, len)       AH == 6
                int     handle;             BX
                char far *buf;              ES:DI
                int     len;                CX

error return:
        carry flag set
        error code                          DH
        possible errors:
                BAD_HANDLE
                NO_SPACE

non-error return:
        carry flag clear
        length                      CX
```

Copies the current local net address of the interface into *buf*. The buffer *buf* is *len* bytes long. The actual number of bytes copied is returned in CX. If the NO_SPACE error is returned, this indicates that *len* was insufficient to hold the local net address. If the address has been changed by *set_address()*, the new address should be returned.

6.9. *reset_interface()*

```
        reset_interface(handle)             AH == 7
                int     handle;             BX

error return:
```

```
        carry flag set
        error code                          DH
        possible errors:
                BAD_HANDLE
                CANT_RESET

non-error return:
        carry flag clear
```

Resets the interface associated with *handle* to a known state, aborting any transmits in process and reinitializing the receiver. The local net address is reset to the default (from ROM), the multicast list is cleared, and the receive *mode* is set to 3 (own address & broadcasts). If multiple handles are open, these actions might seriously disrupt other applications using the interface, so CANT_RESET should be returned.

6.10. *get_parameters()*

high-performance driver function

```
        get_parameters()            AH = 10

error return:
        carry flag set
        error code                          DH
        possible errors:
                BAD_COMMAND                 /* No high-performance support */

non error return:
        carry flag clear
        struct param far *;                 ES:DI

    struct param {
        unsigned char     major_rev;        /* Revision of Packet Driver spec */
        unsigned char     minor_rev;        /*  this driver conforms to. */
        unsigned char     length;           /* Length of structure in bytes */
        unsigned char     addr_len;/* Length of a MAC-layer address */
        unsigned short    mtu;              /* MTU, including MAC headers */
        unsigned short    multicast_aval;   /* Buffer size for multicast addr */
        unsigned short    rcv_bufs;/* (# of back-to-back MTU rcvs) - 1 */
        unsigned short    xmt_bufs;/* (# of successive xmits) - 1 */
        unsigned short    int_num; /* Interrupt # to hook for post-EOI
                                      processing, 0 == none */
    };
```

The performance of an application may benefit from using *get_parameters()* to obtain

a number of driver parameters. This function was added to v1.09 of this specification, and may not be implemented by all drivers (see *driver_info()*).

The *major_rev* and *minor_rev* fields are the major and minor revision numbers of the version of this specification the driver conforms to. For this document, *major_rev* is 1 and *minor_rev* is 9. The *length* field may be used to determine which values are valid, should a later revision of this specification add more values at the end of this structure. For this document, *length* is 14. The *addr_len* field is the length of a MAC address, in bytes. Note the *param* structure is assumed to be packed, such that these fields occupy four consecutive bytes of storage.

In the *param* structure, the *mtu* is the maximum MAC-level packet the driver can handle (on Ethernet this number is fixed, but it may vary on other media, e.g. 802.5 or FDDI). The *multicast_aval* field is the number of bytes required to store all the multicast addresses supported by any "perfect filter" mechanism in the hardware. Calling *get_multicast_list()* with its *len* argument equal to this value should not fail with a NO_SPACE error. A value of zero implies no multicast support.

The *rcv_bufs* and *xmt_bufs* indicate the number of back-to-back receives or transmits the card/driver combination can accomodate, minus 1. The application can use this information to adjust flow-control or transmit strategies. A single-buffered card (for example, an Interlan NI5010) would normally return 0 in both fields. A value of 0 in *rcv_bufs* could also be used by a driver author to indicate that the hardware has limitations which prevent it from receiving as fast as other systems can send, and to recommend that the upper-layer protocols invoke lock-step flow control to avoid packet loss.

The *int_num* field should be set to a hardware interrupt that the application can hook in order to perform interrupt-time protocol processing after the EOI has been sent to the 8259 interrupt controller and the card is ready for more interrupts. A value of zero indicates that there is no such interrupt. Any application hooking this interrupt and finding a non-zero value in the vector **must** pass the interrupt down the chain and wait for its predecessors to return before performing any processing or stack switches. Any driver which implements this function via a separate INT instruction and vector, instead of just using the hardware interrupt, **must** prevent any saved context from being overwritten by a later interrupt. In other words, if the driver switches to its own stack, it must allow re-entrancy.

6.11. *as_send_pkt()*

high-performance driver function

```
        int as_send_pkt(buffer, length, upcall)       AH == 11
                char far *buffer;           DS:SI
                unsigned length;            CX
                int     (far *upcall)();    ES:DI

error return:
        carry flag set
        error code                          DH
        possible errors:
                CANT_SEND           /* transmit error, re-entered, etc. */
                BAD_COMMAND                 /* Level 0 or 1 driver */

non-error return:
        carry flag clear

buffer available upcall:
        (*upcall)(buffer, result)
                int     result;             AX      /* 0 for copy ok */
                char far *buffer; ES:DI     /* from as_send_pkt() call */
```

as_send_pkt() differs from *send_pkt()* in that the *upcall()* routine is called when the application's data has been copied out of the *buffer*, and the application can safely modify or re-use the buffer. The driver may pass a non-zero error code to *upcall()* if the copy failed, or some other error was detected, otherwise it should indicate success, even if the packet hasn't actually been transmitted yet. Note that the *buffer* passed to *send_pkt()* is assumed to be modifiable when that call returns, whereas with *as_send_pkt()*, the *buffer* may be queued by the driver and dealt with later. If an error is returned on the initial call, the *upcall* will not be executed. This function was added in v1.09 of this specification, and may not be implemented by all drivers (see *driver_info()*).

6.12. *set_rcv_mode()*

extended driver function

```
        set_rcv_mode(handle, mode)  AH == 20
                int     handle;             BX
                int     mode;               CX


error return:
        carry flag set
        error code                  DH
        possible errors:
```

BAD_HANDLE
BAD_MODE

non-error return:
carry flag clear

Sets the receive mode on the interface associated with *handle*. The following values are accepted for *mode*:

1	turn off receiver
2	receive only packets sent to this interface
3	mode 2 plus broadcast packets
4	mode 3 plus limited multicast packets
5	mode 3 plus all multicast packets
6	all packets

Note that not all interfaces support all modes. The receive mode affects all packets received by this interface, not just packets associated with the *handle* argument. See the **extended driver functions** *get_multicast_list()* and *set_multicast_list()* for programming "perfect filters" to receive specific multicast addresses.

Note that *mode* 3 is the default, and if the *set_rcv_mode()* function is not implemented, *mode* 3 is assumed to be in force as long as any handles are open (from the first *access_type()* to the last *release_type()*).

6.13. *get_rcv_mode()*

extended driver function
get_rcv_mode(handle, mode) AH == 21
int handle; BX

error return:
carry flag set
error code DH
possible errors:
BAD_HANDLE

non-error return:
carry flag clear
mode AX

Returns the current receive mode of the interface associated with *handle*.

6.14. *set_multicast_list()*

extended driver function

```
        set_multicast_list(addrlst, len)     AH == 22
                char far *addrlst;ES:DI
                int      len;                CX

error return:
        carry flag set
        error code                           DH
        possible errors:
                NO_MULTICAST
                NO_SPACE
                BAD_ADDRESS

non-error return:
        carry flag clear
```

The *addrlst* argument is assumed to point to a *len*-byte buffer containing a number of multicast addresses. BAD_ADDRESS is returned if *len* modulo the size of an address is not equal to 0, or the data is unacceptable for some reason. NO_SPACE is returned (and no addresses are set) if there are more addresses than the hardware supports directly.

The recommended procedure for setting multicast addresses is to issue a *get_multicast_list()*, copy the information to a local buffer, add any addresses desired, and issue a *set_multicast_list()*. This should be reversed when the application exits. If the *set_multicast_list()* fails due to NO_SPACE, use *set_rcv_mode()* to set *mode* 5 instead.

6.15. *get_multicast_list()*

extended driver function

```
        get_multicast_list()                 AH == 23

error return:
        carry flag set
        error code                           DH
        possible errors:
                NO_MULTICAST
                NO_SPACE

non-error return:
        carry flag clear
        char far *addrlst;          ES:DI
        int      len;                        CX
```

On a successful return, *addrlst* points to *len* bytes of multicast addresses currently in use. The application program must not modify this information in place. A NO_SPACE error indicates that there wasn't enough room for all active multicast addresses.

6.16. *get_statistics()*

extended driver function

```
        get_statistics(handle)              AH == 24
                int handle;                 BX

error return:
        carry flag set
        error code                          DH
        possible errors:
                BAD_HANDLE

non-error return:
        carry flag clear
        char far *stats;            DS:SI

    struct statistics {
        unsigned long     packets_in;       /* Totals across all handles */
        unsigned long     packets_out;
        unsigned long     bytes_in;/* Including MAC headers */
        unsigned long     bytes_out;
        unsigned long     errors_in;        /* Totals across all error types */
        unsigned long     errors_out;
        unsigned long     packets_lost;     /* No buffer from receiver(), card */
                                            /*  out of resources, etc. */
    };
```

Returns a pointer to a statistics structure for the interface. The values are stored as to be normal 80xx 32-bit integers.

6.17. *set_address()*

extended driver function

```
        set_address(addr, len)              AH == 25
                char far *addr;             ES:DI
                int len;            CX

error return:
        carry flag set
```

error code DH

possible errors:

CANT_SET

BAD_ADDRESS

non-error return:

carry flag clear

length CX

This call is used when the application or protocol stack needs to use a specific LAN address. For instance, DECnet protocols on Ethernet encode the protocol address in the Ethernet address, requiring that it be set when the protocol stack is loaded. A BAD_ADDRESS error indicates that the Packet Driver doesn't like the *len* (too short or too long), or the data itself. Note that packet drivers should refuse to change the address (with a CANT_SET error) if more than one handle is open (lest it be changed out from under another protocol stack).

Appendix A: Interface classes and types

The following are defined as network interface classes, with their individual types listed immediately following the class.

DEC/Intel/Xerox "Bluebook" Ethernet Class 1

3COM 3C500/3C501	1	
3COM 3C505		2
Interlan Ni5010		3
BICC Data Networks 4110		4
BICC Data Networks 4117		5
MICOM-Interlan NP600		6
Ungermann-Bass PC-NIC		8
Univation NC-516	9	
TRW PC-2000		10
Interlan Ni5210		11
3COM 3C503		12
3COM 3C523		13
Western Digital WD8003		14
Spider Systems S4	15	
Torus Frame Level	16	
1ONET Communications		17
Gateway PC-bus		18
Gateway AT-bus		19
Gateway MCA-bus		20
IMC PCnic	21	
IMC PCnic II		22

IMC PCnic 8bit	23
Tigan Communications	24
Micromatic Research	25
Clarkson "Multiplexor"	26
D-Link 8-bit	27
D-Link 16-bit	28
D-Link PS/2	29
Research Machines 8	30
Research Machines 16	31
Research Machines MCA	32
Radix Microsys. EXM1 16-bit	33
Interlan Ni9210	34
Interlan Ni6510	35
Vestra LANMASTER 16-bit	36
Vestra LANMASTER 8-bit	37
Allied Telesis PC/XT/AT	38
Allied Telesis NEC PC-98	39
Allied Telesis Fujitsu FMR	40
Ungermann-Bass NIC/PS2	41
Tiara LANCard/E AT	42
Tiara LANCard/E MC	43
Tiara LANCard/E TP	44
Spider Comm. SpiderComm 8	45
Spider Comm. SpiderComm 16	46
AT&T Starlan NAU	47
AT&T Starlan-10 NAU	48
AT&T Ethernet NAU	49
Intel smart card	50

ProNET-10 Class 2

Proteon p1300	1
Proteon p1800	2

IEEE 802.5/ProNET-4 Class 3

IBM Token ring adapter	1
Proteon p1340	2
Proteon p1344	3
Gateway PC-bus	4
Gateway AT-bus	5
Gateway MCA-bus	6

Omninet Class 4

Appletalk Class 5

Serial line Class 6

Clarkson 8250-SLIP	1
Clarkson "Multiplexor"	2

Starlan Class 7

(NOTE: Class has been subsumed by Ethernet)

ArcNet Class 8

Datapoint RIM	1

AX.25 Class 9

KISS Class 10

IEE 802.3 w/802.2 hdrs Class 11

Types as given under DIX Ethernet
See Appendix D.

FDDI w/802.2 hdrs Class 12

Internet X.25 Class 13

Western Digital	1
Frontier Technology	2

N.T. LANSTAR (encapsulati ng DIX) Class 14

NT LANSTAR/8	1
NT LANSTAR/MC	2

Appendix B: Function call numbers

The following decimal numbers are used to specify which operation the packet driver should perform. The number is stored in register AH on call to the packet driver.

driver_info	1
access_type	2
release_type	3
send_pkt	4
terminate	5

get_address	6
reset_interface	7
+get_parameters	10
+as_send_pkt	11
*set_rcv_mode	20
*get_rcv_mode	21
*set_multicast_list	22
*get_multicast_list	23
*get_statistics	24
*set_address	25

+ indicates a **high-performance** packet driver function
* indicates an **extended** packet driver function

AH values from 128 through 255 (0x80 through 0xFF) are reserved for user-developed extensions to this specification. While FTP Software cannot support user extensions, we are willing to act as a clearing house for information about them. For more information, contact us.

Appendix C: Error codes

Packet driver calls indicate error by setting the carry flag on return. The error code is returned in register DH (a register not used to pass values to functions *must* be used to return the error code). The following error codes are defined:

1	BAD_HANDLE	Invalid handle number,
2	NO_CLASS	No interfaces of specified class found,
3	NO_TYPE	No interfaces of specified type found,
4	NO_NUMBER	No interfaces of specified number found,
5	BAD_TYPE	Bad packet type specified,
6	NO_MULTICAST	This interface does not support multicast,
7	CANT_TERMINATE	This packet driver cannot terminate,
8	BAD_MODE	An invalid receiver mode was specified,
9	NO_SPACE	Operation failed because of insufficient space,
10	TYPE_INUSE	The *type* had previously been accessed, and not released,
11	BAD_COMMAND	The command was out of range, or not implemented,
12	CANT_SEND	The packet couldn't be sent (usually hardware error),
13	CANT_SET	Hardware address couldn't be changed (more than 1 handle open),
14	BAD_ADDRESS	Hardware address has bad length or format,
15	CANT_RESET	Couldn't reset interface (more than 1 handle open).

Appendix D: 802.3 vs. Blue Book Ethernet

One weakness of the present specification is that there is no provision for simultaneous support of 802.3 and Blue Book (the old DEC-Intel-Xerox standard) Ethernet headers via a single Packet Driver (as defined by its interrupt). The problem is that the "ethertype" of Blue Book packets is in bytes 12 and 13 of the header, and in 802.3 the corresponding bytes are interpreted as a length. In 802.3, the field which would appear to be most useful to begin the *type* check in is the 802.2 header, starting at byte 14. This is only a problem on Ethernet and variants (e.g. Starlan), where 802.3 headers and Blue Book headers are likely to need to coexist for many years to come.

One solution is to redefine class 1 as Blue Book Ethernet, and define a parallel *class* for 802.3 with 802.2 packet headers. This requires that a 2nd Packet Driver (as defined by its interrupt) be implemented where it is necessary to handle both kinds of packets, although they could both be part of the same TSR module.

As of v1.07 of this specification, *class* 11 was assigned to 802.3 using 802.2 headers, to implement the above.

Note: According to this scheme, an application wishing to receive IP encapsulated with an 802.2 SNAP header and "ethertype" of 0x800, per RFC 1042, would specify an *typelen* argument of 8, and *type* would point to:

```
        char    iee_ip[] = {0xAA, 0xAA, 3, 0, 0, 0, 0x00, 0x08};
```

James B. VanBokkelen
jbvb@@ftp.com
...!ftp!jbvb

char iee_ip[] = {0xAA, 0xAA, 3, 0, 0, 0, 0x00, 0x08};
James B. VanBokkelen
jbvb@@ftp.com
...!ftp!jbvb