PCAP

Section: C Library Functions (3)
Updated: 21 November 2003
[INDEX]
From: http://www.tcpdump.org/

NAME

pcap - Packet Capture library

Description

The Packet Capture library provides a high level interface to packet capture systems. All packets on the network, even those destined for other hosts, are accessible through this mechanism.

Edited by Rhett, SJTU, 2011

APIs

1. char errbuf[PCAP_ERRBUF_SIZE];
2. pcap_t *pcap_open_live(const char *device, int snaplen, int promisc, int to_ms, char *errbuf)
3. pcap_t *pcap_open_dead(int linktype, int snaplen)
4. pcap_t *pcap_open_offline(const char *fname, char *errbuf)
5. pcap_dumper_t *pcap_dump_open(pcap_t *p, const char *fname)
6. int pcap_setnonblock(pcap_t *p, int nonblock, char *errbuf);
7. int pcap_getnonblock(pcap_t *p, char *errbuf);
8. int pcap_findalldevs(pcap_if_t **alldevsp, char *errbuf)
9. void pcap_freealldevs(pcap_if_t *alldevs)
10. char *pcap_lookupdev(char *errbuf)
11. int pcap_lookupnet(const char *device, bpf_u_int32 *netp,bpf_u_int32 *maskp, char *errbuf)
12. int pcap_dispatch(pcap_t *p, int cnt, pcap_handler callback, u_char *user)
13. int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)
14. void pcap_dump(u_char *user, struct pcap_pkthdr *h, u_char *sp)
15. int pcap_compile(pcap_t *p, struct bpf_program *fp, char *str, int optimize, bpf_u_int32 netmask)
16. int pcap_setfilter(pcap_t *p, struct bpf_program *fp)
17. void pcap_freecode(struct bpf_program *);
18. const u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)
19. int pcap_next_ex(pcap_t *p, struct pcap_pkthdr **pkt_header, const u_char **pkt_data)
20. void pcap_breakloop(pcap_t *)
21. int pcap_datalink(pcap_t *p)
22. int pcap_list_datalinks(pcap_t *p, int **dlt_buf);
23. int pcap_set_datalink(pcap_t *p, int dlt);
24. int pcap_datalink_name_to_val(const char *name);
25. const char *pcap_datalink_val_to_name(int dlt);
26. const char *pcap_datalink_val_to_description(int dlt);
27. int pcap_snapshot(pcap_t *p)
28. int pcap_is_swapped(pcap_t *p)
29. int pcap_major_version(pcap_t *p)
30. int pcap_minor_version(pcap_t *p)
31. int pcap_stats(pcap_t *p, struct pcap_stat *ps)
32. FILE *pcap_file(pcap_t *p)
33. int pcap_fileno(pcap_t *p)
34. void pcap_perror(pcap_t *p, char *prefix)
35. char *pcap_geterr(pcap_t *p)
36. char *pcap_strerror(int error)
37. const char *pcap_lib_version(void)
38. void pcap_close(pcap_t *p)
39. int pcap_dump_flush(pcap_dumper_t *p)
40. FILE *pcap_dump_file(pcap_dumper_t *p)
41. void pcap_dump_close(pcap_dumper_t *p)

Definition

- **#include <pcap.h>**

- **char errbuf[PCAP_ERRBUF_SIZE];** [INDEX]

NOTE: errbuf in **pcap_open_live(), pcap_open_dead(), pcap_open_offline(), pcap_setnonblock(),pcap_getnonblock(), pcap_findalldevs(), pcap_lookupdev(),** and **pcap_lookupnet()** is assumed to be able to hold at least **PCAP_ERRBUF_SIZE** chars.

- **pcap_t *pcap_open_live(const char *device, int snaplen, int promisc, int to_ms, char *errbuf)** [INDEX]

pcap_open_live() is used to obtain a packet capture descriptor to look at packets on the network. device is a string that specifies the network device to open; on Linux systems with 2.2 or later kernels, a device argument of "any" or **NULL** can be used to capture packets from all interfaces. snaplen specifies the maximum number of bytes to capture. If this value is less than the size of a packet that is captured, only the first snaplen bytes of that packet will be captured and provided as packet data. A value of 65535 should be sufficient, on most if not all networks, to capture all the data available from the packet. promisc specifies if the interface is to be put into promiscuous mode. (**Note that even if this parameter is false, the interface could well be in promiscuous mode for some other reason.**) For now, this doesn't work on the "any" device; if an argument of "any" or **NULL** is supplied, the promisc flag is ignored. to_ms specifies the read timeout in milliseconds. The read timeout is used to arrange that the read not necessarily return immediately when a packet is seen, but that it wait for some amount of time to allow more packets to arrive and to read multiple packets from the OS kernel in one operation. Not all platforms support a read timeout; on platforms that don't, the read timeout is ignored. A zero value *forto_ms*, on platforms that support a read timeout, will cause a read to wait forever to allow enough packets to arrive, with no timeout. errbuf is used to return error or warning text. It will be set to error text when **pcap_open_live()** fails and returns **NULL**. errbuf may also be set to warning text when **pcap_open_live()** succeds; to detect this case the caller should store a zero-length string in errbuf before calling **pcap_open_live()** and display the warning to the user if errbuf is no longer a zero-length string.

- **pcap_t *pcap_open_dead(int linktype, int snaplen)** [INDEX]

pcap_open_dead() is used for creating a **pcap_t** structure to use when calling the other functions in libpcap. It is typically used when just using libpcap for compiling BPF code.

➢ pcap_t *pcap_open_offline(const char *fname, char *errbuf) [INDEX]

pcap_open_offline() is called to open a ``savefile'' for reading. fname specifies the name of the file to open. The file has the same format as those used by tcpdump(1) and tcpslice(1). The name "-" in a synonym for **stdin**. *errbuf* is used to return error text and is only set when **pcap_open_offline()** fails and returns **NULL**.

➢ pcap_dumper_t *pcap_dump_open(pcap_t *p, const char *fname) [INDEX]

pcap_dump_open() is called to open a ``savefile'' for writing. The name "-" in a synonym for **stdout**. **NULL** is returned on failure. *p* is a *pcap* struct as returned by **pcap_open_offline()** or **pcap_open_live().** *fname* specifies the name of the file to open. If **NULL** is returned, **pcap_geterr()** can be used to get the error text.

➢ int pcap_setnonblock(pcap_t *p, int nonblock, char *errbuf); [INDEX]

pcap_setnonblock() puts a capture descriptor, opened with **pcap_open_live()**, into ``non-blocking'' mode, or takes it out of ``non-blocking'' mode, depending on whether the *nonblock* argument is non-zero or zero. It has no effect on ``savefiles''. If there is an error, -1 is returned and *errbuf* is filled in with an appropriate error message; otherwise, 0 is returned. In ``non-blocking'' mode, an attempt to read from the capture descriptor with **pcap_dispatch()** will, if no packets are currently available to be read, return 0 immediately rather than blocking waiting for packets to arrive. **pcap_loop()** and **pcap_next()** will not work in ``non-blocking'' mode.

➢ int pcap_getnonblock(pcap_t *p, char *errbuf); [INDEX]

pcap_getnonblock() returns the current ``non-blocking'' state of the capture descriptor; it always returns 0 on ``savefiles''. If there is an error, -1 is returned and *errbuf* is filled in with an appropriate error message.

➢ int pcap_findalldevs(pcap_if_t **alldevsp, char *errbuf) [INDEX]

pcap_findalldevs() constructs a list of network devices that can be opened with **pcap_open_live()**. (Note that there may be network devices that cannot be opened with **pcap_open_live()** by the process calling **pcap_findalldevs(),** because, for example, that process might not have sufficient privileges to open them for capturing; if so, those devices will not appear on the list.) *alldevsp* is set to point to the first element of the list; each element of the list is of type **pcap_if_t**, and has the following members:

next

if not **NULL**, a pointer to the next element in the list; **NULL** for the last element of the list

name

a pointer to a string giving a name for the device to pass to **pcap_open_live()**

description

if not **NULL**, a pointer to a string giving a human-readable description of the device

addresses

a pointer to the first element of a list of addresses for the interface

flags

interface flags:

PCAP_IF_LOOPBACK

set if the interface is a loopback interface

Each element of the list of addresses is of type **pcap_addr_t**, and has the following members:

next

if not **NULL**, a pointer to the next element in the list; **NULL** for the last element of the list

addr

a pointer to a **struct sockaddr** containing an address

netmask

if not **NULL**, a pointer to a **struct sockaddr** that contains the netmask corresponding to the address pointed to by **addr**

broadaddr

if not **NULL**, a pointer to a **struct sockaddr** that contains the broadcast address corresponding to the address pointed to by **addr;** may be null if the interface doesn't support broadcasts

dstaddr

if not **NULL**, a pointer to a **struct sockaddr** that contains the destination address corresponding to the address pointed to by **addr**; may be null if the interface isn't a point-to-point interface

-1 is returned on failure, in which case *errbuf* is filled in with an appropriate error message; **0** is returned on success.

- **void pcap_freealldevs(pcap_if_t *alldevs)** [INDEX]

pcap_freealldevs() is used to free a list allocated by **pcap_findalldevs()**.

- **char *pcap_lookupdev(char *errbuf)** [INDEX]

pcap_lookupdev() returns a pointer to a network device suitable for use with **pcap_open_live()** and **pcap_lookupnet()**. If there is an error, **NULL** is returned and *errbuf* is filled in with an appropriate error message.

- **int pcap_lookupnet(const char *device, bpf_u_int32 *netp, bpf_u_int32 *maskp, char *errbuf)** [INDEX]

pcap_lookupnet() is used to determine the network number and mask associated with the network device *device*. Both *netp* and *maskp* are **bpf_u_int32** pointers. A return of -1 indicates an error in which case *errbuf* is filled in with an appropriate error message.

➢ int pcap_dispatch(pcap_t *p, int cnt, pcap_handler callback, u_char *user)

[INDEX]

pcap_dispatch() is used to collect and process packets. *cnt* specifies the maximum number of packets to process before returning. This is not a minimum number; when reading a live capture, only one bufferful of packets is read at a time, so fewer than *cnt* packets may be processed. A *cnt* of -1 processes all the packets received in one buffer when reading a live capture, or all the packets in the file when reading a ``savefile''. *callback* specifies a routine to be called with three arguments: a **u_char** pointer which is passed in from **pcap_dispatch()**, a **const struct** *pcap_pkthdr* pointer to a structure with the following members:

ts
: a *struct timeval* containing the time when the packet was captured

caplen
: a **bpf_u_int32** giving the number of bytes of the packet that are available from the capture

len
: a **bpf_u_int32** giving the length of the packet, in bytes (which might be more than the number of bytes available from the capture, if the length of the packet is larger than the maximum number of bytes to capture)

and a **const u_char** pointer to the first **caplen** (as given in the *struct pcap_pkthdr* a pointer to which is passed to the callback routine) bytes of data from the packet (which won't necessarily be the entire packet; to capture the entire packet, you will have to provide a value for *snaplen* in your call to **pcap_open_live()** that is sufficiently large to get all of the packet's data - a value of 65535 should be sufficient on most if not all networks).

The number of packets read is returned. 0 is returned if no packets were read from a live capture (if, for example, they were discarded because they didn't pass the packet filter, or if, on platforms that support a read timeout that starts before any packets arrive, the timeout expires before any packets arrive, or if the file descriptor for the capture device is in non-blocking mode and no packets were available to be read) or if no more packets are available in a “savefile.” A return of -1 indicates an error in which case **pcap_perror()** or **pcap_geterr()** may be used to display the error text. A return of -2 indicates that the loop terminated due to a call to **pcap_breakloop()** before any packets were processed. If your application uses **pcap_breakloop()**, make sure that you explicitly check for -1 and -2, rather than just checking for a return value < 0.

NOTE: when reading a live capture, **pcap_dispatch()** will not necessarily return when the read times out; on some platforms, the read timeout isn't supported, and, on other

platforms, the timer doesn't start until at least one packet arrives. This means that the read timeout should **NOT** be used in, for example, an interactive application, to allow the packet capture loop to ``poll'' for user input periodically, as there's no guarantee that **pcap_dispatch()** will return after the timeout expires.

➢ int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user) [INDEX]

pcap_loop() is similar to **pcap_dispatch()** except it keeps reading packets until *cnt* packets are processed or an error occurs. It does **not** return when live read timeouts occur. Rather, specifying a non-zero read timeout to **pcap_open_live()** and then calling **pcap_dispatch()** allows the reception and processing of any packets that arrive when the timeout occurs. A negative *cnt* causes **pcap_loop()** to loop forever (or at least until an error occurs). -1 is returned on an error; 0 is returned if *cnt* is exhausted; -2 is returned if the loop terminated due to a call to **pcap_breakloop()** before any packets were processed. If your application uses **pcap_breakloop(),** make sure that you explicitly check for -1 and -2, rather than just checking for a return value < 0.

➢ void pcap_dump(u_char *user, struct pcap_pkthdr *h, u_char *sp) [INDEX]

pcap_dump() outputs a packet to the ``savefile'' opened with **pcap_dump_open()**. Note that its calling arguments are suitable for use with **pcap_dispatch()** or **pcap_loop()**. If called directly, the user parameter is of type *pcap_dumper_t* as returned by **pcap_dump_open()**.

➢ int pcap_compile(pcap_t *p, struct bpf_program *fp, char *str, int optimize, bpf_u_int32 netmask) [INDEX]

pcap_compile() is used to compile the string *str* into a filter program. *program* is a pointer to a *bpf_program* struct and is filled in by **pcap_compile()**. *optimize* controls whether optimization on the resulting code is performed. *netmask* specifies the IPv4 netmask of the network on which packets are being captured; it is used only when checking for IPv4 broadcast addresses in the filter program. If the netmask of the network on which packets are being captured isn't known to the program, or if packets are being captured on the Linux "any" pseudo-interface that can capture on more than one network, a value of 0 can be supplied; tests for IPv4 broadcast addresses won't be done correctly, but all other tests in the filter program will be OK. A return of -1 indicates an error in which case **pcap_geterr()** may be used to display the error text.

pcap_compile_nopcap() is similar to **pcap_compile()** except that instead of passing a pcap structure, one passes the snaplen and linktype explicitly. It is intended to be used for compiling filters for direct BPF usage, without necessarily having called **pcap_open()**. A return of -1 indicates an error; the error text is unavailable. (**pcap_compile_nopcap()** is a wrapper around **pcap_open_dead()**, **pcap_compile()**, and **pcap_close()**; the latter three routines can be used directly in order to get the error text for a compilation error.)

- **int pcap_setfilter(pcap_t *p, struct bpf_program *fp)** [INDEX]

pcap_setfilter() is used to specify a filter program. *fp* is a pointer to a *bpf_program* struct, usually the result of a call to **pcap_compile()**. -1 is returned on failure, in which case **pcap_geterr()** may be used to display the error text; **0** is returned on success.

- **void pcap_freecode(struct bpf_program *);** [INDEX]

pcap_freecode() is used to free up allocated memory pointed to by a *bpf_program* struct generated by **pcap_compile()** when that BPF program is no longer needed, for example after it has been made the filter program for a pcap structure by a call to **pcap_setfilter().**

- **const u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)** [INDEX]

pcap_next() reads the next packet (by calling **pcap_dispatch()** with a *cnt* of 1) and returns a **u_char** pointer to the data in that packet. (The *pcap_pkthdr* struct for that packet is not supplied.) **NULL** is returned if an error occured, or if no packets were read from a live capture (if, for example, they were discarded because they didn't pass the packet filter, or if, on platforms that support a read timeout that starts before any packets arrive, the timeout expires before any packets arrive, or if the file descriptor for the capture device is in non-blocking mode and no packets were available to be read), or if no more packets are available in a ``savefile.'' Unfortunately, there is no way to determine whether an error occured or not.

- **int pcap_next_ex(pcap_t *p, struct pcap_pkthdr **pkt_header, const u_char **pkt_data)** [INDEX]

pcap_next_ex() reads the next packet and returns a success/failure indication:

1
: the packet was read without problems

0
: packets are being read from a live capture, and the timeout expired

-1
: an error occurred while reading the packet

-2
: packets are being read from a ``savefile'', and there are no more packets to read from the savefile.

If the packet was read without problems, the pointer pointed to by the *pkt_header* argument is set to point to the *pcap_pkthdr* struct for the packet, and the pointer pointed to by the *pkt_data* argument is set to point to the data in the packet.

- **void pcap_breakloop(pcap_t *)** [INDEX]

pcap_breakloop() sets a flag that will force **pcap_dispatch()** or **pcap_loop()** to return rather than looping; they will return the number of packets that have been processed so far, or -2 if no packets have been processed so far.

This routine is safe to use inside a signal handler on UNIX or a console control handler on Windows, as it merely sets a flag that is checked within the loop.

The flag is checked in loops reading packets from the OS - a signal by itself will not necessarily terminate those loops - as well as in loops processing a set of packets returned by the OS. **Note that if you are catching signals on UNIX systems that support restarting system calls after a signal, and calling pcap_breakloop() in the signal handler, you must specify, when catching those signals, that system calls should NOT be restarted by that signal. Otherwise, if the signal interrupted a call reading packets in a live capture, when your signal handler returns after calling pcap_breakloop(), the call will be restarted, and the loop will not terminate until more packets arrive and the call completes.**

Note that **pcap_next()** will, on some platforms, loop reading packets from the OS; that loop will not necessarily be terminated by a signal, so **pcap_breakloop()** should be used to terminate packet processing even if **pcap_next()** is being used.

pcap_breakloop() does not guarantee that no further packets will be processed by **pcap_dispatch()** or **pcap_loop()** after it is called; at most one more packet might be processed.

If -2 is returned from **pcap_dispatch()** or **pcap_loop()**, the flag is cleared, so a subsequent call will resume reading packets. If a positive number is returned, the flag is not cleared, so a subsequent call will return -2 and clear the flag.

- **int pcap_datalink(pcap_t *p)** [INDEX]

pcap_datalink() returns the link layer type; link layer types it can return include:

DLT_NULL

> BSD loopback encapsulation; the link layer header is a 4-byte field, in *host* byte order, containing a PF_ value from **socket.h** for the network-layer protocol of the packet.
>
> Note that ``host byte order'' is the byte order of the machine on which the packets are captured, and the PF_ values are for the OS of the machine on which the packets are captured; if a live capture is being done, ``host byte order'' is the byte order of the machine capturing the packets, and the PF_ values are those of the OS of the machine capturing the packets, but if a ``savefile'' is being read, the byte order and PF_ values are not necessarily those of the machine reading the capture file.

DLT_EN10MB

Ethernet (10Mb, 100Mb, 1000Mb, and up)

DLT_IEEE802

IEEE 802.5 Token Ring

DLT_ARCNET

ARCNET

DLT_SLIP

SLIP; the link layer header contains, in order:

a 1-byte flag, which is 0 for packets received by the machine and 1 for packets sent by the machine;

a 1-byte field, the upper 4 bits of which indicate the type of packet, as per RFC 1144:

0x40
an unmodified IP datagram (TYPE_IP);

0x70
an uncompressed-TCP IP datagram (UNCOMPRESSED_TCP), with that byte being the first byte of the raw IP header on the wire, containing the connection number in the protocol field;

0x80
a compressed-TCP IP datagram (COMPRESSED_TCP), with that byte being the first byte of the compressed TCP/IP datagram header;

for **UNCOMPRESSED_TCP**, the rest of the modified IP header, and for COMPRESSED_TCP, the compressed TCP/IP datagram header;

for a total of 16 bytes; the uncompressed IP datagram follows the header.

DLT_PPP

PPP; if the first 2 bytes are 0xff and 0x03, it's PPP in HDLC-like framing, with the PPP header following those two bytes, otherwise it's PPP without framing, and the packet begins with the PPP header.

DLT_FDDI

FDDI

DLT_ATM_RFC1483

RFC 1483 LLC/SNAP-encapsulated ATM; the packet begins with an IEEE 802.2 LLC header.

DLT_RAW

raw IP; the packet begins with an IP header.

DLT_PPP_SERIAL

PPP in HDLC-like framing, as per RFC 1662, or Cisco PPP with HDLC framing, as per section 4.3.1 of RFC 1547; the first byte will be 0xFF for PPP in HDLC-like framing, and will be 0x0F or 0x8F for Cisco PPP with HDLC framing.

DLT_PPP_ETHER

PPPoE; the packet begins with a PPPoE header, as per RFC 2516.

DLT_C_HDLC

Cisco PPP with HDLC framing, as per section 4.3.1 of RFC 1547.

DLT_IEEE802_11

IEEE 802.11 wireless LAN

DLT_FRELAY

Frame Relay

DLT_LOOP

OpenBSD loopback encapsulation; the link layer header is a 4-byte field, in *network* byte order, containing a PF_ value from OpenBSD's **socket.h** for the network-layer protocol of the packet.

Note that, if a ``savefile'' is being read, those PF_ values are not necessarily those of the machine reading the capture file.

DLT_LINUX_SLL

Linux "cooked" capture encapsulation; the link layer header contains, in order:

a 2-byte "packet type", in network byte order, which is one of:

0
: packet was sent to us by somebody else

1
: packet was broadcast by somebody else

2
: packet was multicast, but not broadcast, by somebody else

3
: packet was sent by somebody else to somebody else

4
: packet was sent by us

a 2-byte field, in network byte order, containing a Linux ARPHRD_ value for the link layer device type;

a 2-byte field, in network byte order, containing the length of the link layer address of the sender of the packet (which could be 0);

an 8-byte field containing that number of bytes of the link layer header (if there are more than 8 bytes, only the first 8 are present);

a 2-byte field containing an Ethernet protocol type, in network byte order, or containing 1 for Novell 802.3 frames without an 802.2 LLC header or 4 for frames beginning with an 802.2 LLC header.

DLT_LTALK

Apple LocalTalk; the packet begins with an AppleTalk LLAP header.

DLT_PFLOG

OpenBSD pflog; the link layer header contains, in order:

a 4-byte PF_ value, in network byte order;

a 16-character interface name;

a 2-byte rule number, in network byte order;

a 2-byte reason code, in network byte order, which is one of:

0
: match

1
: bad offset

2
: fragment

3
: short

4
: normalize

memory

a 2-byte action code, in network byte order, which is one of:

0
: passed

1
: dropped

2
: scrubbed

a 2-byte direction, in network byte order, which is one of:

0
: incoming or outgoing

1
: incoming

2
: outgoing

DLT_PRISM_HEADER

Prism monitor mode information followed by an 802.11 header.

DLT_IP_OVER_FC

RFC 2625 IP-over-Fibre Channel, with the link-layer header being the Network_Header as described in that RFC.

DLT_SUNATM

SunATM devices; the link layer header contains, in order:

a 1-byte flag field, containing a direction flag in the uppermost bit, which is set for packets transmitted by the machine and clear for packets received by the machine, and a 4-byte traffic type in the low-order 4 bits, which is one of:

0
: raw traffic

1
LANE traffic
2
LLC-encapsulated traffic
3
MARS traffic
4
IFMP traffic
5
ILMI traffic
6
Q.2931 traffic

a 1-byte VPI value;

a 2-byte VCI field, in network byte order.

DLT_IEEE802_11_RADIO

link-layer information followed by an 802.11 header - seehttp://www.shaftnet.org/~pizza/software/capturefrm.txt for a description of the link-layer information.

DLT_ARCNET_LINUX

ARCNET, with no exception frames, reassembled packets rather than raw frames, and an extra 16-bit offset field between the destination host and type bytes.

DLT_LINUX_IRDA

Linux-IrDA packets, with a **DLT_LINUX_SLL** header followed by the IrLAP header.

- **int pcap_list_datalinks(pcap_t *p, int **dlt_buf);** [INDEX]

pcap_list_datalinks() is used to get a list of the supported data link types of the interface associated with the pcap descriptor. **pcap_list_datalinks()** allocates an array to hold the list and sets **dlt_buf*. The caller is responsible for freeing the array. -1 is returned on failure; otherwise, the number of data link types in the array is returned.

- **int pcap_set_datalink(pcap_t *p, int dlt);** [INDEX]

pcap_set_datalink() is used to set the current data link type of the pcap descriptor to the type specified by *dlt*. -1 is returned on failure.

- **int pcap_datalink_name_to_val(const char *name);** [INDEX]

pcap_datalink_name_to_val() translates a data link type name, which is a DLT_ name with the DLT_ removed, to the corresponding data link type value. The translation is case-insensitive. -1 is returned on failure.

- **const char *pcap_datalink_val_to_name(int dlt);** [INDEX]

pcap_datalink_val_to_name() translates a data link type value to the corresponding data link type name. **NULL** is returned on failure.

- **const char *pcap_datalink_val_to_description(int dlt);** [INDEX]

pcap_datalink_val_to_description() translates a data link type value to a short description of that data link type. **NULL** is returned on failure.

- **int pcap_snapshot(pcap_t *p)** [INDEX]

pcap_snapshot() returns the snapshot length specified when **pcap_open_live()** was called.

- **int pcap_is_swapped(pcap_t *p)** [INDEX]

pcap_is_swapped() returns true if the current ``savefile'' uses a different byte order than the current system.

- **int pcap_major_version(pcap_t *p)** [INDEX]

- **int pcap_minor_version(pcap_t *p)** [INDEX]

pcap_major_version() returns the major number of the file format of the savefile; **pcap_minor_version()** returns the minor number of the file format of the savefile. The version number is stored in the header of the savefile.

- **int pcap_stats(pcap_t *p, struct pcap_stat *ps)** [INDEX]

pcap_stats() returns 0 and fills in a *pcap_stat* struct. The values represent packet statistics from the start of the run to the time of the call. If there is an error or the underlying packet capture doesn't support packet statistics, -1 is returned and the error text can be obtained with **pcap_perror()** or **pcap_geterr()**. **pcap_stats()** is supported only on live captures, not on ``savefiles''; no statistics are stored in ``savefiles'', so no statistics are available when reading from a ``savefile''.

- **FILE *pcap_file(pcap_t *p)** [INDEX]

pcap_file() returns the standard I/O stream of the ``savefile,'' if a ``savefile'' was opened with **pcap_open_offline()**, or **NULL**, if a network device was opened with **pcap_open_live()**.

- **int pcap_fileno(pcap_t *p)** [INDEX]

pcap_fileno() returns the file descriptor number from which captured packets are read, if a network device was opened with **pcap_open_live()**, or -1, if a ``savefile'' was opened with **pcap_open_offline()**.

- **void pcap_perror(pcap_t *p, char *prefix)** [INDEX]

pcap_perror() prints the text of the last pcap library error on *stderr*, prefixed by *prefix*.

- **char *pcap_geterr(pcap_t *p)** [INDEX]

pcap_geterr() returns the error text pertaining to the last pcap library error. **NOTE**: the pointer it returns will no longer point to a valid error message string after the *pcap_t* passed to it is closed; you must use or copy the string before closing the *pcap_t*.

- **char *pcap_strerror(int error)** [INDEX]

pcap_strerror() is provided in case **strerror(1)** isn't available.

- **const char *pcap_lib_version(void)** [INDEX]

pcap_lib_version() returns a pointer to a string giving information about the version of the libpcap library being used; note that it contains more information than just a version number.

- **void pcap_close(pcap_t *p)** [INDEX]

pcap_close() closes the files associated with *p* and deallocates resources.

- **int pcap_dump_flush(pcap_dumper_t *p)** [INDEX]

pcap_dump_flush() flushes the output buffer to the ``savefile,'' so that any packets written with **pcap_dump()** but not yet written to the ``savefile'' will be written. -1 is returned on error, 0 on success.

- **FILE *pcap_dump_file(pcap_dumper_t *p)** [INDEX]

pcap_dump_file() returns the standard I/O stream of the ``savefile'' opened by **pcap_dump_open()**.

- **void pcap_dump_close(pcap_dumper_t *p)** [INDEX]

pcap_dump_close() closes the ``savefile.''