SDL_net

3 November 2009

Jonathan Atkins

The latest copy of this document can be found at http://www.jonatkins.org/SDL_net

Table of Contents

1 Overview

A Little Bit About Me

I am currently, as I write this document, a programmer for Raytheon. There I do all sorts of communications, network, GUI, and other general programming tasks in C/C++ on the Solaris and sometimes Linux Operating Systems. I have been programming network code at work and in my free time for about 8 years. I find there is always more to learn about network coding, but the rewards for making something comunicate over what can only be called the most chaotic channel of information transfer are sometimes to much fun to ignore. I have coded only a few things in SDL_net that I would call complete projects. However SDL_net does make network coding easier and more portable than anything I could code using just plain BSD sockets. I was happy that I could finally base my code on something that is portable and small enough to be unintrusive. SDL_net does sometimes seem to oversimplify some things, but in the end I found that if I couldn't do it in SDL_net then perhaps I'm not doing something that is worthwhile. Of course I have and will continue to do things that are not so worthwhile perhaps, using SDL_net. Like a Web Server library, and an IRC client library. But I still enjoy making them anyway! So if you are making a game, or doing a school project, or even making a no-fun application, SDL_net is a viable choice in my opinion for almost any of these tasks. If you are interested in multicast and non-TCP/IPv4 networking then SDL_net is not the right thing for you. Everyone else, please make more network games that are fun to play, and portable enough that more people are out there to play with. I, meanwhile, will continue writting documentation and applications and games that will likely become vapourware before they even see the light of day, but I'll be having fun the whole time.

Feel free to contact me: JonathanCAtkins@gmail.com

I am also usually on IRC at irc.freenode.net in the #SDL channel as LIM

This is the README in the SDL_net source archive.

> SDL_net 1.2
>
> The latest version of this library is available from:
> SDL_net Homepage
>
> This is an example portable network library for use with SDL. It is available under the GNU Library General Public License. The API can be found in the file SDL_net.h This library supports UNIX, Windows, and BeOS. MacOS support is being written.
>
> The demo program is a chat client and server. The chat client requires the sample GUI library available at:
> GUIlib Homepage
> The chat client connects to the server via TCP, registering itself. The server sends back a list of connected clients, and keeps the client updated with the status of other clients. Every line of text from a client is sent via UDP to every other client.
>
> Note that this isn't necessarily how you would want to write a chat program, but it demonstrates how to use the basic features of the network library.
>
> Enjoy!
>
> -Sam Lantinga and Roy Wood

You may want to look at
Beej's Guide to Network Programming, which explains network programming using BSD sockets. You can apply the knowledge from there while using SDL_net.

And here's a bit of humor for you to look at as you deal with networking terms.

Dr. Suess as a Network Wizard

Gene_Ziegler@Cornell.edu

Here's an easy game to play.
Here's an easy thing to say....

If a packet hits a pocket on a socket on a port,
and the bus is interupted as a very last resort,
and the address of the memory makes your floppy disk abort
then the socket packet pocket has an error to report!

If your cursor finds a menu item followed by a dash,
and the double-clicking icon puts your window in the trash,
and your data is corrupted cause the index doesn't hash,
then your situation's hopeless, and your system's gona crash.

You can't say this? What a shame, sir!
We'll find you another game, sir.

If the label on the cable on the table at your house
says the network is connected to the button on your mouse,
but your packets want to tunnel on another protocol,
that's repeatedly rejected by the printer down the hall,
and your screen is all distorted by the side-effects of gauss,
so your icons in the window are as wavy as a souse,
then you may as well reboot and go out with a bang,
cause as sure as I'm a poet, the sucker's gunna hang!

When the copy of your floppy's getting sloppy on the disk,
and the microcode instructions cause unecessary risc,
then you have to flash your memory and you'll want to RAM your ROM.
quickly turn of your computer and be sure to tell your mom!

2 Getting Started

This assumes you have gotten SDL_net and installed it on your system. SDL_net has an INSTALL document in the source distribution to help you get it compiled and installed.

Generally, installation consists of:

```
./configure
make
make install
```

You may also want to look at some demonstration code which may be downloaded from: http://www.jonatkins.org/SDL_net/

2.1 Includes

To use SDL_net functions in a C/C++ source code file, you must use the SDL_net.h include file:

```
#include "SDL_net.h"
```

2.2 Compiling

To link with SDL_net you should use sdl-config to get the required SDL compilation options. After that, compiling with SDL_net is quite easy.
Note: Some systems may not have the SDL_net library and include file in the same place as the SDL library and includes are located, in that case you will need to add more -I and -L paths to these command lines.

Simple Example for compiling an object file:

```
cc -c `sdl-config --cflags` mysource.c
```

Simple Example for compiling an object file:

```
cc -o myprogram mysource.o `sdl-config --libs` -lSDL_net
```

Now `myprogram` is ready to run.

3 Functions

These are the functions in the SDL_net API.

3.1 General

The basic API startup, cleanup and versioning methods, along with a few network byte order data helpers.

3.1.1 SDLNet_Linked_Version

`const SDL_version *`**SDLNet_Linked_Version**`()`
`void` **SDL_NET_VERSION**`(SDL_version *compile_version)`

This works similar to `SDL_Linked_Version` and **SDL_VERSION**.
Using these you can compare the runtime version to the version that you compiled with.
No prior library initialization is needed to call these functions/macros.

```
SDL_version compile_version;
const SDL_version *link_version=SDLNet_Linked_Version();
SDL_NET_VERSION(&compile_version);
printf("compiled with SDL_net version: %d.%d.%d\n",
        compile_version.major,
        compile_version.minor,
        compile_version.patch);
printf("running with SDL_net version: %d.%d.%d\n",
        link_version->major,
        link_version->minor,
        link_version->patch);
```

See Also:
Section 3.1.2 [SDLNet_Init], page 10

3.1.2 SDLNet_Init

`int` **SDLNet_Init**`()`

Initialize the network API.
This must be called before using other functions in this library.
SDL must be initialized before this call.

Returns: 0 on success, -1 on errors

```
if(SDL_Init(0)==-1) {
    printf("SDL_Init: %s\n", SDL_GetError());
    exit(1);
}
if(SDLNet_Init()==-1) {
    printf("SDLNet_Init: %s\n", SDLNet_GetError());
    exit(2);
}
```

See Also:
Section 3.1.3 [SDLNet_Quit], page 11

3.1.3 SDLNet_Quit

`void` **SDLNet_Quit**`()`

Shutdown and cleanup the network API.
After calling this all sockets are closed, and the SDL_net functions should not be used. You may, of course, use SDLNet_Init to use the functionality again.

```
SDLNet_Quit();
// you could SDL_Quit(); here...or not.
```

See Also:
Section 3.1.2 [SDLNet_Init], page 10

3.1.4 SDLNet_GetError

`char` ***SDLNet_GetError**`()`

This is the same as SDL_GetError, which returns the last error set as a string which you may use to tell the user what happened when an error status has been returned from an SDLNet_function.

Returns: a char pointer (string) containing a humam readble version or the reason for the last error that occured.

```
printf("Oh My Goodness, an error : %s", SDLNet_GetError());
```

3.1.5 SDLNet_Write16

`void` **SDLNet_Write16**`(Uint16 value, void *area)`

value The 16bit number to put into the area buffer

area The pointer into a data buffer, at which to put the number

Put the 16bit (a short on 32bit systems) value into the data buffer area in network byte order. This helps avoid byte order differences between two systems that are talking over the network. The value can be a signed number, the unsigned parameter type doesn't affect the data. The *area* pointer need not be at the beginning of a buffer, but must have at least 2 bytes of space left, including the byte currently pointed at.

```
// put my number into a data buffer to prepare for sending to a remote host
char data[1024];
Sint16 number=12345;
SDLNet_Write16((Uint16)number,data);
```

See Also:
Section 3.1.7 [SDLNet_Read16], page 15, Section 3.1.6 [SDLNet_Write32], page 14

3.1.6 SDLNet_Write32

`void` **SDLNet_Write32**`(Uint32 value, void *area)`

value The 32bit number to put into the area buffer

area The pointer into a data buffer, at which to put the number

Put the 32bit (a long on 32bit systems) value into the data buffer area in network byte order. This helps avoid byte order differences between two systems that are talking over the network. The value can be a signed number, the unsigned parameter type doesn't affect the data. The *area* pointer need not be at the beginning of a buffer, but must have at least 4 bytes of space left, including the byte currently pointed at.

```
// put my number into a data buffer to prepare for sending to a remote host
char data[1024];
Uint32 number=0xDEADBEEF;
SDLNet_Write32(number,data);
```

See Also:
Section 3.1.8 [SDLNet_Read32], page 16, Section 3.1.5 [SDLNet_Write16], page 13

3.1.7 SDLNet_Read16

`Uint16` **SDLNet_Read16**`(void *area)`

area The pointer into a data buffer, at which to get the number from

Get a 16bit (a short on 32bit systems) value from the data buffer area which is in network byte order. This helps avoid byte order differences between two systems that are talking over the network. The returned value can be a signed number, the unsigned parameter type doesn't affect the data. The *area* pointer need not be at the beginning of a buffer, but must have at least 2 bytes of space left, including the byte currently pointed at.

```
// get a number from a data buffer to use on this host
//char *ptr; //this points into a previously received data buffer
Sint16 number;
number=(Sint16) SDLNet_Read16(ptr);
// number is now in your hosts byte order, ready to use.
```

See Also:
Section 3.1.5 [SDLNet_Write16], page 13, Section 3.1.8 [SDLNet_Read32], page 16

3.1.8 SDLNet_Read32

`Uint32` **SDLNet_Read32**`(void *area)`

area The pointer into a data buffer, at which to get the number from

Get a 32bit (a long on 32bit systems) value from the data buffer area which is in network byte order. This helps avoid byte order differences between two systems that are talking over the network. The returned value can be a signed number, the unsigned parameter type doesn't affect the data. The *area* pointer need not be at the beginning of a buffer, but must have at least 4 bytes of space left, including the byte currently pointed at.

```
// get a number from a data buffer to use on this host
//char *ptr; //this points into a previously received data buffer
Uint32 number;
number=SDLNet_Read32(ptr);
// number is now in your hosts byte order, ready to use.
```

See Also:
Section 3.1.6 [SDLNet_Write32], page 14, Section 3.1.7 [SDLNet_Read16], page 15

3.2 Name Resolution

These functions are used to resolve hostnames and numerical IPv4 Address to each other.

3.2.1 SDLNet_ResolveHost

`int` **SDLNet_ResolveHost**`(IPaddress *`*address*`, const char *`*host*`, Uint16` *port*`)`

address This points to the IPaddress that will be filled in. It doesn't need to be set before calling this, but it must be allocated in memory.

host For connecting to a server, this is the hostname or IP in a string.
For becoming a server, this is **NULL**. If you do use **NULL**, all network interfaces would be listened to for incoming connections, using the **INADDR_ANY** address.

port For connecting to a server, this is the the servers listening port number.
For becoming a server, this is the port to listen on.
If you are just doing Domain Name Resolution functions, this can be 0.

Resolve the string *host*, and fill in the IPaddress pointed to by *address* with the resolved IP and the port number passed in through *port*.
This is the best way to fill in the IPaddress struct for later use. This function does not actually open any sockets, it is used to prepare the arguments for the socket opening functions.
WARNING: this function will put the *host* and *port* into Network Byte Order into the *address* fields, so make sure you pass in the data in your hosts byte order. (normally not an issue)

Returns: 0 on success. -1 on errors, plus address.host will be INADDR_NONE. An error would likely be that the address could not be resolved.

For a server listening on all interfaces, on port 1234:

```
// create a server type IPaddress on port 1234
IPaddress ipaddress;
SDLNet_ResolveHost(&ipaddress, NULL, 1234);
```

For a client connecting to "host.domain.ext", at port 1234:

```
// create an IPaddress for host name "host.domain.ext" on port 1234
// this is used by a client
IPaddress ipaddress;
SDLNet_ResolveHost(&ipaddress, "host.domain.ext", 1234);
```

See Also:
Section 3.2.2 [SDLNet_ResolveIP], page 19, Section 4.1 [IPaddress], page 54

3.2.2 SDLNet_ResolveIP

`const char *`**SDLNet_ResolveIP**`(IPaddress *`*address*`)`

address This points to the IPaddress that will be resolved to a host name. The address->port is ignored.

Resolve the IPv4 numeric address in *address*->host, and return the hostname as a string.

Returns: a valid char pointer (string) on success. the returned hostname will have host and domain, as in "host.domain.ext". **NULL** is returned on errors, such as when it's not able to resolve the host name. The returned pointer is not to be freed. Each time you call this function the previous pointer's data will change to the new value, so you may have to copy it into a local buffer to keep it around longer.

```
// resolve the host name of the address in ipaddress
//IPaddress ipaddress;
char *host;
if(!(host=SDLNet_ResolveIP(&ipaddress))) {
    printf("SDLNet_ResolveIP: %s\n", SDLNet_GetError());
    exit(1);
}
```

See Also:
Section 3.2.1 [SDLNet_ResolveHost], page 18, Section 4.1 [IPaddress], page 54

3.3 TCP Sockets

These functions are used to work with TCP Sockets. TCP is used with a full connection, whereas UDP is connectionless. TCP also ensures that all packets reach the destionation (when possible). TCP also ensures that packets are received in the same order as sent.

3.3.1 SDLNet_TCP_Open

`TCPsocket` **SDLNet_TCP_Open**`(IPaddress *`*ip*`)`

ip This points to the IPaddress that contains the resolved IP address and port number to use.

Connect to the host and port contained in *ip* using a TCP connection.
If the host is **INADDR_ANY**, then only the port number is used, and a socket is created that can be used to later accept incoming TCP connections.

Returns: a valid TCPsocket on success, which indicates a successful connection has been established, or a socket has been created that is valid to accept incoming TCP connections. **NULL** is returned on errors, such as when it's not able to create a socket, or it cannot connect to host and/or port contained in *ip*.

```
// connect to localhost at port 9999 using TCP (client)
IPaddress ip;
TCPsocket tcpsock;

if(SDLNet_ResolveHost(&ip,"localhost",9999)==-1) {
    printf("SDLNet_ResolveHost: %s\n", SDLNet_GetError());
    exit(1);
}

tcpsock=SDLNet_TCP_Open(&ip);
if(!tcpsock) {
    printf("SDLNet_TCP_Open: %s\n", SDLNet_GetError());
    exit(2);
}
```

```
// create a listening TCP socket on port 9999 (server)
IPaddress ip;
TCPsocket tcpsock;

if(SDLNet_ResolveHost(&ip,NULL,9999)==-1) {
    printf("SDLNet_ResolveHost: %s\n", SDLNet_GetError());
    exit(1);
}

tcpsock=SDLNet_TCP_Open(&ip);
if(!tcpsock) {
    printf("SDLNet_TCP_Open: %s\n", SDLNet_GetError());
    exit(2);
}
```

See Also:
Section 3.3.3 [SDLNet_TCP_Accept], page 24, Section 3.3.2 [SDLNet_TCP_Close], page 23, Section 4.1 [IPaddress], page 54, Section 4.2 [TCPsocket], page 55

3.3.2 SDLNet_TCP_Close

`void` **SDLNet_TCP_Close**(`TCPsocket` *sock*)

sock A valid TCPsocket. This can be a server or client type socket.

This shutsdown, disconnects, and closes the TCPsocket *sock*.
After this, you can be assured that this socket is not in use anymore. You can reuse the sock variable after this to open a new connection with SDLNet_TCP_Open. Do not try to use any other functions on a closed socket, as it is now invalid.

Returns: nothing, this always succeeds for all we need to know.

```
// close the connection on sock
//TCPsocket sock;

SDLNet_TCP_Close(sock);
```

See Also:
Section 3.3.1 [SDLNet_TCP_Open], page 21, Section 4.2 [TCPsocket], page 55

3.3.3 SDLNet_TCP_Accept

`TCPsocket` **SDLNet_TCP_Accept**`(TCPsocket` *server*`)`

server This is the server TCPsocket which was previously created by **SDLNet_TCP_Open**.

Accept an incoming connection on the *server* TCPsocket.
Do not use this function on a connected socket. Server sockets are never connected to a remote host. What you get back is a new TCPsocket that is connected to the remote host. This is a non-blocking call, so if no connections are there to be accepted, you will get a **NULL** TCPsocket and the program will continue going.

Returns: a valid TCPsocket on success, which indicates a successful connection has been established. **NULL** is returned on errors, such as when it's not able to create a socket, or it cannot finish connecting to the originating host and port. There also may not be a connection attempt in progress, so of course you cannot accept nothing, and you get a **NULL** in this case as well.

```
// accept a connection coming in on server_tcpsock
TCPsocket new_tcpsock;

new_tcpsock=SDLNet_TCP_Accept(server_tcpsock);
if(!new_tcpsock) {
    printf("SDLNet_TCP_Accept: %s\n", SDLNet_GetError());
}
else {
    // communicate over new_tcpsock
}
```

See Also:
Section 3.3.1 [SDLNet_TCP_Open], page 21, Section 3.3.4 [SDLNet_TCP_GetPeerAddress], page 25, Section 3.3.2 [SDLNet_TCP_Close], page 23, Section 4.2 [TCPsocket], page 55

3.3.4 SDLNet_TCP_GetPeerAddress

`IPaddress *SDLNet_TCP_GetPeerAddress(TCPsocket sock)`

sock This is a valid TCPsocket.

Get the Peer's (the other side of the connection, the remote side, not the local side) IP address and port number.

Returns: an IPaddress. **NULL** is returned on errors, or when *sock* is a server socket.

```
// get the remote IP and port
//TCPsocket new_tcpsock;
IPaddress *remote_ip;

remote_ip=SDLNet_TCP_GetPeerAddress(new_tcpsock);
if(!remote_ip) {
    printf("SDLNet_TCP_GetPeerAddress: %s\n", SDLNet_GetError());
    printf("This may be a server socket.\n");
}
else {
    // print the info in IPaddress or something else...
}
```

See Also:
Section 3.3.1 [SDLNet_TCP_Open], page 21, Section 3.3.3 [SDLNet_TCP_Accept], page 24, Section 4.1 [IPaddress], page 54, Section 4.2 [TCPsocket], page 55

3.3.5 SDLNet_TCP_Send

`int` **SDLNet_TCP_Send**`(TCPsocket` *sock*`, const void *`*data*`, int` *len*`)`

sock This is a valid, connected, TCPsocket.

data This is a pointer to the data to send over *sock*.

len This is the length (in bytes) if the *data*.

Send *data* of length *len* over the socket *sock*.
This routine is not used for server sockets.

Returns: the number of bytes sent. If the number returned is less than *len*, then an error occured, such as the client disconnecting.

```
// send a hello over sock
//TCPsocket sock;
int len,result;
char *msg="Hello!";

len=strlen(msg)+1; // add one for the terminating NULL
result=SDLNet_TCP_Send(sock,msg,len);
if(result<len) {
    printf("SDLNet_TCP_Send: %s\n", SDLNet_GetError());
    // It may be good to disconnect sock because it is likely invalid now.
}
```

See Also:
Section 3.3.6 [SDLNet_TCP_Recv], page 27, Section 3.3.3 [SDLNet_TCP_Accept], page 24, Section 3.3.1 [SDLNet_TCP_Open], page 21, Section 3.3.4 [SDLNet_TCP_GetPeerAddress], page 25, Section 3.3.2 [SDLNet_TCP_Close], page 23, Section 4.2 [TCPsocket], page 55

3.3.6 SDLNet_TCP_Recv

`int` **SDLNet_TCP_Recv**`(TCPsocket` *sock*`, void *`*data*`, int` *maxlen*`)`

sock This is a valid, connected, TCPsocket.

data This is a pointer to the buffer that receives the data from *sock*.

maxlen This is the maximum length (in bytes) that will be read into *data*.

Receive data of **exactly** length *maxlen* bytes from the socket *sock*, into the memory pointed to by *data*.
This routine is not used for server sockets.
Unless there is an error, or the connection is closed, the buffer will read *maxlen* bytes. If you read more than is sent from the other end, then it will wait until the full requested length is sent, or until the connection is closed from the other end.
You may have to read 1 byte at a time for some applications, for instance, text applications where blocks of text are sent, but you want to read line by line. In that case you may want to find the newline characters yourself to break the lines up, instead of reading some inordinate amount of text which may contain many lines, or not even a full line of text.

Returns: the number of bytes received. If the number returned is less than or equal to zero, then an error occured, or the remote host has closed the connection.

```
// receive some text from sock
//TCPsocket sock;
#define MAXLEN 1024
int result;
char msg[MAXLEN];

result=SDLNet_TCP_Recv(sock,msg,MAXLEN);
if(result<=0) {
    // An error may have occured, but sometimes you can just ignore it
    // It may be good to disconnect sock because it is likely invalid now.
}
printf("Received: \"%s\"\n",msg);
```

See Also:
Section 3.3.5 [SDLNet_TCP_Send], page 26, Section 3.3.3 [SDLNet_TCP_Accept], page 24, Section 3.3.1 [SDLNet_TCP_Open], page 21, Section 3.3.4 [SDLNet_TCP_GetPeerAddress], page 25, Section 3.3.2 [SDLNet_TCP_Close], page 23, Section 4.2 [TCPsocket], page 55

3.4 UDP Sockets

These functions are used to work with UDP Sockets.
UDP is connectionless, but can be used as if it is connected, in the sense that you don't have to address every outgoing packet if you don't want to. This is done by binding a socket to remote IP address and port pairs.
UDP has no delivery guarantees, each packet has a chance of never getting to the destination. UDP packets may also be received completely out of order, as compared to the order of sending them.
All these seeming bad qualities are made up for in speed. UDP is faster than TCP, and so many games and speed sensitive applications, which may also be sending redundant data anyways, such as a game state, prefer to use UDP for the speed benefits.

SDL_net has a concept of channels, which help you to matchup packets to specific clients easier. These channel numbers are not transmitted in the UDP packet data, but rather when a UDP socket receives or sends packets, a channel number may be used instead of an IPaddress to refer to the source or destination. You might prefer not to use channels at all, which is fine. SDL_net provides them only as an optional convenience.

3.4.1 SDLNet_UDP_Open

`UDPsocket` **SDLNet_UDP_Open**`(Uint16` *port*`)`

port This is the port number (in native byte order) on which to receive UDP packets. Most servers will want to use a known port number here so that clients can easily communicate with the server. This can also be zero, which then opens an anonymous unused port number, to most likely be used to send UDP packets from.

Open a socket to be used for UDP packet sending and/or receiving.
If a non-zero *port* is given it will be used, otherwise any open port number will be used automatically.
Unlike TCP sockets, this socket does not require a remote host IP to connect to, this is because UDP ports are never actually connected like TCP ports are.
This socket is able to send and receive directly after this simple creation.

Returns: a valid UDPsocket on success. **NULL** is returned on errors, such as when it's not able to create a socket, or it cannot assign the non-zero *port* as requested.

Note that below I say server, but clients may also open a specific port, though it is prefered that a client be more flexible, given that the port may be already allocated by another process, such as a server. In such a case you will not be able to open the socket, and your program will be stuck, so it is better to just use whatever port you are given by using a specified port of zero. Then the client will always work. The client can inform the server what port to talk back to, or the server can just look at the source of the packets it is receiving to know where to respond to.

```
// create a UDPsocket on port 6666 (server)
UDPsocket udpsock;

udpsock=SDLNet_UDP_Open(6666);
if(!udpsock) {
    printf("SDLNet_UDP_Open: %s\n", SDLNet_GetError());
    exit(2);
}
```

```
// create a UDPsocket on any available port (client)
UDPsocket udpsock;

udpsock=SDLNet_UDP_Open(0);
if(!udpsock) {
    printf("SDLNet_UDP_Open: %s\n", SDLNet_GetError());
    exit(2);
}
```

See Also:
Section 3.4.2 [SDLNet_UDP_Close], page 30, Section 4.3 [UDPsocket], page 56

3.4.2 SDLNet_UDP_Close

`void` **SDLNet_UDP_Close**(`UDPsocket` *sock*)

sock A valid UDPsocket to shutdown, close, and free.

Shutdown, close, and free a UDPsocket.
Don't use the UDPsocket after calling this, except to open a new one.

Returns: nothing, this always succeeds.

```
// unbind all addresses on the UDPsocket channel 0
//UDPsocket udpsock;

SDLNet_UDP_Close(udpsock);
udpsock=NULL; //this helps us know that this UDPsocket is not valid anymore
```

See Also:
Section 3.4.1 [SDLNet_UDP_Open], page 29, Section 4.3 [UDPsocket], page 56

3.4.3 SDLNet_UDP_Bind

`int` **SDLNet_UDP_Bind**`(UDPsocket` *sock*`, int` *channel*`, IPaddress *`*address*`)`

sock the UDPsocket on which to assign the address.

channel The channel to assign address to. This should be less than **SDLNET_MAX_UDPCHANNELS**. If -1 is used, then the first unbound channel will be used, this should only be used for incomming packet filtering, as it will find the first channel with less than **SDLNET_MAX_UDPADDRESSES** assigned to it and use that one.

address The resolved IPaddress to assign to the socket's channel. The host and port are both used.
It is not helpful to bind 0.0.0.0 to a channel.

Bind an address to a channel on a socket.
Incoming packets are only allowed from bound addresses for the socket channel.
All outgoing packets on that channel, regardless of the packets internal address, will attempt to send once on each bound address on that channel.
You may assign up to **SDLNET_MAX_UDPADDRESSES** to each channel.

Returns: The channel number that was bound. -1 is returned on errors, such as no free channels, or this channel has **SDLNET_MAX_UDPADDRESSES** already assigned to it, or you have used a channel higher or equal to **SDLNET_MAX_UDPCHANNELS**, or lower than -1.

```
// Bind address to the first free channel
//UDPsocket udpsock;
//IPaddress *address;
int channel;

channel=SDLNet_UDP_Bind(udpsock, -1, address);
if(channel==-1) {
    printf("SDLNet_UDP_Bind: %s\n", SDLNet_GetError());
    // do something because we failed to bind
}
```

See Also:
Section 3.4.4 [SDLNet_UDP_Unbind], page 32, Section 3.4.5 [SDLNet_UDP_GetPeerAddress], page 33, Section 3.4.1 [SDLNet_UDP_Open], page 29, Section 4.1 [IPaddress], page 54, Section 4.3 [UDPsocket], page 56

3.4.4 SDLNet_UDP_Unbind

`void` **SDLNet_UDP_Unbind**`(UDPsocket` *sock*`, int` *channel*`)`

sock A valid UDPsocket to unbind addresses from.

channel The channel to unbind the addresses from in the UDPsocket.

This removes all previously assigned (bound) addresses from a socket channel.
After this you may bind new addresses to the socket channel.

Returns: nothing, this always succeeds.

```
// unbind all addresses on the UDPsocket channel 0
//UDPsocket udpsock;

SDLNet_UDP_Unbind(udpsock, 0);
```

See Also:
Section 3.4.3 [SDLNet_UDP_Bind], page 31, Section 3.4.2 [SDLNet_UDP_Close], page 30, Section 4.3 [UDPsocket], page 56

3.4.5 SDLNet_UDP_GetPeerAddress

`IPaddress *SDLNet_UDP_GetPeerAddress(UDPsocket sock, int channel)`

sock A valid UDPsocket that probably has an address assigned to the channel.

channel The channel to get the primary address from in the socket. This may also be -1 to get the port which this socket is bound to on the local computer.

Get the primary address assigned to this channel. Only the first bound address is returned. When channel is -1, get the port that this socket is bound to on the local computer, this only means something if you opened the socket with a specific port number.
Do not free the returned IPaddress pointer.

Returns: a pointer to an IPaddress. **NULL** is returned for unbound channels and on any errors.

```
// get the primary address bound to UDPsocket channel 0
//UDPsocket udpsock;
IPaddress *address;

address=SDLNet_UDP_GetPeerAddress(udpsock, 0);
if(!address) {
    printf("SDLNet_UDP_GetPeerAddress: %s\n", SDLNet_GetError());
    // do something because we failed to get the address
}
else {
    // perhaps print out address->host and address->port
}
```

See Also:
Section 3.4.3 [SDLNet_UDP_Bind], page 31, Section 3.4.4 [SDLNet_UDP_Unbind], page 32, Section 4.1 [IPaddress], page 54, Section 4.3 [UDPsocket], page 56

3.4.6 SDLNet_UDP_Send

`int` **SDLNet_UDP_Send**`(UDPsocket` *sock*`, int channel, UDPpacket *`*packet*`)`

sock A valid UDPsocket.

channel what channel to send packet on.

packet The packet to send.

Send *packet* using the specified socket *sock*, use ing the specified *channel* or else the *packet*'s address.
If *channel* is not -1 then the packet is sent to all the socket channels bound addresses. If socket *sock*'s channel is not bound to any destinations, then the packet is not sent at all!
If the channel is -1, then the packet's address is used as the destination.
Don't forget to set the length of the packet in the *len* **element of the** *packet* **you are sending! Note**: the *packet*->channel will be set to the channel passed in to this function.
Note: The maximum size of the packet is limited by the MTU (Maximum Transfer Unit) of the transport medium. It can be as low as 250 bytes for some PPP links, and as high as 1500 bytes for ethernet. Beyond that limit the packet will fragment, and make delivery more and more unreliable as lost fragments cause the whole packet to be discarded.

Returns: The number of destinations sent to that worked. 0 is returned on errors.
Note that since a channel can point to multiple destinations, there should be just as many packets sent, so dont assume it will always return 1 on success. Unfortunately there's no way to get the number of destinations bound to a channel, so either you have to remember the number bound, or just test for the zero return value indicating all channels failed.

```
// send a packet using a UDPsocket, using the packet's channel as the channel
//UDPsocket udpsock;
//UDPpacket *packet;
int numsent;

numsent=SDLNet_UDP_Send(udpsock, packet->channel, packet);
if(!numsent) {
    printf("SDLNet_UDP_Send: %s\n", SDLNet_GetError());
    // do something because we failed to send
    // this may just be because no addresses are bound to the channel...
}
```

Here's a way of sending one packet using it's internal channel setting.
This is actually what **SDLNet_UDP_Send** ends up calling for you.

```
// send a packet using a UDPsocket, using the packet's channel as the channel
//UDPsocket udpsock;
//UDPpacket *packet;
int numsent;

numsent=SDLNet_UDP_SendV(sock, &packet, 1);
if(!numsent) {
    printf("SDLNet_UDP_SendV: %s\n", SDLNet_GetError());
    // do something because we failed to send
    // this may just be because no addresses are bound to the channel...
}
```

See Also:
Section 3.4.3 [SDLNet_UDP_Bind], page 31, Section 3.4.8 [SDLNet_UDP_SendV], page 37, Section 3.4.7 [SDLNet_UDP_Recv], page 36, Section 3.4.9 [SDLNet_UDP_RecvV], page 38, Section 4.4 [UDPpacket], page 57, Section 4.3 [UDPsocket], page 56

3.4.7 SDLNet_UDP_Recv

`int` **SDLNet_UDP_Recv**`(UDPsocket` *sock*`, UDPpacket *`*packet*`)`

sock A valid UDPsocket.

packet The packet to receive into.

Receive a packet on the specified *sock* socket.
The *packet* you pass in must have enough of a data size allocated for the incoming packet data to fit into. This means you should have knowledge of your size needs before trying to receive UDP packets.
The packet will have it's address set to the remote sender's address.
The *socket*'s channels are checked in highest to lowest order, so if an address is bound to multiple channels, the highest channel with the source address bound will be retreived before the lower bound channels. So, the packets channel will also be set to the highest numbered channel that has the remote address and port assigned to it. Otherwise the channel will -1, which you can filter out easily if you want to ignore unbound source address.
Note that the local and remote channel numbers do not have to, and probably won't, match, as they are only local settings, they are not sent in the packet.
This is a non-blocking call, meaning if there's no data ready to be received the function will return.

Returns: 1 is returned when a packet is received. 0 is returned when no packets are received. -1 is returned on errors.

```
// try to receive a waiting udp packet
//UDPsocket udpsock;
UDPpacket packet;
int numrecv;

numrecv=SDLNet_UDP_Recv(udpsock, &packet);
if(numrecv) {
    // do something with packet
}
```

See Also:
Section 3.4.3 [SDLNet_UDP_Bind], page 31, Section 3.4.6 [SDLNet_UDP_Send], page 34, Section 3.4.8 [SDLNet_UDP_SendV], page 37, Section 3.4.9 [SDLNet_UDP_RecvV], page 38, Section 4.4 [UDPpacket], page 57, Section 4.3 [UDPsocket], page 56

3.4.8 SDLNet_UDP_SendV

`int` **SDLNet_UDP_SendV**`(UDPsocket` *sock*`, UDPpacket **`*packetV*`, int` *npackets*`)`

sock A valid UDPsocket.

packetV The vector of packets to send.

npackets number of packets in the packetV vector to send.

Send *npackets* of *packetV* using the specified *sock* socket.
Each packet is sent in the same way as in SDLNet_UDP_Send (see Section 3.4.6 [SDLNet_UDP_Send], page 34).
Don't forget to set the length of the packets in the *len* **element of the** *packets* **you are sending!**

Returns: The number of destinations sent to that worked, for each packet in the vector, all summed up. 0 is returned on errors.

```
// send a vector of 10 packets using UDPsocket
//UDPsocket udpsock;
//UDPpacket **packetV;
int numsent;

numsent=SDLNet_UDP_SendV(udpsock, packetV, 10);
if(!numsent) {
    printf("SDLNet_UDP_SendV: %s\n", SDLNet_GetError());
    // do something because we failed to send
    // this may just be because no addresses are bound to the channels...
}
```

See Also:
Section 3.4.3 [SDLNet_UDP_Bind], page 31, Section 3.4.6 [SDLNet_UDP_Send], page 34, Section 3.4.7 [SDLNet_UDP_Recv], page 36, Section 3.4.9 [SDLNet_UDP_RecvV], page 38, Section 4.4 [UDPpacket], page 57, Section 4.3 [UDPsocket], page 56

3.4.9 SDLNet_UDP_RecvV

`int` **SDLNet_UDP_RecvV**(`UDPsocket` *sock*, `UDPpacket **`*packetV*)

sock A valid UDPsocket.

packet The packet to receive into.

Receive into a packet vector on the specified socket *sock*.
packetV is a **NULL** terminated array. Packets will be received until the **NULL** is reached, or there are none ready to be received.
This call is otherwise the same as **SDLNet_UDP_Recv** (see Section 3.4.7 [SDLNet_UDP_Recv], page 36).

Returns: the number of packets received. 0 is returned when no packets are received. -1 is returned on errors.

```
// try to receive some waiting udp packets
//UDPsocket udpsock;
//UDPpacket **packetV;
int numrecv, i;

numrecv=SDLNet_UDP_RecvV(udpsock, &packetV);
if(numrecv==-1) {
    // handle error, perhaps just print out the SDL_GetError string.
}
for(i=0; i<numrecv; i++) {
    // do something with packetV[i]
}
```

See Also:
Section 3.4.3 [SDLNet_UDP_Bind], page 31, Section 3.4.6 [SDLNet_UDP_Send], page 34, Section 3.4.8 [SDLNet_UDP_SendV], page 37, Section 3.4.7 [SDLNet_UDP_Recv], page 36, Section 4.4 [UDPpacket], page 57, Section 4.3 [UDPsocket], page 56

3.5 UDP Packets

These functions are used to work with the UDPpacket type. This type is used with UDP sockets to transmit and receive data.

3.5.1 SDLNet_AllocPacket

`UDPpacket *`**SDLNet_AllocPacket**`(int` *size*`)`

size Size, in bytes, of the data buffer to be allocated in the new UDPpacket. Zero is invalid.

Create (via malloc) a new UDPpacket with a data buffer of *size* bytes.
The new packet should be freed using **SDLNet_FreePacket** when you are done using it.

Returns: a pointer to a new empty UDPpacket. **NULL** is returned on errors, such as out-of-memory.

```
// create a new UDPpacket to hold 1024 bytes of data
UDPpacket *packet;

packet=SDLNet_AllocPacket(1024);
if(!packet) {
    printf("SDLNet_AllocPacket: %s\n", SDLNet_GetError());
    // perhaps do something else since you can't make this packet
}
else {
    // do stuff with this new packet
    // SDLNet_FreePacket this packet when finished with it
}
```

See Also:
Section 3.5.4 [SDLNet_AllocPacketV], page 43, Section 3.5.2 [SDLNet_ResizePacket], page 41, Section 3.5.3 [SDLNet_FreePacket], page 42, Section 3.4.6 [SDLNet_UDP_Send], page 34, Section 3.4.8 [SDLNet_UDP_SendV], page 37, Section 4.4 [UDPpacket], page 57

3.5.2 SDLNet_ResizePacket

`int` **SDLNet_ResizePacket**`(UDPpacket *`*packet*`, int` *size*`)`

packet A pointer to the UDPpacket to be resized.

size The new desired size, in bytes, of the data buffer to be allocated in the UDPpacket.
Zero is invalid.

Resize a UDPpackets data buffer to *size* bytes. The old data buffer will not be retained, so the new buffer is invalid after this call.

Returns: the new size of the data in the packet. If the number returned is less than what you asked for, that's an error.

```
// Resize a UDPpacket to hold 2048 bytes of data
//UDPpacket *packet;
int newsize;

newsize=SDLNet_ResizePacket(packet, 2048);
if(newsize<2048) {
    printf("SDLNet_ResizePacket: %s\n", SDLNet_GetError());
    // perhaps do something else since you didn't get the buffer you wanted
}
else {
    // do stuff with the resized packet
}
```

See Also:
Section 3.5.1 [SDLNet_AllocPacket], page 40, Section 3.5.4 [SDLNet_AllocPacketV], page 43, Section 3.5.3 [SDLNet_FreePacket], page 42, Section 4.4 [UDPpacket], page 57

3.5.3 SDLNet_FreePacket

`void` **SDLNet_FreePacket**`(UDPpacket *`*packet*`)`

packet A pointer to the UDPpacket to be freed from memory.

Free a UDPpacket from memory. Do not use this UDPpacket after this function is called on it.

Returns: nothing, this always succeeds.

```
// Free a UDPpacket
//UDPpacket *packet;

SDLNet_FreePacket(packet);
packet=NULL; //just to help you know that it is freed
```

See Also:
Section 3.5.1 [SDLNet_AllocPacket], page 40, Section 3.5.4 [SDLNet_AllocPacketV], page 43, Section 3.5.5 [SDLNet_FreePacketV], page 44, Section 3.5.2 [SDLNet_ResizePacket], page 41, Section 4.4 [UDPpacket], page 57

3.5.4 SDLNet_AllocPacketV

`UDPpacket **SDLNet_AllocPacketV(int howmany, int size)`

howmany The number of UDPpackets to allocate.

size Size, in bytes, of the data buffers to be allocated in the new UDPpackets. Zero is invalid.

Create (via malloc) a vector of new UDPpackets, each with data buffers of *size* bytes. The new packet vector should be freed using **SDLNet_FreePacketV** when you are done using it. The returned vector is one entry longer than requested, for a terminating **NULL**.

Returns: a pointer to a new empty UDPpacket vector. **NULL** is returned on errors, such as out-of-memory.

```
// create a new UDPpacket vector to hold 1024 bytes of data in 10 packets
UDPpacket **packetV;

packetV=SDLNet_AllocPacketV(10, 1024);
if(!packetV) {
    printf("SDLNet_AllocPacketV: %s\n", SDLNet_GetError());
    // perhaps do something else since you can't make this packet
}
else {
    // do stuff with this new packet vector
    // SDLNet_FreePacketV this packet vector when finished with it
}
```

See Also:
Section 3.5.1 [SDLNet_AllocPacket], page 40, Section 3.5.3 [SDLNet_FreePacket], page 42, Section 3.5.5 [SDLNet_FreePacketV], page 44, Section 3.5.2 [SDLNet_ResizePacket], page 41, Section 4.4 [UDPpacket], page 57

3.5.5 SDLNet_FreePacketV

`void` **SDLNet_FreePacketV**`(UDPpacket **`*packetV*`)`

packetV A pointer to the UDPpacket vector to be freed from memory.

Free a UDPpacket vector from memory. Do not use this UDPpacket vector, or any UDPpacket in it, after this function is called on it.

Returns: nothing, this always succeeds.

```
// Free a UDPpacket vector
//UDPpacket **packetV;

SDLNet_FreePacketV(packetV);
packetV=NULL; //just to help you know that it is freed
```

See Also:
Section 3.5.4 [SDLNet_AllocPacketV], page 43, Section 3.5.1 [SDLNet_AllocPacket], page 40, Section 3.5.3 [SDLNet_FreePacket], page 42, Section 3.5.2 [SDLNet_ResizePacket], page 41, Section 4.4 [UDPpacket], page 57

3.6 Socket Sets

These functions are used to work with multiple sockets. They allow you to determine when a socket has data or a connection waiting to be processed. This is analogous to polling, or the select function.

3.6.1 SDLNet_AllocSocketSet

`SDLNet_SocketSet` **SDLNet_AllocSocketSet**(`int` *maxsockets*)

maxsockets
: The maximum number of sockets you will want to watch.

Create a socket set that will be able to watch up to *maxsockets* number of sockets. The same socket set can be used for both UDP and TCP sockets.

Returns: A new, empty, SDLNet_SocketSet. **NULL** is returned on errors, such as out-of-memory.

```
// Create a socket set to handle up to 16 sockets
SDLNet_SocketSet set;

set=SDLNet_AllocSocketSet(16);
if(!set) {
    printf("SDLNet_AllocSocketSet: %s\n", SDLNet_GetError());
    exit(1); //most of the time this is a major error, but do what you want.
}
```

See Also:
Section 3.6.2 [SDLNet_FreeSocketSet], page 47, Section 3.6.3 [SDLNet_AddSocket], page 48, Section 4.5 [SDLNet_SocketSet], page 58, Section 4.3 [UDPsocket], page 56, Section 4.2 [TCPsocket], page 55

3.6.2 SDLNet_FreeSocketSet

`void` **SDLNet_FreeSocketSet**`(SDLNet_SocketSet` *set*`)`

set The socket set to free from memory

Free the socket set from memory.
Do not reference the *set* after this call, except to allocate a new one.

Returns: nothing, this call always succeeds.

```
// free a socket set
//SDLNet_SocketSet set;

SDLNet_FreeSocketSet(set);
set=NULL; //this helps us remember that this set is not allocated
```

See Also:
Section 3.6.1 [SDLNet_AllocSocketSet], page 46, Section 3.6.3 [SDLNet_AddSocket], page 48, Section 4.5 [SDLNet_SocketSet], page 58, Section 4.3 [UDPsocket], page 56, Section 4.2 [TCPsocket], page 55

3.6.3 SDLNet_AddSocket

`int` **SDLNet_AddSocket**`(SDLNet_SocketSet` *set*`, SDLNet_GenericSocket` *sock*`)`
`int` **SDLNet_TCP_AddSocket**`(SDLNet_SocketSet` *set*`, TCPsocket` *sock*`)`
`int` **SDLNet_UDP_AddSocket**`(SDLNet_SocketSet` *set*`, UDPsocket` *sock*`)`

set The socket set to add this socket to

sock the socket to add to the socket set

Add a socket to a socket set that will be watched. TCP and UDP sockets should be added using the corrosponding macro (as in `SDLNet_TCP_AddSocket` for a TCP socket). The generic socket function will be called by the TCP and UDP macros. Both TCP and UDP sockets may be added to the same socket set. TCP clients and servers may all be in the same socket set. There is no limitation on the sockets in the socket set, other than they have been opened.

Returns: the number of sockets used in the set on success. -1 is returned on errors.

```
// add two sockets to a socket set
//SDLNet_SocketSet set;
//UDPsocket udpsock;
//TCPsocket tcpsock;
int numused;

numused=SDLNet_UDP_AddSocket(set,udpsock);
if(numused==-1) {
    printf("SDLNet_AddSocket: %s\n", SDLNet_GetError());
    // perhaps you need to restart the set and make it bigger...
}
numused=SDLNet_TCP_AddSocket(set,tcpsock);
if(numused==-1) {
    printf("SDLNet_AddSocket: %s\n", SDLNet_GetError());
    // perhaps you need to restart the set and make it bigger...
}
```

See Also:
Section 3.6.1 [SDLNet_AllocSocketSet], page 46, Section 3.6.4 [SDLNet_DelSocket], page 49, Section 3.6.5 [SDLNet_CheckSockets], page 50, Section 4.5 [SDLNet_SocketSet], page 58, Section 4.3 [UDPsocket], page 56, Section 4.2 [TCPsocket], page 55

3.6.4 SDLNet_DelSocket

`int` **SDLNet_DelSocket**`(SDLNet_SocketSet` *set*`, SDLNet_GenericSocket` *sock*`)`
`int` **SDLNet_TCP_DelSocket**`(SDLNet_SocketSet` *set*`, TCPsocket` *sock*`)`
`int` **SDLNet_UDP_DelSocket**`(SDLNet_SocketSet` *set*`, UDPsocket` *sock*`)`

set The socket set to remove this socket from

sock the socket to remove from the socket set

Remove a socket from a socket set. Use this before closing a socket that you are watching with a socket set. This doesn't close the socket. Call the appropriate macro for TCP or UDP sockets. The generic socket function will be called by the TCP and UDP macros.

Returns: the number of sockets used in the set on success. -1 is returned on errors.

```
// remove two sockets from a socket set
//SDLNet_SocketSet set;
//UDPsocket udpsock;
//TCPsocket tcpsock;
int numused;

numused=SDLNet_UDP_DelSocket(set,udpsock);
if(numused==-1) {
    printf("SDLNet_DelSocket: %s\n", SDLNet_GetError());
    // perhaps the socket is not in the set
}
numused=SDLNet_TCP_DelSocket(set,tcpsock);
if(numused==-1) {
    printf("SDLNet_DelSocket: %s\n", SDLNet_GetError());
    // perhaps the socket is not in the set
}
```

See Also:
Section 3.6.3 [SDLNet_AddSocket], page 48, Section 3.6.2 [SDLNet_FreeSocketSet], page 47, Section 4.5 [SDLNet_SocketSet], page 58, Section 4.3 [UDPsocket], page 56, Section 4.2 [TCPsocket], page 55

3.6.5 SDLNet_CheckSockets

`int` **SDLNet_CheckSockets**`(SDLNet_SocketSet` *set*`, Uint32` *timeout*`)`

set	The socket set to check
timeout	The amount of time (in milliseconds). 0 means no waiting. -1 means to wait over 49 days! (think about it)

Check all sockets in the socket set for activity. If a non-zero timeout is given then this function will wait for activity, or else it will wait for timeout milliseconds.

NOTE: "activity" also includes disconnections and other errors, which would be determined by a failed read/write attempt.

Returns: the number of sockets with activity. -1 is returned on errors, and you may not get a meaningful error message. -1 is also returned for an empty set (nothing to check).

```
// Wait for up to 1 second for network activity
//SDLNet_SocketSet set;
int numready;

numready=SDLNet_CheckSockets(set, 1000);
if(numready==-1) {
    printf("SDLNet_CheckSockets: %s\n", SDLNet_GetError());
    //most of the time this is a system error, where perror might help you.
    perror("SDLNet_CheckSockets");
}
else if(numready) {
    printf("There are %d sockets with activity!\n",numready);
    // check all sockets with SDLNet_SocketReady and handle the active ones.
}
```

See Also:
Section 3.6.6 [SDLNet_SocketReady], page 51, Section 3.6.3 [SDLNet_AddSocket], page 48, Section 3.6.4 [SDLNet_DelSocket], page 49, Section 3.6.1 [SDLNet_AllocSocketSet], page 46, Section 4.5 [SDLNet_SocketSet], page 58, Section 4.3 [UDPsocket], page 56, Section 4.2 [TCPsocket], page 55

3.6.6 SDLNet_SocketReady

`int` **SDLNet_SocketReady**(*sock*)

sock The socket to check for activity.
Both UDPsocket and TCPsocket can be used with this function.

Check whether a socket has been marked as active. This function should only be used on a socket in a socket set, and that set has to have had SDLNet_CheckSockets (see Section 3.6.5 [SDLNet_CheckSockets], page 50) called upon it.

Returns: non-zero for activity. zero is returned for no activity.

```
// Wait forever for a connection attempt
//SDLNet_SocketSet set;
//TCPsocket serversock, client;
int numready;

numready=SDLNet_CheckSockets(set, 1000);
if(numready==-1) {
    printf("SDLNet_CheckSockets: %s\n", SDLNet_GetError());
    //most of the time this is a system error, where perror might help you.
    perror("SDLNet_CheckSockets");
}
else if(numready) {
    printf("There are %d sockets with activity!\n",numready);
    // check all sockets with SDLNet_SocketReady and handle the active ones.
    if(SDLNet_SocketReady(serversock)) {
        client=SDLNet_TCP_Accept(serversock);
        if(client) {
            // play with the client.
        }
    }
}
```

To just quickly do network handling with no waiting, we do this.

```
// Check for, and handle UDP data
//SDLNet_SocketSet set;
//UDPsocket udpsock;
//UDPpacket *packet;
int numready, numpkts;

numready=SDLNet_CheckSockets(set, 0);
if(numready==-1) {
    printf("SDLNet_CheckSockets: %s\n", SDLNet_GetError());
    //most of the time this is a system error, where perror might help you.
    perror("SDLNet_CheckSockets");
}
else if(numready) {
    printf("There are %d sockets with activity!\n",numready);
    // check all sockets with SDLNet_SocketReady and handle the active ones.
    if(SDLNet_SocketReady(udpsock)) {
        numpkts=SDLNet_UDP_Recv(udpsock,&packet);
        if(numpkts) {
            // process the packet.
        }
    }
}
```

See Also:
Section 3.6.5 [SDLNet_CheckSockets], page 50, Section 3.6.3 [SDLNet_AddSocket], page 48, Section 3.6.4 [SDLNet_DelSocket], page 49, Section 3.6.1 [SDLNet_AllocSocketSet], page 46, Section 4.5 [SDLNet_SocketSet], page 58, Section 4.3 [UDPsocket], page 56, Section 4.2 [TCPsocket], page 55

4 Types

These types are defined and used by the SDL_net API.

4.1 IPaddress

```
typedef struct {
    Uint32 host;            /* 32-bit IPv4 host address */
    Uint16 port;            /* 16-bit protocol port */
} IPaddress;
```

host the IPv4 address of a host, encoded in Network Byte Order.

port the IPv4 port number of a socket, encoded in Network Byte Order.

This type contains the information used to form network connections and sockets.

See Also:
Section 3.2 [Name Resolution], page 17, Section 3.3.1 [SDLNet_TCP_Open], page 21, Section 4.4 [UDPpacket], page 57

4.2 TCPsocket

```
typedef struct _TCPsocket *TCPsocket;
```

This is an opaque data type used for TCP connections. This is a pointer, and so it could be **NULL** at times. **NULL** would indicate no socket has been established.

See Also:
Section 3.3 [TCP Sockets], page 20, Section 4.3 [UDPsocket], page 56, Section 4.6 [SDLNet_GenericSocket], page 59

4.3 UDPsocket

```
typedef struct _UDPsocket *UDPsocket;
```

This is an opaque data type used for UDP sockets. This is a pointer, and so it could be **NULL** at times. **NULL** would indicate no socket has been established.

See Also:
Section 3.4 [UDP Sockets], page 28, Section 4.4 [UDPpacket], page 57, Section 4.2 [TCPsocket], page 55, Section 4.6 [SDLNet_GenericSocket], page 59

4.4 UDPpacket

```
typedef struct {
    int channel;          /* The src/dst channel of the packet */
    Uint8 *data;          /* The packet data */
    int len;              /* The length of the packet data */
    int maxlen;           /* The size of the data buffer */
    int status;           /* packet status after sending */
    IPaddress address;    /* The source/dest address of an
                             incoming/outgoing packet */
} UDPpacket;
```

channel The (software) channel number for this packet. This can also be used as a priority value for the packet. If no channel is assigned, the value is -1.

data The data contained in this packet, this is the meat.

len This is the meaningful length of the data in bytes.

maxlen This is size of the data buffer, which may be larger than the meaningful length. This is only used for packet creation on the senders side.

status This contains the number of bytes sent, or a -1 on errors, after sending. This is useless for a received packet.

address This is the resolved IPaddress to be used when sending, or it is the remote source of a received packet.

This struct is used with UDPsockets to send and receive data. It also helps keep track of a packets sending/receiving settings and status. The channels concept helps prioritize, or segregate differring types of data packets.

See Also:
Section 3.5 [UDP Packets], page 39, Section 4.3 [UDPsocket], page 56, Section 4.1 [IPaddress], page 54

4.5 SDLNet_SocketSet

```
typedef struct _SDLNet_SocketSet *SDLNet_SocketSet;
```

This is an opaque data type used for socket sets. This is a pointer, and so it could be **NULL** at times. **NULL** would indicate no socket set has been created.

See Also:
Section 3.6 [Socket Sets], page 45, Section 4.2 [TCPsocket], page 55, Section 4.3 [UDPsocket], page 56

4.6 SDLNet_GenericSocket

```
typedef struct {
    int ready;
} *SDLNet_GenericSocket;
```

ready non-zero when data is ready to be read, or a server socket has a connection attempt ready to be accepted.

This data type is able to be used for both *UDPsocket* and *TCPsocket* types.
After calling **SDLNet_CheckSockets**, if this socket is in SDLNet_SocketSet used, the *ready* will be set according to activity on the socket. This is the only real use for this type, as it doesn't help you know what type of socket it is.

See Also:
Section 3.6 [Socket Sets], page 45, Section 4.2 [TCPsocket], page 55, Section 4.3 [UDPsocket], page 56

5 Defines

SDL_NET_MAJOR_VERSION
: `1`
SDL_net library major number at compilation time

SDL_NET_MINOR_VERSION
: `2`
SDL_net library minor number at compilation time

SDL_NET_PATCHLEVEL
: `7`
SDL_net library patch level at compilation time

INADDR_ANY
: `0x00000000 (0.0.0.0)`
used for listening on all network interfaces.

INADDR_NONE
: `0xFFFFFFFF (255.255.255.255)`
which has limited applications.

INADDR_BROADCAST
: `0xFFFFFFFF (255.255.255.255)`
used as destination when sending a message to all clients on a subnet that allows broadcasts.

SDLNET_MAX_UDPCHANNELS
: 32
The maximum number of channels on a UDP socket.

SDLNET_MAX_UDPADDRESSES
: 4
The maximum number of addresses bound to a single UDP socket channel

6 Glossary

Network Byte Order
Also known as *Big-Endian*. Which means the most significant byte comes first in storage. Sparc and Motorola 68k based chips are MSB ordered.
(SDL defines this as **SDL_BYTEORDER==SDL_BIG_ENDIAN**)

Little-Endian(LSB) is stored in the opposite order, with the least significant byte first in memory. Intel and AMD are two LSB machines.
(SDL defines this as **SDL_BYTEORDER==SDL_LIL_ENDIAN**)

For network addresses, `1.2.3.4` is always stored as `{0x01 0x02 0x03 0x04}`.

Index

(Index is nonexistent)