Chapter 3

Programmer's Manual

This chapter provides an in depth description of the AmiTCP/IP application programming interface. Following sections introduce the socket model of communication (3.1) and the bsdsocket.library function calls implementing the socket abstraction. Some useful supporting routines are described in section 3.2. The client/server model is introduced in section 3.3. Some more advanced topics are discussed in section 3.4. Section 3.5 summarizes the small differences between AmiTCP/IP and 4.3BSD socket APIs. The full function reference of the AmiTCP/IP API functions is in appendix B starting from page 148.

The text in sections 3.1 – 3.4 is based on the [Leffler et al 1991a].

3.1 Socket Concepts

The basic building block for communication is the *socket*. A socket is an endpoint of communication to which a name may be *bound*. Each socket in use has a *type* and one or more associated processes. Sockets exist within *communication domains*. A communication domain is an abstraction introduced to bundle common properties of processes communicating through sockets. One such property is the scheme used to name sockets. Sockets normally exchange data only with sockets in the same domain[1]. The AmiTCP/IP system supports currently only one communication domain: the Internet domain, which is used by processes which communicate using the the DARPA standard communication protocols. The underlying communication facilities provided by the domains have a significant influence on the internal system implementation as well as the interface to socket facilities available to a user.

[1] It may be possible to cross domain boundaries, but only if some translation process is performed.

3.1.1 Socket Types

Sockets are typed according to the communication properties visible to a user. Processes are presumed to communicate only between sockets of the same type, although there is nothing that prevents communication between sockets of different types should the underlying communication protocols support this.

Three types of sockets currently are available to a user. A *stream* socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bidirectionality of data flow, a pair of connected stream sockets provides an interface nearly identical to that of pipes.[2]

A *datagram* socket supports bidirectional flow of data which is not promised to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which it was sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in many contemporary packet switched networks such as the Ethernet.

A *raw* socket provides users access to the underlying communication protocols which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more esoteric facilities of an existing protocol. The use of raw sockets is considered in section 3.4.

Another potential socket type which has interesting properties is the *reliably delivered message* socket. The reliably delivered message socket has similar properties to a datagram socket, but with reliable delivery. There is currently no support for this type of socket, but a reliably delivered message protocol similar to Xerox's Packet Exchange Protocol (PEX) may be simulated at the user level. More information on this topic can be found in section 3.4.

3.1.2 Using The Socket Library

As any other Amiga shared library the bsdsocket.library must be opened to be able to access the functions in the library. This can be done with Exec's `OpenLibrary()` call. The call returns a library base pointer which is *task specfic*, which means that each separate task (or process) must open the library itself. This is because the AmiTCP/IP stores task specific information to the library base structure.

The library base pointer returned by the `OpenLibrary()` must be stored in to a variable accessable from the program (usually global) named `SocketBase`. Example of opening the

[2] In the UNIX systems pipes have been implemented internally as simply a pair of connected stream sockets.

library follows:

```
#include <exec/libraries.h>
 ...
struct Library *SocketBase = NULL;
 ...
    if ((SocketBase = OpenLibrary("bsdsocket.library", 2)) == NULL) {
        /* could not open the library */
         ...
    }
    else {
        /* SocketBase now points to socket base of this task */
         ...
    }
```

Note that the library version argument of the `OpenLibrary()` call is given as `2`, which means that at least version 2 is needed. *This is the minimum version which should be requested, since the version 1 is incompatible with the version 2 and up.* If the application uses features defined for some specific version (and up), a later version number should be specified.

After the application is done with sockets the library must be closed. This is done with `CloseLibrary()` as follows:

```
    if (SocketBase) {
        CloseLibrary(SocketBase);
        SocketBase = NULL;
    }
```

Note that if the application in question is multithreaded, each task (or process) need to open/close its own library base. The base opened by the **net.lib** may be used by the original task only!

Many programs expect the error values of the socket calls to be placed in a global variable named `errno`. By default a shared library cannot know the address (nor size) of the applications variables, however. There are two remedies to this:

1. Use function `Errno()` to retrieve the error value, or

2. Tell the address and the size of the `errno` variable to the AmiTCP/IP by using the `SetErrnoPtr()` call.

The latter method requires only one additional function call to the startup of the application, and is thus the preferred method. The call may look like:

```
#include <errno.h>
#include <sys/socket.h>
 ...
    SetErrnoPtr(&errno, sizeof(errno));
```

All this is done automatically for the application if it is linked with the **net.lib**[3]. See section 3.1.3 for more information about the **net.lib** and about compiling and linking the applications.

3.1.3 Compiling and Linking The Applications

AmiTCP/IP provides standard BSD Unix header files to be used by the applications. Normally they are installed to a directory which is assigned to a name **NETINCLUDE:** (see section 1.1). This means that you should add the **NETINCLUDE:** to the compilers search path for include files.

The include files are decribed briefly in the following subsection:

The NETINCLUDE Header Files

bsdsocket.h This file includes compiler specific prototypes and inline functions for **bsdsocket.library**. Currently supported compilers are GCC and SAS C version 6. The prototypes for the library functions are automatically included by the include files when appropriate, i.e. when the prototypes where declared in the original BSD includes. Thus the **bsdsocket.h** is included by **sys/socket.h**, **netdb.h** and **arpa/inet.h**.

For other compilers only C prototypes are included, so stub routines should be used to call the functions.

errno.h Replacement for the **errno.h** included in the standard C-compiler headers. This includes the file **sys/errno.h**, which defines symbolic constants for the error values returned by socket library calls. This file is BSD compatible and may well replace file provided by the SAS/C 6.

netdb.h Contains definitions and prototypes for the network database functions, such as the `gethostbyname()`.

[3]The **net.lib** is compiler dependent and is currently defined for SASC 6 only. The actual name of the library varies and depends on the compiler options used.

Standard BSD System Headers

sys/errno.h Error code definitions for system functions.

sys/ioctl.h Definitions for socket IO control.

sys/param.h General machine independent parameter definitions.

sys/socket.h Definitions related to sockets: types, address families, options and prototypes.

sys/syslog.h Definations for system logging facilities.

sys/time.h Definition of structure timeval.

sys/types.h Common C type definitions and file descriptor set macros for `select()`.

Internet Related Headers

arpa/inet.h Inet library function prototypes (`inet_addr()` etc.). Included for compatibility and only includes other include files.

netinet/in.h Protocol numbers, port conventions, inet address definitions.

netinet/in_systm.h Some network byte order type definitions.

netinet/ip.h IP packet header, packet options, timestamp.

netinet/ip_icmp.h ICMP packet structure.

netinet/ip_var.h Defines IP statistics, external IP packet header, reassemble queues structures.

netinet/tcp.h Defines the TCP packet structure.

netinet/udp.h Defines the UDP packet structure.

Network Related Headers

net/if.h Defines the interface for network adapter drivers.

net/if_arp.h General protocol independent ARP structures.

net/route.h Routing ioctl definitions.

net/sana2errno.h Sana-II related error definitions.

net/sana2tags.h Tag definitions for configuring the Sana-II software network interface.

Inetd Support

inetd.h Internet daemon interface definitions.

inetdlib.h Internet daemon library definitions.

Prototypes

clib/socket_inlines.h Inline function definitions for those BSD socket API functions, which are not implemented strictly like originals by bsdsocket.library.

clib/socket_protos.h bsdsocket.library function call prototypes.

SAS/C Pragmas

pragmas/socket_pragmas.h SAS/C pragma library calls for bsdsocket.library.

SAS/C Proto -file

proto/socket.h Include file normally included by the SAS/C programs. Defines the socket base variable and includes the files **clib/socket_protos.h** and **pragmas/socket_pragmas.h**.

GCC Inline Functions

inline/socket.h GCC inline functions for the bsdsocket.library functions.

Function Description File

fd/socket_lib.fd Standard fd-file which specifies in which registers the arguments to the bsdsocket.library functions are passed. This file can be used to obtain information needed to call the bsdsocket.library functions by the assembler programs.

Sana-II Header Files

devices/sana2.h Definitions for the Sana-II network device driver interface.

devices/sana2specialstats.h Special statistics definitions for the Sana-II.

Miscellaneous

charread.h Macro package to do buffered byte-by-byte reading from a socket.

lineread.h Definitions for buffered line oriented reading from a socket.

Linking With net.lib

AmiTCP/IP distribution includes a link library named **net.lib** to be used by the applications. It is normally located in the directory which has an assigned name **NETLIB:**.

The library contains compiler dependent code which makes the library itself compiler dependent. Currently only SASC version 6 is supported[4].

net.lib features automatic initialization and termination functions which open and close the bsdsocket.library for the application. Using this feature it is possible to compile some typical BSD Unix socket based applications with AmiTCP/IP without any modifications to the original source code. Note that this base may be used by the process starting the program, i.e. the one that executes the `main()`. This applies to the included utility functions which call the socket library, too.

The library also defines new array of error names to be used by `perror()` library function. This is done because the error name array normally used by Amiga C compilers does not contain enough error entries, resulting `perror()` to print `"Unknown error code"` if some socket error is passed. Note that for `perror()` to work the error value must be placed into the global `errno` variable. This is accomplished by the `SetErrnoPrt()` call made in the automatic initialization function.

For the library functions to take effect, the library must be specified *before* the C compiler own libraries in the link command line.

3.1.4 Socket Creation

To create a socket the `socket()` system call is used:

```
s = socket(domain, type, protocol);
```

This call requests that the system create a socket in the specified `domain` and of the specified `type`. A particular `protocol` may also be requested. If the protocol is left unspecified (a value of `0`), the system will select an appropriate protocol from those protocols which comprise the communication domain and which may be used to support the requested socket type. The user is returned a descriptor (a small integer number) which may be used in later system calls which operate on sockets. The domain is specified as one of the manifest constants defined in the file **sys/socket.h**. For the Internet domain the constant is `AF_INET`[5]. The socket types are also defined in this file and one of `SOCK_STREAM`, `SOCK_DGRAM` or `SOCK_RAW` must be specified. To create a stream socket in the Internet domain the following call might be used:

[4] But since the source for the library is provided, it *can* be used with any C compiler.

[5] The manifest constants are named `AF_whatever` as they indicate the "address format" to use in interpreting names.

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

This call would result in a stream socket being created with the TCP protocol providing the underlying communication support. To create a datagram socket the call might be:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

The default protocol (used when the `protocol` argument to the `socket()` call is `0`) should be correct for most every situation. However, it is possible to specify a protocol other than the default; this will be covered in section 3.4.

There are several reasons a `socket()` call may fail. Aside from the rare occurrence of lack of memory (`ENOBUFS`), a socket request may fail due to a request for an unknown protocol (`EPROTONOSUPPORT`), or a request for a type of socket for which there is no supporting protocol (`EPROTOTYPE`).

3.1.5 Binding Local Names

A socket is created without a name. Until a name is bound to a socket, processes have no way to reference it and, consequently, no messages may be received on it. Communicating processes are bound by an *association*. In the Internet domain, an association is composed of local and foreign *addresses*, and local and foreign *ports*, In most domains, associations must be unique. In the Internet domain there may never be duplicate <protocol, local address, local port, foreign address, foreign port> tuples.

The `bind()` system call allows a process to specify half of an association, <local address, local port>, while the `connect()` and `accept()` calls are used to complete a socket's association.

In the Internet domain, binding names to sockets can be fairly complex. Fortunately, it is usually not necessary to specifically bind an address and port number to a socket, because the `connect()` and `send()` calls will automatically bind an appropriate address if they are used with an unbound socket.

The `bind()` system call is used as follows:

```
bind(s, name, namelen);
```

The bound name is a variable length byte string which is interpreted by the supporting protocol(s). Its interpretation may vary from communication domain to communication domain (this is one of the properties which comprise the domain). As mentioned, in the Internet domain names contain an Internet address and port number.

In binding an Internet address things are a little complicated:

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

The selection of what to place in the address `sin` requires some discussion. We will come back to the problem of formulating Internet addresses in section 3.2 when the library routines used in name resolution are discussed.

3.1.6 Connection Establishment

Connection establishment is asymmetric, with one process a "client" and the other a "server". The server, when willing to offer its advertised services, binds a socket to a well–known address associated with the service and then passively "listens" on its socket. It is then possible for an unrelated process to rendezvous with the server. The client requests services from the server by initiating a "connection" to the server's socket. On the client side the `connect()` call is used to initiate a connection. Using the Internet domain, this might apper as:

```
struct sockaddr_in server;
 ...
connect(s, (struct sockaddr *)&server, sizeof (server));
```

where `server` in the example above would contain Internet address and port number of the server to which the client process wishes to speak. If the client process's socket is unbound at the time of the connect call, the system will automatically select and bind a name to the socket if necessary. This is the usual way that local addresses are bound to a socket.

An error is returned if the connection was unsuccessful (any name automatically bound by the system, however, remains). Otherwise, the socket is associated with the server and data transfer may begin. Some of the more common errors returned when a connection attempt fails are:

`ETIMEDOUT` After failing to establish a connection for a period of time, the system decided there was no point in retrying the connection attempt any more. This usually occurs because the destination host is down, or because problems in the network resulted in transmissions being lost.

ECONNREFUSED The host refused service for some reason. This is usually due to a server process not being present at the requested name.

ENETDOWN or EHOSTDOWN These operational errors are returned based on status information delivered to the client host by the underlying communication services.

ENETUNREACH or EHOSTUNREACH These operational errors can occur either because the network or host is unknown (no route to the network or host is present), or because of status information returned by intermediate gateways or switching nodes. Many times the status returned is not sufficient to distinguish a network being down from a host being down, in which case the system indicates the entire network is unreachable.

For the server to receive a client's connection it must perform two steps after binding its socket. The first is to indicate a willingness to listen for incoming connection requests:

```
listen(s, 5);
```

The second parameter to the `listen()` call specifies the maximum number of outstanding connections which may be queued awaiting acceptance by the server process; this number may be limited by the system. Should a connection be requested while the queue is full, the connection will not be refused, but rather the individual messages which comprise the request will be ignored. This gives a harried server time to make room in its pending connection queue while the client retries the connection request. Had the connection been returned with the ECONNREFUSED error, the client would be unable to tell if the server was up or not. As it is now it is still possible to get the ETIMEDOUT error back, though this is unlikely. The backlog figure supplied with the listen call is currently limited by the system to a maximum of 5 pending connections on any one queue. This avoids the problem of processes hogging system resources by setting an infinite backlog, then ignoring all connection requests.

With a socket marked as listening, a server may accept a connection:

```
struct sockaddr_in from;
 ...
fromlen = sizeof (from);
newsock = accept(s, (struct sockaddr *)&from, &fromlen);
```

A new descriptor is returned on receipt of a connection (along with a new socket). If the server wishes to find out who its client is, it may supply a buffer for the client socket's name. The value–result parameter `fromlen` is initialized by the server to indicate how much space is associated with `from`, then modified on return to reflect the true size of the name. If the client's name is not of interest, the second parameter may be a NULL pointer.

`accept()` normally blocks. That is, `accept()` will not return until a connection is available or the system call is interrupted by a signal[6] to the process. Further, there is no way for a process to indicate it will accept connections from only a specific individual, or individuals. It is up to the user process to consider who the connection is from and close down the connection if it does not wish to speak to the process. If the server process wants to accept connections on more than one socket, or wants to avoid blocking on the accept call, there are alternatives; they will be considered in section 3.4.

3.1.7 Data Transfer

With a connection established, data may begin to flow. To send and receive data there are a number of possible calls. With the peer entity at each end of a connection anchored, a user can send or receive a message without specifying the peer. The calls `send()` and `recv()` may be used:

```
send(s, buf, sizeof (buf), flags);
recv(s, buf, sizeof (buf), flags);
```

While `send()` and `recv()` are virtually identical to the standard I/O routines, the extra `flags` argument is important. The flags, defined in **sys/socket.h**, may be specified as a non–zero value if one or more of the following is required:

`MSG_OOB` Send/receive out of band data.

`MSG_PEEK` Look at data without reading.

`MSG_DONTROUTE` Send data without routing packets.

Out of band data is a notion specific to stream sockets, and one which we will not immediately consider. The option to have data sent without routing applied to the outgoing packets is currently used only by the routing table management process, and is unlikely to be of interest to the casual user. The ability to preview data is, however, of interest. When `MSG_PEEK` is specified with a `recv()` call, any data present is returned to the user, but treated as still "unread". That is, the next `recv()` call applied to the socket will return the data previously previewed.

[6] By default, the `CTRL-C` signal interrupts the system calls, but the application may change this, however.

3.1.8 Discarding Sockets

Once a socket is no longer of interest, it may be discarded by applying a `CloseSocket()` to the descriptor,

```
CloseSocket(s);
```

If data is associated with a socket which promises reliable delivery (e.g. a stream socket) when a close takes place, the system will continue to attempt to transfer the data. However, after a fairly long period of time, if the data is still undelivered, it will be discarded. Should a user have no use for any pending data, it may perform a `shutdown()` on the socket prior to closing it. This call is of the form:

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

where `how` is `0` if the user is no longer interested in reading data, `1` if no more data will be sent, or `2` if no data is to be sent or received.

3.1.9 Connectionless Sockets

To this point we have been concerned mostly with sockets which follow a connection oriented model. However, there is also support for connectionless interactions typical of the datagram facilities found in contemporary packet switched networks. A datagram socket provides a symmetric interface to data exchange. While processes are still likely to be client and server, there is no requirement for connection establishment. Instead, each message includes the destination address.

Datagram sockets are created as before. If a particular local address is needed, the bind operation must precede the first data transmission. Otherwise, the system will set the local address and/or port when data is first sent. To send data, the `sendto()` call is used,

```
sendto(s, buf, buflen, flags, (struct sockaddr *)&to, tolen);
```

The `s`, `buf`, `buflen`, and `flags` parameters are used as before. The `to` and `tolen` values are used to indicate the address of the intended recipient of the message. When using an unreliable datagram interface, it is unlikely that any errors will be reported to the sender. When information is present locally to recognize a message that can not be delivered (for instance when a network is unreachable), the call will return `-1` and the global value `errno` will contain an error number (See section 3.1.2 for discussion about `errno`).

To receive messages on an unconnected datagram socket, the `recvfrom()` call is provided:

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *)&from, &fromlen);
```

Once again, the `fromlen` parameter is handled in a value–result fashion, initially containing the size of the `from` buffer, and modified on return to indicate the actual size of the address from which the datagram was received.

In addition to the two calls mentioned above, datagram sockets may also use the `connect()` call to associate a socket with a specific destination address. In this case, any data sent on the socket will automatically be addressed to the connected peer, and only data received from that peer will be delivered to the user. Only one connected address is permitted for each socket at one time; a second `connect()` will change the destination address, and a `connect()` to a *null* address (family `AF_UNSPEC`) will disconnect. Connect requests on datagram sockets return immediately, as this simply results in the system recording the peer's address (as compared to a stream socket, where a connect request initiates establishment of an end to end connection). `accept()` and `listen()` are not used with datagram sockets.

While a datagram socket is connected, errors from recent `send()` calls may be returned asynchronously. These errors may be reported on subsequent operations on the socket, or a special socket option used with `getsockopt()`, `SO_ERROR`, may be used to interrogate the error status. A `select()` for reading or writing will return true when an error indication has been received. The next operation will return the error, and the error status is cleared. Other of the less important details of datagram sockets are described in section 3.4.

3.1.10 Input/Output Multiplexing

One last facility often used in developing applications is the ability to multiplex i/o requests among multiple sockets. This is done using the `select()` call. The `select()` call provided by AmiTCP/IP is actually a compile time inline function (or normal stub with compilers without inline facility) which calls the `WaitSelect()`. The `WaitSelect()` call is similar to the normal `select()` call, but has one extra argument specifying a pointer to a signal mask for the signals which should break the selection (in addition to the timeouts and the break signal). This makes possible to use `WaitSelect()` instead of normal `Wait()` as a driver for the applications event loop. If the pointer is given as `NULL` the functionality is as with BSD `select()`. The inline (or stub) function for `select()` actually just calls the `WaitSelect()` with last argument as `NULL`.

Here is a brief example of the usage of the `WaitSelect()`:

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

fd_set readmask, writemask, exceptmask;
```

```
struct timeval timeout;
ULONG signalmask;
 ...
WaitSelect(nfds, &readmask, &writemask, &exceptmask, &timeout,
           &signalmask);
```

`WaitSelect()` takes as arguments pointers to three sets, one for the set of file descriptors for which the caller wishes to be able to read data on, one for those descriptors to which data is to be written, and one for which exceptional conditions are pending; out-of-band data is the only exceptional condition currently implemented. If the user is not interested in certain conditions (i.e., read, write, or exceptions), the corresponding argument to the `select()` should be a `NULL` pointer.

Each set is actually a structure containing an array of long integer bit masks; the size of the array is set by the definition `FD_SETSIZE`. The array is long enough to hold one bit for each of `FD_SETSIZE` file descriptors.

The macros `FD_SET(fd, &mask)` and `FD_CLR(fd, &mask)` have been provided for adding and removing file descriptor `fd` in the set `mask`. The set should be zeroed before use, and the macro `FD_ZERO(&mask)` has been provided to clear the set `mask`. The parameter `nfds` in the `select()` call specifies the range of file descriptors (i.e. one plus the value of the largest descriptor) to be examined in a set.

A timeout value may be specified if the selection is not to last more than a predetermined period of time. If the fields in `timeout` are set to `0`, the selection takes the form of a *poll*, returning immediately. If the last parameter is a `NULL` pointer, the selection will block indefinitely[7].

The last argument is a pointer to the mask specifying signals for which the `WaitSelect()` should break. `WaitSelect()` normally returns the number of file descriptors selected; if the `WaitSelect()` call returns due to the timeout expiring, then the value `0` is returned. If the `WaitSelect()` terminates because of an error or interruption, a `-1` is returned with the error number in `errno`, and with the file descriptor masks unchanged. The signal mask is altered on return to hold the bits for the signals which caused the break.

Assuming a successful return, the three sets will indicate which file descriptors are ready to be read from, written to, or have exceptional conditions pending. The status of a file descriptor in a select mask may be tested with the `FD_ISSET(fd, &mask)` macro, which returns a non-zero value if `fd` is a member of the set `mask`, and `0` if it is not.

To determine if there are connections waiting on a socket to be used with an `accept()` call, `select()` can be used, followed by a `FD_ISSET(fd, &mask)` macro to check for read

[7] To be more specific, a return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the system call.

readiness on the appropriate socket. If `FD_ISSET()` returns a non-zero value, indicating permission to read, then a connection is pending on the socket.

As an example, to read data from two sockets, `s1` and `s2` as it is available from each and with a one–second timeout, the following code might be used:

```
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
 ...
fd_set read_template;
struct timeval wait;
int nb;
int s1,s2;
int maxfd;
 ...
maxfd = s1 > s2 ? s1 : s2;
for (;;) {
    wait.tv_sec = 1;        /* one second */
    wait.tv_usec = 0;

    FD_ZERO(&read_template);

    FD_SET(s1, &read_template);
    FD_SET(s2, &read_template);

    nb = select(maxfd, &read_template, NULL, NULL, &wait);
    if (nb <= 0) {
        /* An error occurred during the select, or
           the select timed out. */
    }

    if (FD_ISSET(s1, &read_template)) {
        /* Socket #1 is ready to be read from. */
    }

    if (FD_ISSET(s2, &read_template)) {
        /* Socket #2 is ready to be read from. */
    }
}
```

Note the usage of the `select()`, which calls `WaitSelect()` with `NULL` signal mask pointer.

In 4.2BSD, the arguments to `select()` were pointers to integers instead of pointers to `fd_set`s. This type of call will still work as long as the number of file descriptors being examined is less than the number of bits in an integer; however, the methods illustrated above should be used in all current programs.

`select()` provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of the `SigIO` and `SigURG` signals described in section 3.4.

3.2 Network Library Routines

The discussion in section 3.1 indicated the possible need to locate and construct network addresses when using the interprocess communication facilities in a distributed environment. To aid in this task a number of routines have been added to the Amiga shared socket library. In this section we will consider the routines provided to manipulate network addresses.

Locating a service on a remote host requires many levels of mapping before client and server may communicate. A service is assigned a name which is intended for human consumption; e.g. "the *login* server on host monet". This name, and the name of the peer host, must then be translated into network *addresses* which are not necessarily suitable for human consumption. Finally, the address must then used in locating a physical *location* and *route* to the service. The specifics of these three mappings are likely to vary between network architectures. For instance, it is desirable for a network to not require hosts to be named in such a way that their physical location is known by the client host. Instead, underlying services in the network may discover the actual location of the host at the time a client host wishes to communicate. This ability to have hosts named in a location independent manner may induce overhead in connection establishment, as a discovery process must take place, but allows a host to be physically mobile without requiring it to notify its clientele of its current location.

Standard routines are provided for: mapping host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers and the appropriate protocol to use in communicating with the server process. The file **netdb.h** must be included when using any of these routines.

3.2.1 Host Names

An Internet host name to address mapping is represented by the *hostent* structure:

```
struct  hostent {
    char    *h_name;        /* official name of host */
    char    **h_aliases;    /* alias list */
```

```
    int     h_addrtype;     /* host address type (e.g., AF_INET) */
    int     h_length;       /* length of address */
    char    **h_addr_list;  /* list of addresses, null terminated */
};

#define h_addr  h_addr_list[0]  /* first address, network byte order */
```

The routine `gethostbyname()` takes an Internet host name and returns a hostent structure, while the routine `gethostbyaddr()` maps Internet host addresses into a hostent structure.

The official name of the host and its public aliases are returned by these routines, along with the address type (family) and a null terminated list of variable length addresses. This list of addresses is required because it is possible for a host to have many addresses, all having the same name. The `h_addr` definition is provided for backward compatibility, and is defined to be the first address in the list of addresses in the hostent structure.

The database for these calls is provided either by the configuration file or by use of a name server. Because of the differences in these databases and their access protocols, the information returned may differ. When using the host table version of `gethostbyname()`, only one address will be returned, but all listed aliases will be included. The name server version may return alternate addresses, but will not provide any aliases other than one given as argument.

3.2.2 Network Names

As for host names, routines for mapping network names to numbers, and back, are provided. These routines return a *netent* structure:

```
/*
 * Assumption here is that a network number
 * fits in 32 bits -- probably a poor one.
 */
struct  netent {
    char    *n_name;        /* official name of net */
    char    **n_aliases;    /* alias list */
    int     n_addrtype;     /* net address type */
    int     n_net;          /* network number, host byte order */
};
```

The routines `getnetbyname()`, and`getnetbynumber()` are the network counterparts to the host routines described above. The routines uses data read from AmiTCP/IP configuration file.

3.2.3 Protocol Names

For protocols, the *protoent* structure defines the protocol–name mapping used with the routines `getprotobyname()` and `getprotobynumber()`:

```
struct  protoent {
    char    *p_name;        /* official protocol name */
    char    **p_aliases;    /* alias list */
    int     p_proto;        /* protocol number */
};
```

3.2.4 Service Names

Information regarding services is a bit more complicated. A service is expected to reside at a specific "port" and employ a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Further, a service may reside on multiple ports. If this occurs, the higher level library routines will have to be bypassed or extended. A service mapping is described by the servent structure:

```
struct  servent {
    char    *s_name;        /* official service name */
    char    **s_aliases;    /* alias list */
    int     s_port;         /* port number, network byte order */
    char    *s_proto;       /* protocol to use */
};
```

The routine `getservbyname()` maps service names to a `servent` structure by specifying a service name and, optionally, a qualifying protocol. Thus the call

```
sp = getservbyname("telnet", NULL);
```

returns the service specification for a telnet server using any protocol, while the call

```
sp = getservbyname("telnet", "tcp");
```

returns only that telnet server which uses the TCP protocol. The routine `getservbyport()` is also provided. The `getservbyport()` routine has an interface similar to that provided by `getservbyname()`; an optional protocol name may be specified to qualify lookups.

3.2.5 Miscellaneous

With the support routines described above, an Internet application program should rarely have to deal directly with addresses. This allows services to be developed as much as possible in a network independent fashion. It is clear, however, that purging all network dependencies is very difficult. So long as the user is required to supply network addresses when naming services and sockets there will always some network dependency in a program. For example, the normal code included in client programs, such as the remote login program, is as follows:

Remote Login Client Code

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
 ...
int main(int argc, char *argv[])
{
       struct sockaddr_in server;
       struct servent *sp;
       struct hostent *hp;
       int s;
        ...
       sp = getservbyname("login", "tcp");
       if (sp == NULL) {
          fprintf(stderr, "rlogin: tcp/login: unknown service\n");
          exit(1);
       }
       hp = gethostbyname(argv[1]);
       if (hp == NULL) {
          fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
          exit(2);
       }
       bzero((char *)&server, sizeof (server));
       server.sin_port = sp->s_port;
       bcopy(hp->h_addr, (char *)&server.sin_addr, hp->h_length);
       server.sin_family = hp->h_addrtype;

       s = socket(AF_INET, SOCK_STREAM, 0);
       if (s < 0) {
```

```
            perror("rlogin: socket");
            exit(3);
        }
         ...
        /* Connect does the bind() for us */

        if (connect(s, (struct sockaddr *)&server, sizeof (server)) < 0) {
            perror("rlogin: connect");
            exit(5);
        }
         ...
    }
```

(This example will be considered in more detail in section 3.3.)

If we wanted to make the remote login program independent of the Internet protocols and addressing scheme we would be forced to add a layer of routines which masked the network dependent aspects from the mainstream login code. For the current facilities available in the system this does not appear to be worthwhile.

Aside from the address-related data base routines, there are several other routines available in the run-time library which are of interest to users. These are intended mostly to simplify manipulation of names and addresses. The routines for manipulating variable length byte strings and handling byte swapping of network addresses and values are summarized below:[8].

`bcmp(s1, s2, n)`

Compare byte-strings; 0 if same, not 0 otherwise.

`bcopy(s1, s2, n)`

Copy n bytes from s1 to s2.

`bzero(base, n)`

Zero-fill n bytes starting at base.

`htonl(val)`

Convert 32-bit quantity from host to network byte order.

[8]The byte string functions are provided by the C-compiler. The byte order functions are provided as preprocessor macros.

`htons(val)`

Convert 16-bit quantity from host to network byte order.

`ntohl(val)`

Convert 32-bit quantity from network to host byte order.

`ntohs(val)`

Convert 16-bit quantity from network to host byte order.

The byte swapping routines are provided because the operating system expects addresses to be supplied in network order. On some architectures, such as the VAX, host byte ordering is different than network byte ordering. Consequently, programs are sometimes required to byte swap quantities. The library routines which return network addresses provide them in network order so that they may simply be copied into the structures provided to the system. This implies users should encounter the byte swapping problem only when *interpreting* network addresses. For example, if an Internet port is to be printed out the following code would be required:

```
printf("port number %d\n", ntohs(sp->s_port));
```

On machines where unneeded (as on Amiga) these routines are defined as null macros.

3.3 Client/Server Model

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme client applications request services from a server process. This implies an asymmetry in establishing communication between the client and server which has been examined in section 3.1. In this section we will look more closely at the interactions between client and server, and consider some of the problems in developing client and server applications.

The client and server require a well known set of conventions before service may be rendered (and accepted). This set of conventions comprises a protocol which must be implemented at both ends of a connection. Depending on the situation, the protocol may be symmetric or asymmetric. In a symmetric protocol, either side may play the master or slave roles. In an asymmetric protocol, one side is immutably recognized as the master, with the other as the slave. An example of a symmetric protocol is the TELNET protocol used in the Internet for remote terminal emulation. An example of an asymmetric protocol is the Internet file transfer protocol, FTP. No matter whether the specific protocol used in obtaining a service is symmetric or asymmetric, when accessing a service there is a “client process” and a “server process”. We will first consider the properties of server processes, then client processes.

A server process normally listens at a well known address for service requests. That is, the server process remains dormant until a connection is requested by a client's connection to the server's address. At such a time the server process “wakes up” and services the client, performing whatever appropriate actions the client requests of it.

Alternative schemes which use a service server may be used to eliminate a flock of server processes clogging the system while remaining dormant most of the time. For Internet servers in 4.3BSD, this scheme has been implemented via *inetd*, the so called “internet super-server.” Inetd listens at a variety of ports, determined at start-up by reading a configuration file. When a connection is requested to a port on which inetd is listening, inetd executes the appropriate server program to handle the client. Inetd will be described in more detail in section 3.4.

3.3.1 Servers

In 4.3BSD most servers are accessed at well known Internet addresses or UNIX domain names. For example, the remote login server's main loop is of the form shown below (AmiTCP/IP way):

```
main(int argc, char *argv)
{
    int f;
    struct sockaddr_in from;
    struct servent *sp;

    sp = getservbyname("login", "tcp");
    if (sp == NULL) {
        fprintf(stderr, "rlogind: tcp/login: unknown service\n");
        exit(1);
    }
     ...

    sin.sin_port = sp->s_port;  /* Restricted port */
     ...
    f = socket(AF_INET, SOCK_STREAM, 0);
     ...
    if (bind(f, (struct sockaddr *) &sin, sizeof (sin)) < 0) {
     ...
    }
```

```
      ...
     listen(f, 5);
     for (;;) {
         int g, len = sizeof (from);

         g = accept(f, (struct sockaddr *) &from, &len);
         if (g < 0) {
             if (errno != EINTR)
                 syslog(LOG_ERR, "rlogind: accept: %s", errors[errno]);
             continue;
         }
         /*
          * AmiTCP code follows...
          */
         id = ReleaseSocket(g, UNIQUE_ID);
         startit(id, &from);
     }
 }
```

The first step taken by the server is look up its service definition:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
    fprintf(stderr, "rlogind: tcp/login: unknown service\n");
    exit(1);
}
```

The result of the getservbyname call is used in later portions of the code to define the Internet port at which it listens for service requests (indicated by a connection).

Once a server has established a pristine environment, it creates a socket and begins accepting service requests. The `bind()` call is required to insure the server listens at its expected location.

The main body of the loop is fairly simple:

```
for (;;) {
    int g, len = sizeof (from);

    g = accept(f, (struct sockaddr *)&from, &len);
    if (g < 0) {
        if (errno != EINTR)
            syslog(LOG_ERR, "rlogind: accept: %s", errors[errno]);
```

```
            continue;
        }
        /*
         * AmiTCP code follows...
         */
        id = ReleaseSocket(g, UNIQUE_ID);
        startit(id, &from);
    }
```

An `accept()` call blocks the server until a client requests service. This call could return a failure status if the call is interrupted by a signal such as `CTRL-C` (to be discussed in section 3.4). Therefore, the return value from `accept()` is checked to insure a connection has actually been established.

With a connection in hand, servers using AmiTCP/IP socket library, this new socket is released to an external list inside AmiTCP/IP process via `ReleaseSocket()` call. `ReleaseSocket()` returns an id (unique if requested). `startit()` starts a new AmigaOS *task* and informs the id for it. This new task then uses `ObtainSocket()` with id as argument to receive the socket. The address of the client is also handled the new task because it requires it in authenticating clients.

3.3.2 Clients

The client side of the remote login service was shown earlier in section 3.2. One can see the separate, asymmetric roles of the client and server clearly in the code. The server is a passive entity, listening for client connections, while the client process is an active entity, initiating a connection when invoked.

Let us consider more closely the steps taken by the client remote login process. As in the server process, the first step is to locate the service definition for a remote login:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
    fprintf(stderr, "rlogin: tcp/login: unknown service\n");
    exit(1);
}
```

Next the destination host is looked up with a `gethostbyname()` call:

```
hp = gethostbyname(argv[1]);
if (hp == NULL) {
    fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
```

```
        exit(2);
    }
```

With this accomplished, all that is required is to establish a connection to the server at the requested host and start up the remote login protocol. The address buffer is filled in with the Internet address and rlogin port number of the foreign host.

```
    bzero((char *)&server, sizeof (server));
    server.sin_port = sp->s_port;
    bcopy(hp->h_addr, (char *) &server.sin_addr, hp->h_length);
    server.sin_family = hp->h_addrtype;
```

A socket is created, and a connection initiated. Note that `connect()` implicitly performs a `bind()` call, because `s` is unbound.

```
    s = socket(hp->h_addrtype, SOCK_STREAM, 0);
    if (s < 0) {
        perror("rlogin: socket");
        exit(3);
    }
     ...
    if (connect(s, (struct sockaddr *) &server,
            sizeof (server)) < 0) {
        perror("rlogin: connect");
        exit(4);
    }
```

The details of the remote login protocol will not be considered here.

3.3.3 Connectionless Servers

While connection-based services are the norm, some services are based on the use of datagram sockets. One, in particular, is the "rwho" service which provides users with status information for hosts connected to a local area network. This service, while predicated on the ability to *broadcast* information to all hosts connected to a particular network, is of interest as an example usage of datagram sockets.

A user on any machine running the rwho server may find out the current status of a machine with the **ruptime** program. The output generated is illustrated below.

Ruptime Output

```
arpa       up    9:45,       5 users, load   1.15,   1.39,   1.31
cad        up    2+12:04,    8 users, load   4.67,   5.13,   4.59
calder     up    10:10,      0 users, load   0.27,   0.15,   0.14
dali       up    2+06:28,    9 users, load   1.04,   1.20,   1.65
degas      up    25+09:48,   0 users, load   1.49,   1.43,   1.41
ear        up    5+00:05,    0 users, load   1.51,   1.54,   1.56
ernie    down    0:24
esvax    down    17:04
ingres   down    0:26
kim        up    3+09:16,    8 users, load   2.03,   2.46,   3.11
matisse    up    3+06:18,    0 users, load   0.03,   0.03,   0.05
medea      up    3+09:39,    2 users, load   0.35,   0.37,   0.50
merlin   down    19+15:37
miro       up    1+07:20,    7 users, load   4.59,   3.28,   2.12
monet      up    1+00:43,    2 users, load   0.22,   0.09,   0.07
oz       down    16:09
statvax    up    2+15:57,    3 users, load   1.52,   1.81,   1.86
ucbvax     up    9:34,       2 users, load   6.08,   5.16,   3.28
```

Status information for each host is periodically broadcast by rwho server processes on each machine. The same server process also receives the status information and uses it to update a database. This database is then interpreted to generate the status information for each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

Note that the use of broadcast for such a task is fairly inefficient, as all hosts must process each message, whether or not using an rwho server. Unless such a service is sufficiently universal and is frequently used, the expense of periodic broadcasts outweighs the simplicity.

The rwho server, in a simplified form, is pictured next[9]:

```
BYTE alrmsig;

main()
{
    long on;
    fd_set readfds;
     ...
    sp = getservbyname("who", "udp");
```

[9]A real code must always test the return values of various services against errors. Thes e tests are partly omitted from this code to show the matters important to this section.

```
sin.sin_port = sp->s_port;
net = getnetbyname("localnet");
sin.sin_addr = inet_makeaddr(INADDR_ANY, net);
 ...
s = socket(AF_INET, SOCK_DGRAM, 0);
 ...
on = 1;
if (setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on)) < 0) {
    syslog(LOG_ERR, "rwhod: setsockopt SO_BROADCAST: %s",
           strerror(errno));
    exit(1);
}
bind(s, (struct sockaddr *) &sin, sizeof (sin));
 ...
alrmsig = AllocSignal(-1);
onalrm(); /* activate and handle periodic alarm system */

FD_ZERO(&readfds);
FD_SET(s, &readfds);

for (;;) {
    struct whod wd;
    struct sockaddr_in from;
    int n, cc, whod, len = sizeof (from);
    ULONG alrmmask;

    alrmmask = 1 << alrmsig;
    n = WaitSelect(s, &readfds, NULL, NULL, NULL, &alrmmask);
    if (n < 0) {
        syslog(LOG_ERR, "rwhod: WaitSelect: %s", strerror(errno));
        exit(1);
    }
    if (alrmmask)
      onalrm(); /* handles the alarm */
    if (n > 0) {
        cc = recvfrom(s, (char *)&wd, sizeof (wd), 0,
                      (struct sockaddr *)&from, &len);
        if (cc <= 0) {
            if (cc < 0)
                syslog(LOG_ERR, "rwhod: recv: %s", strerror(errno));
            continue;
```

```
            }
            if (from.sin_port != sin.sin_port) {
                syslog(LOG_ERR, "rwhod: %ld: bad from port",
                       ntohs(from.sin_port));
                continue;
            }
            ...
            if (!verify(wd.wd_hostname)) {
                syslog(LOG_ERR, "rwhod: malformed host name from %lx",
                       ntohl(from.sin_addr.s_addr));
                continue;
            }
            (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
            whod = open(path, O_WRONLY | O_CREAT | O_TRUNC, 0666);
            ...
            (void) time(&wd.wd_recvtime);
            (void) write(whod, (char *)&wd, cc);
            (void) close(whod);
        }
    }
}
```

There are two separate tasks performed by the server. The first task is to act as a receiver of status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the rwho port are interrogated to insure they've been sent by another rwho server process, then are time stamped with their arrival time and used to update a file indicating the status of the host. When a host has not been heard from for an extended period of time, the database interpretation routines assume the host is down and indicate such on the status reports. This algorithm is prone to error as a server may be down while a host is actually up, but serves our current needs.

The second task performed by the server is to supply information regarding the status of its host. This involves periodically acquiring system status information, packaging it up in a message and broadcasting it on the local network for other rwho servers to hear. The supply function is triggered by a timer and runs off a signal. Locating the system status information is somewhat involved, but uninteresting. Deciding where to transmit the resultant packet is somewhat problematical, however.

Status information must be broadcast on the local network. For networks which do not support the notion of broadcast another scheme must be used to simulate or replace broadcasting. One possibility is to enumerate the known neighbors (based on the status messages received from other rwho servers). This, unfortunately, requires some bootstrapping information, for a server will have no idea what machines are its neighbors until it receives status messages from them. Therefore, if all machines on a net are freshly booted, no machine will have any known neighbors and thus never receive, or send, any status information. This is the identical problem faced by the routing table management process in propagating routing status information. The standard solution, unsatisfactory as it may be, is to inform one or more servers of known neighbors and request that theyalways communicate with these neighbors. If each server has at least one neighbor supplied to it, status information may then propagate through a neighbor to hosts which are not (possibly) directly neighbors. If the server is able to support networks which provide a broadcast capability, as well as those which do not, then networks with an arbitrary topology may share status information[10]

It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This would require a separate copy of the server at each host and make maintenance a severe headache. 4.3BSD attempts to isolate host-specific information from applications by providing system calls which return the necessary information[11]. A mechanism exists, in the form of an `IoctlSocket()` call, for finding the collection of networks to which a host is directly connected. Further, a local network broadcasting mechanism has been implemented at the socket level. Combining these two features allows a process to broadcast on any directly connected local network which supports the notion of broadcasting in a site independent manner. This allows 4.3BSD to solve the problem of deciding how to propagate status information in the case of rwho, or more generally in broadcasting: Such status information is broadcast to connected networks at the socket level, where the connected networks have been obtained via the appropriate ioctl calls. The specifics of such broadcastings are complex, however, and will be covered in section 3.4.

3.4 Advanced Topics

A number of facilities have yet to be discussed. For most users of the AmiTCP/IP the mechanisms already described will suffice in constructing distributed applications. However, others will find the need to utilize some of the features which we consider in this section.

3.4.1 Out Of Band Data

The stream socket abstraction includes the notion of "out of band" data. Out of band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out of band data is delivered to the user independently of normal data. The abstraction defines that the out of band data facilities must support the reliable delivery

[10]One must, however, be concerned about loops. That is, if a host is connected to multiple networks, it will receive status information from itself. This can lead to an endless, wasteful, exchange of information.

[11]An example of such a system call is the `gethostname()` call which returns the host's official name.

of at least one out of band message at a time. This message may contain at least one byte of data, and at least one message may be pending delivery to the user at any one time. For communications protocols which support only in-band signaling (i.e. the urgent data is delivered in sequence with the normal data), the system normally extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all the intervening data. It is possible to "peek" (via `MSG_PEEK`) at out of band data. If the socket has an owner, a signal is generated when the protocol is notified of its existence. A process can set the task to be informed by a signal via the appropriate `IoctlSocket()` and `SetSocketSignals()` calls, as described below in section 3.4.3. If multiple sockets may have out of band data awaiting delivery, a `select()` call for exceptional conditions may be used to determine those sockets with such data pending. Neither the signal nor the `select()` indicate the actual arrival of the out-of-band data, but only notification that it is pending.

In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out of band data was sent[12]. The remote login and remote shell applications use this facility to propagate signals between client and server processes. When a signal flushs any pending output from the remote process(es), all data up to the mark in the data stream is discarded.

To send an out of band message the `MSG_OOB` flag is supplied to a `send()` or `sendto()` calls, while to receive out of band data `MSG_OOB` should be indicated when performing a `recvfrom()` or `recv()` call. To find out if the read pointer is currently pointing at the mark in the data stream, the `SIOCATMARK` ioctl is provided:

```
IoctlSocket(s, SIOCATMARK, &yes);
```

If `yes` is a `1` on return, the next read will return data after the mark. Otherwise (assuming out of band data has arrived), the next read will provide data sent by the client prior to transmission of the out of band signal. The routine used in the remote login process to flush output on receipt of an interrupt or quit signal is shown below:

```
#include <sys/ioctl.h>
#include <sys/socket.h>
 ...
oob()
{
    int mark;
    char waste[BUFSIZ];
```

[12] AmiTCP/IP follows the BSD interpretation of the RFC 793 in which the concept of out-of-band data is introduced. The BSD interpretation is in conflict with (later) defined Host Requirements laid down in RFC 1122.

```
        /* flush terminal I/O on receipt of out of band data */

        for (;;) {
            if (IoctlSocket(rem, SIOCATMARK, &mark) < 0) {
                perror("IoctlSocket");
                break;
            }
            if (mark)
                break;
            recv(rem, waste, sizeof (waste), 0);
        }
        if (recv(rem, &mark, 1, MSG_OOB) < 0) {
            perror("recv");
             ...
        }
         ...
    }
```

The normal data up to the mark if first read (discarding it), then the out-of-band byte is read.

A process may also read or peek at the out-of-band data without first reading up to the mark. This is more difficult when the underlying protocol delivers the urgent data in-band with the normal data, and only sends notification of its presence ahead of time (e.g., the TCP protocol used to implement streams in the Internet domain). With such protocols, the out-of-band byte may not yet have arrived when a `recv()` is done with the `MSG_OOB` flag. In that case, the call will return an error of `EWOULDBLOCK`. Worse, there may be enough in-band data in the input buffer that normal flow control prevents the peer from sending the urgent data until the buffer is cleared. The process must then read enough of the queued data that the urgent data may be delivered.

Certain programs that use multiple bytes of urgent data and must handle multiple urgent signals (e.g., **telnet**) need to retain the position of urgent data within the stream. This treatment is available as a socket-level option, `SO_OOBINLINE`; see function reference for `setsockopt()` for usage. With this option, the position of urgent data (the "mark") is retained, but the urgent data immediately follows the mark within the normal data stream returned without the `MSG_OOB` flag. Reception of multiple urgent indications causes the mark to move, but no out-of-band data are lost.

3.4.2 Non-Blocking Sockets

It is occasionally convenient to make use of sockets which do not block; that is, I/O requests which cannot complete immediately and would therefore cause the process to be suspended awaiting completion are not executed, and an error code is returned. Once a socket has been created via the `socket()` call, it may be marked as non-blocking by `IoctlSocket()` as follows:

```
#include <sys/ioctl.h>
 ...
int     s;
long    yes = TRUE;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
 ...
if (IoctlSocket(s, FIONBIO, &yes) < 0)
    perror("IoctlSocket FIONBIO");
    exit(1);
}
 ...
```

When performing non-blocking I/O on sockets, one must be careful to check for the error `EWOULDBLOCK` (stored in the global variable `errno`), which occurs when an operation would normally block, but the socket it was performed on is marked as non-blocking. In particular, `accept()`, `connect()`, `send()`, `sendto()`, `recv()` and `recvto()` can all return `EWOULDBLOCK`, and processes should be prepared to deal with such return codes. If an operation such as a `send()` cannot be done in its entirety, but partial writes are sensible (for example, when using a stream socket), the data that can be sent immediately will be processed, and the return value will indicate the amount actually sent.

3.4.3 Signal Driven Socket I/O

The AmiTCP/IP allows a task to be notified via a signal when a socket has either normal or out-of-band data waiting to be read. Use of this facility requres four steps:

1. The signals to be used must be allocated with Exec `AllocSignal()` call.
2. The allocated signal(s) must be registered to the AmiTCP/IP with the `SetSocketSignals()` call. The signals registered with `SetSocketSignals()` affect all sockets of the calling task, so this is usually done only after `OpenLibrary()` call.
3. The owner of the socket must be set to the task itself (note that the owner of a socket is unspecified by default). This is accomplished by the use of an `IoctlSocket()` call.
4. Asynchronous notification for the socket must be enabled with another `IoctlSocket()` call

Note that it is application's responsibility to react on received signals.

Sample code to allow a given process to receive information on pending I/O requests as they occur for a socket `s` is given below:

```
#include <exec/tasks.h>
#include <sys/ioctl.h>
 ...
BYTE SIGIO = -1, SIGURG = -1;
 ...
struct Task *thisTask = FindTask(NULL); /* our task pointer */
long yes = TRUE;

/* Allocate signals for asynchronous notification */

if ((SIGIO = AllocSignal(-1)) == -1) {
    fprintf(stderr, "allocSignal failed.\n");
    exit(1);
}
atexit(freeSignals); /* free allocated signals on exit */
if ((SIGURG = AllocSignal(-1)) == -1) {
    fprintf(stderr, "allocSignal failed.\n");
    exit(1);
}

/* Set socket signals for this task */

SetSocketSignals(SIGBREAKF_CTRL_C, 1 << SIGIO, 1 << SIGURG);

/* Set the process receiving SIGIO/SIGURG signals to us */

if (IoctlSocket(s, FIOSETOWN, &thisTask) < 0) {
    perror("IoctlSocket FIOSETOWN");
    exit(1);
}

/* Allow receipt of asynchronous I/O signals */

if (IoctlSocket(s, FIOASYNC, &yes) < 0) {
    perror("IoctlSocket FIOASYNC");
    exit(1);
}
```

3.4.4 Selecting Specific Protocols

If the third argument to the `socket()` call is `0`, socket will select a default protocol to use with the returned socket of the type requested. The default protocol is usually correct, and alternate choices are not usually available. However, when using "raw" sockets to communicate directly with lower-level protocols or hardware interfaces, the protocol argument may be important for setting up demultiplexing. For example, raw sockets in the Internet family may be used to implement a new protocol above IP, and the socket will receive packets only for the protocol specified. To obtain a particular protocol one determines the protocol number as defined within the communication domain. For the Internet domain one may use one of the library routines discussed in section 3.2, such as `getprotobyname()`:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
 ...
pp = getprotobyname("newtcp");
s = socket(AF_INET, SOCK_STREAM, pp->p_proto);
```

This would result in a socket s using a stream based connection, but with protocol type of "newtcp" instead of the default "tcp."

3.4.5 Address Binding

As was mentioned in section 3.1, binding addresses to sockets in the Internet domains can be fairly complex. As a brief reminder, these associations are composed of local and foreign addresses, and local and foreign ports. Port numbers are allocated out of separate spaces, one for each system and one for each domain on that system. Through the `bind()` call, a process may specify half of an association, the <local address, local port> part, while the `connect()` and `accept()` calls are used to complete a socket's association by specifying the <foreign address, foreign port> part. Since the association is created in two steps the association uniqueness requirement indicated previously could be violated unless care is taken. Further, it is unrealistic to expect user programs to always know proper values to use for the local address and local port since a host may reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

To simplify local address binding in the Internet domain the notion of a "wildcard" address has been provided. When an address is specified as `INADDR_ANY` (a manifest constant defined in file **netinet/in.h**), the system interprets the address as "any valid address". For example, to bind a specific port number to a socket, but leave the local address unspecified, the following code might be used:

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bzero(sin.sin_zero, sizeof(sin.sin_zero));
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

Sockets with wildcarded local addresses may receive messages directed to the specified port number, and sent to any of the possible addresses assigned to a host. For example, if a host has addresses 128.32.0.4 and 10.0.0.78, and a socket is bound as above, the process will be able to accept connection requests which are addressed to 128.32.0.4 or 10.0.0.78. If a server process wished to only allow hosts on a given network connect to it, it would bind the address of the host on the appropriate network.

In a similar fashion, a local port may be left unspecified (specified as zero), in which case the system will select an appropriate port number for it. For example, to bind a specific local address to a socket, but to leave the local port number unspecified:

```
hp = gethostbyname(hostname);
if (hp == NULL) {
     ...
}
bzero(&sin, sizeof(sin));
bcopy(hp->h_addr, (char *) sin.sin_addr, hp->h_length);
sin.sin_port = htons(0);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

The system selects the local port number based on two criteria. The first is that on 4BSD systems, Internet ports below `IPPORT_RESERVED` (1024) are reserved for privileged processes[13]; Internet ports above `IPPORT_USERRESERVED` (5000) are reserved for non-privileged servers. The second is that the port number is not currently bound to some other socket. In order to find a free Internet port number in the privileged range the **`rresvport()`** library routine may be used as follows to return a stream socket in with a privileged port number:

```
int lport = IPPORT_RESERVED - 1;
```

[13]All processes in AmigaOS are considered as privileged.

```
int s;
s = rresvport(&lport);
if (s < 0) {
    if (errno == EAGAIN)
        fprintf(stderr, "socket: all ports in use\n");
    else
        perror("rresvport: socket");
     ...
}
```

The restriction on allocating ports was done to allow processes executing in a secure environment to perform authentication based on the originating address and port number. For example, the **rlogin** command allows users to log in across a network without being asked for a password, if two conditions hold: First, the name of the system the user is logging in from is in the file **AmiTCP:db/hosts.equiv**[14] on the system he is logging into (or the system name and the user name are in the user's **.rhosts** file in the user's home directory), and second, that the user's rlogin process is coming from a privileged port on the machine from which he is logging. The port number and network address of the machine from which the user is logging in can be determined either by the `from` result of the `accept()` call, or from the `getpeername()` call.

In certain cases the algorithm used by the system in selecting port numbers is unsuitable for an application. This is because associations are created in a two step process. For example, the Internet file transfer protocol, FTP, specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation the system would disallow binding the same local address and port number to a socket if a previous data connection's socket still existed. To override the default port selection algorithm, an option call must be performed prior to address binding:

```
 ...
long    on = 1;
 ...
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

With the above call, local addresses may be bound which are already in use. This does not violate the uniqueness requirement as the system still checks at connect time to be sure any other sockets with the same local address and port do not have the same foreign address and port. If the association already exists, the error `EADDRINUSE` is returned.

[14] In UNIX **/etc/hosts.equiv**

3.4.6 Broadcasting And Determining Network Configuration

By using a datagram socket, it is possible to send broadcast packets on many networks supported by the system. The network itself must support broadcast; the system provides no simulation of broadcast in software. Broadcast messages can place a high load on a network since they force every host on the network to service them. Consequently, the ability to send broadcast packets has been limited to sockets which are explicitly marked as allowing broadcasting. Broadcast is typically used for one of two reasons: it is desired to find a resource on a local network without prior knowledge of its address, or important functions such as routing require that information be sent to all accessible neighbors.

To send a broadcast message, a datagram socket should be created:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

The socket is marked as allowing broadcasting,

```
long    on = 1;

setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof (on));
```

and at least a port number should be bound to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bzero(sin.sin_zero, sizeof(sin.sin_zero));
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

The destination address of the message to be broadcast depends on the network(s) on which the message is to be broadcast. The Internet domain supports a shorthand notation for broadcast on the local network, the address `INADDR_BROADCAST` (defined in **netinet/in.h**). To determine the list of addresses for all reachable neighbors requires knowledge of the networks to which the host is connected. Since this information should be obtained in a host independent fashion and may be impossible to derive, 4.3BSD provides a method of retrieving this information from the system data structures. The `SIOCGIFCONF` `IoctlSocket()` call returns the interface configuration of a host in the form of a single `ifconf` structure; this structure contains a "data area" which is made up of an array of of `ifreq` structures, one for each network interface to which the host is connected. These structures are defined in **net/if.h** as follows:

```
struct ifconf {
    int ifc_len;     /* size of associated buffer */
    union {
```

```
        caddr_t         ifcu_buf;
        struct ifreq *ifcu_req;
    } ifc_ifcu;
};

#define ifc_buf     ifc_ifcu.ifcu_buf/* buffer address */
#define ifc_req     ifc_ifcu.ifcu_req/* array of structures returned */

#define IFNAMSIZ    64

struct ifreq {
    char   ifr_name[IFNAMSIZ]; /* if name, e.g. "en0" */
    union {
        struct sockaddr ifru_addr;
        struct sockaddr ifru_dstaddr;
        struct sockaddr ifru_broadaddr;
        short  ifru_flags;
        caddr_t         ifru_data;
    } ifr_ifru;
};

#define ifr_addr      ifr_ifru.ifru_addr    /* address */
#define ifr_dstaddr   ifr_ifru.ifru_dstaddr /* other end of p-to-p link */
#define ifr_broadaddr ifr_ifru.ifru_broadaddr /* broadcast address */
#define ifr_flags     ifr_ifru.ifru_flags   /* flags */
#define ifr_data      ifr_ifru.ifru_data    /* for use by interface */
```

The actual call which obtains the interface configuration is

```
struct ifconf ifc;
char buf[BUFSIZ];

ifc.ifc_len = sizeof (buf);
ifc.ifc_buf = buf;
if (IoctlSocket(s, SIOCGIFCONF, (char *) &ifc) < 0) {
     ...
}
```

After this call `buf` will contain one `ifreq` structure for each network to which the host is connected, and `ifc.ifc_len` will have been modified to reflect the number of bytes used by the `ifreq` structures.

For each structure there exists a set of "interface flags" which tell whether the network corresponding to that interface is up or down, point to point or broadcast, etc. The `SIOCGIFFLAGS` `IoctlSocket()` retrieves these flags for an interface specified by an `ifreq` structure as follows:

```
struct ifreq *ifr;

ifr = ifc.ifc_req;

for (n = ifc.ifc_len / sizeof (struct ifreq); --n >= 0; ifr++) {
    /*
     * We must be careful that we don't use an interface
     * devoted to an address family other than those intended;
     * if we were interested in NS interfaces, the
     * AF_INET would be AF_NS.
     */
    if (ifr->ifr_addr.sa_family != AF_INET)
        continue;
    if (IoctlSocket(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
         ...
    }
    /*
     * Skip boring cases.
     */
    if ((ifr->ifr_flags & IFF_UP) == 0 ||
        (ifr->ifr_flags & IFF_LOOPBACK) ||
        (ifr->ifr_flags & (IFF_BROADCAST | IFF_POINTTOPOINT)) == 0)
        continue;
```

Once the flags have been obtained, the broadcast address must be obtained. In the case of broadcast networks this is done via the `SIOCGIFBRDADDR` `IoctlSocket()`, while for point-to-point networks the address of the destination host is obtained with `SIOCGIFDSTADDR`.

```
struct sockaddr dst;

if (ifr->ifr_flags & IFF_POINTTOPOINT) {
    if (IoctlSocket(s, SIOCGIFDSTADDR, (char *) ifr) < 0) {
         ...
    }
    bcopy((char *) ifr->ifr_dstaddr, (char *) &dst,
```

```
            sizeof (ifr->ifr_dstaddr));
    } else if (ifr->ifr_flags & IFF_BROADCAST) {
        if (IoctlSocket(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
            ...
        }
        bcopy((char *) ifr->ifr_broadaddr, (char *) &dst,
            sizeof (ifr->ifr_broadaddr));
    }
```

After the appropriate ioctl's have obtained the broadcast or destination address (now in dst), the `sendto()` call may be used:

```
        sendto(s, buf, buflen, 0, (struct sockaddr *)&dst, sizeof (dst));
    }
```

In the above loop one `sendto()` occurs for every interface to which the host is connected that supports the notion of broadcast or point-to-point addressing. If a process only wished to send broadcast messages on a given network, code similar to that outlined above would be used, but the loop would need to find the correct destination address.

Received broadcast messages contain the senders address and port, as datagram sockets are bound before a message is allowed to go out.

AmiTCP/IP specific extensions

Extensions to interface `ioctls`

The following `ioctls` are used to configure protocol and hardware specific properties of a `sana_softc` interface. They are used in the AmiTCP/IP only.

`SIOCSSANATAGS` Set SANA-II specific properties with a tag list.

`SIOCGSANATAGS` Get SANA-II specific properties into the `wiretype_parameters` structure and a user tag list.

These `ioctls` use the following structure as a argument:

```
struct wiretype_parameters
{
  ULONG  wiretype;       /* the wiretype of the interface */
  WORD   flags;                              /* iff_flags */
  struct TagItem *tags;         /* tag list user provides */
};
```

`SIOCGARPT` Get the contents of an ARP mapping cache into a `struct arpreq` table.

This `ioctl` takes the following `arptabreq` structure as an argument:

```
/*
 * An AmiTCP/IP specific ARP table ioctl request
 */
struct arptabreq {
   struct arpreq atr_arpreq;  /* To identify the interface */
   long   atr_size;          /* # of elements in atr_table */
   long   atr_inuse;              /* # of elements in use */
   struct arpreq *atr_table;
};
```

The `atr_arpreq` specifies the used interface. The hardware address for the interface is returned in the `arp_ha` field of `atr_arpreq` structure.

The `SIOCGARPT ioctl` reads at most `atr_size` entries from the cache into the user supplied buffer `atr_table`, if it is not `NULL`. Actual amount of returned entries is returned in `atr_size`. The current amount of cached mappings is returned in the `atr_inuse`.

3.4.7 Socket Options

It is possible to set and get a number of options on sockets via the `setsockopt()` and `getsockopt()` calls. These options include such things as marking a socket for broadcasting, not to route, to linger on close, etc. The general forms of the calls are:

```
setsockopt(s, level, optname, optval, optlen);
```

and

```
getsockopt(s, level, optname, optval, optlen);
```

The parameters to the calls are as follows: `s` is the socket on which the option is to be applied. `level` specifies the protocol layer on which the option is to be applied; in most cases this is the "socket level", indicated by the symbolic constant `SOL_SOCKET`, defined in **sys/socket.h**. The actual option is specified in `optname`, and is a symbolic constant also defined in **sys/socket.h**. `optval` and `optlen` point to the value of the option (in most cases, whether the option is to be turned on or off), and the length of the value of the option, respectively. For `getsockopt()`, `optlen` is a value–result parameter, initially set to the size of the storage area pointed to by `optval`, and modified upon return to indicate the actual amount of storage used.

An example should help clarify things. It is sometimes useful to determine the type (e.g., stream, datagram, etc.) of an existing socket; programs under **inetd** (described in section 3.4.8) may need to perform this task. This can be accomplished as follows via the `SO_TYPE` socket option and the `getsockopt()` call:

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

long type, size;

size = sizeof (type);

if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &type, &size) < 0) {
     ...
}
```

After the `getsockopt()` call, `type` will be set to the value of the socket type, as defined in **sys/socket.h**. If, for example, the socket were a datagram socket, type would have the value corresponding to `SOCK_DGRAM`.

3.4.8 Inetd

One of the daemons provided with AmiTCP/IP is **inetd**, the so called "internet super–server." **Inetd** is invoked at start-up time, and determines the servers, for which it is to listen, from the file **AmiTCP:db/inetd.conf**[15]. Once this information has been read and a pristine environment created, **inetd** proceeds to create one socket for each service it is to listen for, binding the appropriate port number to each socket.

Inetd then performs a `select()` on all these sockets for read availability, waiting for somebody wishing a connection to the service corresponding to that socket. **Inetd** then performs an `accept()` on the socket in question, releases the socket with a `ReleaseSocket()` call and starts the appropriate server.

Servers making use of **inetd** are considerably simplified, as **inetd** takes care of the majority of the work required in establishing a connection. The server invoked by **inetd** expects the socket connected to its client to be found by calling `ObtainSocket()`. The client socket ID for the server is found in a `DaemonMessage` structure given to the server process. Usually the **n**etlib:autoinitd.o module takes care of obtaining the client socket into global variable `server_socket`. For practical purposes the code might assume this socket to be 0.

[15] In UNIX systems **/etc/inetd.conf**.

One call which may be of interest to individuals writing servers under **inetd** is the `getpeername()` call, which returns the address of the peer (process) connected on the other end of the socket. For example, to log the Internet address in "dot notation" (e.g., "128.32.0.4") of a client connected to a server under **inetd**, the following code might be used:

```
struct sockaddr_in name;
int namelen = sizeof (name);
 ...
if (getpeername(0, (struct sockaddr *)&name, &namelen) < 0) {
    syslog(LOG_ERR, "getpeername: %m");
    exit(1);
} else
    syslog(LOG_INFO, "Connection from %s", inet_ntoa(name.sin_addr));
 ...
```

While the `getpeername()` call is especially useful when writing programs to run with **inetd**, it can be used under other circumstances.

Sources for a very simple TCP protocol server is included with AmiTCP/IP as an example.

3.5 Deviation From Berkeley Sockets

This section discusses the differences between the API of the AmiTCP/IP and the 4.3BSD. They are not so numerous as it might seem to, but worth taking attention to when porting existing 4.3BSD software to AmiTCP/IP.

3.5.1 Opening and Closing the Shared Library

Since the API is provided as a shared library, it must be opened to be able to access the functions it provides. Note that any two tasks may not share a socket library base, since the base contains task specific information.

AmiTCP/IP does resource tracking based on the information stored in a library base, so it is essential that the library is closed after use, since the resources used by the base are gone if the application exits without closing the base.

See section 3.1.2 for examples and more discussion on the subject.

3.5.2 Naming Conventions of the API Functions

The API functions which preserve the semantics of the BSD calls are named the same as the original functions. In the name of binary compatibility between different C compilers the functions which either take a structure as an argument or return a structure as a value had to be changed not to do so. These functions are named differently; all words in these function names begin with an upper case letter. Inline functions[16] are provided with the original semantics, however. This makes it possible to keep using the original functions when writing the code and still be binary compatible. The inline functions are mostly trivial; for example the `select()` call is actually an inline which calls `WaitSelect()` with last argument as `NULL`.

This is a matter which should be totally invisible for C users, but assembler programmers should take attention and be sure to pass arguments as described in the function reference for the non-inline versions.

3.5.3 `errno`

Unix libraries return error values in a global variable named `errno`. Since a shared library cannot know the address of any variables of an application, the address of the `errno` must be explicitly told to the AmiTCP/IP. An alternative is to use `Errno()` call to fetch the error value, but since the first method needs no modifications to the existing sources (besides calling `SetErrnoPtr()` once in the beginning), it is the preferred method. Section 3.1.2 contains examples and more discussion about the matter.

3.5.4 New Field in the `sockaddr` Structures

Since AmiTCP/IP is based on the BSD Net/2 release, it has few differences to the 4.3BSD. Most notable one is that the `sockaddr` and `sockaddr_in` structures have a new field telling the length of the structure. These are named as `sa_len` and `sin_len` respectively. These fields are used by the AmiTCP/IP to determine the real length of the address given.

In addition the `sockaddr_in` structure has a field named `sin_zero`, which should be initialized to zero before passed to the AmiTCP/IP, since any garbage left there will be used by the routing facility (which obviously leads to undesired behaviour).

3.5.5 Linger Time

The unit of the linger time of a socket is *a second*. The Net/2 code seemed to use ticks[17].

3.5.6 Transferring Sockets from a Task to Another

Since AmigaOS has no `fork()` call, in which the child process inherits the file descriptors, and hence sockets, a mechanism for transferring sockets from a task to another must be

[16] Or linker stubs in compilers with no inline functions.

[17] One tick is 1/hz:th of a second, where hz is the frequency of the electricity of the wall *socket*.

provided. This is accomplished by the calls `ReleaseSocket()`, `ReleaseCopyOfSocket()` and `ObtainSocket()`, which release a socket, a copy of a socket and obtain a released socket, respectively. An *id* is given to a socket placed in the list of released sockets. This id can be either unique or be based on a service number in cases where it is irrelevant which instance of the server process for the service in question obtains the socket. If an unique id is used, the releasing task is responsible of transferring the id to the obtaining task.

This feature affects the server processes only, since the clients usually create the socket(s) on their own.

3.5.7 Ioctl Differences

The Unix `ioctl()` function is renamed as `IoctlSocket()` in AmiTCP/IP to avoid name clashes with some C runtime libraries.

Following summarizes other differences in the ioctl calls:

1. `FIOCLEX` and `FIONCLEX` are silently ignored, that is, they are accepted, but have no effect. Whether sockets should be closed on `exec()` or not is irrelevant, since AmigaOS has no such feature (see discussion about `fork()` above).

2. `FIOSETOWN` and `SIOCSPGRP` take a pointer to a `struct Task *` as an argument instead of a pointer to a process (or group) id. Note that if the task in question has not opened the bsdsocket.library the owner of the precess is set to `NULL` (disabled). The task pointer is used as the receiver of the asynchronous notification signals if asynchronous notification is enabled.

3. `FIOGETOWN` and `SIOCGPGRP` take a pointer to a `struct Task *` as an argument in which the current task pointer of the owner of the socket in question is placed on return.

3.5.8 Signal Handling

There is a fundamental difference between BSD Unix and Amiga signal handling and the system call interface. In the Unix systems a received signal may interrupt a process executing a system call. If there is a signal handler installed, it can be executed before the system call returns to the main execution branch with an error code.

However, there are *some* system calls which may not be interrupted. If a Unix process has a negative priority, `tsleep()` does not wake up until the specified condition is met.

The interrupted system call does not have any unrecoverable effects, the execution of the program may continue after the `errno` is checked against other errors than `EINTR`.

In the AmigaOS, Exec, there are no specific system calls. All OS functions are provided by shared libraries. There are either no separate kernel and user memory spaces, the one common memory space is shared by all processes. The IO system is based on messages, which are implemented as shared memory areas. When a program receives a message to a port, it is delivered a signal associated with the port.

While it is possible to use signal handlers with Exec, they are even more dangerous to use and restricted than in Unix systems. This is not recommended, since the exception handler must behave like any real interrupt handler. Calls provided by AmiTCP/IP *are not callable from interrupts*. Further, *it is not possible to interrupt a system call implemented as a shared library function.*

The application must itself react on receipt of the signals. The recommended way of handling these signals is by the normal `Wait()` of by AmiTCP/IP call `WaitSelect()`, which allows an application to specify a signal mask which should abort the selection. The application then checks the received signals and calls appropriate handler for the signal.

3.5.9 Asynchronous I/O

AmigaOS does not have any reserved signals for networking, such as `SIGIO` or `SIGURG` in Unix systems, and so the scheme used in asynchronous notification must be changed a little.

The application can set a group of signal masks, with function named `SetSocketSignals()`, to be used by the AmiTCP/IP. First argument specifies the signal mask which should break the blocking of the blocking socket calls. It is by default set to the signal for `CTRL-C`. Second argument specifies the signal(s) to be sent when asynchronous notification for readiness to read is necessary. This mask lets the application define which signal should be used as replacement for `SIGIO` signal of the Unix systems. Third and last argument specifies the corresponding mask for the asynchronous notification of urgent (out-of-band) data (`SIGURG`). These last two masks are zero by default.

Note that there is no way to query the current settings of these signals form the AmiTCP/IP, so the application must store the signal numbers (or masks) for later use. Also note that the break mask must be explicitly given if `SetSocketSignals()` is called, since the values supplied override the default settings.

3.5.10 Constructing an Event Loop

Amiga programs are often constructed around an *event loop*, in which `Wait()` function is used to wait for some subset of given signals to arrive. When a signal is received, some actions are taken and if IO is performed, it is usually asynchronous.

Many Unix programs use to do synchronous IO and let the *signal handlers* to handle special events (window size changes, timeouts, etc.). This can be emulated to some extent with AmigaOS, since it is possible to specify an exeption function to handle reception of given signals. This is very limited, though, since the exeption code is executed at true interrupt level, and may thus pre-empt the main process in an arbitrary location. Also note that a very limited set of shared library functions can be called while in interrupt, especially note that *any* AmiTCP/IP *function may NOT be called from interrupt code*.

AmiTCP/IP offers remedy for this, however. The application can use `WaitSelect()` to handle both Amiga signals and socket IO multiplexing. Selecting assures that the following socket calls will not block[18].

Another possiblility is to use signal driven socket IO (see section 3.4.3).

Yet another possibility is to specify a special break mask with `SetSocketSignals()` function. The signals in the mask cause any blocking socket IO routine to return with the error code `EINTR`. Note that the signals are not cleared in this procedure. The `Wait()` with the same signal mask can be used to determine (and clear) the received signals. This allows the usage of synchronous socket IO, but the `EINTR` error code must be checked after each failing call.

3.5.11 "Killing" the Processes

In AmigaOS the applications must co-operate with the OS for the user to be able to stop them. This is why the blocking operations of the AmiTCP/IP can be aborted. By default the reception of `CTRL-C` signal aborts any blocking call. The call returns an error value (in `errno`) of `EINTR` when aborted. In addition the signal which caused the break will remain set for the application to be able to react on it in its normal event processing. This means that the application need not specially check for `EINTR` after every socket call as long as they eventually check for the break signal.

All sockets left open by the application are closed by the `CloseLibrary()` call. You may left the sockets open when aborting the program, because the socket library is closed automatically during the exit process if either *autotermination function* (specific to SAS C) or ANSI `atexit()` function is installed before the exit is done. .

The signals which cause the abort can be set with the `SetSocketSignals()` call. The break signal mask is given as the first argument. Calling this function discards the previous values of the sockets signal masks. Aborting can be disabled by giving the mask as `0L`. See section 3.5.9 for more discussion about the `SetSocketSignals()` call.

3.5.12 `WaitSelect()`

In AmiTCP/IP no other than socket I/O can be multiplexed with the `select()` call. This may be a major pain as I/O is normally multiplexed with an `Wait()` loop, waiting for given signals to arrive. This is the motivation for the `WaitSelect()` call. It combines the selection

[18] See `NOTES` section of the reference for the `WaitSelect()`.

and waiting in a single call[19]. The `WaitSelect()` takes one argument in addition to the normal `select()` call. It is a pointer to signal mask to wait for in addition to the signals that the AmiTCP/IP uses internally. If any of these signals is received, they are returned as a result in the same signal mask. Signals specified in the given signal mask override the signals of the break mask (see previous section). If the same signal is specified in both the `SIGINTR` mask and the mask given to the `WaitSelect()`, the reception of the signal causes it to be cleared and returned in the mask as the result.

`WaitSelect()` can be used as replacement for the `Wait()` in applications which require to multiplex both socket related and other Amiga I/O.

[19]This feature was really easy to implement, since AmiTCP/IP uses a `Wait()` to wait for I/O events itself.

Appendix B

API Function Reference

This appendix is a complete reference to the functions and concepts provided by the AmiTCP/IP system.

Table of Contents

B.1 Standard BSD Style Socket Functions

B.1.1 accept()

```
NAME
     accept - accept a connection on a socket

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

     ns = accept(s, addr, addrlen)
     D0          D0 A0    A1

     long accept(long, struct sockaddr *, long *);

FUNCTION
     The argument s is a socket that has been created with
     socket(), bound to an address with bind(), and is listen-
     ing for connections after a listen(). accept() extracts the
     first connection on the queue of pending connections,
     creates a new socket with the same properties of s and allo-
     cates a new socket descriptor for the socket. If no pending
     connections are present on the queue, and the socket is not
     marked as non-blocking, accept() blocks the caller until a
     connection is present. If the socket is marked non-blocking
     and no pending connections are present on the queue,
     accept() returns an error as described below. The accepted
     socket is used to read and write data to and from the socket
     which connected to this one; it is not used to accept more
     connections. The original socket s remains open for accept-
     ing further connections.

     The argument addr is a result parameter that is filled in
     with the address of the connecting entity, as known to the
     communications layer. The exact format of the addr parame-
     ter is determined by the domain in which the communication
     is occurring. The addrlen is a value-result parameter; it
     should initially contain the amount of space pointed to by
```

```
     addr; on return it will contain the actual length (in bytes)
     of   the   address   returned.   This  call  is  used  with
     connection-based socket types, currently with SOCK_STREAM.

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

RETURN VALUES
     accept() returns a non-negative descriptor for the  accepted
     socket on success.  On failure, it returns -1 and sets errno
     to indicate the error.

ERRORS
     EBADF        - The descriptor is invalid.

     EINTR        - The operation was interrupted by a break
                    signal.

     EOPNOTSUPP   - The referenced socket is not of type
                    SOCK_STREAM.

     EWOULDBLOCK  - The socket is marked non-blocking and no con-
                    nections are present to be accepted.

SEE ALSO
     bind(), connect(), listen(), select(), SetSocketSignals(),
     socket()
```

B.1.2 bind()

```
NAME
     bind - bind a name to a socket

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

     success = bind(s, name, namelen)
     D0             D0 A0    D1

     long bind(long, struct sockaddr *, long);

FUNCTION
     bind() assigns a name to an unnamed socket.  When  a  socket
     is created with socket(2) it exists in a name space (address
     family) but has no name assigned.  bind() requests that  the
     name pointed to by name be assigned to the socket.

RETURN VALUES
     0  - on success.

     -1 - on failure and sets errno to indicate the error.

ERRORS
     EACCES            - The requested address is protected,  and
                         the  current user has inadequate permis-
                         sion to access it.

     EADDRINUSE        - The specified address is already in use.

     EADDRNOTAVAIL     - The specified address is  not  available
                         from the local machine.

     EBADF             - s is not a valid descriptor.

     EINVAL            - namelen is  not  the  size  of  a  valid
                         address  for  the specified address fam-
```

```
                         ily.

                         The  socket  is  already  bound  to   an
                         address.

SEE ALSO
     connect(), getsockname(), listen(), socket()

NOTES
     The rules used in name binding  vary  between  communication
     domains.
```

B.1.3 CloseSocket()

```
NAME
     CloseSocket - delete a socket descriptor

SYNOPSIS
     success = CloseSocket(s)
     D0                    D0

     long CloseSocket(long);

FUNCTION
     CloseSocket() deletes  a  descriptor  from the  library base
     socket reference table.   If s is the last reference  to the
     underlying object, then the object  will  be deactivated and
     socket  (see socket()),  associated naming  information  and
     queued data are discarded.

     All sockets are automatically closed when the socket library
     is closed, but closing sockets as soon as possible is
     recommended to save system resources.

RETURN VALUES
      0   on success.

     -1   on failure and sets errno to indicate the error.

ERRORS
     EBADF             - s is not an active socket descriptor.

     EINTR             - linger on close was interrupted.
                         The socket is closed, however.

SEE ALSO
     accept(), SetSocketSignals(), shutdown(), socket(),
     exec.library/CloseLibrary()
```

B.1.4 connect()

```
NAME
     connect - initiate a connection on a socket

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

     success = connect(s, name, namelen)
     D0                D0 A0    D1

     long connect(long, struct sockaddr *, long);

FUNCTION
     The parameter s is a socket.  If it  is of  type SOCK_DGRAM,
     then  this call specifies the peer with which the  socket is
     to be associated;  this address  is that  to which datagrams
     are  to be sent, and  the only address from which  datagrams
     are to be received.  If it is of type SOCK_STREAM, then this
     call attempts  to make a connection  to another socket.  The
     other socket is specified by name which is an address in the
     communications space of  the  socket.   Each  communications
     space interprets the  name parameter in  its  own way.  Gen-
     erally, stream sockets may successfully connect() only once;
     datagram sockets may use connect() multiple  times to change
     their association.  Datagram sockets may dissolve the  asso-
     ciation by connecting to  an invalid address, such as a null
     address.

RETURN VALUES
      0   on success.

     -1   on failure and sets errno to indicate the error.

ERRORS
     EADDRINUSE        - The address is already in use.

     EADDRNOTAVAIL     - The specified address is  not  available
```

```
                     on the remote machine.

EAFNOSUPPORT       - Addresses in the specified address  fam-
                     ily cannot be used with this socket.

EALREADY           - The socket is non-blocking and a  previ-
                     ous  connection attempt has not yet been
                     completed.

EBADF              - s is not a valid descriptor.

ECONNREFUSED       - The attempt to  connect  was  forcefully
                     rejected.  The  calling  program should
                     CloseSocket() the socket descriptor, and
                     issue another socket()  call to obtain a
                     new descriptor before attempting another
                     connect() call.

EINPROGRESS        - The socket is non-blocking and the  con-
                     nection cannot be completed immediately.
                     It is possible to select()  for  comple-
                     tion  by  selecting the socket for writ-
                     ing.

EINTR              - The operation was interrupted by a break
                     signal.

EINVAL             - namelen is  not  the  size  of  a  valid
                     address  for  the specified address fam-
                     ily.

EISCONN              The socket is already connected.

ENETUNREACH        - The network is not reachable  from  this
                     host.

ETIMEDOUT          - Connection   establishment   timed   out
                     without establishing a connection.
```

```
SEE ALSO
     accept(), CloseSocket(), connect(), getsockname(), select(),
     socket()
```

B.1.5 Dup2Socket()

```
NAME
    Dup2Socket - duplicate a socket descriptor

SYNOPSIS

    newfd = Dup2Socket(fd1, fd2)
    D0                 D0   D1

    long Dup2Socket(long, long);

DESCRIPTION
    Dup2Socket() duplicates an existing socket descriptor.
    the argument fd1 is small non-negative value that indexes
    the socket on SocketBase descriptor table. The value must
    be less than the size of the table, which is returned by
    getdtablesize(). fd2 specifies the desired value of the new
    descriptor. If descriptor fd2 is already in use, it is
    first deallocated as if it were closed by CloseSocket(). If
    the value if fd2 is -1, the new descriptor used and returned
    is the lowest numbered descriptor that is not currently in
    use by the SocketBase.

RETURN VALUES
    Dup2Socket() returns a new descriptor on  success. On failure
    -1 is returned and errno is set to indicate the error.

ERRORS
    EBADF          fd1 or fd2 is not a valid active descriptor.

    EMFILE         Too many descriptors are active.

SEE ALSO
    accept(), CloseSocket(), getdtablesize(), SetDtableSize(),
    socket()
```

B.1.6 getpeername()

```
NAME
     getpeername - get name of connected peer

SYNOPSIS
     success =  getpeername(s, name, namelen)
     D0                     D0 A0    A1

     long getpeername(long, struct sockaddr *, long *);

FUNCTION
     getpeername() returns the name  of  the  peer  connected  to
     socket  s.   The long  pointed  to  by the namelen parameter
     should be  initialized  to  indicate  the  amount  of  space
     pointed  to  by name.  On return it contains the actual size
     of the name returned (in bytes).  The name is  truncated  if
     the buffer provided is too small.

RETURN VALUE
     A 0 is returned if the call succeeds, -1 if it fails.

ERRORS
     EBADF          - The argument s is not a valid descriptor.

     ENOBUFS        - Insufficient resources were available in  the
                      system to perform the operation.

     ENOTCONN       - The socket is not connected.

SEE ALSO
     accept(), bind(), getsockname(), socket()
```

B.1.7 getsockname()

```
NAME
     getsockname - get socket name

SYNOPSIS

     success = getsockname(s, name, namelen)
     D0                    D0 A0    A1

     long getsockname(long, struct sockaddr *, long *);

FUNCTION
     getsockname() returns the current  name  for  the  specified
     socket.   The  namelen  parameter  should  be initialized to
     indicate the amount of space pointed to by name.  On  return
     it contains the actual size of the name returned (in bytes).

DIAGNOSTICS
     A 0 is returned if the call succeeds, -1 if it fails.

ERRORS
     The call succeeds unless:

     EBADF          s is not a valid descriptor.

     ENOBUFS        Insufficient resources were available in  the
                    system to perform the operation.

SEE ALSO
     bind(), getpeername(), socket()
```

B.1.8 getsockopt()

```
NAME
     getsockopt, setsockopt - get and set options on sockets

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

     success =  getsockopt(s, level, optname, optval, optlen)
     D0                    D0 D1     D2       A0      A1

     long getsockopt(long, long, long, caddr_t, long *);

     success =  setsockopt(s, level, optname, optval, optlen)
     D0                    D0 D1     D2       A0      D3

     long setsockopt(long, long, long, caddr_t, long);

FUNCTION
     getsockopt() and setsockopt() manipulate options  associated
     with  a socket.  Options may exist at multiple protocol lev-
     els; they are always present  at  the  uppermost  ``socket''
     level.

     When manipulating socket options  the  level  at  which  the
     option resides and the name of the option must be specified.
     To manipulate options at  the  ``socket''  level,  level  is
     specified as SOL_SOCKET.  To manipulate options at any other
     level the protocol number of the appropriate  protocol  con-
     trolling  the  option is supplied.  For example, to indicate
     that an option is to be interpreted  by  the  TCP  protocol,
     level  should  be  set  to  the  protocol number of TCP.

     The parameters optval and optlen are used to  access  option
     values  for  setsockopt().  For getsockopt() they identify a
     buffer in which the value for the requested option(s) are to
     be  returned.   For  getsockopt(),  optlen is a value-result
     parameter, initially  containing  the  size  of  the  buffer
```

```
pointed to by optval, and modified on return to indicate the
actual size of the value returned.  If no option value is to
be supplied or returned, optval may be supplied as 0.

optname and any specified options are  passed  uninterpreted
to  the appropriate protocol module for interpretation.  The
include  file  <sys/socket.h>   contains   definitions   for
``socket'' level options, described below.  Options at other
protocol  levels  vary  in  format  and  name.

Most socket-level options take an int parameter for  optval.
For setsockopt(), the parameter should be non-zero to enable
a boolean option, or zero if the option is to  be  disabled.

SO_LINGER  uses  a  struct  linger  parameter,  defined  in
<sys/socket.h>, which specifies the  desired  state  of  the
option and the linger interval (see below).

The following options are recognized at  the  socket  level.
Except  as noted, each may be examined with getsockopt() and
set with setsockopt().

     SO_DEBUG          - toggle   recording   of   debugging
                         information
     SO_REUSEADDR      - toggle local address reuse
     SO_KEEPALIVE      - toggle keep connections alive
     SO_DONTROUTE      - toggle routing bypass for  outgoing
                         messages
     SO_LINGER         - linger on close if data present
     SO_BROADCAST      - toggle   permission   to   transmit
                         broadcast messages
     SO_OOBINLINE      - toggle  reception  of   out-of-band
                         data in band
     SO_SNDBUF         - set buffer size for output
     SO_RCVBUF         - set buffer size for input
     SO_TYPE           - get the type  of  the  socket  (get
                         only)
     SO_ERROR          - get and clear error on  the  socket
                         (get only)
```

SO_DEBUG enables debugging in the underlying protocol modules. SO_REUSEADDR indicates that the rules used in validating addresses supplied in a bind() call should allow reuse of local addresses. SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken. If the process is waiting in select() when the connection is broken, select() returns true for any read or write events selected for the socket. SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER controls the action taken when unsent messags are queued on socket and a CloseSocket() is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the process on the close attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, in seconds, termed the linger interval, is specified in the set- sockopt() call when SO_LINGER is requested). If SO_LINGER is disabled and a CloseSocket() is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

The option SO_BROADCAST requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of the system. With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with recv() or read() calls without the MSG_OOB flag. SO_SNDBUF and SO_RCVBUF are options to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. The system places an absolute limit on these values.

Finally, SO_TYPE and SO_ERROR are options used only with getsockopt(). SO_TYPE returns the type of the socket, such as SOCK_STREAM; it is useful for servers that inherit sockets on startup. SO_ERROR returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

RETURN VALUES

0 - on success.

-1 - on failure and set errno to indicate the error.

ERRORS

EBADF - s is not a valid descriptor.

ENOPROTOOPT - The option is unknown at the level indicated.

SEE ALSO

IoctlSocket(), socket()

BUGS

Several of the socket options should be handled at lower levels of the system.

B.1.9 IoctlSocket()

```
NAME
     IoctlSocket - control sockets

SYNOPSIS

     #include <sys/types.h>
     #include <sys/ioctl.h>

     value = IoctlSocket(fd, request, arg)
     D0              D0  D1       A0

     long IoctlSocket(long, long, caddr_t);

FUNCTION
     IoctlSocket() performs a special function on the object referred
     to by the open  socket descriptor fd. Note: the setsockopt()
     call (see getsockopt()) is the primary  method for operating
     on sockets  as such, rather than on the underlying  protocol
     or network interface.

     For most IoctlSocket() functions, arg is a pointer to data to
     be used by the  function  or to be filled in by the function.
     Other functions may ignore arg or may treat it directly as a
     data item; they may, for example, be passed an int value.

     The following requests are supported:


     FIOASYNC            The argument is a  pointer  to  a  long.
                         Set  or  clear asynchronous I/O.  If the
                         value of that  long is  a  1  (one)  the
                         descriptor  is set for asynchronous I/O.
                         If the value of that long is a  0 (zero)
                         the  descriptor is cleared for asynchro-
                         nous I/O.

     FIOCLEX
```

```
FIONCLEX        Ignored, no use for close-on-exec flag
                in Amiga.

FIOGETOWN
SIOCGPGRP       The argument is pointer to struct Task*.
                Set the value of that pointer to the
                Task that  is  receiving SIGIO or SIGURG
                signals for  the  socket  referred to by
                the descriptor passed to IoctlSocket().

FIONBIO         The argument is a  pointer  to  a  long.
                Set  or  clear non-blocking I/O.  If the
                value of that  long is  a  1  (one)  the
                descriptor  is set for non-blocking I/O.
                If the value of that long is a  0 (zero)
                the  descriptor  is  cleared  for  non-
                blocking I/O.

FIONREAD        The argument is a  pointer  to  a  long.
                Set the value of that long to the number
                of immediately readable characters  from
                the socket fd.

FIOSETOWN
SIOCSPGRP       The argument is pointer to struct Task*,
                pointer  to  the task  that will subseq-
                uently receive  SIGIO or  SIGURG signals
                for  the  socket  referred  to  by  the
                descriptor passed.

SIOCCATMARK     The argument is a pointer to a long.
                Set the value of that long to 1 if the
                read pointer for the socket referred to
                by the descriptor passed to
                IoctlSocket() points to a mark in the
                data stream for an out-of-band message,
                and to 0 if it does not point to a mark.
```

```
RETURN VALUES
     IoctlSocket() returns 0 on success for most requests.   Some
     specialized requests may return non-zero values on success; On
     failure,  IoctlSocket() returns -1 and sets errno to indicate
     the error.

ERRORS
     EBADF          fd is not a valid descriptor.

     EINVAL         request or arg is not valid.

     IoctlSocket() will also fail if the object on which the function
     is  being performed detects an error. In this case, an error
     code specific  to  the  object  and  the  function  will  be
     returned.

SEE ALSO
     getsockopt(), SetSocketSignals(), setsockopt()
```

B.1.10 listen()

```
NAME
     listen - listen for connections on a socket

SYNOPSIS
     success = listen(s, backlog)
     D0               D0 D1

     long listen(long, long);

FUNCTION
     To accept  connections,  a  socket  is  first  created  with
     socket(),  a  backlog for incoming connections is specified
     with listen() and then the  connections  are  accepted  with
     accept().   The  listen()  call  applies only to socket of
     type SOCK_STREAM.

     The backlog parameter defines the maximum length  the  queue
     of pending connections may grow to.  If a connection request
     arrives with the queue full the client will receive an error
     with an indication of ECONNREFUSED.

RETURN VALUES
      0    on success.

     -1    on failure and sets errno to indicate the error.

ERRORS
     EBADF             - s is not a valid descriptor.

     EOPNOTSUPP        - The socket is not of a  type  that  sup-
                         ports listen().

SEE ALSO
     accept(), connect(), socket()

BUGS
     The backlog is currently limited (silently) to 5.
```

B.1.11 recv()

```
NAME
     recv, recvfrom, - receive a message from a socket

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

    nbytes = recv(s, buf, len, flags)
    D0            D0 A0   D1   D2

    long recv(long, char *, long, long);

    nbytes = recvfrom(s, buf, len, flags, from, fromlen)
    D0                D0 A0   D1   D2     A1    A2

    long recvfrom(long, char *, long, long,
                   struct sockaddr *, long *);

FUNCTION
     s is a socket created with socket().  recv() and recvfrom(),
     are used  to  receive messages from  another socket.  recv()
     may  be  used  only on a connected socket  (see  connect()),
     while  recvfrom() may be used  to receive data  on a  socket
     whether it is in a connected state or not.

     If from is not a NULL pointer, the  source  address  of  the
     message  is filled in.  fromlen is a value-result parameter,
     initialized to the size of the buffer associated with  from,
     and  modified  on  return to indicate the actual size of the
     address  stored  there.   The  length  of  the  message   is
     returned.   If  a message is too long to fit in the supplied
     buffer, excess bytes may be discarded depending on the  type
     of socket the message is received from (see socket()).

     If no messages are available at the socket, the receive call
     waits  for  a  message  to arrive, unless the socket is non-
     blocking (see IoctlSocket()) in which case -1  is  returned
```

```
        with the external variable errno set to EWOULDBLOCK.

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

        The flags parameter is formed by ORing one or  more  of  the
        following:

        MSG_OOB      - Read any "out-of-band" data  present  on  the
                       socket,  rather  than  the  regular "in-band"
                       data.

        MSG_PEEK     - "Peek" at the data present on the socket; the
                       data  are returned, but not consumed, so that
                       a subsequent receive operation will  see  the
                       same data.

   RETURN VALUES
        These calls return the number of bytes received, or -1 if an
        error occurred.

   ERRORS
        EBADF             - s is an invalid descriptor.

        EINTR             - The operation was interrupted by a break
                            signal.

        EWOULDBLOCK       - The socket is  marked  non-blocking  and
                            the requested operation would block.

   SEE ALSO
        connect(), getsockopt(), IoctlSocket(), select(), send(),
        SetSocketSignals(), socket()
```

B.1.12 select()

```
NAME
     select -- synchronous I/O multiplexing (stub/inline function)
     WaitSelect -- select() with Amiga Wait() function.

SYNOPSIS
     #include <sys/types.h>
     #include <sys/time.h>

     n = select (nfds, readfds, writefds, exceptfds, timeout)

     long select(long, fd_set *, fd_set *, fd_set *,
                 struct timeval *);

     n = WaitSelect (nfds, readfds, writefds, exceptfds, timeout,
     D0              D0    A0        A1        A2         A3
                 sigmp)
                 D1

     long WaitSelect(long, fd_set *, fd_set *, fd_set *,
                     struct timeval *, long *);

     FD_SET (fd, &fdset)
     FD_CLR (fd, &fdset)
     FD_ISSET (fd, &fdset)
     FD_ZERO (&fdset)
     long fd;
     fd_set fdset;

DESCRIPTION
     select() examines the socket descriptor sets whose addresses
     are passed in readfds,  writefds,  and exceptfds  to  see if
     some of their descriptors are ready  for reading,  ready for
     writing,  or have an exceptional condition pending.  nfds is
     the  number  of bits to be checked in  each  bit  mask  that
     represent a file descriptor; the descriptors from 0  through
     (nfds - 1) in the descriptor sets  are examined.  On return,
     select()  replaces  the  given descriptor sets  with subsets
```

```
     consisting of  those descriptors  that  are  ready  for  the
     requested operation.  The total number  of ready descriptors
     in all the sets is returned.

     WaitSelect() also takes a signal mask which is waited during
     normal select() operation. If one of these singals is recei-
     ved,  WaitSelect() returns  and has  re-set  the signal mask
     to return those signals that  have arrived.  Normal select()
     return values are returned.

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

     If timeout is not a NULL pointer,  it  specifies  a  maximum
     interval  to wait for the selection to complete.  If timeout
     is a NULL  pointer,  the  select  blocks  indefinitely.   To
     effect  a  poll,  the  timeout argument should be a non-NULL
     pointer, pointing to a zero-valued timeval structure.

     Any of readfds, writefds, and exceptfds may be given as NULL
     pointers if no descriptors are of interest.

     Selecting true for reading on a socket descriptor upon which
     a  listen() call has been performed indicates that a subse-
     quent accept() call on that descriptor will not block.

RETURN VALUES
     select() returns a non-negative value on success. A positive
     value indicates the number of ready descriptors in the
     descriptor sets. 0 indicates that the time limit referred to
```

```
     by timeout expired or that the operation was interrupted
     either by a break signal or by arrival of a signal specified
     in *sigmp. On failure, select() returns -1, sets errno to
     indicate the error, and the descriptor sets are not changed.

ERRORS
     EBADF         - One  of  the  descriptor  sets  specified  an
                     invalid descriptor.

     EINTR         - one of the signals in SIGINTR  mask (see Set-
                     SocketSignals())   is  set  and  it  was  not
                     requested in WaitSelect() call.

     EINVAL        - A component of the pointed-to time  limit  is
                     outside  the  acceptable range: t_sec must be
                     between 0 and 10^8, inclusive. t_usec must be
                     greater  than  or  equal  to 0, and less than
                     10^6.

SEE ALSO
     accept(),  connect(), getdtablesize(), listen(), recv(),
     send(), SetDTableSize(), SetSocketSignals()

NOTES
     Under rare  circumstances,  select()  may  indicate  that  a
     descriptor  is  ready for writing when in fact an attempt to
     write would block.  This  can  happen  if  system  resources
     necessary  for  a  write are exhausted or otherwise unavail-
     able.  If an application deems it critical that writes to  a
     file  descriptor not block, it should set the descriptor for
     non-blocking I/O using the FIOASYNC request to IoctlSocket().

     Default   system   limit  for  open  socket  descriptors  is
     currently  64. However,  in  order  to accommodate  programs
     which might  potentially  use  a larger number of open files
     with select, it is possible  to  increase this size within a
     program  by  providing  a  larger definition  of  FD_SETSIZE
     before    the    inclusion    of    <sys/types.h>    and   use
     SetDTableSize(FD_SETSIZE) call directly after OpenLibrary().
```

```
BUGS
    select() should probably return the time remaining from  the
    original  timeout,  if  any,  by modifying the time value in
    place.  This may be implemented in future  versions  of  the
    system.   Thus,  it  is  unwise  to  assume that the timeout
    pointer will be unmodified by the select() call.
```

B.1.13 send()

NAME

send, sendto - send a message from a socket

SYNOPSIS

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

nbytes = send(s, msg, len, flags)
D0            D0 A0   D1   D2

int send(int, char *, int, int);

nbytes = sendto(s, msg, len, flags, to, tolen)
D0              D0 A0   D1   D2     A1  D3

int send(int, char *, int, int, struct sockaddr *, int);
```

FUNCTION

s is a socket created with socket(). send() and sendto() are used to transmit a message to another socket. send() may be used only when the socket is in a connected state, while sendto() may be used at any time.

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

No indication of failure to deliver is implicit in a send(). Return values of -1 indicate some locally detected errors.

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

The flags parameter is formed by ORing one or more of the following:

MSG_OOB - Send ``out-of-band'' data on sockets that support this notion. The underlying protocol must also support ``out-of-band'' data. Currently, only SOCK_STREAM sockets created in the AF_INET address family support out-of-band data.

MSG_DONTROUTE - The SO_DONTROUTE option is turned on for the duration of the operation. This is usually used only by diagnostic or routing programs.

RETURN VALUES

On success, these functions return the number of bytes sent. On failure, they return -1 and set errno to indicate the error.

ERRORS

EBADF - s is an invalid descriptor.

EINTR - The operation was interrupted by a break signal.

EINVAL - len is not the size of a valid address for the specified address family.

EMSGSIZE - The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.

ENOBUFS - The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.

```
    ENOBUFS             - The output queue for a network interface
                          was full.  This generally indicates that
                          the interface has stopped  sending,  but
                          may be caused by transient congestion.

    EWOULDBLOCK         - The socket is  marked  non-blocking  and
                          the requested operation would block.

SEE ALSO
    connect(), getsockopt(), recv(), select(), socket()
```

B.1.14 shutdown()

```
NAME
     shutdown - shut down part of a full-duplex connection

SYNOPSIS
     success = shutdown(s, how)
     D0                 D0 D1

     long shutdown(long, long);

DESCRIPTION
     The shutdown() call causes all or part of a full-duplex con-
     nection on the socket associated with s to be shut down.  If
     how is 0, then further receives will be disallowed.  If  how
     is  1,  then further sends will be disallowed.  If how is 2,
     then further sends and receives will be disallowed.

RETURN VALUES
      0 - on success.

     -1 - on failure and sets errno to indicate the error.

ERRORS
     EBADF        - s is not a valid descriptor.

     ENOTCONN     - The specified socket is not connected.

SEE ALSO
     connect(), socket()

BUGS
     The how values should be defined constants.
```

B.1.15 socket()

```
NAME
     socket - create an endpoint for communication

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

     s = socket(domain, type, protocol)
     D0         D0      D1    D2

     long socket(long, long, long);

FUNCTION
     socket() creates an endpoint for communication and returns a
     descriptor.

     The domain parameter specifies a communications domain
     within which communication will take place; this selects the
     protocol family which should be used. The protocol family
     generally is the same as the address family for the
     addresses supplied in later operations on the socket. These
     families are defined in the include file <sys/socket.h>.
     The currently understood formats are

             PF_INET - (ARPA Internet protocols)

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

             SOCK_STREAM
             SOCK_DGRAM
             SOCK_RAW

     A SOCK_STREAM type provides sequenced, reliable, two-way
     connection based byte streams.  An out-of-band data
     transmission mechanism may be supported.  A SOCK_DGRAM
     socket supports datagrams (connectionless, unreliable mes-
```

sages of a fixed (typically small) maximum length). SOCK_RAW sockets provide access to internal network interfaces.

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

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

The communications protocols used to implement a SOCK_STREAM insure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with -1 returns and with ETIMEDOUT as the specific error code (see Errno()). The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other activity.

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

The operation of sockets is controlled by socket level

```
     options.  These  options  are defined in the file socket.h.
     getsockopt() and  setsockopt()  are  used  to  get  and  set
     options, respectively.

RETURN VALUES
     socket() returns a non-negative descriptor on  success.   On
     failure, it returns -1 and sets errno to indicate the error.

ERRORS
     EACCES          - Permission to create  a  socket  of  the
                       specified   type   and/or   protocol  is
                       denied.

     EMFILE          - The per-process descriptor table is
                       full.

     ENOBUFS         - Insufficient buffer space is available.
                       The socket cannot be created until suf-
                       ficient resources are freed.

     EPROTONOSUPPORT - The protocol type or the specified  pro-
                       tocol is not supported within this
                       domain.

     EPROTOTYPE      - The protocol is the wrong type for the
                       socket.

SEE ALSO
     accept(), bind(), CloseSocket(), connect(), getsockname(),
     getsockopt(), IoctlSocket(), listen(), recv(), select(),
     send(), shutdown(), WaitSelect()
```

B.2 Other BSD Functions Related to Sockets

B.2.1 getdtablesize()

```
NAME
     getdtablesize - get socket descriptor table size

SYNOPSIS

     nfds = getdtablesize()
     D0

     long getdtablesize(void);

FUNCTION
     Return value of maximum  number of open socket  descriptors.
     Larger  socket  descriptor  table  can   be  allocated  with
     SetDTableSize() call.

SEE ALSO
     SetDTableSize()
```

B.2.2 Syslog()

```
NAME
     syslog - write message to AmiTCP/IP log.

SYNOPSIS
     #include <syslog.h>

     void syslog(unsigned long level, char * format, ...);

     Syslog(level, format, ap)
            D0     A0      A1

     VOID Syslog(unsigned long, const char *, va_list);


FUNCTION
     Writes the message given as format string and arguments
     (printf-style) both to the log file and to the console,
     execpt if the level is LOG_EMERG, which is used by panic(),
     in which case only the log file is used since panic()
     generates a User Request.

     The level is selected from an ordered list:

          LOG_EMERG           A panic condition.

          LOG_ALERT           A condition that should be
                              corrected immediately, such as a
                              corrupted system database.

          LOG_CRIT            Critical conditions, such  as  hard
                              device errors.

          LOG_ERR             Errors.

          LOG_WARNING         Warning messages.

          LOG_NOTICE          Conditions that are not error  con-
```

```
                    ditions,  but that may require spe-
                    cial handling.

     LOG_INFO       Informational messages.

     LOG_DEBUG      Messages that  contain  information
                    normally of use only when debugging
                    a program.

INPUTS
     Level     - indicates the type of the message. The levels
                 are defined in sys/syslog.h and listed above.

     format    - This is a printf-style format string as defined
                 in exec.library/RawDoFmt().

     arguments - as in printf().

     ap        - pointer to an array of arguments.

RESULT
     Returns no value.

EXAMPLES
     To  log a  message at priority  LOG_INFO, it would  make the
     following call to syslog:

          syslog(LOG_INFO,  "Connection from host %s",
                 CallingHost);

NOTES
     As Exec RawDoFmt() used to do formatting expects by default
     short (16 bit long) integers you should use the `l'-modifier
     when appopriate. See your compiler documentation about how
     it passes arguments on a vararg list.

     This function is callable from interrupts.

BUGS
```

```
        Because there is a limited number of internal messages used
        by the logging system, some log messages may get lost if a
        high priority task or interrupt handler sends many messages
        in succession. If this happens, the next log message tells
        the fact.

   SEE ALSO
       exec.library/RawDoFmt()
```

B.3 Network Data and Address Manipulation

B.3.1 inet_addr()

```
NAME
     inet_addr,  inet_network,  Inet_MakeAddr,  Inet_LnaOf,
     Inet_NetOf, Inet_NtoA - Internet address manipulation

     inet_makeaddr, inet_lnaof, inet_netof,
     inet_ntoa -- inline/stub functions to handle structure arguments

SYNOPSIS
     #include <netinet/in.h>

     addr = inet_addr(cp)
     D0               A0

     unsigned long inet_addr(char *);

     net = inet_network(cp)
     D0                  A0

     unsigned long inet_network(char *);

     in_addr = Inet_MakeAddr(net, lna)
     D0                      D0   D1

     unsigned long Inet_MakeAddr(long, long);

     lna = Inet_LnaOf(in)
     D0               D0

     long Inet_LnaOf(unsigned long);

     net = Inet_NetOf(in)
     D0               D0

     long Inet_NetOf(unsigned long);
```

```
    addr = Inet_NtoA(in)
    D0               D0

    char * Inet_NtoA(unsigned long);


    in_addr = inet_makeaddr(net, lna)

    struct in_addr inet_makeaddr(long, long);

    lna = inet_lnaof(in)

    int inet_lnaof(struct in_addr);

    net = inet_netof(in)

    int inet_netof(struct in_addr);

    addr = inet_ntoa(in)

    char * inet_ntoa(struct in_addr);

IMPLEMENTATION NOTE
    Return  value  of  Inet_MakeAddr()  and  argument  types  of
    Inet_LnaOf(), Inet_NetOf() and Inet_NtoA() are longs instead
    of  struct  in_addr.  The original behaviour  is achieved by
    using  included  stub  functions (lower case function names)
    which handle structure arguments.

DESCRIPTION
    The routines inet_addr() and inet_network()  each  interpret
    character  strings  representing  numbers  expressed  in the
    Internet standard `.' notation, returning  numbers  suitable
    for  use as Internet addresses and Internet network numbers,
    respectively.  The routine inet_makeaddr() takes an Internet
    network number and a local network address and constructs an
    Internet address from it.   The  routines  inet_netof()  and
    inet_lnaof()  break apart Internet host addresses, returning
    the network number and local network address  part,  respec-
```

tively.

The routine inet_ntoa() returns a pointer to a string in the base 256 notation ''d.d.d.d'' described below.

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

INTERNET ADDRESSES

Values specified using the '.' notation take one of the

following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address. Note: when an Internet address is viewed as a 32-bit integer quantity on little endian systems, the bytes referred to above appear as d.c.b.a. bytes are ordered from right to left.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as "128.net.host".

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as "net.host".

When only one part is given, the value is stored directly in

```
    the network address without any byte rearrangement.

    All numbers supplied as ``parts'' in a `.' notation  may  be
    decimal,  octal,  or  hexadecimal,  as  specified  in  the C
    language (that is, a leading 0x or 0X  implies  hexadecimal;
    otherwise,  a leading 0 implies octal; otherwise, the number
    is interpreted as decimal).

RETURN VALUE
    The value -1 is returned by inet_addr()  and  inet_network()
    for malformed requests.

BUGS
    The problem of host byte ordering versus network byte order-
    ing  is  confusing.  A simple way to specify Class C network
    addresses in a manner similar to that for Class B and  Class
    A is needed.

    The return value from inet_ntoa() points to static buffer
    which  is  overwritten  in  each inet_ntoa() call.
```

B.4 Network, Protocol and Service Queries

B.4.1 gethostbyname()

```
NAME
     gethostbyname, gethostbyaddr  - get network host entry

SYNOPSIS
     #include <sys/types.h>
     #include <sys/socket.h>
     #include <netdb.h>

     hostent = gethostbyname(name)
     D0                      A0

     struct hostent *gethostbyname(char *);

     hostent = gethostbyaddr(addr, len, type)
     D0                      A0    D0   D1

     struct hostent *gethostbyaddr(caddr_t, LONG, LONG);


DESCRIPTION
     gethostbyname() and gethostbyaddr() both return a pointer
     to an object with the following structure containing the
     data received from a name server or the broken-out fields
     of a line in netdb configuration file.  In the case of
     gethostbyaddr(), addr is a pointer to the binary format
     address of length len (not a character string) and type is
     an address family as defined in <sys/socket.h>.

       struct hostent {
         char *h_name;       /* official name of host */
         char **h_aliases;   /* alias list */
         int  h_addrtype;    /* address type */
         int  h_length;      /* length of address */
         char **h_addr_list; /* list of addresses from name server */
       };
```

The members of this structure are:

h_name	Official name of the host.
h_aliases	A zero terminated array of alternate names for the host.
h_addrtype	The type of address being returned; currently always AF_INET.
h_length	The length, in bytes, of the address.
h_addr_list	A pointer to a list of network addresses for the named host. Host addresses are returned in network byte order.

DIAGNOSTICS

A NULL pointer is returned if no matching entry was found or error occured.

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet address format is currently understood.

SEE ALSO

AmiTCP/IP configuration

B.4.2 getnetbyname()

```
NAME
     getnetbyname, getnetbyaddr - get network entry

SYNOPSIS
     #include <netdb.h>

     netent = getnetbyname(name)
     D0                    A0

     struct netent *getnetbyname(char *);

     netent = getnetbyaddr(net, type)
     D0                    D0   D1

     struct netent *getnetbyaddr(long, long);

DESCRIPTION
     getnetbyname(), and getnetbyaddr() both return  a  pointer to
     an  object  with  the  following  structure  containing  the
     broken-out fields of a line in netdb configuration file.

       struct netent {
         char *n_name;       /* official name of net */
         char **n_aliases;   /* alias list */
         int  n_addrtype;    /* net number type */
         long n_net;         /* net number */
       };

     The members of this structure are:

     n_name              The official name of the network.

     n_aliases           A  zero  terminated  list  of  alternate
                         names for the network.

     n_addrtype          The type of the network number returned;
                         currently only AF_INET.
```

```
    n_net               The network number.  Network numbers are
                        returned in machine byte order.

    Network numbers are supplied in host order.

    Type specifies the address type to use, currently only
    AF_INET is supported.

DIAGNOSTICS
    A NULL pointer is returned if no matching entry was found or
    error occured.

BUGS
    All information is contained in a static area so it must  be
    copied if it is to be saved.

    Only Internet network numbers are currently understood.

SEE ALSO
    AmiTCP/IP configuration
```

B.4.3 getprotobyname()

```
NAME
     getprotobyname, getprotobynumber - get protocol entry

SYNOPSIS
     #include <netdb.h>

     protoent = getprotobyname(name)
     D0                        A0

     struct protoent *getprotobyname(char *);

     protoent = getprotobynumber(proto)
     D0                          D0

     struct protoent *getprotobynumber(long);

DESCRIPTION
     getprotobyname() and getprotobynumber() both return a pointer
     to  an  object with the  following structure  containing  the
     broken-out fields of a line in netdb configuration file

       struct    protoent {
         char *p_name;       /* official name of protocol */
         char **p_aliases;   /* alias list */
         int  p_proto;       /* protocol number */
      };

     The members of this structure are:

     p_name              The official name of the protocol.
     p_aliases           A  zero  terminated  list  of  alternate
                         names for the protocol.
     p_proto             The protocol number.


DIAGNOSTICS
     A NULL pointer is returned if no matching entry was found or
```

```
     error occured.

BUGS
     All information is contained in a static area so it must  be
     copied  if  it  is to be saved.  Only the Internet protocols
     are currently understood.

SEE ALSO
     AmiTCP/IP configuration
```

B.4.4 getservbyname()

```
NAME
     getservbyname, getservbyport - get service entry

SYNOPSIS
     #include <netdb.h>

     servent = getservbyname(name, proto)
     D0                      A0    A1

     struct servent *getservbyname(char *, char *)

     servent = getservbyport(port, proto)
     D0                      D0    A0

     struct servent *getservbyport(long, char *);

DESCRIPTION
     getservbyname() and getservbyport() both return a pointer  to
     an  object  with  the  following  structure  containing  the
     broken-out fields of a line in netdb configuration file.

       struct    servent {
         char *s_name;       /* official name of service */
         char **s_aliases;   /* alias list */
         int  s_port;        /* port service resides at */
         char *s_proto;      /* protocol to use */
       };

     The members of this structure are:
          s_name              The official name of the service.
          s_aliases           A zero terminated list of alternate
                              names for the service.
          s_port              The port number at which  the  ser-
                              vice  resides.   Port  numbers  are
                              returned  in  network  short   byte
                              order.
          s_proto             The name of  the  protocol  to  use
```

```
                        when contacting the service.

        The proto argument specifies the protocol for which to the
        sercive is to use. It is a normal C string, e.g. "tcp" or
        "udp".

   DIAGNOSTICS
        A NULL pointer is returned if no matching entry was found or
        error occured.

   BUGS
        All information is contained in a static area so it must  be
        copied  if it is to be saved.  Expecting port numbers to fit
        in a 32 bit quantity is probably naive.

   SEE ALSO
        AmiTCP/IP configuration
```

B.5 AmiTCP/IP Specific Extensions

B.5.1 Errno()

```
NAME
     Errno - get error value after unsuccessful function call

SYNOPSIS
     errno = Errno()
     D0

     LONG Errno(VOID);

FUNCTION
     When  some  function  in  socket  library  return  an  error
     condition value, they also set a specific error value.  This
     error value can be extracted by this function.

RESULT
     Error value  indicating  the error on  last failure  of some
     socket function call.

NOTES
     Return  value  of  Errno()  is not changed  after successful
     function so so it cannot be used to determine success of any
     function call  of this library.  Also, another function call
     to this  library may change  the return value of  Errno() so
     use it right after error occurred.

SEE ALSO
     SetErrnoPtr()
```

B.5.2 ObtainSocket()

```
NAME
     ObtainSocket - get a socket from AmiTCP/IP socket list

SYNOPSIS
     s = ObtainSocket(id, domain, type, protocol)
     D0               D0  D1      D2    D3

     LONG ObtainSocket(LONG, LONG, LONG, LONG);

FUNCTION
     When one task wants to give  a socket to  an another one, it
     releases it (with a key value) to a special socket list held
     by  AmiTCP/IP.  This  function  requests  that  socket  and
     receives it if id and other parameters match.

INPUTS
     id       - a key value given by the socket donator.
     domain   - see documentation of socket().
     type     - see documentation of socket().
     protocol - see documentation of socket().

RESULT
     Non negative socket descriptor on success. On failure, -1 is
     returned and the errno is set to indicate the error.

ERRORS
     EMFILE          - The per-process descriptor table is
                       full.

     EPROTONOSUPPORT - The protocol type or the specified  pro-
                       tocol is not supported within this
                       domain.

     EPROTOTYPE      - The protocol is the wrong type for the
                       socket.

     EWOULDBLOCK     - Matching socket is not found.
```

```
SEE ALSO
     ReleaseCopyOfSocket(), ReleaseSocket(), socket()
```

B.5.3 ReleaseCopyOfSocket()

```
NAME
     ReleaseCopyOfSocket - copy given socket to AmiTCP/IP socket list.

SYNOPSIS
     id = ReleaseCopyOfSocket(fd, id)
     D0                       D0  D1

     LONG ReleaseCopyOfSocket(LONG, LONG);

FUNCTION
     Make a new reference to a given socket (pointed by its descriptor)
     and release it to the socket list held by AmiTCP/IP.

INPUTS
     fd - descriptor of the socket to release.

     id - the key value to identify use of this socket. It can be
          unique or not, depending on its  value.  If id value is
          between 0  and  65535,  inclusively,  it is  considered
          nonunique  and it can  be  used as a port  number,  for
          example.   If  id is greater  than  65535 and less than
          2^31) it  must be unique in currently  held sockets  in
          AmiTCP/IP socket  list,  Otherwise  an  error  will  be
          returned  and  socket  is  not  released.   If  id  ==
          UNIQUE_ID (defined in <sys/socket.h>) an unique id will
          be generated.

RESULT
     id - -1 in case of error and the key value of the socket put
          in the list. Most useful when an unique id is generated
          by this routine.

ERRORS
     EINVAL - Requested unique id is already used.

     ENOMEM - Needed memory couldn't be allocated.
```

```
NOTE
     The socket descriptor is not deallocated.

SEE ALSO
     ObtainSocket(), ReleaseSocket()
```

B.5.4 ReleaseSocket()

```
NAME
     ReleaseSocket - release given socket to AmiTCP/IP socket list.

SYNOPSIS
     id = ReleaseSocket(fd, id)
     D0                 D0  D1

     LONG ReleaseSocket(LONG, LONG);

FUNCTION
     Release the reference of given socket (via  its  descriptor)
     and move the socket to the  socket  list held by  AmiTCP/IP.
     The socket descriptor is deallocated in this procedure.

INPUTS
     fd - descriptor of the socket to release.

     id - the key value to identify use of this socket. It can be
          unique or not, depending on its  value.  If id value is
          between 0  and  65535,  inclusively,  it is  considered
          nonunique  and it can  be  used as a port  number,  for
          example.   If  id is greater  than  65535 and less than
          2^31) it  must be unique in currently  held sockets  in
          AmiTCP/IP socket  list,  Otherwise  an  error  will  be
          returned  and  socket  is  not  released.   If  id  ==
          UNIQUE_ID (defined in <sys/socket.h>) an unique id will
          be generated.

RESULT
     id - -1 in case of error and the key value of the socket put
          in the list. Most useful when an unique id is generated
          by this routine.

ERRORS
     EINVAL - Requested unique id is already used.

     ENOMEM - Needed memory couldn't be allocated.
```

```
SEE ALSO
     ObtainSocket(), ReleaseCopyOfSocket()
```

B.5.5 SetDTableSize()

```
NAME
     SetDTableSize - set socket descriptor table size of the base

SYNOPSIS
     newsize = SetDTableSize(size)
     D0                      D0

     LONG SetDTableSize(UWORD);

FUNCTION
     This  function increases  the  descriptor table size  inside
     library base so more sockets can be open at the same time.

INPUT
     size - the new size of the desctiptor table.

RESULT
     newsize - the new size of the descriptor table. Note that
     this can be less than requested if an error occured.

WARNING
     If the size of fd_set is not adjusted to store the increased
     space needed for  new  socket descriptors some other  memory
     will  be spilled.  Change  the  value of  FD_SETSIZE  before
     including  any  socket  include  files  and  don't  increase
     descriptor  table  to  greater  than  the  new  value  of
     FD_SETSIZE.

SEE ALSO
     getdtablesize(), select()
```

B.5.6 SetErrnoPtr()

```
NAME
     SetErrnoPtr - set new place where the error value will be written

SYNOPSIS
     SetErrnoPtr(ptr, size)
                 A0   D0

     VOID SetErrnoPtr(VOID *, UBYTE);

FUNCTION
     This functions allows caller to redirect error variable inside
     scope of  caller task.  Usually this is  used to make  task's
     global variable errno as error variable.

INPUTS
     ptr     - pointer to error variable that is to be modified on
               every error condition on this library function.
     size    - size of the error variable.

EXAMPLE
     #include <errno.h>

     struct Library;
     struct Library * SocketBase = NULL;

     int main(void)
     {
        ...
       if ((SocketBase = OpenLibrary("bsdsocket.library", 2))
           != NULL) {
         SetErrnoPtr(&errno, sizeof(errno));
        ...
       }
     }

NOTES
     Be sure that this new error variable exists until library base
```

```
     is finally closed or SetErrnoPtr() is called again for another
     variable.

SEE ALSO
     Errno()
```

B.5.7 SetSocketSignals()

```
NAME
     SetSocketSignals - inform AmiTCP/IP of INTR, IO and URG signals

SYNOPSIS
     SetSocketSignals(sigintrmask, sigiomask, sigurgmask)
                      D0           D1         D2

     VOID SetSocketSignals(ULONG, ULONG, ULONG);

FUNCTION
     SetSocketSignals()  tells the AmiTCP/IP which  signal  masks
     corresponds UNIX SIGINT, SIGIO and SIGURG signals to be used
     in  this  implementation.  sigintrmask mask  is  used  to
     determine  which  Amiga signals  interrupt  blocking library
     calls.  sigio-  and sigurgmasks  are sent when asynchronous
     notification of socket events is done and  when  out-of-band
     data arrives, respectively.

     Note that  the supplied  values write over old ones. If this
     function is used and CTRL-C is still wanted to interrupt the
     calls (the  default behaviour), the value BREAKF_CTRL_C must
     be explicitly given.

SEE ALSO
     IoctlSocket(), recv(), send(), WaitSelect()
```

Appendix C

AmiTCP/IP Network Link Library

This appendix describes the functions located in the **net.lib**.

Table of Contents

C.1 net.lib Functions

C.1.1 autoinit

```
NAME
    autoinit - SAS C Autoinitialization Functions

SYNOPSIS
    _STIopenSockets()

    void _STIopenSockets(void)

    _STDcloseSockets()

    void _STDcloseSockets(void)

FUNCTION
    These functions open and close the bsdsocket.library at the
    startup and exit of the program, respectively. For a
    program to use these functions, it must be linked with
    netlib:net.lib.

    If the library can be opened, the _STIopenSockets() calls
    bsdsocket.library function SetErrnoPtr() to tell the
    library the address and the size of the errno variable of
    the calling program.

NOTES
    _STIopenSockets() also checks that the system version is at
    least 37. It puts up a requester if the bsdsocket.library
    is not found or is of wrong version.

    The autoinitialization and autotermination functions are
    features specific to the SAS C6. However, these functions
    can be used with other (ANSI) C compilers, too. Example
    follows:

    \* at start of main() *\

    atexit(_STDcloseSockets);
    _STDopenSockets();
```

```
BUGS

SEE ALSO
    bsdsocket.library/SetErrnoPtr(),
    SAS/C 6 User's Guide p. 145 for details of
    autoinitialization and autotermination functions.
```

C.1.2 autoinitd

```
NAME
    autoinitd - SAS C Autoinitialization Functions for Daemons

SYNOPSIS
    void _STIopenSockets(void);
    void _STDcloseSockets(void);
    long server_socket;

DESCRIPTION
    These are SASC autoinitialization functions for internet daemons
    started by inetd, Internet super-server. Upon startup, the server
    socket is obtained with ObtainSocket() library call. If successful,
    the socket id is stored to the global variable server_socket. If the
    socket is not obtainable, the server_socket contains value -1.
    If the server_socket is not valid, the server may try to accept() a
    new connection and act as a stand-alone server.

RESULT
    server_socket - positive socket id for success or -1 for failure.

NOTES
    _STIopenSockets() also checks that the system version is at
    least 37. It puts up a requester if the bsdsocket.library
    is not found or is of wrong version.

    The autoinitialization and autotermination functions are
    features specific to the SAS C6. However, these functions
    can be used with other (ANSI) C compilers, too. Example
    follows:

    \* at start of main() *\

    atexit(_STDcloseSockets);
    _STDopenSockets();

AUTHOR
    Jarno Rajahalme, Pekka Pessi,
    the AmiTCP/IP Group <amitcp-group@hut.fi>,
    Helsinki University of Technology, Finland.
```

```
SEE ALSO
    serveraccept(), netutil/inetd
```

C.1.3 charRead

```
NAME
    charRead -- read characters from socket one by one.

SYNOPSIS
    initCharRead(rc, fd)

    void initCharRead(struct CharRead *, int);


    character = charRead(rc)

    int charRead(struct CharRead *);


DESCRIPTION
    charRead is a macro package which return characters one by one
    from given socket input stream. The socket where data is to be read
    is set by calling initCharRead(): rc is the pointer to charread
    structure previously allocated. fd is the (socket) descriptor where
    reading is to be done.

    charRead() returns the next character from input stream or one of
    the following:

    RC_DO_SELECT    (-3)    - read input buffer is returned. Do select
                              before next call if you don't want charread
                              to block.

    RC_EOF          (-2)    - end-of-file condition has occurred.

    RC_ERROR        (-1)    - there has been an error while filling new
                              charread buffer. Check the value of Errno()

NOTE
    Always use variable of type int to store return value from charRead()
    since the numeric value of characters returned may vary between
    0 -255 (or even greater). As you may know, -3 equals 253 if of type
    unsigned char.

EXAMPLE
```

```
/*
 * This piece of code shows how to use charread with select()
 */
#include <sys/types.h>
#include <sys/socket.h>
#include <charread.h>

main_loop(int sock)
{
  struct CharRead rc;
  fd_set readfds;
  int c;

  initCharRead(&rc, sock);

  FD_ZERO(&readfds);

  while(1) {
    FD_SET(sock, &readfds);

    if (select(sock + 1. &readfds, NULL, NULL, NULL)) < 0) {
      perror("select");
      break;
    }
    if (FD_ISSET(sock, &readfds)) {
      while((c = charRead(&rc)) >= 0)
        handle_next_input_character(c);
      if (c == RC_EOF)
        break;
      if (c == RC_ERROR) {
        perror("charRead");
        break;
      }
    }
  }
}
```

```
PORTABILITY
   The source file charread.h should be able to be used in
   UNIX programs as is.
```

```
AUTHORS
   Tomi Ollila,
   the AmiTCP/IP Group <amitcp-group@hut.fi>,

SEE ALSO
   lineRead(), bsdsocket.library/recv()
```

C.1.4 gethostname

```
NAME
    gethostname -- get the name of the host

SYNOPSIS
    error = gethostname(name, namelen);

    int gethostname(char *, int);

FUNCTION
    Get the name of the host to the buffer name of length namelen.
    The name is taken from the environment variable "HOSTNAME"
    where it SHOULD reside.

INPUTS
    name    - Pointer to the buffer where the name should be
              stored.
    namelen - Length of the buffer name.

RESULT
    error   - 0 on success, -1 in case of an error. The global
              variable errno will be set to indicate the error as
              follows:

              ENOENT - The environment variable "HOSTNAME" is not
                       found.

EXAMPLE
    char hostname[MAXHOSTNAMELEN];
    int error;

    error = gethostname(hostname, sizeof(hostname));
    if (error < 0)
      exit(10);

    printf("My name is \"%s\".\n", hostname);

NOTES
    This function is included for source compatibility with Unix
    systems.
    The ENOENT errno value is AmiTCP/IP addition.
```

```
BUGS
    Unlike the Unix version, this version assures that the
    resulting string is always NULL-terminated.

SEE ALSO
    getenv()
```

C.1.5 lineRead

```
NAME
    lineRead -- read newline terminated strings from socket

SYNOPSIS
    initLineRead(rl, fd, lftype, bufsize)

    void initLineRead(struct LineRead *, int, int, int);


    length = lineRead(rl)

    int lineread(struct LineRead *);


DESCRIPTION
    lineRead() reads newline terminated strings from given descriptor
    very efficiently. All the options needed are set by calling
    initLineRead(): rl is the pointer to lineread structure previously
    allocated. fd is the (socket) descriptor where reading is to be
    done. lftype can have following 3 values:

        RL_LFNOTREQ - Newline terminated strings are returned unless
                      there is no newlines left in currently buffered
                      input. In this case remaining buffer is returned.

        RL_LFREQLF  - If there is no newlines left in currently buffered
                      input the remaining input data is copied at the
                      start of buffer. Caller is informed that next
                      call will fill the buffer (and it may block).
                      Lines are always returned with newline at the end
                      unless the string is longer than whole buffer.

        RL_LFREQNUL  - Like LF_REQLF, but remaining newline is removed.
                      Note here that lenght is one longer that actual
                      string length since line that has only one
                      newline at the end would return length as 0
                      which indigate string incomplete condition.

    bufsize is used to tell lineread how big the receive buffer is.
    always put RL_BUFSIZE here since that value is used to determine
```

```
    the memory allocated for the buffer. This option is given to you
    so you may decide to use different buffer size than the default
    1024.

    lineRead() returns the newline terminated string in rl_line field
    of lineread structure. Return values of lineRead() are:

         1 - RL_BUFSIZE     - normal length of returned string.

         0                  - If zero is returned just after select(),
                              end-of-file condition has occurred.
                              Otherwise string is not completed yet.
                              Make sure you call select() (or use non-
                              blocking IO) if you don't want next call
                              to block.

        -1                  - if rl_Line field of lineread structure
                              is NULL, it indicates error condition.
                              If rl_Line points to start of string
                              buffer, input string has been longer
                              than buffer. In this case rl_Line points
                              to zero terminated string of length
                              RL_BUFSIZE.

    You may modify the zero terminated string returned by lineRead() in
    any way, but memory around the string is private lineread memory.

EXAMPLE
    /*
     * The following code shows how to use lineread with select()
     */
    #ifdef USE_LOW_MEMORY_BUFFER
    #define RL_BUFSIZE 256
    #endif

    #include <sys/types.h>
    #ifdef AMIGA
    #include <bsdsocket.h>
    #endif
    #include <lineread.h>
```

```
#define NULL 0

...

main_loop(int sock)
{
  struct LineRead * rl;
  int length;
  fd_set reafdfs;

  if (rl = (struct LineRead *)AllocMem(sizeof (*rl), 0)) {

    initLineRead(rl, sock, LF_REQLF, RL_BUFSIZE);

    FD_ZERO(&readfds);

    while(1) {
      FD_SET(sock, &readfds);

      if (select(sock + 1, &readfds, NULL, NULL, NULL)) < 0) {
        perror("select");
        break;
      }
      if (FD_ISSET(sock, &readfds))
        if ((length = lineRead(rl)) == 0) /* EOF */
          break;
        do {
          if (length > 0)
            write(1, rl->rl_Line, length); /* stdout. write() for */
                                           /* speed demonstration */
          else { /* length == -1 */
            if (rl->rl_Line == NULL); {
              perror("lineRead");
              break;
            }
            else {
              fprintf(stderr, "lineread input buffer overflow!\n");
              write(1, rl->rl_Line, RL_BUFSIZE);
              write(1, "\n", 1);
            }
          }
```

```
            } while ((length = lineRead(rl)) != 0); /* 0 -> do select() */
          }
        FreeMem(rl, sizeof (*rl);
        }
        else
          fprintf("AllocMem: Out Of memory\n");
      }
```

```
PORTABILITY
   The source modules lineread.c and lineread.h should compile
   in UNIX machines as is.

AUTHORS
   Tomi Ollila,
   the AmiTCP/IP Group <amitcp-group@hut.fi>,

SEE ALSO
   readChar(), bsdsocket.library/recv()
```

Appendix D

Protocols and Network Interfaces

The AutoDoc file **protocol.doc** contains on-line manual pages for protocols and network interfaces.

Table of Contents

D.1 Protocols and Network Interfaces

D.1.1 arp

NAME

arp - Address Resolution Protocol

CONFIG

Any SANA-II device driver using ARP

SYNOPSIS

```
#include <sys/socket.h>
#include <net/if_arp.h>
#include <netinet/in.h>

s = socket(AF_INET, SOCK_DGRAM, 0);
```

DESCRIPTION

ARP is a protocol used to dynamically map between Internet Protocol (IP) and hardware addresses. It can be used by most the SANA-II network interface drivers. The current implementation supports only Internet Protocol (and is tested only with Ethernet). However, ARP is not limited to only that combination.

ARP caches IP-to-hardware address mappings. When an interface requests a mapping for an address not in the cache, ARP queues the message which requires the mapping and broadcasts a message on the associated network requesting the address mapping. If a response is provided, the new mapping is cached and any pending message is transmitted. ARP will queue at most one packet while waiting for a mapping request to be responded to; only the most recently transmitted packet is kept.

The address mapping caches are separate for each interface. The amount of mappings in the cache may be specified with an IoctlSocket() request.

To facilitate communications with systems which do not use ARP, IoctlSocket() requests are provided to enter and delete entries in the IP-to-Ethernet tables.

```
USAGE
    #include <sys/ioctl.h>
    #include <sys/socket.h>
    #include <net/if.h>
    #include <net/if_arp.h>

    struct arpreq arpreq;

    IoctlSocket(s, SIOCSARP, (caddr_t)&arpreq);
    IoctlSocket(s, SIOCGARP, (caddr_t)&arpreq);
    IoctlSocket(s, SIOCDARP, (caddr_t)&arpreq);

    These three IoctlSocket()s take the same structure as an argument.
    SIOCSARP sets an ARP entry, SIOCGARP gets an ARP entry, and SIOCDARP
    deletes an ARP entry. These IoctlSocket() requests may be applied to
    any socket descriptor (s). The arpreq structure contains:

    /* Maximum number of octets in protocol/hw address */
    #define MAXADDRARP  16

    /*
     * ARP ioctl request.
     */
    struct arpreq {
            struct  sockaddr arp_pa;  /* protocol address */
            struct  {                 /* hardware address */
              u_char sa_len;         /* actual length + 2 */
              u_char sa_family;
              char   sa_data[MAXADDRARP];
            }  arp_ha;
            int     arp_flags;                   /* flags */
    };

    /*  arp_flags and at_flags field values */
    #define ATF_INUSE       0x01          /* entry in use */
    #define ATF_COM         0x02       /* completed entry */
    #define ATF_PERM        0x04       /* permanent entry */
    #define ATF_PUBL        0x08         /* publish entry */
```

```
    The interface whose ARP table is manipulated is specified by
    arp_pa sockaddr. The address family for the arp_pa sockaddr
    must be AF_INET; for the arp_ha sockaddr it must be AF_UNSPEC.
    The length of arp_ha must match the length of used hardware
    address. Maximum length for the hardware address is MAXADDRARP
    bytes. The only flag bits which may be written are ATF_PERM
    and ATF_PUBL. ATF_PERM makes the entry permanent if the
    IoctlSocket() call succeeds. ATF_PUBL specifies that the ARP
    code should respond to ARP requests for the indicated host
    coming from other machines.  This allows a host to act as an
    ARP server which may be useful in convincing an ARP-only
    machine to talk to a non-ARP machine.

UNSUPPORTED IN AmiTCP/IP

AmiTCP/IP EXTENSIONS
    There is an extension to the standard BSD4.4 ARP ioctl interface to
    access the contents of the whole ARP mapping cache. (In the BSD4.4
    the static ARP table is accessed via the /dev/kmem.) The SIOCGARPT
    ioctl takes the following arptabreq structure as an argument:

    /*
     * An AmiTCP/IP specific ARP table ioctl request
     */
    struct arptabreq {
            struct arpreq atr_arpreq;  /* To identify the interface */
            long   atr_size;          /* # of elements in atr_table */
            long   atr_inuse;               /* # of elements in use */
            struct arpreq *atr_table;
    };

    The atr_arpreq specifies the used interface. The hardware address
    for the interface is returned in the arp_ha field of atr_arpreq
    structure.

    The SIOCGARPT ioctl reads at most atr_size entries from the cache
    into the user supplied buffer atr_table, if it is not NULL. Actual
    amount of returned entries is returned in atr_size. The current
    amount of cached mappings is returned in the atr_inuse.

    The SIOCGARPT ioctl has following usage:
```

```
    struct arpreq cache[N];
    struct arptabreq arptab = { N, 0, cache };

    IoctlSocket(s, SIOCGARPT, (caddr_t)&arptabreq);
```

DIAGNOSTICS

ARP watches passively for hosts impersonating the local host (that is, a host which responds to an ARP mapping request for the local host's address).

"duplicate IP address a.b.c.d!!"
"sent from hardware address: %x:%x:...:%x:%x"

ARP has discovered another host on the local network which responds to mapping requests for its own Internet address.

BUGS

The ARP is tested only with Ethernet. Other network hardware may require special ifconfig configuration.

SEE ALSO

inet, netutil/arp, netutil/ifconfig, <net/if_arp.h>

Plummer, Dave, ''An Ethernet Address Resolution Protocol -or- Converting Network Protocol Addresses to 48.bit Ethernet Addresses for Transmission on Ethernet Hardware,'' RFC 826, Network Information Center, SRI International, Menlo Park, Calif., November 1982. (Sun 800-1059-10)

D.1.2 icmp

```
NAME
    icmp - Internet Control Message Protocol

SYNOPSIS
    #include <sys/socket.h>
    #include <netinet/in.h>

    int
    socket(AF_INET, SOCK_RAW, proto)

DESCRIPTION
    ICMP is the error and control message protocol used by IP and the
    Internet protocol family.  It may be accessed through a ``raw
    socket'' for network monitoring and diagnostic functions.  The proto
    parameter to the socket call to create an ICMP socket is obtained
    from getprotobyname().  ICMP sockets are connectionless, and are
    normally used with the sendto() and recvfrom() calls, though the
    connect() call may also be used to fix the destination for future
    packets (in which case the recv() and send() socket library calls
    may be used).

    Outgoing packets automatically have an IP header prepended to them
    (based on the destination address).  Incoming packets are received
    with the IP header and options intact.

DIAGNOSTICS
    A socket operation may fail with one of the following errors
    returned:

    [EISCONN]        when trying to establish a connection on a socket
                     which already has one, or when trying to send a
                     datagram with the destination address specified and
                     the socket is already connected;

    [ENOTCONN]       when trying to send a datagram, but no destination
                     address is specified, and the socket hasn't been
                     connected;

    [ENOBUFS]        when the system runs out of memory for an internal
                     data structure;
```

```
    [EADDRNOTAVAIL]  when an attempt is made to create a socket with a
                     network address for which no network interface
                     exists.

SEE ALSO
    bsdsocket.library/send(),  bsdsocket.library/recv(), inet,  ip

HISTORY
    The icmp protocol is originally from 4.3BSD.
```

D.1.3 if

```
NAME
    if - Network Interface to SANA-II devices

DESCRIPTION
    Each network interface in the AmiTCP/IP corresponds to a path
    through which messages may be sent and received.  A network
    interface usually has a SANA-II device driver associated with it,
    though the loopback interface, "lo", do not. The network interface
    in the AmiTCP/IP (sana_softc) is superset of the BSD Unix network
    interface.

    When the network interface is first time referenced, AmiTCP/IP tries
    to open the corresponding SANA-II device driver. If successful, a
    software interface to the SANA-II device is created. The "network/"
    prefix is added to the SANA-II device name, if needed. Once the
    interface has acquired its address, it is expected to install a
    routing table entry so that messages can be routed through it.

    The SANA-II interfaces must be configured before they will allow
    traffic to flow through them. It is done after the interface is
    assigned a protocol address with a SIOCSIFADDR ioctl. Some
    interfaces may use the protocol address or a part of it as their
    hardware address. On interfaces where the network-link layer address
    mapping is static, only the network number is taken from the ioctl;
    the remainder is found in a hardware specific manner. On interfaces
    which provide dynamic network-link layer address mapping facilities
    (for example, Ethernets or Arcnets using ARP), the entire address
    specified in the ioctl is used.

    The following ioctl calls may be used to manipulate network
    interfaces. Unless specified otherwise, the request takes an ifreq
    structure as its parameter. This structure has the form

    struct ifreq {
      char ifr_name[IFNAMSIZ]; /* interface name (eg. "slip.device/0")*/
      union {
        struct sockaddr ifru_addr;
        struct sockaddr ifru_dstaddr;
        short           ifru_flags;
      } ifr_ifru;
```

```
#define ifr_addr    ifr_ifru.ifru_addr                 /* address */
#define ifr_dstaddr ifr_ifru.ifru_dstaddr   /* end of p-to-p link */
#define ifr_flags   ifr_ifru.ifru_flags                  /* flags */
};

SIOCSIFADDR      Set interface address. Following the address
                 assignment, the ``initialization'' routine for
                 the interface is called.

SIOCGIFADDR      Get interface address.

SIOCSIFDSTADDR   Set point to point address for interface.

SIOCGIFDSTADDR   Get point to point address for interface.

SIOCSIFFLAGS     Set interface flags field. If the interface is
                 marked down, any processes currently routing
                 packets through the interface are notified.

SIOCGIFFLAGS     Get interface flags.

SIOCGIFCONF      Get interface configuration list. This request
                 takes an ifconf structure (see below) as a
                 value-result parameter. The ifc_len field should be
                 initially set to the size of the buffer pointed to
                 by ifc_buf. On return it will contain the length,
                 in bytes, of the configuration list.

/*
 * Structure used in SIOCGIFCONF request.
 * Used to retrieve interface configuration
 * for machine (useful for programs which
 * must know all networks accessible).
 */
struct ifconf {
  int  ifc_len;                      /* size of associated buffer */
  union {
    caddr_t       ifcu_buf;
    struct ifreq *ifcu_req;
  } ifc_ifcu;
#define ifc_buf ifc_ifcu.ifcu_buf              /* buffer address */
```

```
#define ifc_req ifc_ifcu.ifcu_req /* array of structures returned */
};
```

UNSUPPORTED IN AmiTCP/IP

These standard BSD ioctl codes are not currently supported:

```
SIOCADDMULTI     Enable a multicast address for the interface.

SIOCDELMULTI     Disable a previously set multicast address.

SIOCSPROMISC     Toggle promiscuous mode.
```

AmiTCP/IP EXTENSIONS

The following ioctls are used to configure protocol and hardware specific properties of a sana_softc interface. They are used in the AmiTCP/IP only.

```
SIOCSSANATAGS    Set SANA-II specific properties with a tag list.

SIOCGSANATAGS    Get SANA-II specific properties into a
                 wiretype_parameters structure and a user tag list.

struct wiretype_parameters
{
  ULONG  wiretype;              /* the wiretype of the interface */
  WORD   flags;                                    /* iff_flags */
  struct TagItem *tags;                /* tag list user provides */
};
```

SEE ALSO

arp, lo, netutil/arp, netutil/ifconfig, <sys/ioctl.h>, <net/if.h>, <net/sana2tags.h>

D.1.4 inet

```
NAME
    inet - Internet protocol family

SYNOPSIS
    #include <sys/types.h>
    #include <netinet/in.h>

DESCRIPTION
    The Internet protocol family implements a collection of protocols
    which are centered around the Internet Protocol (IP) and which share
    a common address format.  The Internet family provides protocol
    support for the SOCK_STREAM, SOCK_DGRAM, and SOCK_RAW socket types.

PROTOCOLS
    The Internet protocol family is comprised of the Internet Protocol
    (IP), the Address Resolution Protocol (ARP), the Internet Control
    Message Protocol (ICMP), the Transmission Control Protocol (TCP),
    and the User Datagram Protocol (UDP).

    TCP is used to support the SOCK_STREAM abstraction while UDP is used
    to support the SOCK_DGRAM abstraction; (SEE ALSO tcp, SEE ALSO udp).
    A raw interface to IP is available by creating an Internet socket of
    type SOCK_RAW; (SEE ALSO ip).  ICMP is used by the kernel to handle
    and report errors in protocol processing.  It is also accessible to
    user programs; (SEE ALSO icmp).  ARP is used to translate 32-bit IP
    addresses into varying length hardware addresses; (SEE ALSO arp).

    The 32-bit IP address is divided into network number and host number
    parts.  It is frequency-encoded; the most significant bit is zero in
    Class A addresses, in which the high-order 8 bits are the network
    number.  Class B addresses have their high order two bits set to 10
    and use the highorder 16 bits as the network number field.  Class C
    addresses have a 24-bit network number part of which the high order
    three bits are 110.  Sites with a cluster of local networks may
    chose to use a single network number for the cluster; this is done
    by using subnet addressing.  The local (host) portion of the address
    is further subdivided into subnet number and host number parts.
    Within a subnet, each subnet appears to be an individual network;
    externally, the entire cluster appears to be a single, uniform
    network requiring only a single routing entry.  Subnet addressing is
```

```
    enabled and examined by the following ioctl commands on a datagram
    socket in the Internet domain; they have the same form as the
    SIOCIFADDR (SEE ALSO if) command.

    SIOCSIFNETMASK      Set interface network mask.  The network mask
                        defines the network part of the address; if it
                        contains more of the address than the address
                        type would indicate, then subnets are in use.

    SIOCGIFNETMASK      Get interface network mask.

ADDRESSING
    IP addresses are four byte quantities, stored in network byte order
    (the native Amiga byte order)

    Sockets in the Internet protocol family  use  the  following
    addressing structure:
         struct sockaddr_in {
              short     sin_family;
              u_short   sin_port;
              struct    in_addr sin_addr;
              char sin_zero[8];
         };

    Functions in bsdsocket.library are provided to manipulate structures
    of this form.

    The sin_addr field of the sockaddr_in structure specifies a local or
    remote IP address.  Each network interface has its own unique IP
    address.  The special value INADDR_ANY may be used in this field to
    effect "wildcard" matching.  Given in a bind() call, this value
    leaves the local IP address of the socket unspecified, so that the
    socket will receive connections or messages directed at any of the
    valid IP addresses of the system.  This can prove useful when a
    process neither knows nor cares what the local IP address is or when
    a process wishes to receive requests using all of its network
    interfaces.  The sockaddr_in structure given in the bind() call must
    specify an in_addr value of either IPADDR_ANY or one of the system's
    valid IP addresses.  Requests to bind any other address will elicit
    the error EADDRNOTAVAIL. When a connect() call is made for a socket
    that has a wildcard local address, the system sets the sin_addr
```

```
    field of the socket to the IP address of the network interface that
    the packets for that connection are routed via.

    The sin_port field of the sockaddr_in structure specifies a port
    number used by TCP or UDP. The local port address specified in a
    bind() call is restricted to be greater than IPPORT_RESERVED
    (defined in <netinet/in.h>) unless the creating process is running
    as the super-user, providing a space of protected port numbers.  In
    addition, the local port address must not be in use by any socket of
    same address family and type.  Requests to bind sockets to port
    numbers being used by other sockets return the error EADDRINUSE.  If
    the local port address is specified as 0, then the system picks a
    unique port address greater than IPPORT_RESERVED.  A unique local
    port address is also picked when a socket which is not bound is used
    in a connect() or send() call.  This allows programs which do not
    care which local port number is used to set up TCP connections by
    sim- ply calling socket() and then connect(), and to send UDP
    datagrams with a socket() call followed by a send() call.

    Although this implementation restricts sockets to unique local port
    numbers, TCP allows multiple simultaneous connections involving the
    same local port number so long as the remote IP addresses or port
    numbers are different for each connection.  Programs may explicitly
    override the socket restriction by setting the SO_REUSEADDR socket
    option with setsockopt (see getsockopt()).

SEE ALSO
    bsdsocket.library/bind(), bsdsocket.library/connect(),
    bsdsocket.library/getsockopt(), bsdsocket.library/IoctlSocket(),
    bsdsocket.library/send(), bsdsocket.library/socket(),
    bsdsocket.library/gethostent(), bsdsocket.library/getnetent(),
    bsdsocket.library/getprotoent(), bsdsocket.library/getservent(),
    bsdsocket.library/inet_addr(), arp, icmp, ip, tcp, udp

    Network Information Center, DDN Protocol Handbook (3 vols.),
    Network  Information  Center, SRI International, Menlo Park,
    Calif., 1985.
    A AmiTCP/IP Interprocess Communication Primer

WARNING
    The Internet protocol support is subject to change as the Internet
```

```
protocols develop.  Users should not depend on details of the
current implementation, but rather the services exported.
```

D.1.5 ip

NAME

ip - Internet Protocol

SYNOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>

int
socket(AF_INET, SOCK_RAW, proto)
```

DESCRIPTION

IP is the transport layer protocol used by the Internet protocol family. Options may be set at the IP level when using higher-level protocols that are based on IP (such as TCP and UDP). It may also be accessed through a ''raw socket'' when developing new protocols, or special purpose applica- tions.

A single generic option is supported at the IP level, IP_OPTIONS, that may be used to provide IP options to be transmitted in the IP header of each outgoing packet. Options are set with setsockopt() and examined with getsockopt(). The format of IP options to be sent is that specified by the IP protocol specification, with one exception: the list of addresses for Source Route options must include the first-hop gateway at the beginning of the list of gateways. The first-hop gateway address will be extracted from the option list and the size adjusted accordingly before use. IP options may be used with any socket type in the Internet family.

Raw IP sockets are connectionless, and are normally used with the sendto and recvfrom calls, though the connect() call may also be used to fix the destination for future packets (in which case the recv() and send() system calls may be used).

If proto is 0, the default protocol IPPROTO_RAW is used for outgoing packets, and only incoming packets destined for that protocol are received. If proto is non-zero, that protocol number will be used on outgoing packets and to filter incoming packets.

Outgoing packets automatically have an IP header prepended to them (based on the destination address and the protocol number the socket is created with). Incoming packets are received with IP header and options intact.

DIAGNOSTICS

A socket operation may fail with one of the following errors returned:

[EISCONN]	when trying to establish a connection on a socket which already has one, or when trying to send a datagram with the destination address specified and the socket is already connected;
[ENOTCONN]	when trying to send a datagram, but no destination address is specified, and the socket hasn't been connected;
[ENOBUFS]	when the system runs out of memory for an internal data structure;
[EADDRNOTAVAIL]	when an attempt is made to create a socket with a network address for which no network interface exists.

The following errors specific to IP may occur when setting or getting IP options:

[EINVAL]	An unknown socket option name was given.
[EINVAL]	The IP option field was improperly formed; an option field was shorter than the minimum value or longer than the option buffer provided.

SEE ALSO

bsdsocket.library/getsockopt(), bsdsocket.library/send(), bsdsocket.library/recv(), icmp, inet

HISTORY

The ip protocol appeared in 4.2BSD.

D.1.6 lo

```
NAME
    lo - Software Loopback Network Interface

SYNOPSIS
    pseudo-device
    loop

DESCRIPTION
    The loop interface is a software loopback mechanism which may be
    used for performance analysis, software testing, and/or local
    communication.  There is no SANA-II interface associated with lo.
    As with other network interfaces, the loopback interface must have
    network addresses assigned for each address family with which it is
    to be used.  These addresses may be set or changed with the
    SIOCSIFADDR ioctl. The loopback interface should be the last
    interface configured, as protocols may use the order of
    configuration as an indication of priority.  The loopback should
    never be configured first unless no hardware interfaces exist.

DIAGNOSTICS
    "lo%d: can't handle af%d."
    The interface was handed a message with ad- dresses formatted in an
    unsuitable address family; the packet was dropped.

SEE ALSO
    inet, if, netutil/ifconfig

BUGS
    Older BSD Unix systems enabled the loopback interface
    automatically, using a nonstandard Internet address (127.1).  Use
    of that address is now discouraged; a reserved host address for the
    local network should be used instead.
```

D.1.7 routing

```
NAME
    routing - system supporting for local network packet routing

DESCRIPTION
    The network facilities provided general packet routing,
    leaving routing table maintenance to applications processes.

    A simple set of data structures comprise a ``routing table''
    used in selecting the appropriate network interface when
    transmitting packets.  This table contains a single entry for
    each route to a specific network or host.  A user process, the
    routing daemon, maintains this data base with the aid of two
    socket specific ioctl commands, SIOCADDRT and SIOCDELRT.
    The commands allow the addition and deletion of a single
    routing table entry, respectively.  Routing table
    manipulations may only be carried out by super-user.

    A routing table entry has the following form, as defined  in
    <net/route.h>:
         struct rtentry {
              u_long    rt_hash;
              struct    sockaddr rt_dst;
              struct    sockaddr rt_gateway;
              short     rt_flags;
              short     rt_refcnt;
              u_long    rt_use;
              struct    ifnet *rt_ifp;
         };
    with rt_flags defined from:
      #define   RTF_UP         0x1              /* route usable */
      #define   RTF_GATEWAY    0x2  /* destination is a gateway */
      #define   RTF_HOST  0x4     /* host entry (net otherwise) */

    Routing table entries come in three flavors: for a specific
    host, for all hosts on a specific network, for any destination
    not matched by entries of the first two types (a wildcard
    route).  When the system is booted, each network interface
    autoconfigured installs a routing table entry when it wishes
    to have packets sent through it.  Normally the interface
    specifies the route through it is a ``direct'' connection to
```

```
    the destination host or network.  If the route is direct, the
    transport layer of a protocol family usually requests the
    packet be sent to the same host specified in the packet.
    Otherwise, the interface may be requested to address the
    packet to an entity different from the eventual recipient
    (that is, the packet is forwarded).

    Routing table entries installed by a user process may not
    specify the hash, reference count, use, or interface fields;
    these are filled in by the routing routines.  If a route is in
    use when it is deleted (rt_refcnt is non-zero), the resources
    associated with it will not be reclaimed until all references
    to it are removed.

    The routing code returns EEXIST if requested to duplicate an
    existing entry, ESRCH if requested to delete a non-existent
    entry, or ENOBUFS if insufficient resources were available to
    install a new route.

    The rt_use field contains the number of packets sent along the
    route.  This value is used to select among multiple routes to
    the same destination.  When multiple routes to the same
    destination exist, the least used route is selected.

    A wildcard routing entry is specified with a zero destination
    address value.  Wildcard routes are used only when the system
    fails to find a route to the destination host and network.
    The combination of wildcard routes and routing redirects can
    provide an economical mechanism for routing traffic.

SEE ALSO
    bsdsocket.library/IoctlSocket(), netutil/route
```

D.1.8 tcp

```
NAME
    tcp - Internet Transmission Control Protocol

SYNOPSIS
    #include <sys/socket.h>
    #include <netinet/in.h>

    int
    socket(AF_INET, SOCK_STREAM, 0)

DESCRIPTION
    The TCP protocol provides reliable, flow-controlled, two-way
    transmission of data.  It is a byte-stream protocol used to support
    the SOCK_STREAM abstraction.  TCP uses the standard Internet address
    format and, in addition, provides a per-host collection of ``port
    addresses''. Thus, each address is composed of an Internet address
    specifying the host and network, with a specific TCP port on the
    host identifying the peer entity.

    Sockets utilizing the tcp protocol are either ``active'' or
    ``passive''.  Active sockets initiate connections to passive
    sockets.  By default TCP sockets are created active; to create a
    passive socket the listen() bsdsocket.library function call must be
    used after binding the socket with the bind() bsdsocket.library
    function call.  Only passive sockets may use the accept() call to
    accept incoming connections.  Only active sockets may use the
    connect() call to initiate connections.

    Passive sockets may ``underspecify'' their location to match
    incoming connection requests from multiple networks.  This
    technique, termed ``wildcard addressing'', allows a single server to
    provide service to clients on multiple networks.  To create a socket
    which listens on all networks, the Internet address INADDR_ANY must
    be bound.  The TCP port may still be specified at this time; if the
    port is not specified the bsdsocket.library function will assign
    one.  Once a connection has been established the socket's address is
    fixed by the peer entity's location.  The address assigned the
    socket is the address associated with the network interface through
    which packets are being transmitted and received.  Normally this
    address corresponds to the peer entity's network.
```

```
    TCP supports one socket option which is set with setsockopt() and
    tested with getsockopt(). Under most circumstances, TCP sends data
    when it is presented; when outstanding data has not yet been
    acknowledged, it gathers small amounts of output to be sent in a
    single packet once an acknowledgement is received. For a small
    number of clients, such as X Window System functions that
    send a stream of mouse events which receive no replies, this
    packetization may cause significant delays. Therefore, TCP provides
    a boolean option, TCP_NODELAY (from <netinet/tcp.h>, to defeat this
    algorithm. The option level for the setsockopt call is the protocol
    number for TCP, available from getprotobyname().

    Options at the IP transport level may be used with TCP; SEE ALSO ip.
    Incoming connection requests that are source-routed are noted, and
    the reverse source route is used in responding.

DIAGNOSTICS
    A socket operation may fail with one of the following errors
    returned:

    [EISCONN]        when trying to establish a connection on a socket
                     which already has one;

    [ENOBUFS]        when the AmiTCP/IP runs out of memory for an internal
                     data structure;

    [ETIMEDOUT]      when a connection was dropped due to excessive
                     retransmissions;

    [ECONNRESET]     when the remote peer forces the connection to be
                     closed;

    [ECONNREFUSED]   when the remote peer actively refuses connection
                     establishment (usually because no process is
                     listening to the port);

    [EADDRINUSE]     when an attempt is made to create a socket with a
                     port which has already been allocated;

    [EADDRNOTAVAIL]  when an attempt is made to create a socket with a
```

```
                    network address for which no network interface
                    exists.

SEE ALSO
    bsdsocket.library/getsockopt(), bsdsocket.library/socket(),
    bsdsocket.library/bind(), bsdsocket.library/listen(),
    bsdsocket.library/accept(), bsdsocket.library/connect(), inet,
    ip, <sys/socket.h>, <netinet/tcp.h>, <netinet/in.h>

HISTORY
    The tcp protocol stack appeared in 4.2BSD.
```

D.1.9 udp

```
NAME
    udp - Internet User Datagram Protocol

SYNOPSIS
    #include <sys/socket.h>
    #include <netinet/in.h>

    int
    socket(AF_INET, SOCK_DGRAM, 0)

DESCRIPTION
    UDP is a simple, unreliable datagram protocol which is used to
    support the SOCK_DGRAM abstraction for the Internet protocol family.
    UDP sockets are connectionless, and are normally used with the
    sendto() and recvfrom() calls, though the connect() call may also be
    used to fix the destination for future packets (in which case the
    recv() and send() function calls may be used).

    UDP address formats are identical to those used by TCP. In
    particular UDP provides a port identifier in addition to the normal
    Internet address format.  Note that the UDP port space is separate
    from the TCP port space (i.e. a UDP port may not be ``connected'' to
    a TCP port). In addition broadcast packets may be sent (assuming the
    underlying network supports this) by using a reserved ``broadcast
    address''; this address is network interface dependent.

    Options at the IP transport level may be used with UDP; SEE ALSO ip.

DIAGNOSTICS
    A socket operation may fail with one of the following errors
    returned:

    [EISCONN]         when trying to establish a connection on a socket
                      which already has one, or when trying to send a
                      datagram with the destination address specified and
                      the socket is already connected;

    [ENOTCONN]        when trying to send a datagram, but no destination
                      address is specified, and the socket hasn't been
                      connected;
```

```
    [ENOBUFS]         when the system runs out of memory for an
                      internal data structure;

    [EADDRINUSE]      when an attempt is made to create a socket with a
                      port which has already been allocated;

    [EADDRNOTAVAIL]   when an attempt is made to create a socket with a
                      network address for which no network interface
                      exists.

SEE ALSO
    bsdsocket.library/getsockopt(), bsdsocket.library/recv(),
    bsdsocket.library/send(), bsdsocket.library/socket(), inet, ip

HISTORY
    The udp protocol appeared in 4.2BSD.
```