Webserv

Resources:
RFC 7231 (HTTP/1.1 messages): https://datatracker.ietf.org/doc/html/rfc7231
*****Overall introduction:** https://medium.com/from-the-scratch/http-server-what-do-you-need-to-know-to-build-a-simple-http-server-from-scratch-d1ef8945e4fa
*****https://www.youtube.com/watch?v=YwHErWJIh6Y (Guy who implements the overall instruction in C++)
Web Server Concepts: https://www.youtube.com/watch?v=9J1nJOivdyw
What http is: https://www.youtube.com/watch?v=0OrmKCB0UrQ&t=111s
GET vs POST: https://www.youtube.com/watch?v=K8HJ6DN23zI
Usage of select: https://www.youtube.com/watch?v=Y6pFtgRdUts&list=PL9IEJIKnBJjH_zM5LnovnoaKIXML5qh17&index=8&ab_channel=JacobSorber
HTTP Made Really Easy - A Practical Guide to Writing Clients and Servers: https://www.jmarshall.com/easy/http/
CGI Made Really Easy or, Writing CGI scripts to process Web forms: https://www.jmarshall.com/easy/cgi/
I/O Multiplexing (select vs. poll vs. epoll/kqueue): https://nima101.github.io/io_multiplexing
Header fields (can also be found in RFC7231): https://en.wikipedia.org/wiki/List_of_HTTP_header_fields
Beej's Guide to Network Programming: https://beej.us/guide/bgnet/html/
*https://www.digitalocean.com/community/tutorials/understanding-nginx-server-and-location-block-selection-algorithms
client_max_body_size: https://linuxhint.com/what-is-client-max-body-size-nginx/ || https://stackoverflow.com/questions/5840148/how-can-i-get-a-files-size-in-c || https://stackoverflow.com/questions/10634629/what-are-the-usage-differences-between-size-t-and-off-t

https://adrienblanc.com/projects/42/webserv
http://dwise1.net/pgm/sockets/blocking.html
https://www.cs.toronto.edu/~penny/teaching/csc309-01f/lectures/40/CGI.pdf
https://www.mkssoftware.com/docs/man3/select.3.asp
https://www.linuxtoday.com/blog/blocking-and-non-blocking-i-0/
https://docs.nginx.com/nginx/admin-guide/basic-functionality/managing-configuration-files/
https://www.ibm.com/docs/en/was/9.0.5?topic=in-editing-web-server-configuration-files
https://nginx.org/en/docs/beginners_guide.html
https://www.digitalocean.com/community/tutorials/understanding-the-nginx-configuration-file-structure-and-configuration-contexts
https://www.guru99.com/cpp-file-read-write-open.html
https://githubhot.com/repo/cclaude42/webserv
https://www.jetbrains.com/help/ruby/exploring-http-syntax.html#using_request_vars

https://datatracker.ietf.org/doc/html/rfc7230
https://developer.mozilla.org/en-US/docs/Web/HTTP/Redirections
http://www.wijata.com/cgi/cgispec.html#4.0
http://www.mnuwer.dbasedeveloper.co.uk/dlearn/web/session01.htm
https://rostlab.org/owiki/images/3/39/Sebastian_Hollizeck-BiolabExpertTalk.pdf

https://youtu.be/Yt1nesKi5Ec
https://youtu.be/thJSev60yfg
https://youtu.be/JhpUch6lWMw
https://youtu.be/wB9tIg209-8
https://youtu.be/CfEShMmsUus
https://youtu.be/ThNTJFBYL0Q
https://youtu.be/rFaRFCyewpA

https://www.youtube.com/watch?v=esXw4bdaZkc
https://www.youtube.com/watch?v=NEf3CFjN0Dg
https://www.youtube.com/watch?v=UObINRj2EGY
https://www.youtube.com/watch?v=8aAr9uxv-gU

https://en.wikipedia.org/wiki/Web_server
https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
https://en.wikipedia.org/wiki/Common_Gateway_Interface
https://en.wikipedia.org/wiki/Nginx
https://en.wikipedia.org/wiki/Network_socket
https://en.wikipedia.org/wiki/Request_for_Comments

Webserv Test Plan

-Go through PDF
-Intra Tester
-Go through Eval Sheet —> https://github.com/mharriso/school21-checklists/blob/master/ng_5_webserv.pdf
-https://github.com/t0mm4rx/webserv/tree/main/tests
-https://github.com/fredrikalindh/webserv_tester
-https://github.com/hygoni/webserv_tester

Everything in C++ 98.

htons	**#include <arpa/inet.h> uint16_t htons(uint16_t hostshort);**
htonl	**#include <arpa/inet.h> uint32_t htonl(uint32_t hostlong);**
ntohs	**#include <arpa/inet.h> uint16_t ntohs(uint16_t netshort);**
ntohl	**#include <arpa/inet.h> uint32_t ntohl(uint32_t netlong);**

These routines convert 16 bit, 32 bit, and 64 bit quantities between network byte order and host byte order. (Network byte order is big endian, or most significant byte first.) On machines which have a byte order which is the same as the network order, routines are defined as null macros.
These routines are most often used in conjunction with Internet addresses and ports as returned by gethostbyname(3) and getservent(3).
The byteorder functions except are expected to conform with IEEE Std POSIX.1-200x ("POSIX.1")

select   #include <sys/select.h>   int select(int nfds, fd_set *restrict readfds, fd_set *restrict writefds, fd_set *restrict errorfds, struct timeval *restrict timeout);
DESCRIPTION
select() examines the I/O descriptor sets whose addresses are passed in readfds, writefds, and errorfds to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first nfds descriptors are checked in each set; i.e., the descriptors from 0 through nfds-1 in the descriptor sets are examined. (Example: If you have set two file descriptors "4" and "17", nfds should not be "2", but rather "17 + 1" or "18".) On return, select() replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. select() returns the total number of ready descriptors in all the sets.

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets: FD_ZERO(&fdset) initializes a descriptor set fdset to the null set. FD_SET(fd, &fdset) includes a particular descriptor fd in fdset. FD_CLR(fd, &fdset) removes fd from fdset. FD_ISSET(fd, &fdset) is non-zero if fd is a member of fdset, zero otherwise. FD_COPY(&fdset_orig, &fdset_copy) replaces an already allocated &fdset_copy file descriptor set with a copy of &fdset_orig. The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to FD_SETSIZE, which is normally at least equal to the maximum number of descriptors supported by the system.

If timeout is not a null pointer, it specifies a maximum interval to wait for the selection to complete.
If timeout is a null pointer, the select blocks indefinitely.

To effect a poll, the timeout argument should be not be a null pointer, but it should point to a zero-valued timeval structure.

timeout is not changed by select(), and may be reused on subsequent calls, however it is good style to re-initialize it before each invocation of select().
Any of readfds, writefds, and errorfds may be given as null pointers if no descriptors are of interest.

select() returns the number of ready descriptors that are contained in the descriptor sets, or -1 if an error occurred. If the time limit expires, select() returns 0. If select() returns with an error, including one due to an interrupted call, the descriptor sets will be unmodified and the global variable errno will be set to indicate the error.

poll   #include <poll.h> int poll(struct pollfd fds[], nfds_t nfds, int timeout);
poll() examines a set of file descriptors to see if some of them are ready for I/O or if certain events have occurred on them. The fds argument is a pointer to an array of pollfd structures, as defined in ⟨poll.h⟩ (shown below). The nfds argument specifies the size of the fds array.

epoll
The epoll API performs a similar task to poll(2): monitoring multiple file descriptors to see if I/O is possible on any of them. The epoll API can be used either as an edge-triggered or a level-triggered interface and scales well to large numbers of watched file descriptors.

The central concept of the epoll API is the epoll *instance*, an in-kernel data structure which, from a user-space perspective, can be considered as a container for two lists:

The *interest* list (sometimes also called the epoll set): the set of file descriptors that the process has registered an interest in monitoring.

The *ready* list: the set of file descriptors that are "ready" for I/O. The ready list is a subset of (or, more precisely, a set of references to) the file descriptors in the interest list. The ready list is dynamically populated by the kernel as a result of I/O activity on those file descriptors.

epoll_create   #include <sys/epoll.h> int epoll_create(int *size*)
epoll_create() creates a new epoll(7) instance. Since Linux 2.6.8, the *size* argument is ignored, but must be greater than zero; see NOTES.

epoll_create() returns a file descriptor referring to the new epoll instance. This file descriptor is used for all the subsequent calls to the epoll interface. When no longer required, the file descriptor returned by epoll_create() should be closed by using close(2). When all file descriptors referring to an epoll instance have been closed, the kernel destroys the instance and releases the associated resources for reuse.

If *flags* is 0, then, other than the fact that the obsolete *size* argument is dropped, epoll_create1() is the same as epoll_create(). The following value can be included in *flags* to obtain different behavior:

Set the close-on-exec (FD_CLOEXEC) flag on the new file descriptor. See the description of the O_CLOEXEC flag in open(2) for reasons why this may be useful.
On success, these system calls return a file descriptor (a nonnegative integer). On error, -1 is returned, and *errno* is set to indicate the error.

epoll_ctl **#include <sys/epoll.h> int epoll_ctl(int *epfd*, int *op*, int *fd*, struct epoll_event ****event*);**
This system call is used to add, modify, or remove entries in the interest list of the epoll(7) instance referred to by the file descriptor *epfd*. It requests that the operation *op* be performed for the target file descriptor, *fd*.

epoll_wait **#include <sys/epoll.h> int epoll_wait(int *epfd*, struct epoll_event ****events*, int *maxevents*, int *timeout*);**
The epoll_wait() system call waits for events on the epoll(7) instance referred to by the file descriptor *epfd*. The buffer pointed to by *events* is used to return information from the ready list about file descriptors in the interest list that have some events available. Up to *maxevents* are returned by epoll_wait(). The *maxevents* argument must be greater than zero.

The *timeout* argument specifies the number of milliseconds that epoll_wait() will block. Time is measured against the CLOCK_MONOTONIC clock.

A call to epoll_wait() will block until either:
• a file descriptor delivers an event;
• the call is interrupted by a signal handler; or
• the timeout expires.

Note that the *timeout* interval will be rounded up to the system clock granularity, and kernel scheduling delays mean that the blocking interval may overrun by a small amount. Specifying a *timeout* of -1 causes epoll_wait() to block indefinitely, while specifying a *timeout* equal to zero cause epoll_wait() to return immediately, even if no events are available.

kqueue **#include <sys/types.h> #include <sys/event.h> #include <sys/time.h> int kqueue(void);**
kevent **#include <sys/types.h> #include <sys/event.h> #include <sys/time.h> int kevent(int kq, const struct kevent *changelist, int nchanges, struct kevent *eventlist, int nevents, const struct timespec *timeout);**
The kqueue() system call allocates a kqueue file descriptor. This file descriptor

provides a generic method of notifying the user when a kernel event (kevent) happens or a condition holds, based on the results of small pieces of kernel code termed filters.

A kevent is identified by an (ident, filter, and optional udata value) tuple. It specifies the interesting conditions to be notified about for that tuple. An (ident, filter, and optional udata value) tuple can only appear once in a given kqueue. Subsequent attempts to register the same tuple for a given kqueue will result in the replacement of the conditions being watched, not an addition. Whether the udata value is considered as part of the tuple is controlled by the EV_UDATA_SPECIFIC flag on the kevent.

The filter identified in a kevent is executed upon the initial registration of that event in order to detect whether a preexisting condition is present, and is also executed whenever an event is passed to the filter for evaluation. If the filter determines that the condition should be reported, then the kevent is placed on the kqueue for the user to retrieve.

The filter is also run when the user attempts to retrieve the kevent from the kqueue. If the filter indicates that the condition that triggered the event no longer holds, the kevent is removed from the kqueue and is not returned.

Multiple events which trigger the filter do not result in multiple kevents being placed on the kqueue; instead, the filter will aggregate the events into a single struct kevent. Calling close() on a file descriptor will remove any kevents that reference the descriptor.

The kqueue() system call creates a new kernel event queue and returns a descriptor. The queue is not inherited by a child created with fork(2).

The kevent,() kevent64() and kevent_qos() system calls are used to register events with the queue, and return any pending events to the user. The changelist argument is a pointer to an array of kevent, kevent64_s or kevent_qos_s structures, as defined in ⟨sys/event.h⟩. All changes contained in the changelist are applied before any pending events are read from the queue. The nchanges argument gives the size of changelist.

socket #include <sys/socket.h> int socket(int domain, int type, int protocol);
socket() creates an endpoint for communication and returns a descriptor.

The domain parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. These families are defined in the include file ⟨sys/socket.h⟩. The currently understood formats are

PF_LOCAL	Host-internal protocols, formerly called PF_UNIX,
PF_UNIX	Host-internal protocols, deprecated, use PF_LOCAL,
PF_INET	Internet version 4 protocols,
PF_ROUTE	Internal Routing protocol,
PF_KEY	Internal key-management function,
PF_INET6	Internet version 6 protocols,
PF_SYSTEM	System domain,
PF_NDRV	Raw access to network device,
PF_VSOCK	VM Sockets protocols

The socket has the indicated type, which specifies the semantics of communication. Currently defined types are:

SOCK_STREAM
SOCK_DGRAM
SOCK_RAW

A SOCK_STREAM type provides sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism may be supported. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). SOCK_RAW sockets provide access to internal network protocols and interfaces. The type SOCK_RAW, which is available only to the super-user.

The protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the "communication domain" in which communication is to take place; see protocols(5).

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a connect(2) or connectx(2) call. Once connected, data may be transferred using read(2) and write(2) calls or some variant of the send(2) and recv(2) calls. When a session has been completed a close(2) may be performed. Out-of-band data may also be transmitted as described in send(2) and received as described in recv(2).

The communications protocols used to implement a SOCK_STREAM insure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with -1 returns and with ETIMEDOUT as the specific code in the global variable errno. The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (e.g. 5 minutes). A SIGPIPE signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.

SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in send(2) calls. Datagrams are generally received with recvfrom(2), which returns the next datagram with its return address.

An fcntl(2) call can be used to specify a process group to receive a SIGURG signal when the out-of-band data arrives. It may also enable non-blocking I/O and asynchronous notification of I/O events via SIGIO.

The operation of sockets is controlled by socket level options. These options are defined in the file ⟨sys/socket.h⟩. Setsockopt(2) and getsockopt(2) are used to set and get options, respectively.

A -1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

accept   #include <sys/socket.h>   int accept(int socket, struct sockaddr *restrict address, socklen_t *restrict address_len);
The argument socket is a socket that has been created with socket(2), bound to an address with bind(2), and is listening for connections after a listen(2). accept() extracts the first connection request on the queue of pending connections, creates a new socket with the same properties of socket, and allocates a new file descriptor for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, accept() blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, accept() returns an error as described below. The accepted socket may not be used to accept more connections. The original socket socket, remains open.

The argument address is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the address parameter is determined by the domain in which the communication is occurring. The address_len is a value-result parameter; it should initially contain the amount of space pointed to by address; on return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with SOCK_STREAM.

It is possible to select(2) a socket for the purposes of doing an accept() by selecting it for read.

For certain protocols which require an explicit confirmation, such as ISO or DATAKIT, accept() can be thought of as merely dequeuing the next connection request and not implying confirmation. Confirmation can be implied by a normal read or write on the new file descriptor, and rejection can be implied by closing the new socket.

One can obtain user connection request data without confirming the connection by issuing a recvmsg(2) call with an msg_iovlen of 0 and a non-zero msg_controllen, or by issuing a getsockopt(2) request. Similarly, one can provide user connection rejection information by issuing a sendmsg(2) call with providing only the control information, or by calling setsockopt(2).

The call returns -1 on error and the global variable errno is set to indicate the error. If it succeeds, it returns a non-negative integer that is a descriptor for the accepted socket.

listen   #include <sys/socket.h>   int listen(int socket, int backlog);
Creation of socket-based connections requires several operations. First, a socket is created with socket(2). Next, a willingness to accept incoming connections and a queue limit for incoming connections are specified with listen(). Finally, the connections are accepted with accept(2). The listen() call applies only to sockets of type SOCK_STREAM.

The backlog parameter defines the maximum length for the queue of pending connections. If a connection request arrives with the queue full, the client may receive an error with an indication of ECONNREFUSED. Alternatively, if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.

The listen() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable errno is set to indicate the error.

send   #include <sys/socket.h>   ssize_t send(int socket, const void *buffer, size_t length, int flags);
send(), sendto(), and sendmsg() are used to transmit a message to another socket. send() may be used only when the socket is in a connected state, while sendto() and sendmsg() may be used at any time.

The address of the target is given by dest_addr with dest_len specifying its size. The length of the message is given by length. If the message is too long to pass atomically through the underlying protocol, the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a send(). Locally detected errors are indicated by a return value of -1.

If no messages space is available at the socket to hold the message to be transmitted, then send() normally blocks, unless the socket has been placed in non-blocking I/O mode. The select(2) call may be used to determine when it is possible to send more data.

The flags parameter may include one or more of the following:

```
#define MSG_OOB        0x1  /* process out-of-band data */
#define MSG_DONTROUTE  0x4  /* bypass routing, use direct interface */
```

The flag MSG_OOB is used to send “out-of-band” data on sockets that support this notion (e.g. SOCK_STREAM); the underlying protocol must also support “out-of-band” data. MSG_DONTROUTE is usually used only by diagnostic or routing programs.

The sendmsg() system call uses a msghdr structure to minimize the number of directly supplied arguments. The msg_iov and msg_iovlen fields of message specify zero or more buffers containing the data to be sent. msg_iov points to an array of iovec structures; msg_iovlen shall be set to the dimension of this array. In each iovec structure, the iov_base field specifies a storage area and the iov_len field gives its size in bytes. Some of these sizes can be zero. The data from each storage area indicated by msg_iov is sent in turn. See recv(2) for a complete description of the msghdr structure.

Upon successful completion, the number of bytes which were sent is returned. Otherwise, -1 is returned and the global variable errno is set to indicate the error.

recv #include <sys/socket.h>   ssize_t recv(int socket, void *buffer, size_t length, int flags);
The **recvfrom**() and **recvmsg**() system calls are used to receive messages from a socket, and may be used to receive data on a socket whether or not it is connection-oriented.

If address is not a null pointer and the socket is not connection-oriented, the source address of the message is filled in. The address_len argument is a value- result argument, initialized to the size of the buffer associated with address, and modified on return to indicate the actual size of the address stored there.

The **recv**() function is normally used only on a connected socket (see connect(2) or connectx(2)) and is identical to **recvfrom**() with a null pointer passed as its address argument. As it is redundant, it may not be supported in future releases.

All three routines return the length of the message on successful completion. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see socket(2)).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking (see fcntl(2)) in which case the value -1 is returned and the external variable errno set to EAGAIN. The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested; this behavior is affected by the socket-level options SO_RCVLOWAT and SO_RCVTIMEO described in getsockopt(2).

The select(2) system call may be used to determine when more data arrive.

If no messages are available to be received and the peer has performed an orderly shutdown, the value 0 is returned.

bind #include <sys/socket.h>   int bind(int socket, const struct sockaddr *address, socklen_t address_len);
Shell builtin commands are commands that can be executed within the running shell's process. Note that, in the case of csh(1) builtin commands, the command is executed in a subshell if it occurs as any component of a pipeline except the last.

If a command specified to the shell contains a slash “/”, the shell will not execute a builtin command, even if the last component of the specified command matches the name of a builtin command. Thus, while specifying “**echo**” causes a builtin command to be executed under shells that support the **echo** builtin command, specifying “**/bin/echo**” or “**./echo**” does not.

While some builtin commands may exist in more than one shell, their operation may be different under each shell which supports them. Below is a table which lists shell builtin commands, the standard shells that support them and whether they exist as standalone utilities.

Only builtin commands for the csh(1) and sh(1) shells are listed here. Consult a shell's manual page for details on the operation of its builtin commands. Beware that the sh(1) manual page, at least, calls some of these commands “built-in commands” and some of them “reserved words”. Users of other shells may need to consult an info(1) page or other sources of documentation.

connect #include <sys/types.h> #include <sys/socket.h>   int connect(int socket, const struct sockaddr *address, socklen_t address_len);
The parameter socket is a socket. If it is of type SOCK_DGRAM, this call specifies the peer with which the socket is to be associated; this address is

that
to which datagrams are to be sent, and the only address from which datagrams are to be received. If the socket is of type SOCK_STREAM, this call attempts to make a connection to another socket. The other socket is specified by address, which is an address in the communications space of the socket.

Each communications space interprets the address parameter in its own way. Generally, stream sockets may successfully **connect**() only once; datagram sockets may use **connect**() multiple times to change their association. Datagram sockets may dissolve the association by calling disconnectx(2), or by connecting to an invalid address, such as a null address or an address with the address family set to AF_UNSPEC (the error EAFNOSUPPORT will be harmlessly returned).

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the global integer variable errno is set to indicate the error.

inet_addr   #include <arpa/inet.h> in_addr_t inet_addr(const char *cp);
The routines inet_aton(), inet_addr() and inet_network() interpret character strings representing numbers expressed in the Internet standard '.' notation.

The inet_pton() function converts a presentation format address (that is, printable form as held in a character string) to network format (usually a struct in_addr or some other internal binary representation, in network byte order). It returns 1 if the address was valid for the specified address family, or 0 if the address was not parseable in the specified address family, or -1 if some system error occurred (in which case errno will have been set). This function is presently valid for AF_INET and AF_INET6.

The inet_aton() routine interprets the specified character string as an Internet address, placing the address into the structure provided. It returns 1 if the string was successfully interpreted, or 0 if the string is invalid. The inet_addr() and inet_network() functions return numbers suitable for use as Internet addresses and Internet network numbers, respectively.

The function inet_ntop() converts an address *src from network format (usually a struct in_addr or some other binary form, in network byte order) to presentation format (suitable for external display purposes). The size argument specifies the size, in bytes, of the buffer *dst. INET_ADDRSTRLEN and INET6_ADDRSTRLEN define the maximum size required to convert an address of the respective type. It returns NULL if a system error occurs (in which case, errno will have been set), or it returns a pointer to the destination string. This function is presently valid for AF_INET and AF_INET6.

The routine inet_ntoa() takes an Internet address and returns an ASCII string representing the address in '.' notation. The routine inet_makeaddr() takes an

Internet network number and a local network address and constructs an Internet address from it. The routines inet_netof() and inet_lnaof() break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine byte order integer values.

setsockopt   #include <sys/socket.h>   int setsockopt(int socket, int level, int option_name, const void *option_value, socklen_t option_len);
getsockopt() and setsockopt() manipulate the options associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost "socket" level.

When manipulating socket options the level at which the option resides and the name of the option must be specified. To manipulate options at the socket level, level is specified as SOL_SOCKET. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, level should be set to the protocol number of TCP; see getprotoent(3).

The parameters option_value and option_len are used to access option values for setsockopt(). For getsockopt() they identify a buffer in which the value for the requested option(s) are to be returned. For getsockopt(), option_len is a value- result parameter, initially containing the size of the buffer pointed to by option_value, and modified on return to indicate the actual size of the value returned. If no option value is to be supplied or returned, option_value may be NULL.

option_name and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file ⟨sys/socket.h⟩ contains definitions for socket level options, described below. Options at other protocol levels vary in format and name; consult the appropriate entries in section 4 of the manual.

Most socket-level options utilize an int parameter for option_value. For setsockopt(), the parameter should be non-zero to enable a boolean option, or zero if the option is to be disabled. SO_LINGER uses a struct linger parameter, defined in ⟨sys/socket.h⟩, which specifies the desired state of the option and the linger interval (see below). SO_SNDTIMEO and SO_RCVTIMEO use a struct timeval parameter, defined in ⟨sys/time.h⟩.

getsockname #include <sys/socket.h>   int getsockname(int socket,

struct sockaddr *restrict address, socklen_t *restrict address_len);
The getsockname() function returns the current address for the specified socket.

The address_len parameter should be initialized to indicate the amount of space pointed to by address. On return it contains the actual size of the address returned (in bytes).

The address is truncated if the buffer provided is too small.

Note: For the UNIX domain, the address length returned is the address_len parameter passed to the previous bind(2) system call and not the sa_len field of the address parameter passed to bind(2).

The getsockname() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable errno is set to indicate the error.

fcntl#include <fcntl.h>int fcntl(int fildes, int cmd, ...);
—> ONLY USE AS —>   fcntl(fd, F_SETFL, O_NONBLOCK);
fcntl() provides for control over descriptors. The argument fildes is a descriptor to be operated on by cmd as follows: https://man7.org/linux/man-pages/man2/fcntl.2.html