Sega Dreamcast™

Planetweb Stack

Sega Dreamcast

Planetweb Stack Table of Contents

1. Introduction

This document describes the interface to the Planetweb network library for Sega systems. It is based upon the draft POSIX network specification (P1003.1g/D6.1 May 1995) with additions for the single-threaded nature of most applications, and for the functions usually carried out at boot time and not specified by POSIX.

It is highly recommended to obtain a copy of this draft specification—it answers many questions about the inter-relatedness of the sockets functions in much greater detail than this document describes.

Not all of the POSIX specification is implemented: only: only the “sockets” implementation is provided (not the “XTI” interface), and the only address family that is supported is the Internet family (i.e.: no UNIX family sockets).

Some of the functions specified by POSIX are actually additions to the existing file system calls (i.e.: `close()`, `fcntl()`, and so on). Since Planetweb does not have the ability to modify the file system routines, these are actually implemented as separate routines called `net_close()`, `net_fcntl()`, and so on. The details will be given below.

As of this writing, there are a few areas that aren’t well implemented. It is instructive to recognize that this network library was initially developed to support an Internet Web Browser, and as such is tailored to that task. More and more of the POSIX specification will be added as time goes on, but if you wonder why an item is missing, it’s probably because it was not important for web browsing.

The namespace that the network library uses is not very clean. This is an area Planetweb will work on, and we reserve the leading characters `PW_`. Nevertheless, there may be namespace clashes until the library is finally cleaned.

Note: The concept of TCP urgent data is not well implemented, and probably never will be. Urgent data is not a well understood concept, and you can get much better clarity of code and more reliable programs if you simply open two TCP connections: one for ordinary data and one for urgent data. (Urgent data is called out-of-band data in the socket vocabulary.)

A Note About Header Files

Many of the descriptions of functions that follow indicate that you should include the system header file `<sys/types.h>`. Occasionally, you will also need to include `<sys/types2.h>`, in order to actually successfully compile your programs. This is because the `<sys/types.h>` header file that comes with your compiler is not POSIX compliant, and Planetweb is not proposing to modify it, since the maintenance of that task is too onerous. `<sys/types2.h>` is a header file provided by Planetweb which contains the POSIX types needed by the network code that are not present in `<sys/types.h>`.

In the same manner, Planetweb provides a header file <errno2.h> that contains network error codes missing from the system header `<errno.h>`. If your code uses the error codes mentioned below, you may also need to include `<errno2.h>` in order to correctly compile.

2. Connecting and Disconnecting

Connecting to the network—dialing up and communicating with an ISP—happens the first time an application tries to do something that requires Internet connectivity. Until then, the network is basically idle. Information necessary to establish a connection is specified in `InitializeNetwork()` and so must be available before the network is initialized.

In addition the routines `ConnectToNetwork()` and `DisconnectFromNetwork()` can be used to explicitly connect and disconnect the network. Neither of these functions blocks.

3. Kernel Level and Interface Routines

InitializeNetwork

Get Memory and Initialize the Network.

Description

`InitializeNetwork` obtains heap memory and sets up initial network data structures for subsequent network activity. It does not perform an actual connection to the Internet.

It takes as it single argument a pointer to a `struct netInfo` structure (defined in `<pweb_net.h>`), containing values used in initialization. `maxTCP` is the maximum number of simultaneously open TCP network connections, `maxUDP` is the maximum number of simultaneously open UDP ports, `maxSocket` is the maximum number of simultaneously open sockets, `netMalloc` is a pointer to a heap allocation function that the network should use for its memory functions, `netFree` is a pointer to the corresponding deallocation function, and `netRealloc` is a pointer to a reallocation function.

The network uses the heap extensively, and you may wish to customize either the memory region or the style of memory allocation to help the network fit in with your own application. In nearly all cases, out of memory errors are treated as dropped packets, so transmission protocols like TCP can handle them transparently, provided that the low memory conditions are transient.

`progressCallback` is an optional pointer to a function that is called every time the modem's state changes (typically, this is used to update the on-screen U/I to let the user know that things are actually happening). See `<pweb_net.h>` for the different modem states. If no action is desired for modem state changes, set `progressCallback` to `NULL`.

`modemStringsStart`, `modemStringsFinish`, and `modemString` are all callback functions used to implement something like a terminal window for strings that the modem emits. `modemStringsStart` is called when the low level state machine thinks that printable strings are likely to come from the modem (typically at the initial dialup stage). `modemStringsFinish` is called when the data connection switches to PPP packet mode and printable strings become unlikely. `modemString` is called when there's a line of text from the modem. Any of these functions can be set to `NULL`, in which case the corresponding function is not called.

`dlgHandler` is a callback function used to inform the user of exceptional conditions and to request guidance on how to proceed. `dlgHandler` is called for the following conditions:

ME_BROKEN	This indicates that the modem has stopped responding at the register level. On those platforms where this is possible, it usually means the user has removed the modem device from the system, or somehow knocked it loose. Valid responses for a broken modem error are RESP_RETRY or RESP_CANCEL.
ME_BUSY	This indicates that the modem is reporting that the telephone is busy. Valid responses are RESP_CANCEL, RESP_RETRY, or RESP_RETRY_TIL_CONNECTED.
ME_NODIALTONE	The modem was unable to detect a dialtone. Usually, this means the user hasn't plugged in the telephone. Valid responses are RESP_CANCEL or RESP_RETRY.
ME_CARRIERLOSS	The connection was dropped. Many ISPs have inactivity timeouts on their lines, and a ME_CARRIERLOSS error is the result of that. Valid responses are RESP_CANCEL or RESP_RETRY.
ME_NOANSWER	There was no answer at the other end of the phone line. Valid responses are RESP_CANCEL, RESP_RETRY, or RESP_RETRY_TIL_CONNECTED.
ME_BADPASSWORD	The network was informed that the username/password pair was incorrect. (Note that not all login methods report this information.) The only valid response is RESP_CANCEL.

nl_pClockTicks is a pointer to a function returning the number of clock ticks since some reasonable time (since power-on is acceptable). This function will be used to generate timeout values for the network, and to measure round trip times. It probably shouldn't wrap more than once per day, meaning the clock should run more slowly than 49,000 ticks per second. It also should not run too slowly: faster than once per second is desirable. We have found clocks ticking at 10 ticks per second to be acceptable. nl_ticksPerSecond contains the value of the clock frequency. This clock should be running at the time InitializeNetwork is called.

nl_flags is a logical OR of zero or more of the following flags: NETL_USETONE, which instructs the modem to use tone (as opposed to pulse) dialing; NETL_BLINDDIAL, which instructs the modem to not try to detect a dialtone before dialing (useful in foreign countries); and NETL_DIALAREA, which instructs the modem to always use the area code of the phone number, even if it matches the area code for the dialing location.

nl_outline is a string containing the digits to be dialed to get an outside line. (If none is required, use the empty string.)

nl_callwait is a string containing the digits to be dialed to turn off call waiting. These are usually *70 or 1170. If call waiting is not on the phone line, then use the empty string.

nl_areacode is a string containing the area code that the call is originating from. It is used to distinguish between long distance and local calls.

nl_longdist is a string containing the digits to be dialed for a long distance connection. It is usually 1, but can be a telephone company access code or a credit card dialing sequence.

nl_telephone is a string containing the telephone number of the ISP, in the form (408) 490-0610. (This form allows checking against the local area code to figure out if the call is long distance or not.)

nl_alttelephone is an alternate telephone number that is used if the first fails.

nl_loginID is a string used to identify the user to the ISP. It may have a strange syntax; refer to the technical documentation of the ISP for any details.

`nl_password` is a string transmitted to the ISP when a password is requested.

`nl_modem` is an `AT` command string that should be sent to the modem in addition to any setup already performed. If nothing extra is desired, this should be the empty string.

`nl_primaryDNS` and `nl_secondaryDNS` are IP addresses of the DNS servers associated with a given ISP. These are only used if the network configuration was unable to obtain DNS addresses at dialup.

If `InitializeNetwork` is successful, it returns `0`. Otherwise it returns a negative value.

Note: `InitializeNetwork` has no POSIX or BSD equivalent, since this function is carried out at boot time and in the kernel for UNIX-style systems.

Format

```
#include <pweb_net.h>
int InitializeNetwork(const struct netInfo *pNI)

struct netInfo {
int       maxTCP;
int       maxUDP;
int       maxSocket;
void *    (*netMalloc)(size_t size);
void      (*netFree)(void *pMem);
void *    (*netRealloc)(void *pOld, size_t newSize);
void
(*progressCallback)(modemProgressState);
void      (*modemStringsStart)(void);
void      (*modemStringsFinish)(void);
void      (*modemString)(const char *);
errorResponse(*dlgHandler)(modemErrorType);
unsigned long(*nl_pClockTicks)(void);
unsigned longnl_ticksPerSecond;
int       nl_flags;
const char *nl_outline;
const char *nl_callwait;
const char *nl_areacode;
const char *nl_longdist;
const char *nl_telephone;
const char *nl_alttelephone;
const char *nl_loginID;
const char *nl_password;
const char *nl_modem;
struct in_addrnl_primaryDNS;
struct in_addrnl_secondaryDNS;
};
```

NetworkIdle

Make the Network Go.

Description

The `NetworkIdle` routine is where the bulk of the network's work actually takes place. Since this implementation is a single-threaded one, there are no separate `inet` processes running in the background, making the network go. Every place where application code delays, `NetworkIdle` should be called.

All API-level network functions call `NetworkIdle` as part of their process, however, so if the application is simply polling the network, this is happening by default.

There is no POSIX equivalent to `NetworkIdle`.

Format

```
#include <pweb_net.h>

void NetworkIdle(void);
```

ConnectToNetwork

Queue a request to connect to the Internet.

Description

`ConnectToNetwork()` sets the “administrative” state of the network interface to `UP`. This means that during subsequent calls to `NetworkIdle()`, the network will try to establish a good link to the Internet.

There is no POSIX equivalent to `ConnectToNetwork()`.

Format

```
#include <pweb_net.h>
void ConnectToNetwork(void);
```

DisconnectFromNetwork

Queue a request to disconnect from the Internet.

Format

```
#include <pweb_net.h>
void DisconnectFromNetwork(void);
```

Description

`DisconnectFromNetwork()` sets the “administrative” state of the network interface to `DOWN`. This means that during subsequent calls to `NetworkIdle()`, the network will drop the connection. It also winds up hanging up the phone.

There is no POSIX equivalent to `DisconnectFromNetwork()`.

NetworkAdmState

Get the current administrative state of the network.

Description

`NetworkAdmState()` returns the current "administrative" state of the network. (The "administrative" state is that state which the network is trying to achieve.) It will be one of `NIS_UP` or `NIS_DOWN`.

Format

```
#include <pweb_net.h>
networkState NetworkAdmState(void);
```

NetworkActualState

Get the current actual state of the network.

Format

```
#include <pweb_net.h>
networkState NetworkActualState(void);
```

Description

`NetworkActualState()` returns the actual state of the network. It will be one of the following:

`NIS_UP`	The network is up and able to send and receive Internet packets.
`NIS_DOWN`	The network is down and unable to send or receive Internet packets. In addition, the telephone connection has been terminated, and no further processing remains to be done in order to close the network.
`NIS_OPENING`	The network is performing tasks in order to enter the `NIS_UP` state. (Typically, this is dialing the connection and configuring the PPP connection.)
`NIS_CLOSING`	The network is performing tasks in order to enter the `NIS_DOWN` state. (The phone may still be off hook.)
`NIS_BROKEN`	The network detected something wrong with the network. Internet transmission and reception is not possible. When the network is in the `NIS_BROKEN` state, the network will not try to automatically start. Use `ConnectToNetwork()` to reconnect to the network once the user has rectified the problem.

4. "Socket" Interface and Routines

Asynchronous Errors

If any of the following conditions occur asynchronously for a socket, the corresponding value listed below will become the "pending" error for the socket:

`[ECONNABORTED]`	The connection was aborted locally.
`[ECONNREFUSED]`	For a connection mode socket attempting a non-blocking connection, the attempt to connect was forcefully rejected. For a connectionless mode socket, an attempt to deliver a datagram was forcefully rejected.
`[ECONNRESET]`	The peer has aborted the connection.
`[EHOSTDOWN]`	The destination host has been determined to be down or disconnected.
`[EHOSTUNREACH]`	The destination host is not reachable.
`[ENETDOWN]`	The local network connection is not operational.
`[ENETRESET]`	The connection was aborted by the network.
`[ENETUNREACH]`	The destination network is not reachable.
`[ETIMEDOUT]`	The connection timed out during or after connection establishment.

socket

Create an endpoint for communication.

Description

The `socket()` function creates an endpoint for communication, returning a descriptor.

The protofamily argument specifies a particular protocol family, defined in `<sys/socket.h>`. Currently, only `PF_INET` is supported.

type is one of `SOCK_STREAM`, `SOCK_DGRAM`, `SOCK_RAW`, or `SOCK_SEQPACKET`. (Currently, only `SOCK_STREAM` and `SOCK_DGRAM` are supported.) For the internet protocol family, `SOCK_STREAM` corresponds to TCP, and `SOCK_DGRAM` corresponds to UDP.

The protocol argument specifies a particular protocol to be used with the socket. If protocol is zero, then the default protocol shall be used for the given protofamily and type. Note that not all protocols can be supported by all values of protofamily and type. For the internet protocol family, the protocols are `IPPROTO_ICMP`, `IPPROTO_TCP`, and `IPPROTO_UDP` (defined in <`netinet/in.h`>).

After completing successfully, `socket()` returns `0`. Otherwise, it returns `-1` and sets `errno` to one of the following values:

`[ENFILE]`	No more socket structures are available. (File table full.)
`[ENOBUFS]`	No more network structures are available.
`[EPROTONOSUPPORT]`	The protocol family is unsupported, or the type or particular protocol is not supported within the protocol family.
`[ESOCKTNOSUPPORT]`	The socket type is not supported.

Format

```
#include <sys/types.h>
#include <sys/socket.h>

int socket (int protofamily, int type, int protocol)
```

bind

Bind a socket address to a socket.

Description

[POSIX 5.4.3.2]

The `bind()` function assigns a local socket-address to a socket identified by descriptor s that has no local socket-address assigned. When a socket is created with `socket()` it is associated with a specific protocol from the protocol family but has no local socket-address assigned. The `bind()` function requests that the local socket-address specified in the `sockaddr` structure `socketaddress` be assigned to the socket. The address family of the socket-address, specified by the `sa_family` field of the socket-address, must be an address family supported by the protocol (The only address family supported by the Planetweb network stack is `AF_INET`). The format of the socket-address depends on the address family specified in the `sockaddr` structure. For example, if the address family is `AF_INET`, the socket address has the format of an Internet socket-address which is an IP address and port number combination (This format is known as a `sockaddr_in`, which is defined in `<netinet/in.h>`.).

Since `bind()` must know the available IP addresses of the system, it will block waiting for a network connection (i.e.: it will wait for the dialer to finish its job). Otherwise, the function returns immediately.

If the function completes successfully, it returns `0`. Otherwise, it returns `–1` and sets `errno` to one of the following:

`[EADDRINUSE]`	The specified socket-address is already in use.
`[EADDRNOTAVAIL]`	The specified socket-address is not available from the local machine.
`[EAFNOSUPPORT]`	Address in the specified address family cannot be used with this socket.
`[EBADF]`	The parameter s is not a valid descriptor.
`[EINVAL]`	The socket is already associated with a local socket-address or the parameter `addresslen` is not the size of a valid socket-address for the specified address family.

Format

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int s, const struct sockaddr *socketaddress, size_t addresslen)
```

connect

Initiate a connection on a socket.

Description

The `connect()` function sets the peer address with which a socket is associated, and attempts to create a connection with that peer if the socket is a connection mode socket.

The parameter s shall be a descriptor referring to a socket. The peer is specified by `socketaddress`, which is a socket-address valid for the protocol of the socket.

If the socket has not yet been bound to a local socket-address, `connect()` will bind the socket to an unused local socket-address.

If the socket is a connectionless mode socket (of type `SOCK_DGRAM`), then this function specifies the peer with which the socket is to be associated; this socket-address is that to which datagrams are to be sent, and the only socket-address from which datagrams are to be received. Calling the `connect()` function on a connectionless mode socket in `OPEN` state shall change the peer with which the socket is associated. Applications using connectionless mode sockets shall be able to dissolve the association by connecting to an address of family `AF_UNSPEC`. This call is valid in all states for connectionless mode sockets. Upon successful completion, the socket state shall be `OPEN`. If the connectionless mode socket was in the `OPEN` state before the call, unsuccessful completion shall leave the socket in the `BOUND` state.

If the socket is a connection mode socket (of type `SOCK_STREAM`), then this function attempts to make a connection to the peer. For such sockets, this call is valid only in the `GROUND` and `BOUND` states.

If a socket of type `SOCK_STREAM` has not been marked as non-blocking, `connect()` shall block until either the connection has been established or an error is detected. The time period for which attempts to make the connection will continue is unspecified. If the socket has been marked as non-blocking, `connect()` will return immediately.

Upon successful completion, the function returns `0`. If a socket of type `SOCK_STREAM` has been marked as non-blocking and the connection does not complete immediately, the function returns `–1` and sets `errno` to `[EINPROGRESS]`, and the state shall change to `CONNECTING`. Otherwise, it shall return `–1` and set `errno` to indicate the error.

If the function fails for a connection mode socket, the state of the socket shall be `FAILED`. Applications are expected to use the `net_close()` function to deallocate the socket and descriptor, and to create a new socket if attempting to re-initiate the connection.

On error, `errno` is set to one of the following values:

`[EADDRINUSE]`	The socket-address is already in use.
`[EADDRNOTAVAIL]`	The specified socket-address has a valid address family, but is invalid or not available.
`[EAFNOSUPPORT]`	Addresses in the specified address family cannot be used with this socket.
`[EALREADY]`	A previous connection attempt on the socket has not yet completed.
`[EBADF]`	The parameter s is not a valid descriptor.
`[ECONNREFUSED]`	The attempt to connect was forcefully rejected.
`[EHOSTUNREACH]`	The host specified is not reachable.
`[EINPROGRESS]`	The socket is non-blocking and the connection cannot be completed immediately.
`[EINVAL]`	The parameter `addresslen` is not the size of a valid socket-address for the specified address family, or the value of `sa_family` in `socketaddress` is not one supported by the protocol.
`[EISCONN]`	The socket is a connection-mode socket and is already connected.

[ENETDOWN]	The local network connection is not operational.
[ENETUNREACH]	The network is not reachable from this host.
[ETIMEDOUT]	Connection establishment timed out without establishing a connection.

Format

```
#include <sys/types.h>
#include <sys/socket.h>

int connect (int s, struct sockaddr *socketaddress, size_t addresslen)
```

accept

Dequeue a connection indication on a socket.

Description

The `accept()` function extracts the first connection request on the queue of pending connections for a listening socket. The parameter `s` is a descriptor referring to a socket that has been created with `socket()`, bound to a local socket-address with `bind()`, and is listening for connections after a `listen()`. The `accept()` function creates a new socket with similar properties to s except that it is not listening for connection requests, but instead is associated with a specific connection or connection request. The function allocates a new file descriptor, `ns`, 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 descriptor for the accepted socket, `ns`, can not be used to accept more connections. The original socket descriptor s remains open.

If the parameter `socketaddress` is not `NULL`, it specifies a buffer in which to return the socket-address of the connecting entity, as known to the communications layer. The exact format of the socket-address is determined by the protocol family specified on the call used to create the socket. If the parameter `socketaddress` is not `NULL`, the `addresslen` parameter refers to a location that is value-result; it should initially contain the amount of space pointed to by `socketaddress`; on return it will contain the actual length (in bytes) of the socket-address returned. The socket-address is truncated if the buffer provided is too small, in which case the value to which `addressing` refers will indicate the length before truncation. If the value of `socketaddress` is `NULL`, `addresslen` is ignored.

Upon successful completion, the function returns a non-negative integer that is a descriptor for the accepted socket. Otherwise, it returns a value of `–1` and sets `errno` to one of the following:

`[EBADF]`	The parameter s is not a valid descriptor.
`[ECONNABORTED]`	The peer has closed the connection before the application has processed the request.
`[EINVAL]`	The parameter s refers to a socket that is not in the `LISTENING` state.
`[ENFILE]`	The socket table is full.
`[ENOBUFS]`	Insufficient resources were available in the system to perform the operation.
`[EWOULDBLOCK]`	The socket is marked non-blocking and no connections are present to be accepted.

Format

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int s, struct sockaddr *socketaddress, size_t *addresslen)
```

listen — Listen for connections on a socket.

Format

```
#include <sys/types.h>
#include <sys/socket.h>

int listen(int s, int backlog)
```

Description

The s parameter is an open socket descriptor. To accept connections, a socket is first created with `socket()`, bound to a local socket-address with `bind()`, a willingness to accept incoming connections and a queue limit for incoming connections are specified with `listen()`, and then the connection requests are dequeued with `accept()`. The `listen()` function applies only to sockets of type `SOCK_STREAM`.

The backlog parameter defines the maximum length the queue of pending connections may grow to. Applications shall not supply values of backlog that are less than `1`. If a connection request arrives with the queue full, the protocol engine may report an error causing the client to receive an error, or the protocol engine may discard the request so that retries may succeed. Values of backlog up to `SOMAXCONN`, defined in `<sys/socket.h>`, are supported. Greater values are silently reduced to `SOMAXCONN`.

Upon successful completion, the function returns `0`. Otherwise, it returns `–1` and sets `errno` to the following:

`[EBADF]`	The parameter s is not a valid descriptor.

recv, recvfrom

Receive a message from a socket.

Description

The function `recvfrom()` is used to receive messages from a socket, whether or not it is a connection mode socket. The `s` parameter is an open socket descriptor.

If from is non-`NULL`, it specifies a buffer in which to return the source socket-address of the message. If the from parameter is not `NULL`, the `fromlen` parameter refers to a location that is value-result, initialized to the size of the buffer associated with from, and modified on return to indicate the actual size of the socket-address stored there. The socket-address shall be truncated if the buffer provided is too small, in which case the value to which `fromlen` refers shall indicate the length before truncation. If the value of from is `NULL`, the value of `fromlen` is unused. The `from` and `fromlen` parameters are ignored if the socket is a connection mode socket.

The `recv()` function is normally used on a connected socket and is identical to `recvfrom()` with from and `fromlen` parameters set to `NULL`.

The flags parameter shall contain the value zero, or the bitwise `OR` of any combination of the values `MSG_PEEK`, `MSG_OOB`, and `MSG_WAITALL`.

Note: None of these currently function.

The amount of data requested by a `recv()` or `recvfrom()` call is indicated by the `len` parameter.

A call to `recv()` or `recvfrom()` shall return immediately when one of the following conditions is true:

- A data segment containing data is present at the beginning of the receive queue, and either the segment is terminated, or the amount of data in the segment is at least as much as the smaller of the amount of data requested and the low-water mark for the socket.
- The socket has a pending error.
- For a connection mode socket, and end-of-file indication has been received or the connection has been terminated.

If none of these conditions are true and the socket is marked as non-blocking, the function will return `–1` and set `errno` to `[EWOULDBLOCK]`.

If none of these conditions are true and the socket is not marked as non-blocking, the function will block until one of these conditions becomes true.

If any of the conditions are true and a data segment is present in the receive queue, the function shall return as much data as possible from the first segment in the queue. Otherwise, if more than one of the conditions is true, the condition listed first is processed.

If the function returns because data are available, the amount of data returned is the smaller of the data available or the amount requested. The data returned by the function is removed from the socket receive queue. If the socket is of type `SOCK_DGRAM`, and the amount of data requested by a call is smaller than the amount of data contained in the segment, the remainder of the data in the segment is discarded.

If the function returns because of a pending error for the socket, the function returns `–1` and sets `errno` to the appropriate value. The pending error value is then cleared.

If the function returns because of an end-of-file condition or other connection termination with no pending error, the function will return a value of zero.

Upon successful completion, the functions return the number of bytes received. A return value of zero indicates either an end-of-file indication (`SOCK_STREAM`) or a zero-length record (`SOCK_DGRAM`). Otherwise the functions return `–1` and set `errno` to one of the following:

`[EBADF]`	The parameter s is not a valid descriptor.
`[EWOULDBLOCK]`	The socket is marked non-blocking and the receive operation would block.9
`[ENOTCONN]`	The socket is of type `SOCK_STREAM` and has not been connected.
[Section 5.1]	Any one of the asynchronous errors.

Format

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recv (int s, void *buf, size_t len, int flags)

ssize_t recvfrom (int s, void *buf, size_t len, int flags, struct sockaddr *from, size_t
*fromlen)
```

send, sendto, sendmsg

Send a message from a socket.

Description

The `send()` and `sendto()` functions are used to transmit data to a peer via socket `s`.

The peer socket-address may have been specified in advance (by the use of the `connect()` function), in which case no destination address should be specified. If the peer socket-address has not been pre-specified, a socket-address must be provided via the to argument to the `sendto()` function, with `tolen` specifying its size. A `NULL` value for the to argument indicates that no socket-address is specified, in which case the corresponding length field shall be ignored.

The `send()` function is equivalent to the `sendto()` function call with a `NULL` to parameter.

The length of the data to be sent is given by the len argument to the functions, with the `buf` argument specifying the buffer containing data to be sent.

For connectionless mode sockets (type `SOCK_DGRAM`), the specified data is transmitted as a single datagram. If the message is too long to pass atomically through the underlying protocol, then the `errno` is set to `[EMSGSIZE]`, and the message is not transmitted. For connection mode sockets (type `SOCK_STREAM`), transmission is not necessarily atomic, and part or all of the data may be transmitted.

No indication of failure to deliver is implicit in a `send()`.

If the socket has a pending error, the function returns `-1` and sets `errno` to the value of the pending error. The pending error is then cleared.

If the socket is not marked as non-blocking and no space is available at the socket to hold the message to be transmitted due to flow control, then these functions block until all the data can be transmitted.

The flags field is currently ignored. (Set it to zero for future compatibility.)

Upon successful completion, the function returns the number of bytes accepted for transmission. Otherwise, it sends no data, returns `-1`, and sets `errno` to one of the following:

`[EAFNOSUPPORT]`	Addresses in the specified address family cannot be used with this socket.
`[EBADF]`	The parameter s is not a valid descriptor.
`[EHOSTUNREACH]`	The destination host is not reachable.
`[EISCONN]`	The socket is already connected, and a destination address was specified with the `sendto()` function.
`[EINVAL]`	The parameter `tolen` is not the size of a valid socket-address for the specified address family.
`[EMSGSIZE]`	The socket requires that the message be sent atomically, and the size of the message to be sent made this impossible.
`[ENOTCONN]`	The socket in the send() call is not connected, or otherwise has not had the peer pre-specified.
`[EWOULDBLOCK]`	The socket is marked non-blocking and the requested operation would block.

Format

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send (int s, const void *buf, size_t len, int flags)
ssize_t sendto (int s, const void *buf, size_t len, int flags, const struct sockaddr *to,
size_t tolen)
```

shutdown

Shut down part of a full-duplex connection.

Description

The `shutdown()` function causes all or part of a full-duplex connection on the socket associated with `s` to be shut down. If how is `SHUT_RD`, then further receives will be disallowed. If how is `SHUT_WR`, then further sends will be disallowed. If how is `SHUT_RDWR`, then further sends and receives will be disallowed.

Upon successful completion, the function returns a value of `0`. Otherwise, it returns `–1` and sets `errno` to indicate the error:

`[EINVAL]`	The value of how is invalid.
`[ENOTCONN]`	The referenced socket is not connected.

Format

```
#include <sys/socket.h>
int shutdown (int s, int how)
```

net_close

Release resources used for a network connection.

Description

The function `net_close()` is used to release resources associated with the specified socket.

If the socket is in the `LISTENING` state, all pending connections to the socket are aborted, whether or not the new connection was ready for return by the `accept()` function.

If the socket is connection mode, and is connected, this call begins the shutdown sequence.

Data that has been accepted by the network system but which may not yet have been accepted by the peer may or may not actually be delivered.

All data in the socket's receive queue is discarded.

Upon successful completion, the function returns `0`. Otherwise, it returns `–1` and sets `errno` appropriately.

Format

```
#include <pweb_net.h>
int net_close(int s)
```

net_fcntl

Set options for a socket.

Description

(This only currently sets `O_NONBLOCK`!)

Format

5. Internet Address Manipulation

htonl, htons, ntohl, ntohs	Convert Values Between Host and Network Byte Order.
net_addr, inet_network, inet_makeaddr, inet_lnaof, inet_netof	Internet Address Manipulation.

6. Network Database Routines

getaddrinfo, freeaddrinfo, AIsetup, AIidle, AIabort

Get Address Information.

Description

In the Planetweb implementation of the network, `getaddrinfo()` is used to translate a host name into the IP address portion of an internet socket descriptor. The name of the host is passed in the name argument.

For this implementation, the service argument must be `NULL`, since service databases are unavailable on systems without large permanent storage. In addition, since only one type of information will ever be returned, the values pointed to by the `req` argument are useless, and so `req` should also be `NULL`.

When the function returns successfully, the location to which `pai` points shall refer to a linked list of `addrinfo` structures, each of which refer to the given `hostname`. (Hostnames which correspond to multiple IP addresses have one list item per address.) The `ai_family`, `ai_socktype`, and `ai_protocol` fields are usable as arguments to the `socket()` function, and the `ai_addr` and `ai_addrlen` fields are usable as arguments to the `connect()` function.

`getaddrinfo()` can take considerable time to complete, as it often waits for DNS responses from the Internet. If non-blocking operation is required, use the `AIsetup()`, `AIidle()`, and `AIabort()` functions.

The `freeaddrinfo()` function frees the storage associated with the `ai` argument, as well as any supplemental storage to which it refers. If the `ai_next` field is not `NULL`, then the entire list is deallocated.

The `AIsetup()` function sets up a request for network information with exactly the same form as `getaddrinfo()`; however `AIsetup()` stores internal state information in the `pai` field instead of the `addrinfo` list. After `AIsetup()` is called, repeated calls to `AIidle()` with the same `pai` field will cause the query to eventually complete. `AIidle()` returns the value `EAI_COMPLETE` if the query has been completed (and the `addrinfo` list placed in the location pointed to by `pai`). While things are still waiting, `AIidle()` returns `0`. If an error happens, `AIidle()` returns a negative error code. If the application needs to abort the lookup before `AIidle()` returns `EAI_COMPLETE`, use `AIabort()`, which frees the state information, and terminates the lookup.

`AIsetup()`, `AIidle()`, and `AIabort()` are non-standard.

Format

```
#include <netdb.h>
#include <pweb_net.h> /* for the AI...() functions */

int getaddrinfo (const char *name, const char *service, const struct addrinfo *req, struct
addrinfo **pai)

void freeaddrinfo (struct addrinfo *ai)

int AIsetup (const char *name, const char *service, const struct addrinfo *req, struct
addrinfo **pai)

int AIidle (struct addrinfo **pai)

void AIabort (struct addrinfo **pai)
```