User's Guide to

SN TCP/IP Stack for PlayStation®2

SN Systems Ltd
Version 1.02
July 2000

UG-V1.02 / SN TCP/IP Stack for PlayStation 2 / July 2000

Copyright notice

The KwikNet TCP/IP Stack is derived from the University of California's Berkeley Software Distribution (BSD). Some components have been adapted from software made available by the Massachusetts Institute of Technology and Carnegie Mellon University. Use of this software requires the following copyright acknowledgements.

Contents

Preface

Overview of SN TCP/IP Stack for PlayStation 2

SN TCP/IP Stack for PlayStation 2 enables you to add networking capabilities to your PlayStation 2 application, at minimum cost and effort. Whether it's for online gaming, or any other internet application for the PlayStation 2, such as a web browser or an e-mail client, all of this can be achieved through an intuitive API.

SN TCP/IP Stack for PlayStation 2 provides a socket API on the EE processor, that closely resembles the BSD socket API but with extensions so that multiple threads on the EE can use the socket API. The TCP/IP stack and drivers run on the IOP processor so that it doesn't use up valuable EE processing time. Sockets can be blocking or non-blocking, depending on how the application wants to use the multi-threaded support provided in the socket API.

What does it include?

Build components:

- TCP/IP Stack and USB modem driver or Ethernet driver supplied as an IOP executable (IRX file). Please note that v1.01 only contains modem support and v1.02 only contains Ethernet support. A future release will combine both modem and Ethernet support.
- EE socket and modem API supplied as a C header file and object module.

Main features:

- Small footprint: IOP - 276KB and EE - 19KB *
- Modular C code.
- Support for USB analogue modems conforming to the USB comms class specification Abstract Control Model (hard modems).
- Support for USB-Ethernet adapters

* Non-debug version only. The footprint sizes may change in later releases of the product.

Supported protocols:

- TCP/IP - Transport Control Protocol / Internet Protocol
- UDP - User Datagram Protocol

- PPP - Point-to-Point Protocol (modem version only)
- ARP – Address Resolution Protocol (Ethernet version only)
- ICMP - Internet Control Message Protocol
- DNS client

How does it work?

Building the SN TCP/IP Stack into your game:

- Integrate with your application using either SN Systems' or Sony's PlayStation 2 software development tools.
- Two versions of the IOP IRX files are provided: one with debug tracing/logging for use during game development, another for release in games.
- Code written to use our socket / modem APIs is designed to be portable to other consoles.

The organization of the SN TCP/IP Stack for PlayStation 2 is shown in Figure 1:

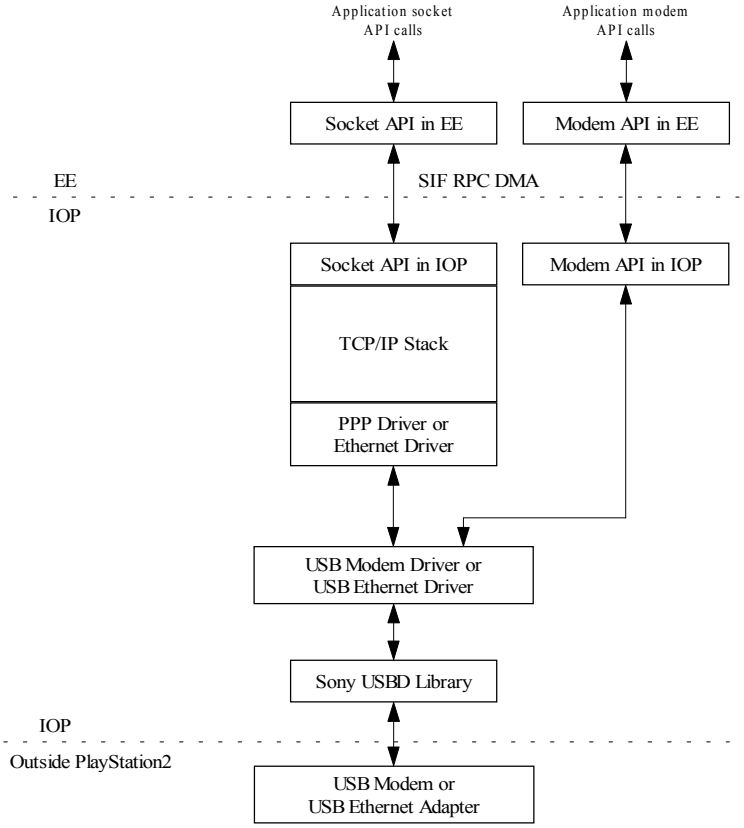

Figure 1: TCP/IP Stack Block Diagram

Figure 1 shows the following software components:

1 Socket API In EE

2 Modem API In EE (Modem version only)

3 Socket API In IOP

4 Modem API In IOP (Modem version only)

5 TCP/IP Stack

6 PPP Driver (Modem version only)

7 Ethernet Driver (Ethernet version only)

8 USB Modem Driver (Modem version only)

9 USB Ethernet Driver (Ethernet version only)

10 Sony USBD Library

Currently the SN TCP/IP Stack for PlayStation 2 software components are supplied as follows:

- The Sony USBD library is supplied by Sony as part of the Sony PlayStation 2 libraries (`usbd.irx`). This component will need to be loaded before loading the TCP/IP stack executable (See below for file name).
- EE socket and modem APIs are supplied as C header files and object library.
- All other components, including the TCP/IP Stack , the USB Modem driver and the USB Ethernet driver, are supplied in one of the following IOP executable files:
  - `sntcpip.irx` – Modem version for released titles – no run time debug logging
  - `sndbgip.irx` – Modem version for title development – includes run time debug logging
  - `sntcpet.irx` – Ethernet version for released titles – no run time debug logging
  - `sndbget.irx` – Ethernet version for title development – includes run time debug logging

About this manual

This manual contains information on the installation and design of the SN TCP/IP Stack for Play Station 2, together with a full API reference.

This manual is broken down into the following sections:

Chapter 1: Getting started describes which USB modem or Ethernet adapter to use, how to install the SN TCP/IP Stack software, and how to build and run the example programs.

Chapter 2: Software design details the design of the SN TCP/IP Stack for PlayStation 2.

Chapters 3-5: Socket API in EE, *Modem API in EE* and *EE utilities* describe the data structures and API associated with the two programmable components of the SN TCP/IP Stack for PlayStation 2.

Updates and technical support

There will be regular updates to SN TCP/IP Stack for PlayStation 2. These will be available to be downloaded from the technical support area of the SN Systems web site, so remember to check out:

http://www.snsys.com

We recommend that you make regular use of this service and quickly take advantage of any new features added to the software, report or download bug reports, gain answers to questions that may be causing you difficulty and keep up-to-date on news concerning the development industry.

This product is backed by SN Systems' commitment to continual enhancement, development and technical support.

If you experience any difficulties, please do not hesitate to contact our technical support at SN Systems:

Mail: SN Systems Ltd
4th Floor - Redcliff Quay
120 Redcliff Street
Bristol BS1 6HU
United Kingdom

Tel.: +44 (0)117 929 9733

Fax: +44 (0)117 929 9251

WWW: http://www.snsys.com

E-mail (support): support@snsys.com

E-mail (sales): sales@snsys.com

Chapter 1: **Getting started**

Installation

This chapter provides notes on installing SN TCP/IP Stack for PlayStation 2, including hardware and software requirements, and how to build and run the sample programs supplied.

Please note that v1.01 only contains modem support and v1.02 only contains Ethernet support. A future release will combine both modem and Ethernet support.

Supported modems

The following modems are supported in release 1.01:

- Actiontec Call Waiting USB modem

Supported USB-Ethernet Adapters

The following Ethernet adapters are supported in release 1.02:

Corega FEther USB-TX (Available in Japan)

D-Link DU-E100 (Available in Europe)

D-Link DSB-650TX (Available in North America)

Sony libraries

For information about which version of the Sony libraries has been used to test versions 1.01 and 1.02, please see the relevant release notes and readme file.

Installing the software

The TCP/IP stack header files, library, IOP executables and sample programs are installed by downloading a `ZIP` file.

- Extracting this `ZIP` file will copy the software to the `\usr\local\sce` directory and its subdirectories.
- A `README.TXT` file will also be created in `\usr\local\sce\doc\sntcpip`. Please read the `README.TXT` file for release notes relating to the installed version.

Building and running the sample programs

SN TCP/IP sample programs are located in `\usr\local\sce\ee\sample\sntcpip`.

Each example program comes with its own `makefile`, which is similar to the Sony example `makefiles`. Just type "`make`" and the program should build.

For detailed information about how to run any particular example program, consult the `readme_e.txt` file in the example program's directory.

Chapter 2: **Software design**

General design considerations

SN TCP/IP Stack for PlayStation 2 provides a socket API on the EE processor, that closely resembles the BSD socket API but with extensions so that multiple threads on the EE can use the socket API. Sockets can be blocking or non-blocking, depending on how the application wants to use the multi-threaded support provided in the socket API.

The TCP/IP stack and drivers run on the IOP processor so that it doesn't use up valuable EE processing time.

EE to IOP DMA method

The RPC (Remote Procedure Call) DMA API provided in the EE and IOP kernel is used to perform the DMA transfers.

Multithreaded access to EE Socket API

In order to make the blocking behavior for applications calling the socket API in the EE, the same as it would be if the stack was also in the EE, the following strategy has been taken:

- The EE socket API is made aware of the maximum number of threads that will access the socket API via a parameter to the socket API initialization function. An RPC server is created for each of these threads, so there is a (virtual) DMA channel for each EE thread that uses the EE socket API.
- Hence if a call is made to a blocking socket function from a particular EE thread, the RPC server for that call won't complete until the socket operation completes in the IOP and the response is sent back to the RPC client in the EE. Whilst this is happening, the thread in the EE that made the call will be blocked.
- Each thread in the EE that is going to use the socket API must register itself with the socket API before making use of other functions provided by the EE socket API. This will allow the EE socket API to get the thread ID for the registering thread and allocate an RPC server to that thread. Whilst the thread is registered, any calls made by it to the EE socket API will check the thread ID of the caller and make sure that it has an RPC server registered to it; if not an error will be returned. When a thread is about to terminate, or no longer

requires access to the EE socket API, it should call a function in the socket API to deregister itself, otherwise a point might be reached where the IOP has allocated its maximum number of threads (currently 10).

The above strategy enables most of the reentrancy problems associated with the traditional BSD socket API to be resolved, e.g. functions such as `gethostbyname` which return a pointer to a static buffer contained in the socket API can be made to support calls from more than one thread by having one copy of the static buffer per thread.

Chapter 3: **Socket API in EE**

Introduction

The socket API in EE consists of a header file, `snsocket.h`, which exports the functions implemented in `snsocket.c`.

Some of the functions are entirely implemented on the EE and require no dialogue with the IOP. An example of such a function is `htonl` which converts a long integer from host to network byte order.

Most of the functions require a dialogue with the IOP. An example of such a function is `send` which sends data on a connected socket. The implementation of the `send` function passes the function parameters and the data to be sent to the IOP via DMA and then waits for the IOP to pass back the function return value via DMA. On receiving the return value from the IOP the `send` function returns the same value to the caller.

Header files

All EE socket applications should include `SNSOCKET.H`. This file includes `SNSKDEFS.H` and `SNTYPES.H`, so there is no need to include those separately.

If needed, `SNEEUTIL.H` and/or `SNTCUTIL.H` should be included after `SNSOCKET.H`.

Socket API in EE service summary

The socket API in the EE provides a header file called `snsocket.h` and is implemented in a file called `snsocket.c`.

The following list summarizes all of the SN sockets functions which are accessible to the user:

`sockAPIinit`	Initializes the socket API
`sockAPIregthr`	Registers a thread for use by the socket API
`sockAPIderegthr`	Deregisters a thread from use by the socket API
`socket`	Create a socket (an endpoint for communication)
`FD_ZERO`	Initializes a socket descriptor set to the empty set
`FD_SET`	Adds a socket descriptor to a descriptor set

FD_ISSET	Tests whether a socket descriptor is a member of a set
FD_CLR	Removes a socket descriptor from a set
bind	Bind a local address to a socket
connect	Connect a socket to a specific address
listen	Requests a socket to listen for connection requests
accept	Accept a connection request and establish a new socket
closesocket	Close a socket
recv	Receive data from a connected socket
recvfrom	Receive data from a socket (gets sender's address)
recvmsg	Receive scattered data from a socket (gets sender's address)
send	Send data to a socket
sendto	Send data to a socket (with destination address)
sendmsg	Send scattered data to a socket (with destination address)
sn_errno	Fetch most recent error recorded for socket
sn_h_errno	Returns the error code from the most recent failing call to gethostbyname made by the calling thread.
sn_stack_state	Returns the current state of the TCP/IP stack.
sndbg_print_stats	Prints debug statistics to IOP printf channel
sndns_add_server	Adds a DNS server to the DNS server list held in the stack
shutdown	Shutdown all or part of a full duplex socket connection
select	Select sockets ready to receive or send data
gethostname	Gets standard name for local machine
gethostbyname	Converts domain name to IP address
getpeername	Get the address of the remote end of a connected socket
getsockname	Get the local address of a socket
getsockopt	Get a particular socket option
setsockopt	Set a particular socket option

The following IP services are also useful:

htonl	Covert long from host to network endian form
htons	Convert short from host to network endian form
ntohl	Convert long from network to host endian form

ntohs	Convert short from network to host endian form
inet_addr	Convert dotted IP address to numeric form
inet_aton	Convert dotted IP address to numeric form
inet_ntoa	Convert numeric IP address to dotted form

Details of how to use each of these functions is covered in the following sections:

accept

Prototype

```
sn_int32 accept(
 sn_int32                  s,
 struct sockaddr*          addr,
 sn_int32*                 addrlen);
```

Description

Accepts a connection request.

Parameters

The parameters for this function are as follows:

`s`

Input: Socket descriptor identifying the socket on which to wait; `socket`, `bind` and `listen` must already have been called for this socket. *Output:* None.

`addr`

Input: Points to where the address of the client which is making the connection request will be returned by this function. If this is not required, set this parameter to `NULL`. *Output:* `*addr` contains the address of the client (if the function was successful and `addr` was not set to `NULL`).

`addrlen`

Input: Points to a value specifying the maximum size in bytes of the storage area pointed to by `addr`. If `addr` was set to `NULL`, also set this to `NULL`. *Output:* `*addrlen` contains the actual length of the address in bytes (if the function was successful and `addr` was not set to `NULL`).

Returns

If successful: A positive (non-zero) socket descriptor will be returned; `*addr` and `*addrlen` will contain additional returned information as described above. *On failure:* The error status –1 is returned; `*addr` and `*addrlen` will not be modified; `sn_errno` can be used to obtain the reason for failure:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_ESTKDOWN`	The TCP/IP Stack has not been started
`EBADF`	The socket descriptor s is invalid
`EINVAL`	One of the following conditions: The socket is no longer accepting connections `addr` is not `NULL` but `addrlen` is `NULL` `*addrlen` specified a size which is not large enough to hold the address
`ECONNABORTED`	The connection was aborted
`EOPNOTSUPP`	`s` is not of type `SOCK_STREAM`
`EWOULDBLOCK`	The socket is in non-blocking mode and no connection request is pending

ENOMEM	Out of memory
SN_REQSIZE	Internal error (IOP RPC size mismatch)

bind

Prototype

```
sn_int32 bind(
 sn_int32                  s,
 const struct sockaddr*    addr,
 sn_int32                  addrlen);
```

Description

Binds a local address to a socket.

Parameters

The parameters for this function are as follows:

`s` — *Input:* Socket descriptor identifying the socket to be bound. *Output:* None.

`addr` — *Input:* Points to the local address to which the socket should be bound. If `addr` is `NULL` an AFINET address of all zeroes will be used and `addrlen` will be ignored. *Output:* None.

`addrlen` — *Input:* The size in bytes of the address contained in `*addr`. Ignored if `addr` is `NULL`. *Output:* None.

Returns

If successful: A value of zero is returned. *On failure:* The error status –1 is returned; `sn_errno` can be used to obtain the reason for failure:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_ESTKDOWN`	The TCP/IP Stack has not been started
`EBADF`	The socket descriptor `s` is invalid
`EADDRNOTAVAIL`	The specified address is not available at the local host
`EADDRINUSE`	The specified address is already being used
`EINVAL`	One of the following conditions: The socket is already bound to an address `addrlen` is invalid
`ENOMEM`	Out of memory
`SN_REQSIZE`	Internal error (IOP RPC size mismatch)

closesocket

Prototype

```
sn_int32 closesocket( sn_int32    s);
```

Description

Closes a socket.

Parameters

The parameters for this function are as follows:

`s` *Input*: Socket descriptor for the socket to be closed. *Output*: None.

Returns

If successful: A value of zero is returned. *On failure*: The error status –1 is returned; `sn_errno` can be used to obtain the reason for failure:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_ESTKDOWN`	The TCP/IP Stack has not been started
`EBADF`	The socket descriptor `s` is invalid
`SN_REQSIZE`	Internal error (IOP RPC size mismatch)

connect

Prototype

```
sn_int32 connect(
 sn_int32                  s,
 const struct sockaddr*    addr,
 sn_int32                  addrlen);
```

Description

Connects a socket to a specific address.

Parameters

The parameters for this function are as follows:

`s`

Input: Socket descriptor identifying the socket which is to be connected. *Output:* None.

`addr`

Input: Points to the address to which the socket should be connected. *Output:* None.

`addrlen`

Input: The size in bytes of the address contained in `*addr`. *Output:* None.

Returns

If successful: A value of zero is returned. *On failure:* The error status –1 is returned; `sn_errno` can be used to obtain the reason for failure:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_ESTKDOWN`	The TCP/IP Stack has not been started
`EBADF`	The socket descriptor s is invalid
`EADDRNOTAVAIL`	The specified address is not available at the local host
`EADDRINUSE`	The specified address is already being used
`EAFNOSUPPORT`	The address family specified by `*addr` cannot be used with this socket.
`EINVAL`	One of the following conditions:

The socket is already bound to an address
Parameter addr is `NULL`
Parameter addrlen is not valid.

`EISCONN`	The socket is already connected
`ETIMEDOUT`	Timed out before connection established
`ECONNREFUSED`	The connection was rejected
`EINPROGRESS`	The socket is non-blocking and the connection could not be immediately established
`EALREADY`	The socket is non-blocking and a previous connection attempt is in progress
`EOPNOTSUPP`	This socket does not support this operation

ENOMEM	Out of memory
SN_REQSIZE	Internal error (IOP RPC size mismatch)

FD_CLR, FD_ISSET, FD_SET, FD_ZERO

These are implemented as macros in `snsocket.h`. Each of these macros operates on a socket descriptor set of type `fd_set`, which is also defined in `snsocket.h`.

In the following descriptions, `s` is a socket descriptor and `p` is a pointer to type `fd_set`.

- **`FD_CLR`**`( s, p )`
  Removes the descriptor `s` from set.
- **`FD_ISSET`**`( s, p )`
  Non-zero if `s` is a member of the set. Otherwise, zero.
- **`FD_SET`**`( s, p )`
  Adds descriptor `s` to set.
- **`FD_ZERO`**`( p )`
  Initializes the set to the empty set.

gethostbyaddr

The standard BSD function `gethostbyaddr` is not supported.

gethostbyname

Prototype

```
struct hostent* gethostbyname( const sn_char*  name);
```

Description

Converts a domain name to an IP address. This function blocks the calling thread whilst it attempts to resolve the domain name. One `hostent` structure is allocated per registered thread, so it's OK to call this function from more than one thread, but be sure to copy all the required information from the returned `hostent` before calling this function again from the same thread. There is currently no support for a non-blocking version of this function. To perform a non-blocking `gethostbyname`, create a new thread that calls `gethostbyname` and have the new thread signal back to the main thread when it completes. This function can be used to obtain the IP address of the PS2 by calling it like this `gethostbyname(LOCAL_NAME)`.

Parameters

The parameters for this function are as follows:

`name` — *Input:* A pointer to the null-terminated name of the host to resolve. *Output:* None.

Returns

If successful: A pointer to a `hostent` structure is returned, with the fields set up as follows:

`h_name`	points to a copy of the string supplied in the `*name` parameter
`h_aliases`	alternate names not supported, always points to a null
`h_addrtype`	always `AF_INET`
`h_length`	the length in bytes of each address (4)
`h_addr_list`	points to a list containing one address which is the IP address resolved from `*name`.

On failure: A `NULL` pointer is returned; `sn_h_errno` can be used to obtain the reason for failure. Unlike some stacks, the global variable `h_errno` is not supported, instead the function `sn_h_errno` provides an error value which is unique to the calling thread:

`SN_ESTKDOWN`	The TCP/IP Stack has not been started
`EINVAL`	`name` was too long or was a `NULL` pointer

The following standard BSD error codes are not supported in the current release; any of these failures will cause the general error `SN_EDNSFAIL` to be returned:

`HOST_NOT_FOUND`	Authoritative answer host not found
`TRY_AGAIN`	Non-authoritive host not found, or `SERVERFAIL`

NO_RECOVERY	Non-recoverable, FORMERR, REFUSED, NOTIMP
NO_DATA	Valid name, no data record of requested type
NO_ADDRESS	No address, look for MX record

gethostname

Prototype

```
sn_int32 gethostname(
 sn_char*                  name,
 sn_int32                  namelen);
```

Description

Gets standard host name for the local machine.

Parameters

The parameters for this function are as follows:

`name`

Input: A pointer to where the name will be returned. *Output:* `*name` contains the returned name.

`namelen`

Input: The size in bytes of the area pointed to by name. *Output:* None.

Returns

If successful: A value of zero is returned and `*name` contains the host name. *On failure:* The error status –1 is returned and `*name` will not be modified. It may fail for one of the following reasons, but the reason for failure is not available to the caller:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_ESTKDOWN`	The TCP/IP Stack has not been started
`EINVAL`	`namelen` is not large enough to hold the name
`SN_REQSIZE`	Internal error (IOP RPC size mismatch)

getpeername

Prototype

```
sn_int32 getpeername(
 sn_int32                  s,
 struct sockaddr*          addr,
 sn_int32*                 addrlen);
```

Description

Gets the address of the remote end of a connected socket.

Parameters

The parameters for this function are as follows:

`s`

Input: Socket descriptor identifying the socket for which the remote address is required. *Output:* None.

`addr`

Input: Points to where the remote address will be returned by this function. *Output:* `*addr` contains the remote address.

`addrlen`

Input: Points to a value specifying the maximum size in bytes of the storage area pointed to by `addr`. *Output:* `*addrlen` contains the actual length of the address in bytes.

Returns

If successful: A positive (non-zero) socket descriptor will be returned; `*addr` and `*addrlen` will contain additional returned information as described above. *On failure:* The error status –1 is returned; `*addr` and `*addrlen` will not be modified; `sn_errno` can be used to obtain the reason for failure:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_ESTKDOWN`	The TCP/IP Stack has not been started
`EBADF`	The socket descriptor `s` is invalid
`ENOMEM`	Out of memory
`ENOTCONN`	The socket is not connected
`EINVAL`	One of the following conditions:
`addr` is `NULL`	
`addrlen` is `NULL`	
`*addrlen` is invalid	
`SN_REQSIZE`	Internal error (IOP RPC size mismatch)

getsockname

Prototype

```
sn_int32 getsockname(
 sn_int32                  s,
 struct sockaddr*          addr,
 sn_int32*                 addrlen);
```

Description

Gets the local address of a socket.

Parameters

The parameters for this function are as follows:

`s` — *Input:* Socket descriptor identifying the socket for which the local address is required. *Output:* None.

`addr` — *Input:* Points to where the local address will be returned by this function. *Output:* `*addr` contains the local address.

`addrlen` — *Input:* Points to a value specifying the maximum size in bytes of the storage area pointed to by `addr`. *Output:* `*addrlen` contains the actual length of the address in bytes.

Returns

If successful: A value of zero will be returned; `*addr` and `*addrlen` will contain additional returned information as described above. *On failure:* The error status –1 is returned; `*addr` and `*addrlen` will not be modified; `sn_errno` can be used to obtain the reason for failure:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_ESTKDOWN`	The TCP/IP Stack has not been started
`EBADF`	The socket descriptor `s` is invalid
`ENOMEM`	Out of memory
`EINVAL`	One of the following conditions:
`addr` is `NULL`	
`addrlen` is `NULL`	
`*addrlen` is invalid	
`SN_REQSIZE`	Internal error (IOP RPC size mismatch)

getsockopt

Prototype

```
sn_int32 getsockopt(
 sn_int32                  s,
 sn_int32                  level,
 sn_int32                  optname,
 void*                     optval,
 sn_int32*                 optlen);
```

Description Get a particular socket option.

Parameters The parameters for this function are as follows:

`s` *Input:* Socket descriptor identifying the socket for which option information is required. *Output:* None.

`level` *Input:* Indicates the protocol level for which socket information is required, only `SOL_SOCKET` is supported. *Output:* None.

`optname` *Input:* Specifies the option for which information is to be obtained (although all of the options listed are recognised as being valid, only those marked ► are operational). *Output:* None

`SO_REUSEADDR`	`bool`	►	Local address reuse
`SO_KEEPALIVE`	`bool`	►	Keep connections alive
`SO_DONTROUTE`	`bool`	►	Routing bypass for outgoing messages
`SO_BROADCAST`	`bool`	►	Permission to transmit broadcast messages
`SO_OOBINLINE`	`bool`	►	Allow out-of-band data in band
`SO_LINGER`	`struct`	►	Linger on close if data present
`SO_SNDBUF`	`int`		Buffer size for send
`SO_RCVBUF`	`int`		Buffer size for receive
`SO_SNDLOWAT`	`int`		Buffer low limit for send
`SO_RCVLOWAT`	`int`		Buffer low limit for receive
`SO_SNDTIMEO`	`struct`		Timeout limit for send
`SO_RCVTIMEO`	`struct`		Timeout limit for receive
`SO_TYPE`	`int`	►	Get socket type
`SO_ERROR`	`int`	►	Get and clear error on socket

`optval` *Input:* Points to where the option information will be returned by this function. *Output:* `*optval` contains the option information. The size of each option is indicated above, `bool` and `int` are returned in a 32-bit integer, for `bool` a value of zero is false and non-zero is true. The `linger`

structure required by SO_LINGER. The timeval structures required by SO_SNDTIMEO and SO_RCVTIMEO are defined in snsocket.h.

optlen

Input: Points to a value specifying the maximum size in bytes of the storage area pointed to by optval. *Output:* *optlen contains the actual length of the returned option information in bytes.

Returns

If successful: A value of zero will be returned; *optval and *optlen will contain additional returned information as described above. *On failure:* The error status –1 is returned; *optval and *optlen will not be modified; sn_errno can be used to obtain the reason for failure:

SN_ENOTINIT	The socket API has not been initialized
SN_ETHNOTREG	The calling thread is not registered
SN_ESTKDOWN	The TCP/IP Stack has not been started
EBADF	The socket descriptor s is invalid
ENOPROTOOPT	The option is not valid for the level specified
EINVAL	One of the following conditions:
optval is NULL	
optlen is NULL	
*optlen is not large enough to hold the requested option	
SN_REQSIZE	Internal error (IOP RPC size mismatch)

htonl and htons

These are implemented as macros in `snsocket.h`. They convert 32-bit / 16-bit values from host to network byte order.

Because the EE (and IOP) run in little-endian mode, it is necessary for these macros to implement byte-swapping as network byte order is big-endian. These macros call the functions `sn_swap32` and `sn_swap16` which are implemented in `snsocket.c`.

Prototype `sn_uint32 htonl(sn_uint32)`

Description Convert a `long` from host to network endian form.

Prototype `sn_uint16 htons(sn_uint16)`

Description Convert a `short` from host to network endian form.

inet_addr

Prototype

```
sn_uint32 inet_addr( const sn_char*     cp);
```

Description

Converts the dotted IP address supplied in `cp` to a 32-bit IP address in network byte order. This function is re-entrant, it can be called before the socket API is initialized, and it can be called from any thread without the need for the thread to be registered. Note that this function is deprecated and any new code should use `inet_aton`.

Parameters

The parameters for this function are as follows:

`cp`

Input: Points to null-terminated string containing dotted IP address.
Output: None.

Returns

If successful: The 32-bit IP address is returned. *On failure:* `ONADDR_NONE` is returned.

inet_aton

Prototype

```
sn_uint32 inet_aton(
 const sn_char*            cp,
 struct in_addr*           addr);
```

Description

Converts the dotted IP address supplied via `cp` to a 32-bit IP address in network byte order. This function is reentrant, it can be called before the socket API is initialized, and it can be called from any thread without the need for the thread to be registered.

Parameters

The parameters for this function are as follows:

`cp`

Input: Points to null-terminated string containing dotted IP address. O*utput:* None.

`addr`

Input: Points to where the IP address will be returned, or set it to `NULL` if you just want to validate the dotted IP address. *Output:* If successful `*addr` will contain the IP address.

Returns

If successful: A value of 1 is returned and `*addr` (unless it was `NULL`) contains the IP address. *On failure:* A value of 0 is returned and `*addr` is not updated.

inet_ntoa

Prototype

```
sn_char* inet_ntoa(
 struct in_addr            in);
```

Description

Converts the 32-bit IP address supplied in `in`, into a dotted IP address. The socket API must be initialized and the calling thread must be registered before calling this function. One `char` buffer for the return value of this function is allocated per registered thread, so it's OK to call this function from more than one thread, but be sure to copy all the required information from the returned pointer before calling this function again from the same thread.

Parameters

The parameters for this function are as follows:

`in`

Input: The address (in network byte order) to be converted into dotted form.
Output: None.

Returns

If successful: A character pointer is returned pointing to a null-terminated string containing the IP address in standard dot form. *On failure:* A `NULL` pointer is returned.

listen

Prototype

```
sn_int32 listen(
 sn_int32                  s,
 sn_int32                  backlog);
```

Description

Request a socket to listen for connection requests.

Parameters

The parameters for this function are as follows:

`s`

Input: Socket descriptor identifying the socket that should listen for connection requests. *Output:* None.

`backlog`

Input: Maximum number of connection requests which the socket will be able to queue. *Output:* None.

Returns

If successful: A value of zero is returned. *On failure:* The error status –1 is returned; `sn_errno` can be used to obtain the reason for failure:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_ESTKDOWN`	The TCP/IP Stack has not been started
`EBADF`	The socket descriptor `s` is invalid
`EOPNOTSUPP`	The operation is not supported by this type of socket
`SN_REQSIZE`	Internal error (IOP RPC size mismatch)

ntohl and ntohs

These are implemented as macros in `snsocket.h`. They convert 32-bit / 16-bit values from network to host byte order.

Because the EE (and IOP) run in little-endian mode, it is necessary for these macros to implement byte-swapping because network byte order is big-endian. These macros call the functions `sn_swap32` and `sn_swap16` which are implemented in `snsocket.c`.

Prototype `sn_uint32 ntohl(sn_uint32)`

Description Convert a `long` from network to host endian form.

Prototype `sn_uint16 ntohs(sn_uint16)`

Description Convert a `short` from network to host endian form.

recv

Prototype

```
sn_int32 recv(
 sn_int32                s,
 void*                   buf,
 sn_int32                len,
 sn_int32                flags);
```

Description

Receive data from a connected socket.

Parameters

The parameters for this function are as follows:

`s`

Input: Socket descriptor identifying the socket that data should be received from. *Output:* None.

`buf`

Input: Points to the buffer where received data will be returned. *Output:* `*buf` contains the received data.

`len`

Input: The size of `*buf` in bytes. *Output:* None.

`flags`

Input: Specifies how the receive process should operate, by setting one or more of the flags. *Output:* None.

`MSG_OOB`	Read out-of-band data
`MSG_PEEK`	Read the received data, but do not remove it from the socket

Returns

If successful: The number of bytes of data stored in `*buf` is returned, and `*buf` contains the received data. If the socket has been closed by the sender 0 is returned. *On failure:* The error status –1 is returned; `sn_errno` can be used to obtain the reason for failure:

SN_ENOTINIT	The socket API has not been initialized
SN_ETHNOTREG	The calling thread is not registered
SN_ESTKDOWN	The TCP/IP Stack has not been started
EBADF	The socket descriptor s is invalid
ENOTCONN	The socket is not connected
ECONNRESET	The connection has been reset
EINVAL	One of the following conditions: len is < 0 buf is NULL an error was detected whilst processing out-of-band data
EWOULDBLOCK	The socket is in non-blocking mode and no data is available
SN_REQSIZE	Internal error (IOP RPC size mismatch)

recvfrom

Prototype

```
sn_int32 recvfrom(
 sn_int32                  s,
 void*                     buf,
 sn_int32                  len,
 sn_int32                  flags,
 struct sockaddr*          from,
 sn_int32*                 fromlen);
```

Description

Receives data from a socket

Parameters

The parameters for this function are as follows:

`s`

Input: Socket descriptor identifying the socket that data should be received from. *Output:* None.

`buf`

Input: Points to the buffer where received data will be returned. *Output:* `*buf` contains the received data.

`len`

Input: The size of `*buf` in bytes. *Output:* None.

`flags`

Input: Specifies how the receive process should operate, by setting one or more of the flags (below). *Output:* None.

`MSG_OOB`	Read out-of-band data
`MSG_PEEK`	Read the received data, but do not remove it from the socket

`from`

Input: A pointer to where this function will return the address of the sender of the received data. Set this to `NULL` if the address is not required. *Output:* `*from` contains the address of the sender.

`fromlen`

Input: Points to a value specifying the maximum size in bytes of the storage area pointed to by `from`. If the parameter `from` is set to `NULL`, also set this to `NULL`. *Output:* `*fromlen` contains the actual length of the address `*from` in bytes.

Returns

If successful: The number of bytes of data stored in `*buf` is returned; `*buf` will contain the received data; `*from` and `*fromlen` will contain additional returned information as described above. If the socket has been closed by the sender, 0 is returned. *On failure:* The error status –1 is returned; `sn_errno` can be used to obtain the reason for failure:

SN_ENOTINIT	The socket API has not been initialized
SN_ETHNOTREG	The calling thread is not registered
SN_ESTKDOWN	The TCP/IP Stack has not been started
EBADF	The socket descriptor `s` is invalid
ENOTCONN	The socket is not connected
ECONNRESET	The connection has been reset
EINVAL	One of the following conditions: `len` is < 0 an error was detected whilst processing out-of-band data `fromlen` was not large enough to hold the address.
EWOULDBLOCK	The socket is in non-blocking mode and no data is available
SN_REQSIZE	Internal error (IOP RPC size mismatch)

recvmsg

Prototype

```
sn_int32 recvmsg(
 sn_int32                  s,
 struct msghdr*            msg,
 sn_int32                  flags);
```

Description

Receives scattered data from a socket

Parameters

The parameters for this function are as follows:

`s`

Input: Socket descriptor identifying the socket that data should be received from. *Output:* None.

`msg`

Input: Points to a structure defining how the received data should be processed. *Output:* None.

`struct msghdr` is defined in `snsocket.h` as follows:

```
struct msghdr{
       struct sockaddr* msg_name;
       sn_int32         msg_namelen;
       struct iovec*    msg_iov;
       sn_int32         msg_iovlen;
       void*            msg_control;
       sn_int32         msg_controllen;
       sn_int32         msg_flags;
       };
```

`msg_name`
Input: A pointer to where this function will return the address of the sender of the received data. Set this to `NULL` if the address is not required. *Output:* `*msg_name` contains the address of the sender.

`msg_namelen`
Input: Specifies the maximum size in bytes of the storage area pointed to by `msg_name`. If `msg_name` is set to `NULL`, set this to zero. *Output:* the actual length of `*msg_name`.

`msg_iov`
Input: points to an array of data vectors describing the locations of storage buffers for the received data. *Output:* None.

`struct iovec` is defined in `snsocket.h` as follows:

```
struct iovec {
       void*    iov_base;
       sn_int32 iov_len;
              };
```

`iov_base`
Input: points to storage for the received data. *Output:* incremented by number of bytes received using this vector and *(original value of `iov_base`) contains the received data.

`iov_len`
Input: size in bytes of storage area pointed to by `iov_base`. *Output:* decremented by the number of bytes received using this vector.

`msg_iovlen`
Input: defines the number of `iovecs` in the array pointed to by `msg_iov`. *Output:* None.

`msg_control`
Input: This parameter is not used, set it to `NULL`. *Output:* None.

`msg_controllen`
Input: This parameter is not used, set it to zero. *Output:* None.

`msg_flags`
Input: This parameter is not used, set it to zero. *Output:* None.

`flags` — *Input:* Specifies how the receive process should operate, by setting one or more of the flags. *Output:* None

`MSG_OOB`	Read out-of-band data
`MSG_PEEK`	Read the received data, but do not remove it from the socket

Returns

If successful: The total number of bytes of received data is returned, and various other parameters are updated as described above. If the socket has been closed by the sender 0 is returned. *On failure:* The error status –1 is returned; `sn_errno` can be used to obtain the reason for failure:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_ESTKDOWN`	The TCP/IP Stack has not been started
`EBADF`	The socket descriptor `s` is invalid
`ENOTCONN`	The socket is not connected
`ECONNRESET`	The connection has been reset
`EINVAL`	One of the following conditions:

`msg` is `NULL`
`msg_iovlen` is not in range `1..IOV_MAX`
The buffer length in a data vector is < 0
The buffer pointer in a data vector is `NULL`
`msg_name` isn't `NULL` and `msg_namelen` specifies a value which is too small to hold the address

`msg_control` is not `NULL`
`msg_controllen` is not zero
`msg_flags` is not zero
An error was detected whilst processing out-of-band data

`EWOULDBLOCK`	The socket is in non-blocking mode and no data is available
`SN_REQSIZE`	Internal error (IOP RPC size mismatch)

select

Prototype

```
sn_int32 select(
 sn_int32                  nfds,
 fd_set*                   readfds,
 fd_set*                   writefds,
 fd_set*                   exceptfds,
 struct timeval*           timeout);
```

Description

Selects sockets ready for receive or send. Note about the operation of the SN implementation of `select`. There is NO clear specification for the BSD socket `select` function. The SN TCP/IP stack operates as follows:
Function `select returns >=0` if there has been no problem handling the request. That does NOT necessarily mean that the interrogated sockets are error free.
For example, if you call `select` to see if a set of sockets is ready to read (write), the function will return a positive value indicating that "something" of interest on one or more of the sockets is worth noting.
If a particular socket has no outstanding error conditions pending, then the socket can be assumed to be readable (writable). Otherwise, all that you can surmise is that the socket has a problem (read or write or connection) and might still be readable (writable).
The error value can be retrieved and reset by calling `getsockopt` with the optname set to `SO_ERROR`. If the returned `optval` is non-zero, you must call `select` again to see if the socket is really readable (writable). Only when `select` says that the socket is ready and `getsockopt` indicates that there is no error, should you proceed to read (write) the socket, expecting success.

Parameters

The parameters for this function are as follows:

`nfds`

Input: The maximum socket id to be examined in the `fd_sets` or can be set to –1 to examine all possible socket IDs. *Output:* None.

`readfds`

Input: Points to a socket set that specifies which sockets should be checked for received data. Set it to `NULL` if checking for received data is not required. *Output:* Providing it wasn't `NULL`, `*readfds` specifies which sockets have data ready to receive.

`writefds`

Input: Points to a socket set that specifies which sockets should be checked for sent data. Set it to `NULL` if checking for sent data is not required. *Output:* Providing it wasn't `NULL`, `*writefds` specifies which sockets have no data remaining to be sent.

`exceptfds`	*Input:* Points to a socket set that specifies which sockets should be checked for outstanding exceptions. Set it to `NULL` if checking for exceptions is not required. *Output:* Providing it wasn't `NULL`, `*exceptfds` specifies which sockets have outstanding exceptions. The only supported exception is out-of-band data received.
`timeout`	*Input:* Points to a structure that defines the maximum amount of time that the caller wants to wait for at least one socket to meet the selected criteria. Set it to `NULL` to wait indefinitely, set the `timeout` period to zero to return immediately. *Output:* None.

Returns

If successful: Returns the total number of sockets identified in the returned descriptor sets. *On failure:* The error status –1 is returned.

send

Prototype

```
sn_int32 send(
 sn_int32                s,
 const void*             buf,
 sn_int32                len,
 sn_int32                flags);
```

Description

Sends data to a connected socket

Parameters

The parameters for this function are as follows:

`s`

Input: Socket descriptor identifying the socket that data should be sent on. *Output:* None.

`buf`

Input: Points to the buffer containing the data to be sent. *Output:* None.

`len`

Input: The amount of data contained in `*buf` in bytes. *Output:* None.

`flags`

Input: Specifies how the transmit process should operate, by setting one or more of the flags. *Output:* None.

`MSG_OOB`	Send out-of-band data
`MSG_DONTROUTE`	Do not use routing
`MSG_DONTWAIT`	Send message in a non-blocking manner

Returns

If successful: The number of bytes of data that were sent is returned. *On failure:* The error status –1 is returned; `sn_errno` can be used to obtain the reason for failure:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_ESTKDOWN`	The TCP/IP Stack has not been started
`EBADF`	The socket descriptor s is invalid
`ENOTCONN`	The socket is not connected
`ECONNRESET`	The connection has been reset
`EPIPE`	Cannot send any more out of socket s
`EINVAL`	One of the following conditions:
`len` is < 0	
`buf` is `NULL`	
`len` is invalid for the socket's protocol	
`EWOULDBLOCK`	The socket is in non-blocking mode and would need to block to complete the operation
`EMSGSIZE`	The message is larger than the maximum allowed

`SN_REQSIZE`	Internal error (IOP RPC size mismatch)

sendmsg

Prototype

```
sn_int32 sendmsg(
 sn_int32                  s,
 const struct msghdr*      msg,
 sn_int32                  flags);
```

Description

Sends scattered data to a socket.

Parameters

The parameters for this function are as follows:

`s`

Input: Socket descriptor identifying the socket that data should be sent on. *Output:* None.

`msg`

Input: Points to a structure defining the data to be sent and how it should be processed. *Output:* None.

`struct msghdr` is defined in `snsocket.h` as follows:

```
struct msghdr {
       struct sockaddr* msg_name;
       sn_int32         msg_namelen;
       struct iovec*    msg_iov;
       sn_int32         msg_iovlen;
       void*            msg_control;
       sn_int32         msg_controllen;
       sn_int32         msg_flags;
              };
```

`msg_name`
Input: A pointer to the address to which the data should be sent. If using a connected socket set it to `NULL`. *Output:* None.

`msg_namelen`
Input: The length of `*msg_name`, set it to zero if `msg_name` is `NULL`. *Output:* None.

`msg_iov`
Input: points to an array of data vectors describing the locations of storage buffers containing the data to be sent. *Output:* None.

`struct iovec` is defined in `snsocket.h` as follows:

```
struct iovec {
       void*    iov_base;
       sn_int32 iov_len;
              };
```

`iov_base`
Input: points to storage for the transmitted data. *Output:* incremented by number of bytes transmitted using this vector.

`iov_len`
Input: size in bytes of storage area pointed to by `iov_base`. *Output:* decremented by the number of bytes transmitted using this vector.

`msg_iovlen`
Input: defines the number of `iovecs` in the array pointed to by `msg_iov`. *Output:* None.

`msg_control`
Input: This parameter is not used; set it to `NULL`. *Output:* None.

`msg_controllen`
Input: This parameter is not used; set it to zero. *Output:* None.

`msg_flags`
Input: This parameter is not used; set it to zero. *Output:* None.

`flags` *Input:* Specifies how the transmit process should operate, by setting one or more of the flags. *Output:* None.

`MSG_OOB`	Send out-of-band data
`MSG_DONTROUTE`	Do not use routing
`MSG_DONTWAIT`	Send message in a non-blocking manner

Returns

If successful: The total number of bytes sent is returned, and the io vectors are updated as described above. *On failure:* The error status –1 is returned; `sn_errno` can be used to obtain the reason for failure:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_ESTKDOWN`	The TCP/IP Stack has not been started
`EBADF`	The socket descriptor `s` is invalid
`ENOTCONN`	The socket is not connected
`ECONNRESET`	The connection has been reset
`EPIPE`	Can not send any more data out of socket `s`
`EINVAL`	One of the following conditions:

The buffer length in a data vector is < 0 or is invalid for the socket's protocol

`msg_namelen` specifies a value which is too small to hold the address

`EWOULDBLOCK`	The socket is in non-blocking mode and it would be necessary to block in order to complete the operation
`EMSGSIZE`	The message is larger than the maximum

	message length supported by the protocol
ENOBUFFS	Memory is not available to complete the request
EDESTADDRREQ	A destination address is required but not specified
SN_REQSIZE	Internal error (IOP RPC size mismatch)

sendto

Prototype

```
sn_int32 sendto(
 sn_int32                  s,
 const void*               buf,
 sn_int32                  len,
 sn_int32                  flags,
 const struct sockaddr*    to,
 sn_int32                  tolen);
```

Description

Sends data to a socket.

Parameters

The parameters for this function are as follows:

`s`

Input: Socket descriptor identifying the socket that data should be sent on. *Output:* None.

`buf`

Input: Points to the buffer containing the data to send. *Output:* None.

`len`

Input: The amount of data contained in `*buf` in bytes. *Output:* None.

`flags`

Input: Specifies how the transmit process should operate, by setting one or more of the flags. *Output:* None.

`MSG_OOB`	Send out-of-band data
`MSG_DONTROUTE`	Do not use routing
`MSG_DONTWAIT`	Send message in a non-blocking manner

`to`

Input: A pointer to the address to which the data should be sent. Set it to `NULL` if not required. *Output:* None

`tolen`

Input: The size in bytes of the address contained in `*to`, set it to zero if `to` is `NULL`. *Output:* None.

Returns

If successful: The number of bytes sent is returned. *On failure:* The error status –1 is returned; `sn_errno` can be used to obtain the reason for failure:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_ESTKDOWN`	The TCP/IP Stack has not been started
`EBADF`	The socket descriptor s is invalid
`ENOTCONN`	The socket is not connected
`ECONNRESET`	The connection has been reset

EPIPE	Cannot send any more data out of socket `s`
EINVAL	One of the following conditions:
`len` is < 0	
`len` is invalid for the socket's protocol	
`tolen` specifies an invalid address length	
EWOULDBLOCK	The socket is in non-blocking mode and would have to block in order to complete the operation
EMSGSIZE	The message is larger than the maximum supported by the protocol
ENOBUFS	Memory is not available to complete the request
EDESTADDRREQ	A destination address is required but is not available
SN_REQSIZE	Internal error (IOP RPC size mismatch)

setsockopt

Prototype

```
sn_int32 setsockopt(
 sn_int32                s,
 sn_int32                level,
 sn_int32                optname,
 const void*             optval,
 sn_int32                optlen);
```

Description Sets a particular socket option.

Parameters The parameters for this function are as follows:

`s` *Input:* Socket descriptor identifying the socket for which the option should be set. *Output:* None.

`level` *Input:* Indicates the protocol level for which the socket option should be set, only `SOL_SOCKET` is supported. *Output:* None.

`optname` *Input:* Specifies the option for which is to be set (although all of the options listed are recognised as being valid, only those marked with ► are operational). *Output:* None

`SO_REUSEADDR`	`bool`	►	Local address reuse
`SO_KEEPALIVE`	`bool`	►	Keep connections alive
`SO_DONTROUTE`	`bool`	►	Routing bypass for outgoing messages
`SO_BROADCAST`	`bool`	►	Permission to transmit broadcast messages
`SO_OOBINLINE`	`bool`	►	Allow out-of-band data in band
`SO_LINGER`	`struct`	►	Linger on close if data present
`SO_SNDBUF`	`int`		Buffer size for send
`SO_RCVBUF`	`int`		Buffer size for receive
`SO_SNDLOWAT`	`int`		Buffer low limit for send
`SO_RCVLOWAT`	`int`		Buffer low limit for receive
`SO_SNDTIMEO`	`struct`		Timeout limit for send
`SO_RCVTIMEO`	`struct`		Timeout limit for receive
`SO_NBIO`	`int`	►	Set non-blocking mode
`SO_BIO`	`int`	►	Set blocking mode

`optval` *Input:* Points to where the option information to be setup is contained. Note that although the option value for `SO_NBIO` and `SO_BIO` is ignored, it must still be present. *Output:* None.

`optlen`	*Input:* The size in bytes of the option information pointed to by optval. *Output:* None.

Returns

If successful: Avalue of zero will be returned. *On failure:* The error status –1 is returned; `sn_errno` can be used to obtain the reason for failure:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_ESTKDOWN`	The TCP/IP Stack has not been started
`EBADF`	The socket descriptor s is invalid
`ENOPROTOOPT`	The option is not valid for the level specified
`EINVAL`	One of the following conditions: `optval` is `NULL` `optlen` is less than the size required to hold the specified option
`SN_REQSIZE`	Internal error (IOP RPC size mismatch)

shutdown

Prototype

```
sn_int32 shutdown(
 sn_int32                    s,
 sn_int32                    how);
```

Description

Shuts down all or part of a full-duplex socket connection.

Parameters

The parameters for this function are as follows:

`s`

Input: Socket descriptor identifying the socket to be shut down. *Output:* None.

`how`

Input: Specifies how the shutdown is to be performed:
0 = no further receives are allowed
1 = no further sends are allowed
2 = no further sends or receives are allowed
Output: None.

Returns

If successful: A value of zero is returned. *On failure:* The error status –1 is returned; `sn_errno` can be used to obtain the reason for failure:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_ESTKDOWN`	The TCP/IP Stack has not been started
`EBADF`	The socket descriptor s is invalid
`ENOTCONN`	The socket is not connected
`SN_REQSIZE`	Internal error (IOP RPC size mismatch)

sn_errno

Prototype `sn_int32 sn_errno( sn_int32 s);`

Description Gets error code from most recent socket operation.

Parameters The parameters for this function are as follows:

`s` *Input:* Socket descriptor identifying the socket for which error information is to be returned. *Output:* None.

Returns *If successful:* The most recent error for a socket operation performed on socket descriptor `s` is returned (see the various error values defined by the other socket functions), or, this function call can fail itself for one of the following reasons.

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_ESTKDOWN`	The TCP/IP Stack has not been started
`EBADF`	The socket descriptor `s` is invalid
`SN_REQSIZE`	Internal error (IOP RPC size mismatch)
Other values	As described in the various socket functions

sn_h_errno

Prototype `sn_int32 sn_h_errno( void );`

Description Returns the error code from the most recent failing call to `gethostbyname` made by the calling thread .

Parameters None

Returns If successful, the most recent error for a failing call to `gethostbyname` by this thread is returned, or, this function call can fail itself for one of the following reasons.

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
Other values	As described in `gethostbyname` errors

sn_stack_state

Prototype

```
sn_int32 sn_stack_state(
 sn_int32           new_state,
 sn_int32*          current_state);
```

Description

Can be used to determine the current state (started / stopped) of the TCP/IP stack, or, can be used to change the state from started to stopped, or, vice versa. The stack is initially in the stopped state and must be started via a call to this function before it can be used. In the current version of the software, a modem or Ethernet adapter must be attached when this function is called to start the stack, otherwise the stack initialization will fail. A check can be done for an attached device using `snmdm_get_attached`.

Note: The stack can only be started / stopped once per load of the `sntcpip.irx` file. Trying to start the stack a second time after it has been stopped with this command will result in an error as the stack uses preset data.

Parameters

The parameters for this function are as follows:

`new_state`

Input: This may be set to one of the following values:

`SN_STACK_STATE_READ`	Do not change the state of the stack
`SN_STACK_STATE_START`	Start the stack
`SN_STACK_STATE_STOP`	Stop the stack

Output: None.

`current_state`

Input: Points to the location where the current state of the stack will be returned or set to `NULL` if not required. *Output:* If `current_state` was not `NULL`, and the function worked successfully, one of the following two values will be returned: `SN_STACK_STATE_START` or `SN_STACK_STATE_STOP`.

Returns

If successful: a value of zero will be returned. *On failure:* one of the following error codes will be returned:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`EINVAL`	Parameter `new_state` is invalid
`SN_REQSIZE`	Internal error (IOP RPC size mismatch)

sndbg_print_stats

Prototype `sn_int32 sndbg_print_stats( void );`

Description If using a debug build of the TCP/IP IRX file, causes the IOP to print the network statistics debug information on the `printf` channel.

Parameters None.

Returns *If successful:* a value of zero will be returned. *On failure:* one of the following error codes will be returned:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_ESTKDOWN`	The TCP/IP Stack has not been started
`SN_REQSIZE`	Internal error (IOP RPC size mismatch)

sndns_add_server

Prototype

```
sn_int32 sndns_add_server(
 sn_int32           netend_ip_addr);
```

Description

Adds a DNS server to the list of DNS servers maintained within the TCP/IP stack. The socket API interface must have been initialized before this function is called. It shouldn't be necessary for an application to use this function, as the higher level function `sntc set dns server list` should be used instead.

Parameters

The parameters for this function are as follows:

`netend_ip_addr`

Input: 32-bit IP address of a DNS server, in network byte order. *Output:* None.

Returns

If successful: a value of zero will be returned. *On failure:* one of the following error codes will be returned:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_REQSIZE`	Internal error (IOP RPC size mismatch)
`EINVAL`	The internal DNS server list is full

sockAPIinit

Prototype

```
sn_int32 sockAPIinit( sn_int32   maxthreads);
```

Description

Initializes the socket API. Must be called exactly once by an application that wants to use the EE socket API, before making calls to any of the other functions provided by the socket API (unless the function description expressly allows it). This function performs the necessary initialization required to support access from `maxthreads` for the EE end of the interface, and causes the IOP end of the interface to be initialized. On successful return from this function, the interface will be ready to accept calls to `sockAPIregthr`.

Parameters

The parameters for this function are as follows:

`maxthreads` Specifies the maximum number of threads in the application that will require access to the socket API at any one time. This should be a positive value (> 0) but less than or equal to the maximum currently allowed (10).

Returns

If successful: A value of 0 is returned. *On failure:* One of the values defined below will be returned.

`SN_EMAXTHREAD`	The specified number of threads is larger than the allowed value (currently 10) or <= 0
`SN_EBINDFAIL`	The call to `sceSifBindRpc` failed
`SN_EIOPNORESP`	The IOP failed to initialize its end of the interface within the allowed time. Note that the normal reason for this error is because the `IRX` files have not been preloaded.
`SN_ENOMEM`	There was not enough memory available to complete the initialization
`SN_EALRDYINIT`	This isn't the first time that this function has been called by the application
`SN_REQSIZE`	Internal error (IOP RPC size mismatch)

sockAPIregthr

Prototype

```
sn_int32 sockAPIregthr( void );
```

Description

Register a thread for use of the socket API. sockAPIinit must have been called (not necessarily by the same thread) before this function is called. Must be called once by a thread that wishes to use the socket API. On successful return from this function, the interface will be ready to accept calls to the socket API functions from the thread that called this function. If this function is called when the calling thread is already registered it will return a successful indication and will not cause the socket API to erroneously register two copies of the thread.

Parameters

None

Returns

If successful: A value of 0 is returned. *On failure:* One of the values defined below will be returned.

SN_ENOTINIT	The socket API has not been initialized
SN_EMAXTHREAD	The maximum number of allowed threads are already registered
SN_EINVTHREAD	The thread id of the caller is invalid.

Note: The design relies on a thread ID of zero being invalid. Observation of the operation of the EE seems to bear this out, but it is not explicit in the EE kernel documentation. This error should never be returned.

sockAPIderegthr

Prototype

```
sn_int32 sockAPIderegthr( void );
```

Description

Deregister a thread from use by the socket API. Should be called by a thread that has previously called sockAPIregthr when the thread is about to terminate, or, when it no longer requires access to the socket API in order to free up the store of registered threads in the socket API. If this function is not called the thread will remain registered and may cause the maximum number of allowed threads to be exceeded in other calls to sockAPIregthr.

Parameters

None

Returns

If successful: A value of 0 is returned. *On failure:* One of the values defined below will be returned.

SN_ENOTINIT	The socket API has not been initialized
SN_ETHNOTREG	The calling thread was not registered
SN_EINVTHREAD	The thread id of the caller is invalid

Note: The design relies on a thread ID of zero being invalid. Observation of the operation of the EE seems to bear this out, but it is not explicit in the EE kernel documentation. This error should never be returned.

socket

Prototype

```
sn_int32 socket(
 sn_int32                  af,
 sn_int32                  type,
 sn_int32                  protocol);
```

Description Creates a socket

Parameters The parameters for this function are as follows:

`af` *Input:* This must be set to `AF_INET`. *Output:* None.

`type` *Input:* This specifies the type of socket to be created, must be set to either `SOCK_STREAM` or `SOCK_DGRAM`. *Output:* None.

`protocol` *Input:* This is ignored, as the only address family supported is `AF_INET`, a protocol of `PF_INET` is always used. *Output:* None.

Returns *If successful:* A socket descriptor with a value of > 0 is returned. *On failure:* The error status –1 is returned.

Chapter 4: **Modem API in EE**

Introduction

The modem API in the EE is declared in the same header file as the socket API (`snsocket.h`).

The functions which are exported via `snsocket.h` are as follows:

- `snmdm_connect`
- `snmdm_disconnect`
- `snmdm_get_attached`
- `snmdm_get_connect_err`
- `snmdm_get_state`
- `snmdm_set_mdm_init`
- `snmdm_set_phone_no`
- `snmdm_set_script`

In addition to the above (low-level) modem API functions, there are some higher level utilities which should be used by the application. These higher level utilities are described in "TCP/IP-specific EE utilities (sntcutil.h)" on page 71.

Most of the functions require a dialogue with the IOP, e.g. `snmdm_set_phone_no`. The implementation of this function passes the function parameters to the IOP via DMA and then waits for the IOP to pass back the function return value via DMA. On receiving the return value from the IOP the function returns the same value to the caller.

With one exception, the modem API functions listed above should only be used if you are using a version of the SN TCP/IP stack with modem support. If you attempt to use any of these functions with an Ethernet only version of the SN TCP/IP stack, the error code `SN_RPCBAD` will be returned.

The exception is the function `snmdm_get_attached`, which will return information on the attached Ethernet adapter if an Ethernet version of the SN TCP/IP stack is in use.

snmdm_connect

Prototype `sn_int32 snmdm_connect( void );`

Description This function is used to initiate a modem connection to an ISP. Because this function only initiates the modem connection, it will generally return before the PPP link is up. The function `snmdm_get_state` can be used to determine when the PPP link is up. The socket API must have been initialized and the stack must have been started before calling this function. It shouldn't be necessary for an application to use this function, as the higher-level function `sntc_connect_modem` should be used.

Parameters None.

Returns *If successful:* A value of 0 is returned. *On failure:* A non-zero error code is returned, which may be one of the following:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_ESTKDOWN`	The TCP/IP Stack has not been started
`SN_REQSIZE`	Internal error (IOP RPC size mismatch)
`SN_RPCBAD`	Function not supported – Ethernet version of the Stack in use

snmdm_disconnect

Prototype `sn_int32 snmdm_disconnect( void );`

Description This function is used to terminate a modem connection to an ISP, or clean up after a failed attempt to connect to an ISP. Because this function only initiates the modem disconnection, it will generally return before the modem is ready to establish another connection. The function `snmdm_get_state` can be used to determine when the modem is ready to dial. The socket API must have been initialized and the stack must have been started before calling this function. It shouldn't be necessary for an application to use this function, as the higher-level function `sntc_disconnect_modem` should be used.

Parameters None.

Returns *If successful:* A value of 0 is returned. *On failure:* A non-zero error code is returned, which may be one of the following:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_ESTKDOWN`	The TCP/IP Stack has not been started
`SN_REQSIZE`	Internal error (IOP RPC size mismatch)
`SN_RPCBAD`	Function not supported – Ethernet version of the Stack in use

snmdm_get_attached

Prototype

```
sn_int32 snmdm_get_attached(
 sn_bool*                  attached,
 sn_int16*                 idVendor,
 sn_int16*                 idProduct);
```

Description This function is used to determine whether a compatible USB modem or Ethernet adapter is attached, and if so exactly what type of device it is. The socket API must have been initialized before calling this function.

Parameters The parameters for this function are as follows:

`attached` *Input:* Points to where this function will return the value indicating whether or not a compatible device is attached. This can be set to `NULL` if this is not required. *Output:* If a non-`NULL` pointer was supplied, the location pointed to will be set to `SN_TRUE` if a compatible device is attached, or, it will be set to `SN_FALSE` if no device, or, an incompatible device is attached.

`idVendor` *Input:* Points to where this function will return the value indicating the USB vendor ID for the device. *Output:* If a non-`NULL` pointer was supplied, and a compatible device is attached, the location pointed to will be set to the USB vendor ID of the device.

`idProduct` *Input:* Points to where this function will return the value indicating the USB product id for the device. *Output:* If a non-`NULL` pointer was supplied, and a compatible device is attached, the location pointed to will be set to the USB product ID of the device.

Returns *If successful:* A value of 0 is returned. *On failure:* A non-zero error code is returned, which may be one of the following:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_REQSIZE`	Internal error (IOP RPC size mismatch)

snmdm_get_connect_err

Prototype

```
sn_int32 snmdm_get_connect_err(sn_int32* connect_err);
```

Description

This function is used to determine why an attempt to dial a phone number and connect to it with the modem failed.
If you are using `sntc_connect_modem` to make the connection, you will not need to call this function, as it is called from within `sntc_connect_modem`.
If you are using the lower level `snmdm_connect` function to make the connection, you may wish to use this function.
This function returns information in `*connect_err` indicating whether any error string was sent by the modem such as "BUSY" in response to the modem dial command.

Parameters

The parameters for this function are as follows:

`connect_err` *Input:* Points to where this function will return the value indicating the connection error. *Output:* The location pointed to will be set to one of the possible error codes defined by the macros `SN_CONERR_...`

`...UNKNOWN`	The connection did not fail as a result of an error message from the modem.
`...SUCCESS`	The connection was successful (the modem reported "CONNECT").
`...BUSY`	The connection failed because the dialled phone number was busy (engaged).
`...NOCARRIER`	The connection failed because the remote modem did not respond.
`...NOANSWER`	The connection failed because the dialled phone was not answered.
`...NODIALTONE`	The connection failed because the local modem could not get a dial tone.

Returns

If successful, a value of zero is returned.
On failure, a non zero error code is returned, which may be one of the following:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_REQSIZE`	Internal error (IOP RPC size mismatch)
`SN_RPCBAD`	Function not supported – Ethernet version of the Stack in use

snmdm_get_state

Prototype `sn_int32 snmdm_get_state( sn_int32*  modem_state );`

Description This function is used to determine the current state of a modem connection. The socket API must have been initialized before calling this function.

Parameters The parameters for this function are as follows:

`modem_state` *Input:* Points to where this function will return the value indicating the modem state. *Output:* If successful, the location pointed to will be set to one of the possible modem states defined by the macros `SN_MODEM...`

Returns *If successful:* A value of 0 is returned and `*modem_state` contains the modem state. *On failure:* A non-zero error code is returned, which may be one of the following:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_REQSIZE`	Internal error (IOP RPC size mismatch)
`SN_EIFSTATE`	Internal error (`inet_ifstate`)
`SN_RPCBAD`	Function not supported – Ethernet version of the Stack in use

snmdm_set_mdm_init

Prototype `sn_int32 snmdm_set_mdm_init( sn_char*  modem_init_str);`

Description This function is used to store the modem initialization string that will be sent to the modem prior to any attempt to dial. The socket API must have been initialized before calling this function.

Parameters The parameters for this function are as follows:

`modem_init_str` *Input:* Null-terminated string (not a `NULL` pointer) containing the modem initialization string. *Output:* None.

Returns *If successful:* A value of 0 is returned. *On failure:* A non-zero error code is returned, which may be one of the following:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_REQSIZE`	Internal error (IOP RPC size mismatch)
`EINVAL`	One of the following conditions:
`modem_init_str` is a `NULL` pointer	
`modem_init_str` is longer than `SN_MAX_MDMINIT_LEN`	
`SN_RPCBAD`	Function not supported – Ethernet version of the Stack in use

snmdm_set_phone_no

Prototype `sn_int32 snmdm_set_phone_no( sn_char*  phone_no);`

Description This function is used to store the phone number that the modem will dial. It shouldn't be necessary for an application to use this function, as the higher level function `sntc_connect_modem` should be used which sets up the `phone_no`. The socket API must have been initialized before calling this function.

Parameters The parameters for this function are as follows:

`phone_no` *Input:* Null-terminated string (not a `NULL` pointer) containing the phone number to dial.*Output:* None.

Returns *If successful:* A value of 0 is returned. *On failure:* A non-zero error code is returned, which may be one of the following:

`SN_ENOTINIT`	The socket API has not been initialized
`SN_ETHNOTREG`	The calling thread is not registered
`SN_REQSIZE`	Internal error (IOP RPC size mismatch)

`EINVAL`	One of the following conditions:
	`phone_no` is `NULL`
	`phone_no` is longer than `SN_MAX_PHONE_LEN`

`SN_RPCBAD`	Function not supported – Ethernet version of the Stack in use

snmdm_set_script

Prototype

```
sn_int32 snmdm_set_script( const sn_char*  script_str);
```

Description

Stores one line of a login script string in the modem API. The higher level function `sntc_connect_modem` should be used unless there is a need to build a login script file in a format that isn't supported by `sntc_connect_modem`. This function is used to store one line of a login script file. The stack must have been started before calling this function.

Parameters

The parameters for this function are as follows:

`script_str`

Input: Null-terminated string containing the next line of the login script file or a zero length string may be sent to reset the file to empty. This should be done at the start of each file write.
The format of the lines of a login script file is shown in the following examples:
`"input 30 ogin:"` – Wait for up to 30 seconds for the ISP to send the login prompt, which in this case contains `"ogin:"`. The `"L"` or `"l"` has been omitted because the comparison is case sensitive.
`"output myname\\r"` – Send `"myname"` followed by a new line to the ISP. Note that output lines should be terminated with `"\\r"`.
`"wait 10"` – Wait 10 seconds before proceeding with the script.
Output: None.

Returns

If successful: a value of zero is returned. *On failure:* a non-zero error code is returned, which may be one of the following:

`SN_ENOTINIT`	The socket API has not been initialised
`SN_ETHNOTREG`	The calling thread is not registered
`SN_REQSIZE`	Internal error (IOP RPC size mismatch)
`EINVAL`	One of the following conditions:
`script_str` is `NULL`	
`Script_str` is longer than `SN_MAX_SCRIPT_LEN`	
number of lines in login script is greater than `SN_MAX_SCRIPT_LINES`	
`SN_RPCBAD`	Function not supported – Ethernet version of the Stack in use

Chapter 5: **EE utilities**

Utility modules for EE

In addition to the main socket API module (`snsocket.h`), there are two other modules used by the EE:

There is a general utility module, which is not specific to TCP/IP support, called `sneeutil.h`. This contains general purpose utilities for use in the EE. There is also a TCP/IP utility module, which contains TCP/IP specific utilities, called `sntcutil.h`.

The functions provided by these modules are described in this chapter.

General EE utilities (sneeutil.h)

The functions exported via this module are:

- `sn_delay`
- `sn_strcat`

TCP/IP-specific EE utilities (sntcutil.h)

The functions exported via this module are:

- `sntc_set_dns_server_list`
- `sntc_connect_modem`
- `sntc_disconnect_modem`
- `sntc_reset_modem`
- `sntc_str_modem_state`
- `sntc_test_gethostbyname`
- `sntc_connect_to_tcpip_server`
- `sntc_recv_nbytes`
- `sntc_send_nbytes`

Each of these functions is described in the following sections

sn_delay

Prototype

```
sn_int32 sn_delay( sn_int32      ms);
```

Description

Puts the calling thread into the sleep state for the specified number of milliseconds. Uses an alarm to wake up the thread again.

Parameters

The parameters for this function are as follows:

`ms`

Input: Delay period in milliseconds. *Output:* None.

Returns

If successful: a value of zero is returned. *On failure:* a value of –1 is returned, which can only be as a result of the EE kernel functions `SetAlarm` or `SleepThread` returning an error indication; if this happens the function will return an error message.

Note: in version 1.0 the error is `printf`'d. In later releases it is planned to return an error code indicating the reason for failure

sn_strcat

Prototype

```
sn_char* sn_strcat(
  sn_char*                dest,
  const sn_char*          src);
```

Description

Same as the standard C library `strcat`function, replacing the faulty `strcat` in version 1.5 of the Sony libraries.

Note: The intention is to remove this function when the Sony version is fixed.

Parameters

Same as `strcat`.

Returns

Same as `strcat`.

sntc_set_dns_server_list

Prototype

```
sn_int32 sntc_set_dns_server_list(
  const sn_char**          dns_servers );
```

Description

This function sets up the list of DNS servers to be used by the TCP/IP stack. If you want to use DNS (gethostbyname) then you must call this function and supply at least one DNS server.

Note: There is a lower level function sndns_add_server defined in snsocket.h, which adds an individual DNS server.

The socket API must have been initialized before calling this function. It may be called before or after the stack is started. This function should be called only once during initialization, as the DNS server list is not cleared. No checks are made for duplicate entries, and the maximum number of DNS servers that can be stored in total is defined by SN_MAX_DNS_SERVERS.

Parameters

The parameters for this function are as follows:

dns_servers — *Input:* Points to the first element of an array of null-terminated strings. Each element should contain a dotted IP address of a DNS server. The list should be terminated by an element containing a null string (i.e. a string of length zero not a NULL pointer). *Output:* None.

Returns

If successful: a value of zero (SNTC_ERR_NONE) is returned. *On failure:* one of the following error codes will be returned:

SNTC_ERR_INVAL	One (or more) of the dotted IP addresses supplied was invalid (failed to convert with inet_aton).
SNTC_ERR_IOPDNS	Problem communicating with the IOP using sndns_add_server. This includes trying to send more than SN_MAX_DNS_SERVERS in total since the interface was initialized.

sntc_connect_modem

Prototype

```
sn_int32 sntc_connect_modem(
  sn_char*                 phone_no,
  sn_int32                 isp_type,
  sn_char*                 user_name,
  sn_char*                 password,
  sn_int32                 timeout_secs,
  sntc_mdmstate_callback   callback,
  sn_char**                error_message);
```

Description

This function attempts to establish a PPP link to an ISP, using the specified `phone_no`, `isp_type`, `user_name` and `password`. The stack must have been started before calling this function.

Note: This function should not be used with a version of the SN TCP/IP stack that provides only Ethernet support.

Parameters

The parameters for this function are as follows:

`phone_no` — *Input:* Null terminated string containing the phone number to be dialled. It's length must not be greater than `SN_MAX_PHONE_LEN`. If required pauses can be inserted in the dial string using commas. *Output:* None.

`isp_type` — *Input:* Currently only two values are supported. `SNTC_ISP_GENERIC` should be used if the ISP prompts for protocol, otherwise `SNTC_ISP_GRIC` should be used. *Output:* None.

`user_name` — *Input:* Null-terminated string containing the user name to be used with the ISP account. Its length must not be greater than `SNTC_UN_PW_MAX_LEN`. *Output:* None.

`password` — *Input:* Null-terminated string containing the password to be used with the ISP account. Its length must not be greater than `SNTC_UN_PW_MAX_LEN`. *Output:* None.

`timeout_secs` — *Input:* The timeout (in seconds) that this function should wait in total from the point at which the dial command is sent to the modem to the point at which the PPP link is established. Most error conditions will cause this function to return as soon as the error is detected, and so it will not normally wait this long on a failed connection. *Output:* None.

`callback` — *Input:* If required the `callback` parameter can be set to point to a function which will be called each time this function detects a change in

the modem_state. Because some states are short-lived, it generally may not be called for every state, but it will be called for any that persist for a significant amount of time. The purpose of this callback is to provide some feedback on how the connection is progressing. If this isn't required, set callback to NULL. *Output:* None.

callback function spec

The prototype for the callback function should be like this (any name can be used for the function):

```
void sntc_callback(
  sn_int32                  modem_state);
```

The modem_state parameter will have one of the values defined by the macros SN_MODEM_ ...

error_message

Input: If required the error_message parameter can be set to point to a location where this function will return a pointer to a null-terminated string describing any error that was detected (or "Success" if it worked OK). If this isn't required set error_message to NULL. *Output:* If error_message isn't NULL, the location pointed to by error_message will contain a pointer to a null-terminated string describing any error encountered by this function (or "Success" if it worked OK).

Returns

If successful: a value of zero (SNTC_ERR_NONE) is returned. *On failure:* one of the following error codes will be returned:

SNTC_ERR_INVAL	Invalid parameter passed to function
SNTC_ERR_MDMAPI	A lower level modem API function failed - usually because Ethernet version of the Stack is in use
SNTC_ERR_NOMODEM	A compatible modem is not attached
SNTC_ERR_BSCRIPT	Error building log in script
SNTC_ERR_CONNECT	Modem went to bad state during connect
SNTC_ERR_BUSY	The dialled phone number was busy (engaged).
SNTC_ERR_NOCARRIER	The remote modem did not respond.
SNTC_ERR_NOANSWER	The dialled phone number was not answered.
SNTC_ERR_NODIALTONE	The local modem could not get a dial tone.
SNTC_ERR_TIMEOUT	Timed out before PPP link came up

sntc_disconnect_modem

Prototype

```
sn_int32 sntc_disconnect_modem(
  sn_int32                  timeout_secs,
  sntc_mdmstate_callback    callback,
  sn_char**                 error_message);
```

Description

Disconnects the modem from the ISP. This function should be called following a call to sntc_connect_modem, either to terminate a call after a connection or to get the modem back to the ready to dial state after an unsucessful attempt to connect. The stack must have been started before calling this function.

Note: This function should not be used with a version of the SN TCP/IP stack that provides only Ethernet support.

Parameters

The parameters for this function are as follows:

`timeout_secs`

Input: The timeout (in seconds) that this function should wait in total from the point at which the function is called, to the point at which the modem goes to a state where it is ready to accept a dial command (ready for another call to sntc_connect_modem). Most error conditions will cause this function to return as soon as the error is detected, and so it will not normally wait this long on a faulty modem. *Output:* None.

`callback`

Input: If required the `callback` parameter can be set to point to a function which will be called each time this function detects a change in the `modem_state`. Because some states are short-lived, it generally may not be called for every state, but it will be called for any that persist for a significant amount of time. The purpose of this callback is to provide some feedback on how the connection is progressing. If this isn't required, set `callback` to `NULL`. *Output:* None.

`callback function spec`

The prototype for the `callback` function should be like this (any name can be used for the function):

```
void sntc_callback(
  sn_int32                  modem_state);
```

The `modem_state` parameter will have one of the values defined by the macros `SN_MODEM_ ...`

`error_message`

Input: If required the `error_message` parameter can be set to point to a location where this function will return a pointer to a null-terminated string describing any error that was detected (or "`Success`" if it worked OK). If this isn't required set `error_message` to `NULL`. *Output:* If

error_message isn't NULL, the location pointed to by error_message will contain a pointer to a null-terminated string describing any error encountered by this function (or "Success" if it worked OK).

Returns

If successful: a value of zero (SNTC_ERR_NONE) is returned. *On failure:* one of the following error codes will be returned:

SNTC_ERR_INVAL	Invalid parameter passed to function
SNTC_ERR_MDMAPI	A lower level modem API function failed – usually because Ethernet version of the Stack is in use
SNTC_ERR_NOMODEM	A compatible modem is not attached
SNTC_ERR_TIMEOUT	Timed out before modem became ready to accept a dial command

sntc_reset_modem

Prototype

```
sn_int32 sntc_reset_modem(
  sn_int32                  timeout_secs,
  sntc_mdmstate_callback    callback,
  sn_char**                 error_message);
```

Description

Resets the modem. This function attempts to get the modem into the state where it is ready to dial. This function would normally be called only if sntc_disconnect_modem returned a fail code. The stack must have been started before calling this function.

Note: This function should not be used with a version of the SN TCP/IP stack that provides only Ethernet support.

Parameters

The parameters for this function are as follows:

`timeout_secs`

Input: The timeout (in seconds) that this function should wait in total from the point at which the function is called, to the point at which the modem goes to a state where it is ready to accept a dial command (ready for another call to sntc_connect_modem). *Output:* None.

`callback`

Input: If required the `callback` parameter can be set to point to a function which will be called each time this function detects a change in the `modem_state`. Because some states are short-lived, it generally may not be called for every state, but it will be called for any that persist for a significant amount of time. The purpose of this callback is to provide some feedback on how the connection is progressing. If this isn't required, set `callback` to `NULL`. *Output:* None.

`callback function spec`

The prototype for the `callback` function should be like this (any name can be used for the function):

```
void sntc_callback(
  sn_int32                  modem_state);
```

The `modem_state` parameter will have one of the values defined by the macros `SN_MODEM_ ...`

`error_message`

Input: If required the `error_message` parameter can be set to point to a location where this function will return a pointer to a null-terminated string describing any error that was detected (or "`Success`" if it worked OK). If this isn't required set `error_message` to `NULL`. *Output:* If `error_message` isn't `NULL`, the location pointed to by `error_message` will contain a pointer to a null-terminated string describing any error encountered by this function (or "`Success`" if it worked OK).

Returns

If successful: a value of zero (`SNTC_ERR_NONE`) is returned. *On failure:* one of the following error codes will be returned:

`SNTC_ERR_INVAL`	Invalid parameter passed to function
`SNTC_ERR_MDMAPI`	A lower level modem API function failed – usually because Ethernet version of the Stack is in use
`SNTC_ERR_NOMODEM`	A compatible modem is not attached
`SNTC_ERR_TIMEOUT`	Timed out before modem became ready to accept a dial command

sntc_str_modem_state

Prototype

```
sn_char* sntc_str_modem_state( sn_int32 modem_state);
```

Description

Converts modem state into a string description of the modem state. This function returns a pointer to a null-terminated string containing a description of the `modem_state`. This function can be called at any time, there is no need for the interface to have been initialized and it doesn't matter whether the stack has been started or not.

Note: This function should not be used with a version of the SN TCP/IP stack that provides only Ethernet support.

Parameters

The parameters for this function are as follows:

`modem_state`

Input: The `modem_state` parameter passed to this function should be a value that was returned by the function `snmdm_get_state`, or the value passed as the `modem_state` parameter to the callback function for one of the functions `sntc_connect_modem`, `sntc_disconnect_modem`, or `sntc_reset_modem`. *Output:* None.

Returns

A pointer to a string containing a description of the `modem_state`; this string may be from 4 to 31 characters long (in the current software release but may change in future releases). If an invalid `modem_state` is passed to this function, it will return a pointer to the string: "`Invalid`".

sntc_test_gethostbyname

Prototype

```
void sntc_test_gethostbyname( sn_char*  name );
```

Description

Simple test utility to demonstrate that DNS is working. This function calls `gethostbyname` for the supplied name, and `printfs` the IP address corresponding to the name, or an error message if the name cannot be resolved to an IP address. Before calling this function, the modem must be connected (PPP link up) and at least one DNS server must have been specified.
Example: to print the IP address of the SN web server:

```
sntc_test_gethostbyname("www.snsys.com");
```

to print the IP address of the PS2:

```
sntc_test_gethostbyname(LOCAL_NAME);
```

Parameters

The parameters for this function are as follows:

`name` — *Input:* A null-terminated string containing the name which is to be converted to an IP address. *Output:* None.

Returns

None

sntc_connect_to_tcpip_server

Prototype

```
sn_int32 sntc_connect_to_tcip_server(
  sn_char*          server_addr,
  sn_uint16         server_port,
  sn_int32*         sock_ptr,
  sn_int32*         connection_state);
```

Description

This function creates a TCP/IP client socket and attempts to connect it to the TCP/IP server specified by `server_addr, server_port`.

Parameters

The parameters for this function are as follows:

`server_addr` — *Input:* Points to a null-terminated string, either containing the name of the server or the dotted IP address of the server. *Output:* None.

`server_port` — *Input:* The server port number in host byte order. *Output:* None.

`sock_ptr` — *Input:* Points to a location where this function will return a socket descriptor. *Output:* The loction pointed to by `sock_ptr` will contain the client socket descriptor. If this function is not successful, this socket descriptor may be invalid and will have been closed.

`connection_state` — *Input:* Points to a location where this function will return the connection state. *Output:* The loction pointed to by `connection_state` will contain a value indicating how far the connection process got to, which will be `SNTC_STATE_CONNECTED` for a successful connection.
The other values which will only be returned for an error condition are covered under the "Returns" section.

Returns

Normally this function will return zero indicating success, in which case `*sock_ptr` will contain the socket descriptor for the client socket and `*connection_state` will contain a value of `SNTC_STATE_CONNECTED`.
On error this function will return non zero, the meaning of the value returned by this function will depend on how far the connection got before encountering the error, as defined by the value returned in `*connection_state` as defined below:
`SNTC_STATE_CREATE_SOCK` - the call to `socket` failed, the return value of this function is whatever value `socket` returned.
`SNTC_STATE_BIND` – the call to `bind` failed, the return value of this function is whatever `sn_errno` returned for the socket following the call to `bind`.
`SNTC_STATE_CONV_NAME` – `server_addr` was interpreted as a name

and the call to gethostbyname failed, the return value of this function is whatever sn_h_errno returned following the call to gethostbyname.

SNTC_STATE_CONV_DOTTED – server_addr was interpreted as a dotted IP address and the call to inet_aton failed. A value of -1 is returned by this function.

SNTC_STATE_CONNECTING – the call to connect failed, the return value of this function is whatever sn_errno returned for the socket following the call to connect.

sntc_recv_nbytes

Prototype

```
sn_int32 sntc_recv_nbytes(
  sn_int32          s,
  sn_char*          buf,
  sn_int32          len);
```

Description

This function provides a wrapper for the recv function that will call recv as many times as necessary to receive the number of bytes specified by len. The recv flags parameter will be set to zero. This function should not be called for non blocking sockets.

Parameters

The parameters for this function are as follows:

s	*Input:* Socket descriptor identifying the socket that data should be received from. *Output:* None.
buf	*Input:*Points to the buffer where received data will be returned. *Output:* *buf contains the received data.
len	*Input:* The number of bytes to be received. *Output:* None.

Returns

If successful, the number of bytes of data stored in *buf is returned (which will be equal to the len parameter), and *buf contains the received data.
If the socket has been closed by the sender 0 is returned.
On failure, the error status -1 is returned. sn_errno can be used to obtain the reason for failure. See recv for possible reasons for failure.

sntc_send_nbytes

Prototype

```
sn_int32 sntc_send_nbytes(
  sn_int32          s,
  sn_void*          buf,
  sn_int32          len);
```

Description

This function provides a wrapper for the send function that will call send as many times as necessary to send the number of bytes specified by len. The send flags parameter will be set to zero. This function should not be called for non blocking sockets.

Parameters

The parameters for this function are as follows:

s	*Input:* Socket descriptor identifying the socket that data should be sent on. *Output:* None.
buf	*Input:*Points to the buffer containing the data to be sent. *Output:* None.
len	*Input:* The amount of data contained in *buf in bytes. *Output:* None.

Returns

If successful, the number of bytes of data that were sent is returned (which will be equal to the len parameter).
On failure, the error status -1 is returned. sn_errno can be used to obtain the reason for failure. See send for possible reasons for failure.

Glossary of Terms

Blocking / non-blocking

When operations are performed using a socket, the caller requesting the action is usually forced to wait until the operation completes; the socket is said to be in *blocking mode*. An alternative mode of operation is available in which the socket action is allowed to proceed without blocking the caller. In this case the socket is said to be in *non-blocking mode*.

BSD

Univerity of California's Berkeley Software Distribution.

DNS

Domain Name System. The distributed database used to map machine names to IP addresses.

IP

Internet Protocol. The protocol used to handle the delivery of IP datagrams across the internet.

IP address

A 32-bit address used to identify a host computer on a network. Can also be expressed as "dotted IP" addresses, e.g. "123.4.321.8".

PPP

Point-to-point protocol, A network protocol used to control the delivery of IP datagrams between two hosts connected by a serial link.

Semaphore

A data structure used to provide an event-signalling mechanism or mutual exclusion in a real-time operating system.

TCP

Transport Control Protocol. The protocol used to provide reliable, full-duplex delivery of data streams across a logical connection established between two end points.

Index

N

R

S

T

U