Socket Quick Reference

Author: Jialong He
Email: Jialong_he@bigfoot.com
http://www.bigfoot.com/~jialong_he

Berkeley Socket Functions

accept	An incoming connection is acknowledged and associated with an immediately created socket. The original socket is returned to the listening state.
bind	Assign a local name to an unnamed socket.
closesocket	Remove a socket from the per-process object reference table. Only blocks if SO_LINGER is set with a non-zero timeout on a blocking socket.
connect	Initiate a connection on the specified socket.
gethostbyaddr	retrieves the host information corresponding to a network address
gethostbyname	retrieves host information corresponding to a host name from a host database
gethostname	returns the standard host name for the local machine
getprotobyname	retrieves the protocol information corresponding to a protocol name
getprotobynumber	retrieves protocol information corresponding to a protocol number
getservbyname	retrieves service information corresponding to a service name and protocol
getservbyport	retrieves service information corresponding to a port and protocol
getpeername	Retrieve the name of the peer connected to the specified socket.
getsockname	Retrieve the local address to which the specified socket is bound.
getsockopt	Retrieve options associated with the specified socket.
htonl	Convert a 32-bit quantity from host byte order to network byte order.
htons	Convert a 16-bit quantity from host byte order to network byte order.
inet_addr	Converts a character string representing a number in the Internet standard "." notation to an Internet address value.
inet_ntoa	Converts an Internet address value to an ASCII string in "." notation i.e. "a.b.c.d".
ioctlsocket	Provide control for sockets.
listen	Listen for incoming connections on a specified socket.
ntohl	Convert a 32-bit quantity from network byte order to host byte order.
ntohs	Convert a 16-bit quantity from network byte order to host byte order.
recv	Receive data from a connected or unconnected socket.
recvfrom	Receive data from either a connected or unconnected socket.
select	Perform synchronous I/O multiplexing.
send	Send data to a connected socket.
sendto	Send data to either a connected or unconnected socket.
setsockopt	Store options associated with the specified socket.
shutdown	Shut down part of a full-duplex connection.
socket	Create an endpoint for communication and return a socket descriptor.

Windows Extension Functions

WSAAccept	An extended version of accept which allows for conditional acceptance and socket grouping.
WSAAsyncGetHostByAddr WSAAsyncGetHostByName WSAAsyncGetProtoByName WSAAsyncGetProtoByNumber WSAAsyncGetServByName WSAAsyncGetServByPort	A set of functions which provide asynchronous versions of the standard Berkeley getXbyY functions. For example, the **WSAAsyncGetHostByName** function provides an asynchronous, message-based implementation of the standard Berkeley **gethostbyname** function.
WSAAsyncSelect	Perform asynchronous version of select
WSACancelAsyncRequest	Cancel an outstanding instance of a WSAAsyncGetXByY function.
WSACleanup	Sign off from the underlying Windows Sockets DLL.
WSACloseEvent	Destroys an event object.
WSAConnect1	An extended version of connect which allows for exchange of connect data and QOS specification.
WSACreateEvent	Creates an event object.
WSADuplicateSocket	Allow an underlying socket to be shared by creating a virtual socket.
WSAEnumNetworkEvents	Discover occurrences of network events.
WSAEnumProtocols	Retrieve information about each available protocol.
WSAEventSelect	Associate network events with an event object.
WSAGetLastError	Obtain details of last Windows Sockets error
WSAGetOverlappedResult	Get completion status of overlapped operation.
WSAGetQOSByName	Supply QOS parameters based on a well-known service name.
WSAHtonl	Extended version of htonl
WSAHtons	Extended version of htons
WSAIoctl	Overlapped-capable version of ioctl
WSAJoinLeaf	Add a multipoint leaf to a multipoint session
WSANtohl	Extended version of ntohl
WSANtohs	Extended version of ntohs
WSAProviderConfigChange	Receive notifications of service providers being installed/removed.
WSARecv	An extended version of recv which accommodates scatter/gather I/O, overlapped sockets and provides the *flags* parameter as IN OUT
WSARecvFrom	An extended version of recvfrom which accommodates scatter/gather I/O, overlapped sockets and provides the *flags* parameter as IN OUT
WSAResetEvent	Resets an event object.
WSASend	An extended version of send which accommodates scatter/gather I/O and overlapped sockets
WSASendTo 1	An extended version of sendto which accommodates scatter/gather I/O and overlapped sockets
WSASetEvent	Sets an event object.
WSASetLastError	Set the error to be returned by a subsequent WSAGetLastError
WSASocket	An extended version of socket which takes a WSAPROTOCOL_INFO struct as input and allows overlapped sockets to be created. Also allows socket groups to be formed.
WSAStartup	Initialize the underlying Windows Sockets DLL.
WSAWaitForMultipleEvents 1	Blocks on multiple event objects.

Marcos

HIBYTE	retrieves the high-order byte from the given 16-bit value
LOBYTE	retrieves the low-order byte from the given 16-bit value
MAKEWORD	creates an unsigned 16-bit integer by concatenating two given unsigned character values
HIWORD	retrieves the high-order word from the given 32-bit value
LOWORD	retrieves the low-order word from the given 32-bit value
MAKELONG	creates an unsigned 32-bit value by concatenating two given 16-bit values

Client Example

```
#include <stdio.h>
#include <winsock.h>

void main(int argc, char **argv)
{
        WORD wVersionRequested = MAKEWORD(1,1);
        WSADATA wsaData;
        SOCKET  theSocket;
        SOCKADDR_IN saServer;
        int nRet;
        short nPort=80;
        char szBuf[256] = "Hi, there!\0";
        LPHOSTENT lpHostEntry;

        lpHostEntry = gethostbyname("www.microsoft.com");
```

```
	//------------------------------------------
	// Initialize WinSock and check the version
	//------------------------------------------
	nRet = WSAStartup(wVersionRequested, &wsaData);

	//-------------------------------
	// Create a TCP/IP stream socket
	//-------------------------------
	theSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
	saServer.sin_family = AF_INET;
	saServer.sin_addr = *((LPIN_ADDR)*lpHostEntry->h_addr_list);
	saServer.sin_port = htons(nPort);

	//-------------------------------
	// connect, send, receive
	//-------------------------------
	nRet = connect(theSocket, (LPSOCKADDR)&saServer, sizeof(struct sockaddr));
	nRet = send(theSocket, szBuf, strlen(szBuf), 0);
	nRet = recv(theSocket, szBuf, sizeof(szBuf), 0);

	//---------------------
	// Release WinSock
	//---------------------
	shutdown(theSocket, SD_SEND);
	closesocket(theSocket);
	WSACleanup();
}
```

Server Example

```
#include <stdio.h>
#include <winsock.h>

void main(int argc, char **argv)
{
	WORD wVersionRequested = MAKEWORD(1,1);
	WSADATA wsaData;
	int nRet;
	short nPort = 80;
	SOCKET  listenSocketm, remoteSocket;
	SOCKADDR_IN saServer;
	char szBuf[256];


	nRet = WSAStartup(wVersionRequested, &wsaData);

	saServer.sin_family = AF_INET;
	saServer.sin_addr.s_addr = INADDR_ANY;
	saServer.sin_port = htons(nPort);


	listenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
	nRet = bind(listenSocket, (LPSOCKADDR)&saServer, sizeof(struct sockaddr));
	nRet = listen(listenSocket, SOMAXCONN);

	remoteSocket = accept(listenSocket, NULL, NULL);
	memset(szBuf, 0, sizeof(szBuf));
	nRet = recv(remoteSocket, szBuf, sizeof(szBuf), 0);

	strcpy(szBuf, "How are you doing?");
	nRet = send(remoteSocket, szBuf, strlen(szBuf), 0);
```

WinPcap – Packet Capture Library

Where to get it:
http://netgroup-serv.polito.it/wincap
Install developer's pack

How to compile an application
(1) include <pcap.h> in the source code and add include file search path for "pcap.h";
(2) add "wpcap.lib", "wsock32.lib" to the link library list and set library search path for "wpcap.lib";

pcap_t *pcap_open_live(char *device, int snaplen, int promisc, int to_ms, char *ebuf);
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. 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. ebuf is used to return error text and is only set when pcap_open_live() fails and returns NULL.

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

char *pcap_lookupdev(char *errbuf)
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(char *device, bpf_u_int32 *netp, bpf_u_int32 *maskp, char *errbuf);
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);
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 pointer to the pcap_pkthdr struct (which precede the actual network headers and data), and a u_char pointer to the packet data.

int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user);
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).

u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h);
pcap_next() returns a u_char pointer to the next packet.

int pcap_stats(pcap_t *p, struct pcap_stat *ps);
int 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 under lying packet capture doesn't support packet statistics, -1 is returned and the error text can be obtained with pcap_perror() or pcap_geterr().

int pcap_compile(pcap_t *p, struct bpf_program *fp, char *str, int optimize, bpf_u_int32 netmask);
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 netmask of the local net. A return of -1 indicates an error in which case pcap_geterr() may be used to display the error text.

int pcap_setfilter(pcap_t *p, struct bpf_program *fp);
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 *);
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().

pcap_t *pcap_open_dead(int linktype, int snaplen);
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(char *fname, char *ebuf);
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. ebuf 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, char *fname);
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.

void pcap_dump(u_char *user, struct pcap_pkthdr *h, u_char *sp);
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().

void pcap_dump_close(pcap_dumper_t *p);
closes the ``savefile.''

FILE *pcap_file(pcap_t *p);
int pcap_fileno(pcap_t *p);
pcap_file() returns the name of the ``savefile.''
pcap_fileno() returns the file descriptor number of the ``savefile.''

int pcap_datalink(pcap_t *p);
int pcap_snapshot(pcap_t *p);
int pcap_is_swapped(pcap_t *p);
int pcap_major_version(pcap_t *p);
int pcap_minor_version(pcap_t *p);

pcap_datalink() returns the link layer type, e.g. DLT_EN10MB.
pcap_snapshot() returns the snapshot length specified when pcap_open_live was called.
pcap_is_swapped() returns true if the current ``savefile'' uses a different byte order than the current system.
pcap_major_version() returns the major number of the version of the pcap used to write the savefile.
pcap_minor_version() returns the minor number of the version of the pcap used to write the savefile.

void pcap_perror(pcap_t *p, char *prefix);
char *pcap_geterr(pcap_t *p);
char *pcap_strerror(int error);
pcap_perror() prints the text of the last pcap library error on stderr, prefixed by prefix.

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.

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

int pcap_setbuff(pcap_t *p, int dim);
pcap_setbuff() sets the size of the kernel buffer associated with the adapter p to dim bytes. Return value is 0 when the call succeeds, -1 otherwise. If an old buffer was already created with a previous call to pcap_setbuff(), it is deleted and the packets contained are discarded. pcap_open_live() creates a 1MB buffer by default.

int pcap_setmode(pcap_t *p, int mode);
pcap_setmode() sets the working mode of the interface p to mode. Valid values for mode are MODE_CAPT (default capture mode) and MODE_STAT (statistical mode). If the interface is in statistical mode, the callback function set by pcap_dispatch() or pcap_loop() is invoked every to_ms milliseconds (where to_ms is the timeout passed as an input parameter to pcap_open_live()). The received data contains two 64 bit integers indicating respectively the number of packets and the amount of total bytes that satisfied the BPF filter set with pcap_setfilter().

int pcap_setmintocopy(pcap_t *p, int size);
pcap_setmintocopy() changes the minimum amount of data in the kernel buffer that causes a read from the packet driver to return (unless the timeout expires). If size is big, the kernel is forced to wait the arrival of several packets before copying the data to the user. This guarantees a low number of system calls, i.e. low processor usage, and is a good setting for applications like packet-sniffers and protocol analyzers. Vice versa, in presence of a small value for this variable, the kernel will copy the packets as soon as the application is ready to receive them. This is useful for real time applications that need the best responsiveness from the kernel.

HANDLE pcap_getevent(pcap_t *p)
pcap_getevent() returns the handle of the event associated with the interface p. This event can be passed to functions like **WaitForSingleObject** or **WaitForMultipleObjects** to wait until the driver's buffer contains some data without performing a read.

int pcap_sendpacket(pcap_t *p, u_char *buf, int size);

pcap_sendpacket() allows to send a raw packet to the network using libpcap instead of accessing directly the underlying APIs. p is the interface that will be used to send the packet, buf contains the data of the packet to send (including the various protocol headers), size is the dimension of the buffer pointed by buf. The MAC CRC doesn't need to be calculated and added to the packet, because it is transparently put after the end of the data portion by the network interface.

```
const struct sniff_ethernet *ethernet; /* The ethernet header */
const struct sniff_ip *ip; /* The IP header */
const struct sniff_tcp *tcp; /* The TCP header */
const char *payload; /* Packet payload */
/* For readability, we'll make variables for the sizes of each of the structures */
int size_ethernet = sizeof(struct sniff_ethernet);
int size_ip = sizeof(struct sniff_ip);
int size_tcp = sizeof(struct sniff_tcp);
And now we do our magical typecasting:
ethernet = (struct sniff_ethernet*)(packet);
ip = (struct sniff_ip*)(packet + size_ethernet);
tcp = (struct sniff_tcp*)(packet + size_ethernet + size_ip);
payload = (u_char *)(packet + size_ethernet + size_ip + size_tcp);
```

```
/* Ethernet header */
struct sniff_ethernet {
    u_char ether_dhost[ETHER_ADDR_LEN]; /* Destination host address */
    u_char ether_shost[ETHER_ADDR_LEN]; /* Source host address */
    u_short ether_type; /* IP? ARP? RARP? etc */
};
```

```
/* IP header */
struct sniff_ip {
    #if BYTE_ORDER == LITTLE_ENDIAN
    u_int ip_hl:4, /* header length */
    ip_v:4; /* version */
    #if BYTE_ORDER == BIG_ENDIAN
    u_int ip_v:4, /* version */
    ip_hl:4; /* header length */
    #endif
    #endif /* not _IP_VHL */
    u_char ip_tos; /* type of service */
    u_short ip_len; /* total length */
    u_short ip_id; /* identification */
    u_short ip_off; /* fragment offset field */
    #define IP_RF 0x8000 /* reserved fragment flag */
    #define IP_DF 0x4000 /* dont fragment flag */
    #define IP_MF 0x2000 /* more fragments flag */
    #define IP_OFFMASK 0x1fff /* mask for fragmenting bits */
    u_char ip_ttl; /* time to live */
    u_char ip_p; /* protocol */
    u_short ip_sum; /* checksum */
    struct in_addr ip_src,ip_dst; /* source and dest address */
};
```

```
/* TCP header */
struct sniff_tcp {
    u_short th_sport; /* source port */
    u_short th_dport; /* destination port */
    tcp_seq th_seq; /* sequence number */
    tcp_seq th_ack; /* acknowledgement number */
    #if BYTE_ORDER == LITTLE_ENDIAN
    u_int th_x2:4, /* (unused) */
    th_off:4; /* data offset */
    #endif
    #if BYTE_ORDER == BIG_ENDIAN
    u_int th_off:4, /* data offset */
    th_x2:4; /* (unused) */
    #endif
    u_char th_flags;
    #define TH_FIN 0x01
    #define TH_SYN 0x02
    #define TH_RST 0x04
    #define TH_PUSH 0x08
    #define TH_ACK 0x10
    #define TH_URG 0x20
    #define TH_ECE 0x40
    #define TH_CWR 0x80
    #define TH_FLAGS (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)
    u_short th_win; /* window */
    u_short th; /* checksum */
    u_short th_urp; /* urgent pointer */
};
```

```
#include <pcap.h>
#include <stdio.h>
/*====================================*/
/*    PrintDeviceList               */
/*====================================*/
        void PrintDeviceList(const char* device) {

        DWORD dwVersion;
        DWORD dwWindowsMajorVersion;
        const WCHAR* t;
        const char* t95;
        int i=0;
        int DescPos=0;
        char *Desc;
        int n=1;

        dwVersion=GetVersion();
        dwWindowsMajorVersion =  (DWORD)(LOBYTE(LOWORD(dwVersion)));
        if (dwVersion >= 0x80000000 && dwWindowsMajorVersion >= 4)
        // Windows '95
        {
                t95=(char*)device;

                while(*(t95+DescPos)!=0 || *(t95+DescPos-1)!=0){
                        DescPos++;
                }

                Desc=(char*)t95+DescPos+1;

                printf("%d.",n++);

                while (!(t95[i]==0 && t95[i-1]==0))
                {
                        if (t95[i]==0){
                                putchar(' ');
                                putchar('(');
                                while(*Desc!=0){
                                        putchar(*Desc);
                                        Desc++;
                                }
                                Desc++;
                                putchar(')');
                                putchar('\n');
                        }
                        else putchar(t95[i]);

                        if((t95[i]==0) && (t95[i+1]!=0)){
                                printf("%d.",n++);
                        }

                        i++;
                }
                putchar('\n');
        }

        else{                   //Windows NT

                t=(WCHAR*)device;
                while(*(t+DescPos)!=0 || *(t+DescPos-1)!=0){
                        DescPos++;
                }
```