OpenBSD TCP/IP Stack Port and SNMP for eCos

November 2000

Copying terms

Trademarks

Contents

TCP/IP

TCP/IP Networking for eCos now provides a complete TCP/IP networking stack, which is derived from the latest stable release of OpenBSD. The networking support is fully featured and well tested within the eCos environment.

Ethernet drivers are currenty provided for the following standard supported platforms:

- Motorola PowerPC MBX/860
- Cirrus Logic EDB72xx, based on the Crystal CS8900
- Socket Communications Low Power Compact Flash Ethernet adaptor
- Intel EBSA-285 + EtherPRO 10/100+

Networking Stack Features

Since this networking package is based on BSD code, it is very complete and robust. The eCos implementation includes support for the following protocols:

- IPv4
- UDP
- TCP
- ICMP
- raw packet interface

These additional features are also present in the package, but are not yet supported:

- Berkeley Packet Filter
- Multi-cast and uni-cast support, including multi-casting routing
- IPv6

Ethernet Driver Design

Currently, the networking stack only supports ethernet based networking.

The network drivers use a two-layer design. One layer is hardware independent and contains all the stack specific code. The other layer is platform dependent and communicates with the hardware independent layer via a very simple API. In this way, hardware device drivers can actually be used with other stacks, if the same API can be provided by that stack. We designed the drivers this way to encourage the development of other stacks in eCos while allowing re-use of the actual hardware specific code.

Complete documentation of the ethernet device driver and the associated API can be found in the file `net/drivers/eth/common/VERSION/doc/driver_doc`. The driver and API is the same as the minimal debug stack used by the RedBoot application. See the RedBoot documentation for further information.

Sample Code

Many examples using the networking support are provided. These are arranged as eCos test programs, primarily for use in verifying the package, but they can also serve as useful frameworks for program design. We have taken a KISS approach to building programs which use the network. A single include file <`network.h`> is all that is required to access the stack. A complete, annotated test program can be found at `net/tcpip/VERSION/tests/ftp_test.c`, with its neighbouring files.

OpenBSD TCP/IP stack port

This document describes how to get started with the OpenBSD TCP/IP network stack.

Installation

The stack is usually distributed in the eCos package distribution format. It is installed by adding the file `net-10b`*`y`*`.epk` where *y* is the minor version number of this release (located in the root of the software distribution) to an existing eCos installation, using the eCos **Package Administration Tool**.

To add the new EPKs to the eCos 1.4.*x* repository, run up the GUI Package Administration Tool, click '**Add...**' and select the net* .epk file (hold down the **Ctrl** key to extend the selection).

In the config tool, most packages should be version "v1_4_*x*". The EBSA ethernet driver should be version "v1_4_*x*". When you add the "Networking" package to the config, it should be "v1_0b*y*".

If you have an existing configuration customized to your application which already uses the network driver, you can switch over to the newer versions of the network driver package using the **Build->Packages** dialog.

For command-line use (eg. under Linux), use the script ecosadmin.tcl which will be found in the packages directory. Ensure that either the environment variable ECOS_REPOSITORY is set correctly, or unset it and run the script from within the packages directory itself.

```
cd packages
./ecosadmin.tcl add net-1.0by.epk
```

Targets

A number of device drivers may be supported. The default configuration supports two by default, and you will need to write your own driver if you should add additional ones.

The target for your board will normally be supplied with an ethernet driver, in which case it may simply be added to your network interface, as above. If your target is not supplied with an ethernet driver, you will need to use loopback (see “Loopback tests” on page 9).

Building the Network Stack

Using the **Build->Packages** dialog, add the two packages “Networking” and “Common Ethernet Support” to your configuration. Their macro names are CYGPKG_NET and CYGPKG_NET_ETH_DRIVERS respectively.

The platform-specific ethernet device driver for your platform will be added as part of the target selection (in the **Build->Templates** “Hardware” item), along with the PCI I/O subsystem and the appropriate serial device driver. For example, the PowerPC MBX target selection adds the package PKG_NET_QUICC_ETH_DRIVERS, and the Cirrus Logic EDB7xxx target selection adds the package CYGPKG_NET_EDB7XXX_ETH_DRIVERS. After this, eCos and its tests can be built exactly as usual.

NOTE: By default, most of the network tests are not built. This is because some of them require manual intervention, i.e. they are to be run “by hand”, and are not suitable for automated testing. To build the full set of network tests, set the configuration option CYGPKG_NET_BUILD_TESTS “Build networking tests (demo programs)” within “Networking support build options”.

Configuring IP Addresses

Each interface (“eth0” and “eth1”) has independent configuration of its setup. Each can be set up manually (in which case you must write code to do this), or by using BOOTP/DHCP, or explicitly, with configured values. If additional interfaces are added, these must be configured manually.

The configurable values are:

- IP address
- netmask
- broadcast address
- gateway/router
- server address.

Server address is the DHCP server if applicable, but in addition, many test cases use it as “the machine to talk to” in whatever manner the test exercises the protocol stack.

The initialization is invoked by calling the C routine

```
  void init_all_network_interfaces(void);
```

refer to the test cases, “`.../packages/net/tcpip/VERSION/tests/ftp_test.c`” for example usage, and the source files in `packages/net/tcpip/VERSION/src/lib/bootp_support.c` and `network_support.c` to see what that call does.

This assumes that the MAC address is already defined in the serial EEPROM or however the particular target implements this; there is no (tested) support for setting the MAC address in this release.

DHCP support is active by default, and there are configuration options to control it. Firstly, in the top level of the “Networking” configuration tree, “Use full DHCP instead of BOOTP” enables DHCP, and it contains an option to have the system provide a thread to renew DHCP leases and manage lease expiry. Secondly, the individual interfaces “eth0” and “eth1” each have new options within the “Use BOOTP/DHCP to initialize ‘*ethX*’” to select whether to use DHCP rather than BOOTP.

Tests and Demonstrations

Loopback tests

By default, only tests which can execute on any target will be built. These therefore do not actually use external network interfaces (though they may configure and initialize them) but are limited to testing via the loopback interface.

ping_lo_test - ping test of the loopback address
tcp_lo_select - simple test of select with TCP via loopback
tcp_lo_test - trivial TCP test via loopback
udp_lo_test - trivial UDP test via loopback
multi_lo_select - test of multiple select() calls simultaneously

Building the Network Tests

To build further network tests, ensure that the configuration option CYGPKG_NET_BUILD_TESTS is set as described above and then make the tests in the usual way. Alternatively (with that option set) use

```
make -C net/tcpip/VERSION/ tests
```

after building the eCos library to build only the network tests.

This should give test executables in install/tests/net/tcpip/VERSION/tests including the following:

socket_test - trivial test of socket creation API
mbuf_test - trivial test of mbuf allocation API
ftp_test - simple FTP test, connects to “server”
ping_test - pings “server” and non-existent host to test timeout
tcp_echo - data forwarding program for performance test
nc_test_master - network characterization master (unsupported)
nc_test_slave - network characterization slave (unsupported)
server_test - a very simple server example
tftp_client_test - performs a tftp get and put from/to “server”
tftp_server_test - runs a tftp server for a short while
dhcp_test - ping test, but also relinquishes and
reacquires DHCP leases periodically
flood - a flood ping test; use with care
set_mac_address - set MAC address(es) of interfaces in NVRAM
bridge - contributed network bridge code

Standalone Tests

socket_test - trivial test of socket creation API
mbuf_test - trivial test of mbuf allocation API

These two do not communicate over the net; they just perform simple API tests then exit.

ftp_test `- simple FTP test, connects to “server”`

This test initializes the interface(s) then connects to the FTP server on the “server” machine for for each active interface in turn, confirms that the connection was successful, disconnects and exits. This tests interworking with the server.

ping_test `- pings “server” and non-existent host to test timeout`

This test initializes the interface(s) then pings the server machine in the standard way, then pings address “32 up” from the server in the expectation that there is no machine there. This confirms that the successful ping is not a false positive, and tests the receive timeout. If there is such a machine, of course the 2nd set of pings succeeds, confirming that we can talk to a machine not previously mentioned by configuration or by bootp. It then does the same thing on the other interface, eth1.

dhcp_test `- ping test, but also manipulates DHCP leases`

This test is very similar to the ping test, but in addition, provided the network package is not configured to do this automatically, it manually relinquishes and reclaims DHCP leases for all available interfaces. This tests the external API to DHCP. See section below describing this.

flood `- a flood ping test; use with care`

This test performs pings on all interfaces as quickly as possible, and only prints status information periodically. Flood pinging is bad for network performance; so do not use this test on general purpose networks unless protected by a switch.

Performance Test

tcp_echo `- data forwarding program for performance test`

`tcp_echo` is one part of the standard performance test we use. The other parts are host programs `tcp_source` and `tcp_sink`. To make these (under LINUX) cd to the tests source directory in the eCos repository and type “`make -f make.linux`” - this should build `tcp_source` and `tcp_sink`.

The LINUX program “`tcp_source`” sends data to the target. On the target, “`tcp_echo`” sends it onwards to “`tcp_sink`” running on LINUX. So the target must receive and send on all the data that `tcp_source` sends it; the time taken for this is measured and the data rate is calculated.

To invoke the test, first start `tcp_echo` on the target board and wait for it to become quiescent - it will report work to calibrate a CPU load which can be used to simulate real operating conditions for the stack.

Then on your LINUX machine, in one terminal window, invoke `tcp_sink` giving it the IP address (or hostname) of one interface of the target board. For example "`tcp_sink 10.130.39.66`". `tcp_echo` on the target will print something like "`SINK connection from 10.130.39.13:1143`" when `tcp_sink` is correctly invoked.

Next, in another LINUX terminal window, invoke `tcp_source`, giving it the IP address (or hostname) of an interface of the target board, and optionally a background load to apply to the target while the test runs. For example, "`tcp_source 194.130.39.66`" to run the test with no additional target CPU load, or "`tcp_source 194.130.39.66 85`" to load it up to 85% used. The target load must be a multiple of 5. `tcp_echo` on the target will print something like "`SOURCE connection from 194.130.39.13:1144`" when tcp_source is correctly invoked.

You can connect tcp_sink to one target interface and tcp_source to another, or both to the same interface. Similarly, you can run `tcp_sink` and `tcp_source` on the same LINUX machine or different ones. TCP/IP and ARP look after them finding one another, as intended.

nc_test_master `- network characterization master (unsupported)`
nc_test_slave `- network characterization slave (unsupported)`

These tests talk to each other to measure network performance. They can each run on either a test target or a LINUX host computer given some customization to your local environment. As provided, `nc_test_slave` must run on the test target, and `nc_test_master` must be run on a LINUX host, and be given the test target's IP address or hostname.

The tests print network performance for various packet sizes over UDP and TCP, versus various additional CPU loads on the target.

Interactive Tests

server_test `- a very simple server example`

This test simply awaits a connection on port 7734 and after accepting a connection, gets a packet (with a timeout of a few seconds) and prints it.

The connection is then closed. We then loop to await the next connection, and so on. To use it, telnet to the target on port 7734 then type something (quickly!)

```
% telnet 172.16.19.171 7734
Hello target board
```

and the test program will print something like:

```
connection from 172.16.19.13:3369
buf = ’Hello target board’
```

tftp_client_test `- performs a tftp get and put from/to “server”`

This is only partially interactive. You need to set things up on the “server” in order for this to work, and you will need to look at the server afterwards to confirm that all was well.

For each interface in turn, this test attempts to read by tftp from the server, a file called “`tftp_get`” and prints the status and contents it read (if any). It then writes the same data to a file called “`tftp_put`” on the same server.

In order for this to succeed, both files must already exist. The TFTP protocol does not require that a WRQ request _create_ a file, just that it can write it. The TFTP server on Linux certainly will only allow writes to an existing file, given the appropriate permission. Thus, you need to have these files in place, with proper permission, before running the test.

The conventional place for the tftp server to operate in LINUX is /tftpboot/; you will likely need root privileges to create files there. The data contents of `tftp_get` can be anything you like, but anything very large will waste lots of time printing it on the test’s stdout, and anything above 32kB will cause a buffer overflow and unpredictable failure.

Creating an empty tftp_put file (eg. by copying /dev/null to it) is neatest. So before the test you should have something like:

```
 -rw-rw-rw- 1 root        1076 May  1 11:39 tftp_get
 -rw-rw-rw- 1 root        0 May  1 15:52 tftp_put
```

note that both files have public permissions wide open. After running the test, `tftp_put` should be a copy of `tftp_get`.

```
 -rw-rw-rw-  1 root       1076 May  1 11:39 tftp_get
 -rw-rw-rw-  1 root       1076 May  1 15:52 tftp_put
```

tftp_server_test `- runs a tftp server for a short while`

This test is truly interactive, in that you can use a standard tftp application to get and put files from the server, during the 5 minutes that it runs. The dummy filesystem which underlies the server initially contains one file, called “uu” which contains part of a familiar text and some padding. It also accommodates creation of 3 further files of up to 1Mb in size and names of up to 256 bytes. Exceeding these limits will cause a buffer overflow and unpredictable failure.

The dummy filesystem is an implementation of the generic API which allows a true filesystem to be attached to the tftp server in the network stack.

We have been testing the tftp server by running the test on the target board, then using two different host computers connecting to the different target interfaces, putting a file from each, getting the “uu” file, and getting the file from the other computer. This verifies that data is preserved during the transfer as well as interworking with standard tftp applications.

Maintenance Tools

set_mac_address `- set MAC address(es) of interfaces in NVRAM`

This program makes an example ioctl() call SIOCSIFHWADDR "Socket IO Set InterFace HardWare ADDRess" to set the MAC address on targets where this is supported and enabled in the configuration. You must edit the source to choose a MAC address and further edit it to allow this very dangerous operation. Not all ethernet drivers support this operation, because most ethernet hardware does not support it - or it comes pre-set from the factory. Do not use this program.

APIs

Standard networking

The APIs for the standard networking calls such as socket(), recv() and so on, are in header files relative to the top-level include directory, within the standard subdirectories as conventionally found in /usr/include. For example:

```
-rw-r-----     5582 May  5 11:36 install/include/arpa/tftp.h
-rw-r-----     5250 Apr 20 14:20 install/include/netinet/tcpip.h
-rw-r-----    17292 Apr 20 14:20 install/include/sys/socket.h
-rw-r-----    15022 May  3 13:05 install/include/sys/socketvar.h
-rw-r-----     9615 Apr 20 14:20 install/include/sys/sockio.h
```

`network.h` at the top level defines various extensions, for example the API `init_all_network_interfaces(void)` described above. We advise including `network.h` whether you use these features or not.

In general, using the networking code may require definition of two symbols: _KERNEL and __ECOS. _KERNEL is not normally required; __ECOS is normally required. So add this to your compile lines for files which use the network stack:

```
    -D__ECOS
```

To expand a little, it’s like this because this is a port of a standard distribution external to Red Hat. One goal is to perturb the sources as little as possible, so that upgrading and maintenance from the external distribution is simplified. The __ECOS symbol marks out Red Hat’s additions in making the port. The _KERNEL symbol is traditional UNIX practice: it distinguishes a compilation which is to be linked into the kernel from one which is part of an application. eCos applications are fully linked, so this distinction does not apply. _KERNEL can however be used to control the visibility of the internals of the stack, so depending on what features your application uses, it may or may not be necessary.

The include file `network.h` undefines _KERNEL unconditionally, to provide an application-like compilation environment. If you were writing code which, for example, enumerates the stack's internal structures, that is a kernel-like compilation environment, so you would need to define _KERNEL (in addition to __ECOS) and avoid including `network.h`.

Enhanced Select()

The network stack supports an extension to the standard select semantics which allows all threads that are waiting to be restarted even if the select conditions are not satisfied.

The standard select() API:

```
int
select(int nfd, fd_set *in, fd_set *out, fd_set *ex,
          struct timeval *tv);
```

does not support the restart.

The additional API:

```
int
cyg_select_with_abort(int nfd, fd_set *in, fd_set *out, fd_set *ex,
                       struct timeval *tv)
```

behaves exactly as select() with the additional feature that a call to

```
void cyg_select_abort(void)
```

will cause all threads waiting in any `cyg_select_with_abort()` call to cease waiting and continue execution.

TFTP

The TFTP client and server are described in `tftp_support.h`; the client API is simple and can be easily understood by reading `tftp_client_test.c`.

The server is more complex. It requires a filesystem implementation to be supplied by the user, and attached to the tftp server by means of a vector of function pointers:

```
struct tftpd_fileops {
             int (*open)(const char *, int);
             int (*close)(int);
             int (*write)(int, const void *, int);
             int (*read)(int, void *, int);
};
```

These functions have the obvious semantics. The structure describing the filesystem is an argument to the `tftpd_start(int, struct tftpd_fileops *);` call. The first argument is the port to use for the server.

As discussed in the description of the tftp_server_test above, an example filesystem is provided in `net/tcpip/VERSION/src/lib/tftp_dummy_file.c` for use by the tftp server test. The dummy filesystem is not a supported part of the network stack, it exists purely for demonstration purposes.

DHCP

This API publishes a routine to maintain DHCP state, and a semaphore that is signalled when a lease requires attention: this is your clue to call the aforementioned routine.

The intent with this API is that a simple DHCP client thread, which maintains the state of the interfaces, can go as follows: (after `init_all_networks` is called from elsewhere)

```
while ( 1 ) {
        while ( 1 ) {
            cyg_semaphore_wait( &dhcp_needs_attention );
            if ( ! dhcp_bind() ) // a lease expired
                break; // If we need to re-bind
         }
        dhcp_halt(); // tear everything down
        init_all_network_interfaces(); // re-initialize
}
```

and if the application does not want to suffer the overhead of a separate thread and its stack for this, this functionality can be placed in the app’s server loop in an obvious fashion. That is the goal of breaking out these internal elements. For example, some server might be arranged to poll DHCP from time to time like this:

```
while ( 1 ) {
    init_all_network_interfaces();
    open-my-listen-sockets();
    while ( 1 ) {
       serve-one-request();
       // sleeps if no connections, but not forever;
       // so this loop is polled a few times a minute...
       if ( cyg_semaphore_trywait( &dhcp_needs_attention )) {
             if ( ! dhcp_bind() ) {
                 close-my-listen-sockets();
                 dhcp_halt();
                 break;
             }
       }
    }
}
```

If the configuration option CYGOPT_NET_DHCP_DHCP_THREAD is defined, then eCos provides a thread as described initially. Independent of this option, initialization of the interfaces still occurs in `init_all_network_interfaces()` and your startup code must call that. It will start the DHCP management thread if configured. If a lease fails to be renewed, the management thread will shut down all interfaces and attempt to initialize all the interfaces again from scratch. This may cause chaos in the app, which is why managing the DHCP state in an application aware thread is actually better, just far less convenient for testing.

TCP/IP Library Reference

- accept
- bind
- close
- connect
- gethostbyname, gethostbyaddr
- getpeername
- getprotobyname, getprotobynumber
- getservent, getservbyname, getservbyport
- getsockname
- getsockopt
- inet_aton, inet_addr, inet_ntoa
- ioctl
- listen
- read
- recvfrom
- select
- sendto
- setsockopt
- shutdown
- socket
- write

accept

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

SYNOPSIS
       #include <network.h>

       int accept(int s, struct sockaddr *addr, int *addrlen);

DESCRIPTION
       The  argument  s  is  a  socket that has been created with
       socket(2), bound to an address with bind(2), and  is  lis-
       tening  for  connections  after  a  listen(2).  The accept
       function extracts the  first  connection  request  on  the
       queue  of  pending  connections, creates a new socket with
       the same  properties  of  s,  and  allocates  a  new  file
       descriptor  for the socket.  If no pending connections are
       present on the queue, and the socket is not marked as non-
       blocking,  accept  blocks the caller until a connection is
       present.  If the socket  is  marked  non-blocking  and  no
       pending  connections  are  present  on  the  queue, accept
       returns an error as described below.  The socket  returned
       by accept may not be used to accept more connections.  The
       original socket s remains open.

       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 param-
       eter  is  determined by the domain in which the communica-
       tion is occurring.  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(2) a socket for the purposes of
       doing an accept by selecting it for read.

       For certain protocols which require an explicit  confirma-
       tion,  such  as DECNet, accept can be thought of as merely
       dequeuing the next connection  request  and  not  implying
       confirmation.   Confirmation  can  be  implied by a normal
       read or write on the new file  descriptor,  and  rejection
       can  be  implied by closing the new socket. Currently only
       DECNet has these semantics on Linux.

NOTES
       If you want accept to never  block  the  listening  socket
       needs  to  have  the  non blocking flag set. Assuming that
       there is always a connection waiting after select returned
       true  is  not  reliable,  because  the connection might be
       removed by  an  asynchronous  network  error  between  the
       select/poll returning and the accept call. The application
       would hang then if the listen socket is not non  blocking.

RETURN VALUES
       The  call returns -1 on error.  If it succeeds, it returns
       a non-negative  integer  that  is  a  descriptor  for  the
       accepted socket.

ERRORS
       EBADF   The descriptor is invalid.

       ENOTSOCK
               The descriptor references a file, not a socket.

       EOPNOTSUPP
               The referenced socket is not of type  SOCK_STREAM.

       EAGAIN  The socket is marked non-blocking and  no  connec-
               tions are present to be accepted.
```

```
ENOBUFS, ENOMEM
        Not enough free memory.
```

bind

NAME

bind - bind a name to a socket

SYNOPSIS

```
#include <network.h>

int   bind(int   sockfd,  struct  sockaddr  *my_addr,  int
addrlen);
```

DESCRIPTION

bind gives the socket sockfd the local address my_addr. my_addr is addrlen bytes long. Traditionally, this is called “assigning a name to a socket.” (When a socket is created with socket(2), it exists in a name space (address family) but has no name assigned.)

Before a SOCK_STREAM socket is put into the LISTEN state to receive connections, you usually need to first assign a local address using bind to make the socket visible.

NOTES

Binding a name that is not in the abstract namespace in the UNIX domain creates a socket in the file system that must be deleted by the caller when it is no longer needed (using unlink(2)).

The rules used in name binding vary between communication domains. Consult the manual entries in section 4 for detailed information. For IP see ip(4) and for PF_UNIX see unix(4). If you want to listen to every local interface for IPv4 set the sin_addr member of the IP-specific sockaddr_in to INADDR_ANY. For IP only one socket may be bound to a specific local address/port pair. For TCP a bound local socket endpoint (address/port pair) is unavailable for some time after closing the socket, unless the SO_REUSEADDR flag is set. Note that carelessly setting SO_REUSEADDR might make TCP more unreliable unless PAWS is used (see tcp(4)); the delay is needed to handle old packets still in the network.

IP sockets may also bind to a broadcast or multicast address.

RETURN VALUE

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

ERRORS

EBADF   sockfd is not a valid descriptor.

EINVAL The socket is already bound to an address. This may change in the future: see linux/unix/sock.c for details.

ENOTSOCK

Argument is a descriptor for a file, not a socket.

close

```
NAME
       close - close a file descriptor

SYNOPSIS

       int close(int fd);

DESCRIPTION
       close  closes  a  file  descriptor,  so  that it no longer
       refers to any file and may be reused.

       If fd is the last copy of a particular file descriptor the
       resources associated with it are freed.

RETURN VALUE
       close returns zero on success, or -1 if an error occurred.

ERRORS
       EBADF  fd isn’t a valid open file descriptor.
```

connect

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

SYNOPSIS
       #include <network.h>

       int  connect(int  sockfd,  struct sockaddr *serv_addr, int
       addrlen);

DESCRIPTION
       The parameter sockfd is a socket.  If  the  socket  is  of
       type  SOCK_DGRAM,  this call specifies the peer with which
       the socket is to be associated; this address  is  that  to
       which  datagrams are to be sent, and the only address from
       which datagrams are to be received.  If the socket  is  of
       type  SOCK_STREAM, this call attempts to make a connection
       to another socket.   The  other  socket  is  specified  by
       serv_addr, which is an address in the communications space
       of the socket.  Each communications space  interprets  the
       serv_addr  parameter  in  its  own way.  Generally, stream
       sockets may successfully connect only once; datagram sock-
       ets may use connect multiple times to change their associ-
       ation.  Datagram sockets may dissolve the  association  by
       connecting  to an address with the sa_family sockaddr mem-
       ber set to AF_UNSPEC.

RETURN VALUE
       If the connection or binding succeeds, zero  is  returned.
       On  error, -1 is returned, and errno is set appropriately.

ERRORS
       The following are general socket errors only.   There  may
       be other domain-specific error codes.

       EBADF   Bad descriptor.

       ENOTSOCK
               The descriptor is not associated with a socket.

       EISCONN The socket is already connected.

       ECONNREFUSED
               Connection refused at server.

       ETIMEDOUT
               Timeout while attempting connection.

       ENETUNREACH
               Network is unreachable.

       EADDRINUSE
               Address is already in use.

       EINPROGRESS
               The socket is non-blocking and the connection can-
               not  be  completed immediately.  It is possible to
               select(2) or poll(2) for completion  by  selecting
               the  socket  for  writing.  After select indicates
               writability,  use  getsockopt(2)   to   read   the
               SO_ERROR  option  at level SOL_SOCKET to determine
               whether connect completed  successfully  (SO_ERROR
               is zero) or unsuccessfully (SO_ERROR is one of the
               usual error codes  listed  above,  explaining  the
               reason for the failure).

       EALREADY
               The  socket is non-blocking and a previous connec-
               tion attempt has not yet been completed.

       EAFNOSUPPORT
```

```
        The passed address didn’t have the correct address
        family in its sa_family field.

EACCES  The  user  tried to connect to a broadcast address
        without having the socket broadcast flag  enabled.
```

gethostbyname, gethostbyaddr

NAME

gethostbyname, gethostbyaddr, herror, hstrerror - get network host entry

SYNOPSIS

```
#include <network.h>

struct hostent *gethostbyname(const char *name);

struct hostent *gethostbyaddr(const char *addr, int len, int type);

void herror(const char *s);

const char * hstrerror(int err);
```

DESCRIPTION

The gethostbyname() function returns a structure of type hostent for the given host name. Here name is either a host name, or an IPv4 address in standard dot notation, or an IPv6 address in colon (and possibly dot) notation. (See RFC 1884 for the description of IPv6 addresses.) If name is an IPv4 or IPv6 address, no lookup is performed and gethostbyname() simply copies name into the h_name field and its struct in_addr equivalent into the h_addr_list[0] field of the returned hostent structure. If name doesn’t end in a dot and the environment variable HOSTALIASES is set, the alias file pointed to by HOSTALIASES will first be searched for name. (See hostname(7) for the file format.) The current domain and its parents are searched unless name ends in a dot.

The gethostbyaddr() function returns a structure of type hostent for the given host address addr of length len and address type type. The only valid address type is currently AF_INET.

The (obsolete) herror() function prints the error message associated with the current value of h_errno on stderr.

The (obsolete) hstrerror() function takes an error number (typically h_errno) and returns the corresponding message string.

The domain name queries carried out by gethostbyname() and gethostbyaddr() use a combination of any or all of the name server named(8), a broken out line from /etc/hosts, and the Network Information Service (NIS or YP), depending upon the contents of the order line in /etc/host.conf. (See resolv+(8)). The default action is to query named(8), followed by /etc/hosts.

The hostent structure is defined in <netdb.h> as follows:

```
struct hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* host address type */
        int     h_length;       /* length of address */
        char    **h_addr_list;  /* list of addresses */
}
#define h_addr  h_addr_list[0]  /* for backward compatibility */
```

The members of the hostent structure are:

h_name The official name of the host.

h_aliases

A zero-terminated array of alternative names for the host.

h_addrtype

```
                     The type of address; always AF_INET at present.

              h_length
                     The length of the address in bytes.

              h_addr_list
                     A  zero-terminated  array  of network addresses for
                     the host in network byte order.

              h_addr The first address in h_addr_list for backward  com-
                     patibility.

       RETURN VALUE
              The  gethostbyname()  and gethostbyaddr() functions return
              the hostent structure  or  a  NULL  pointer  if  an  error
              occurs.   On  error,  the  h_errno variable holds an error
              number.

       ERRORS
              The variable h_errno can have the following values:

              HOST_NOT_FOUND
                     The specified host is unknown.

              NO_ADDRESS or NO_DATA
                     The requested name is valid but does not have an IP
                     address.

              NO_RECOVERY
                     A non-recoverable name server error occurred.

              TRY_AGAIN
                     A temporary error occurred on an authoritative name
                     server.  Try again later.
```

getpeername

```
NAME
       getpeername - get name of connected peer

SYNOPSIS
       #include <network.h>

       int  getpeername(int  s,  struct sockaddr *name, socklen_t
       *namelen);

DESCRIPTION
       Getpeername returns the name  of  the  peer  connected  to
       socket  s.  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
       On  success,  zero is returned.  On error, -1 is returned,
       and errno is set appropriately.

ERRORS
       EBADF   The argument s is not a valid descriptor.

       ENOTSOCK
               The argument s is a file, not a socket.

       ENOTCONN
               The socket is not connected.

       ENOBUFS Insufficient resources were available in the  sys-
               tem to perform the operation.
```

getprotobyname, getprotobynumber

```
NAME
       getprotobyname,  getprotobynumber - get protocol entry

SYNOPSIS
       #include <network.h>

       struct protoent *getprotobyname(const char *name);

       struct protoent *getprotobynumber(int proto);

DESCRIPTION
       The getprotobyname() function returns a protoent structure
       for the line from /etc/protocols that matches the protocol
       name name.

       The  getprotobynumber() function returns a protoent struc-
       ture for the line that matches the protocol number number.

       The protoent structure is defined in  as follows:

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

       The members of the protoent structure are:

       p_name The official name of the protocol.

       p_aliases
              A zero terminated list of alternative names for the
              protocol.
       p_proto
              The protocol number.

RETURN VALUE
       The getprotobyname() and getprotobynumber()
       functions return the protoent structure, or a NULL pointer
       if an error occurs.
```

getservent, getservbyname, getservbyport

```
NAME
       getservent, getservbyname, getservbyport - get service entry

SYNOPSIS
       #include <network.h>

       struct servent *getservbyname(const char *name, const char *proto);

       struct servent *getservbyport(int port, const char *proto);

DESCRIPTION
       The  getservbyname()  function returns a servent structure
       for the line from /etc/services that matches  the  service
       name using protocol proto.

       The  getservbyport()  function returns a servent structure
       for the line that matches the port port given  in  network
       byte order using protocol proto.

       The servent structure is defined in <netdb.h> as follows:

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

       The members of the servent structure are:

       s_name The official name of the service.

       s_aliases
              A zero terminated list of alternative names for the
              service.

       s_port The  port  number  for the service given in network
              byte order.

       s_proto
              The name of the protocol to use with this  service.

RETURN VALUE
       The   getservbyname()  and  getservbyport()
       functions return the servent structure, or a NULL  pointer
       if an error occurs.
```

getsockname

```
NAME
       getsockname - get socket name

SYNOPSIS
       #include <network.h>

       int   getsockname(int   s  ,  struct  sockaddr  *  name  ,
       socklen_t * namelen )

DESCRIPTION
       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).

RETURN VALUE
       On success, zero is returned.  On error, -1  is  returned,
       and  errno  is  set appropriately.  A 0 is returned if the
       call succeeds, -1 if it fails.

ERRORS
       EBADF   The argument s is not a valid descriptor.

       ENOTSOCK
               The argument s is a file, not a socket.

       ENOBUFS Insufficient resources were available in the  sys-
               tem to perform the operation.
```

getsockopt

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

SYNOPSIS
       #include <network.h>

       int  getsockopt(int  s, int level, int optname, void *opt-
       val, socklen_t *optlen);

       int setsockopt(int s, int level, int optname,  const  void
       *optval, socklen_t optlen);

DESCRIPTION
       Getsockopt  and  setsockopt manipulate the options associ-
       ated with a socket.  Options may exist at multiple  proto-
       col  levels;  they  are  always  present  at the uppermost
       socket level.

       When manipulating socket options the level  at  which  the
       option  resides  and the name of the option must be speci-
       fied.  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  proto-
       col  controlling  the option is supplied.  For example, to
       indicate that an option is to be interpreted  by  the  TCP
       protocol,  level  should  be set to the protocol number of
       TCP; see getprotoent(3).

       The parameters 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 NULL.

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

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

       For  a  description  of  the  available socket options see
       socket(7) and the appropriate protocol man pages.

RETURN VALUE
       On success, zero is returned.  On error, -1  is  returned,
       and errno is set appropriately.

ERRORS
       EBADF   The argument s is not a valid descriptor.

       ENOTSOCK
               The argument s is a file, not a socket.

       ENOPROTOOPT
               The option is unknown at the level indicated.
```

inet_aton, inet_addr, inet_ntoa

```
NAME
       inet_aton, inet_addr, inet_ntoa -  Internet  address
       manipulation routines

SYNOPSIS
       #include <network.h>

       int inet_aton(const char *cp, struct in_addr *inp);

       unsigned long int inet_addr(const char *cp);

       char *inet_ntoa(struct in_addr in);

DESCRIPTION
       inet_aton() converts the Internet host address cp from the
       standard numbers-and-dots notation into  binary  data  and
       stores  it  in the structure that inp points to. inet_aton
       returns nonzero if the address is valid, zero if not.

       The  inet_addr()  function  converts  the  Internet   host
       address cp from numbers-and-dots notation into binary data
       in  network  byte  order.   If  the  input   is   invalid,
       INADDR_NONE (usually -1) is returned.  This is an obsolete
       interface to inet_aton, described immediately above; it is
       obsolete  because -1 is a valid address (255.255.255.255),
       and inet_aton provides a cleaner  way  to  indicate  error
       return.

       The   inet_ntoa()  function  converts  the  Internet  host
       address in given in network byte  order  to  a  string  in
       standard   numbers-and-dots   notation.    The  string  is
       returned in a statically allocated  buffer,  which  subse-
       quent calls will overwrite.
```

ioctl

```
NAME
       ioctl - control device

SYNOPSIS
       #include

       int ioctl(int d, int request, ...)

       [The  “third”  argument  is  traditionally char *argp, and
       will be so named for this discussion.]

DESCRIPTION
       The  ioctl  function  manipulates  the  underlying  device
       parameters  of special files.  In particular, many operat-
       ing characteristics of sockets and network devices
       may be controlled with ioctl requests.  The argu-
       ment d must be an open file descriptor.

       An ioctl request has encoded in it whether the argument is
       an  in  parameter  or  out  parameter, and the size of the
       argument argp in bytes.  Macros and defines used in speci-
       fying   an   ioctl   request   are  located  in  the  file
       .

RETURN VALUE
       On success, zero is returned.  On error, -1  is  returned,
       and errno is set appropriately.

ERRORS
       EBADF  d is not a valid descriptor.

       EFAULT argp references an inaccessible memory area.

       ENOTTY d  is  not  associated  with  a  character  special
              device.

       ENOTTY The specified request does not apply to the kind of
              object that the descriptor d references.

       EINVAL Request or argp is not valid.
```

listen

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

SYNOPSIS
       #include <network.h>

       int listen(int s, int backlog);

DESCRIPTION
       To  accept  connections,  a  socket  is first created with
       socket(2), a willingness to  accept  incoming  connections
       and  a  queue limit for incoming connections are specified
       with listen, and then the connections  are  accepted  with
       accept(2).   The  listen  call  applies only to sockets of
       type SOCK_STREAM or SOCK_SEQPACKET.

       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 may receive
       an  error  with  an  indication of ECONNREFUSED or, if the
       underlying protocol supports retransmission,  the  request
       may be ignored so that retries may succeed.

RETURN VALUE
       On success, zero is returned.  On error, -1  is  returned,
       and errno is set appropriately.

ERRORS
       EBADF   The argument s is not a valid descriptor.

       ENOTSOCK
               The argument s is not a socket.

       EOPNOTSUPP
               The socket is not of a type that supports the lis-
               ten operation.
```

read

```
NAME
       read - read from a file descriptor

SYNOPSIS
       ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
       read()  attempts  to  read  up  to  count  bytes from file
       descriptor fd into the buffer starting at buf.

       If count is zero, read() returns zero  and  has  no  other
       results.


RETURN VALUE
       On success, the number of bytes  read  is  returned  (zero
       indicates  end of file), and the file position is advanced
       by this number.  It is not an  error  if  this  number  is
       smaller  than the number of bytes requested; this may hap-
       pen for example because fewer bytes are actually available
       right  now (maybe because we were close to end-of-file, or
       because we are reading from a pipe, or from  a  terminal),
       or  because read() was interrupted by a signal.  On error,
       -1 is returned, and errno is set  appropriately.  In  this
       case  it is left unspecified whether the file position (if
       any) changes.

ERRORS
       EINTR   The call was interrupted by a  signal  before  any
               data was read.

       EAGAIN  Non-blocking  I/O  has  been selected using O_NON-
               BLOCK and no data was  immediately  available  for
               reading.

       EIO     I/O  error.  This will happen for example when the
               process is in a background process group, tries to
               read  from  its  controlling tty, and either it is
               ignoring or blocking SIGTTIN or its process  group
               is  orphaned.   It  may also occur when there is a
               low-level I/O error while reading from a  disk  or
               tape.

      EISDIR  fd refers to a directory.

       EBADF   fd  is  not a valid file descriptor or is not open
               for reading.

       EINVAL  fd is attached to an object  which  is  unsuitable
               for reading.

       Other  errors may occur, depending on the object connected
       to fd.  POSIX allows a  read  that  is  interrupted  after
       reading  some  data to return -1 (with errno set to EINTR)
       or to return the number of bytes already read.
```

recvfrom

NAME

recvfrom - receive a message from a socket

SYNOPSIS

```
#include <network.h>

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

DESCRIPTION

The recvfrom call is used to receive messages from a socket, and may be used to receive data on a socket whether or not it is connection-oriented.

If from is not NULL, and the socket is not connection-oriented, 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 routine returns the length of the message on successful completion. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see socket(2)).

If no messages are available at the socket, the receive calls wait for a message to arrive, unless the socket is nonblocking (see fcntl(2)) in which case the value -1 is returned and the external variable errno set to EAGAIN. The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested.

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

The flags argument to a recvfrom call is formed by OR'ing one or more of the following values:

MSG_OOB

This flag requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at the head of the normal data queue, and thus this flag cannot be used with such protocols.

MSG_PEEK

This flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data.

MSG_WAITALL

This flag requests that the operation block until the full request is satisfied. However, the call may still return less data than requested if a signal is caught, an error or disconnect occurs, or the next data to be received is of a different type than that returned.

MSG_ERRQUEUE

Receive packet from the error queue

MSG_NOSIGNAL

This flag turns off raising of SIGPIPE on stream sockets when the other end disappears.

MSG_ERRQUEUE

This flag specifies that queued errors should be received from the socket error queue. The error is passed in a ancilliary message with a type dependent on the protocol (for IP IP_RECVERR). The error is supplied in a sock_extended_error structure:

```
#define SO_EE_ORIGIN_NONE       0
#define SO_EE_ORIGIN_LOCAL      1
#define SO_EE_ORIGIN_ICMP       2
#define SO_EE_ORIGIN_ICMP6      3

struct sock_extended_err
{
    __u32           ee_errno;   /* error number */
    __u8            ee_origin;  /* where the error originated */
    __u8            ee_type;    /* type */
    __u8            ee_code;    /* code */
    __u8            ee_pad;
    __u32           ee_info;    /* additional information */
    __u32           ee_data;    /* other data */
};

struct sockaddr *SOCK_EE_OFFENDER(struct sock_extended_err *);
```

ee_errno contains the errno number of the queued error. ee_origin is the origin code of where the error originated. The other fields are protocol specific. SOCK_EE_OFFENDER returns a pointer to the address of the network object where the error originated from. If this address is not known, the sa_family member of the sockaddr contains AF_UNSPEC and the other fields of the sockaddr are undefined. The payload of the packet that caused the error is passed as normal data.

For local errors, no address is passed (this can be checked with the cmsg_len member of the cmsghdr). For error receives, the MSG_ERRQUEUE is set in the msghdr. After a error has been passed, the pending socket error is regenerated based on the next queued error and will be passed on the next socket operation.

The msg_flags field is set on return according to the message received. MSG_EOR indicates end-of-record; the data returned completed a record (generally used with sockets of type SOCK_SEQPACKET). MSG_TRUNC indicates that the trailing portion of a datagram was discarded because the datagram was larger than the buffer supplied. MSG_CTRUNC indicates that some control data were discarded due to lack of space in the buffer for ancillary data. MSG_OOB is returned to indicate that expedited or out-of-band data were received. MSG_ERRQUEUE indicates that no data was received but an extended error from the socket error queue.

RETURN VALUES

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

ERRORS

These are some standard errors generated by the socket layer. Additional errors may be generated and returned from the underlying protocol modules; see their manual pages.

EBADF   The argument s is an invalid descriptor.

ENOTSOCK
        The argument s does not refer to a socket.

EAGAIN The socket is marked non-blocking and the receive

```
                operation would block, or a  receive  timeout  had
                been  set  and the timeout expired before data was
                received.

        EINTR   The receive was interrupted by delivery of a  sig-
                nal before any data were available.

        EINVAL  Invalid argument passed.
```

select

```
NAME
       select,  FD_CLR,  FD_ISSET,  FD_SET, FD_ZERO - synchronous
       I/O multiplexing

SYNOPSIS
       #include <network.h>

       int  select(int  n,  fd_set  *readfds,  fd_set  *writefds,
       fd_set *exceptfds, struct timeval *timeout);

       FD_CLR(int fd, fd_set *set);
       FD_ISSET(int fd, fd_set *set);
       FD_SET(int fd, fd_set *set);
       FD_ZERO(fd_set *set);

DESCRIPTION
       select  waits  for  a number of file descriptors to change
       status.

       Three independent sets of descriptors are watched.   Those
       listed  in  readfds  will  be watched to see if characters
       become available for reading, those in  writefds  will  be
       watched  to  see if it is ok to immediately write on them,
       and those in exceptfds will be watched for exceptions.  On
       exit,  the  sets  are  modified in place to indicate which
       descriptors actually changed status.

       Four macros are provided to manipulate the sets.   FD_ZERO
       will clear a set.  FD_SET and FD_CLR add or remove a given
       descriptor from  a  set.   FD_ISSET  tests  to  see  if  a
       descriptor is part of the set; this is useful after select
       returns.

       n is the highest-numbered descriptor in any of  the  three
       sets, plus 1.

       timeout  is  an  upper bound on the amount of time elapsed
       before select returns. It may be zero, causing  select  to
       return  immediately.  If  timeout  is  NULL  (no timeout),
       select can block indefinitely.

RETURN VALUE
       On success, select returns the number of descriptors  con-
       tained  in  the  descriptor sets, which may be zero if the
       timeout expires before anything interesting  happens.   On
       error, -1 is returned, and errno is set appropriately; the
       sets and timeout become undefined, so do not rely on their
       contents after an error.

ERRORS
       EBADF   An invalid file descriptor was given in one of the
               sets.

       EINTR   A non blocked signal was caught.

       EINVAL  n is negative.

       ENOMEM  select was unable to allocate memory for  internal
               tables.

NOTES
       Some  code calls select with all three sets empty, n zero,
       and a non-null timeout as a fairly portable way  to  sleep
       with subsecond precision.

EXAMPLE
       #include
       #include
       #include
       #include
```

```
int
main(void)
{
    fd_set rfds;
    struct timeval tv;
    int retval;

    /* Watch stdin (fd 0) to see when it has input. */
    FD_ZERO(&rfds);
    FD_SET(0, &rfds);
    /* Wait up to five seconds. */
    tv.tv_sec = 5;
    tv.tv_usec = 0;

    retval = select(1, &rfds, NULL, NULL, &tv);
    /* Don’t rely on the value of tv now! */

    if (retval)
        printf(“Data is available now.\n”);
        /* FD_ISSET(0, &rfds) will be true. */
    else
        printf(“No data within five seconds.\n”);

    exit(0);
}
```

Generally portable to/from non-BSD systems supporting clones of the BSD socket layer (including System V variants). However, note that the System V variant typically sets the timeout variable before exit, but the BSD variant does not.

sendto

```
NAME
       sendto - send a message from a socket

SYNOPSIS
       #include <network.h>

       int sendto(int s, const void *msg, int len,  unsigned  int
       flags, const struct sockaddr *to, int tolen);

DESCRIPTION
       Sendto is used to transmit  a  message
       to  another socket.

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

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

       When  the message does not fit into the send buffer of the
       socket, send normally blocks, unless the socket  has  been
       placed  in non-blocking I/O mode.  In non-blocking mode it
       would return EAGAIN in this case.  The select(2) call  may
       be  used  to  determine  when  it is possible to send more
       data.

       The flags parameter may include one or more of the follow-
       ing:

              #define MSG_OOB           0x1     /* process out-of-band data */
             #define MSG_DONTROUTE     0x4    /* bypass routing, use direct interface
*/
              #define MSG_DONTWAIT      0x40    /* don’t block */
              #define MSG_NOSIGNAL      0x2000  /* don’t raise SIGPIPE */

       MSG_OOB
              Sends out-of-band data on sockets that support this
              notion (e.g.  SOCK_STREAM); the underlying protocol
              must also support out-of-band data.

       MSG_DONTROUTE
              Bypasses  the  usual routing table lookup and sends
              the packet directly to the interface  described  by
              the  destination address. This is usually used only
              by diagnostic or routing programs.

       MSG_DONTWAIT
              Enables non-blocking operation;  if  the  operation
              would block, EAGAIN is returned.

       MSG_NOSIGNAL
              Requests  not  to  send SIGPIPE on errors on stream
              oriented sockets when the other end breaks the con-
              nection. The EPIPE error is still returned.

       See recv(2) for a description of the msghdr structure. You
       may send control information  using  the  msg_control  and
       msg_controllen  members. The maximum control buffer length
       the kernel can process is  limited  by  the  net.core.opt-
       mem_max sysctl; see socket(4).

RETURN VALUES
       The  calls  return the number of characters sent, or -1 if
       an error occurred.

ERRORS
```

These are some standard errors generated by the socket layer. Additional errors may be generated and returned from the underlying protocol modules; see their respective manual pages.

EBADF An invalid descriptor was specified.

ENOTSOCK
The argument s is not a socket.

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

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

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

EINTR A signal occurred.

ENOMEM No memory available.

EINVAL Invalid argument passed.

EPIPE The local end has been shut down on a connection oriented socket. In this case the process will also receive a SIGPIPE unless MSG_NOSIGNAL is set.

setsockopt

NAME

getsockopt, setsockopt - get and set options on sockets

SYNOPSIS

```
#include <network.h>

int  getsockopt(int  s, int level, int optname, void *opt-
val, socklen_t *optlen);

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

DESCRIPTION

Getsockopt and setsockopt manipulate the options associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost socket level.

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

The parameters 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 NULL.

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

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

For a description of the available socket options see socket(7) and the appropriate protocol man pages.

RETURN VALUE

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

ERRORS

EBADF   The argument s is not a valid descriptor.

ENOTSOCK
The argument s is a file, not a socket.

ENOPROTOOPT
The option is unknown at the level indicated.

shutdown

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

SYNOPSIS
       #include <network.h>

       int shutdown(int s, int how);

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, further receives will be disallowed.  If how
       is 1, further sends will be disallowed.  If how is 2, fur-
       ther sends and receives will be disallowed.

RETURN VALUE
       On  success,  zero is returned.  On error, -1 is returned,
       and errno is set appropriately.

ERRORS
       EBADF   s is not a valid descriptor.

       ENOTSOCK
               s is a file, not a socket.

       ENOTCONN
               The specified socket is not connected.
```

socket

NAME

socket - create an endpoint for communication

SYNOPSIS

```
#include <network.h>

int socket(int domain, int type, int protocol);
```

DESCRIPTION

Socket creates an endpoint for communication and returns a descriptor.

The domain parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. These families are defined in <network.h>. The currently understood formats include:

PF_INET

IPv4 Internet protocols; see ip(4)

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

SOCK_STREAM

Provides sequenced, reliable, two-way connection-based byte streams. An out-of-band data transmission mechanism may be supported.

SOCK_DGRAM

Supports datagrams (connectionless, unreliable messages of a fixed maximum length).

SOCK_SEQPACKET

Provides a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer is required to read an entire packet with each read system call.

SOCK_RAW

Provides raw network protocol access.

The protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the “communication domain” in which communication is to take place; see protocols(5). See getprotoent(3) on how to map protocol name strings to protocol numbers.

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

The communications protocols which implement a SOCK_STREAM ensure 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 When SO_KEEPALIVE is enabled on the socket the protocol checks in a proto-

```
       col-specific  manner  if  the other end is still alive.

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

       When  the network signals an error condition to the proto-
       col module (e.g.  using a ICMP message for IP) the pending
       error  flag  is set for the socket.  The next operation on
       this socket will return the  error  code  of  the  pending
       error.  For some protocols it is possible to enable a per-
       socket error queue to retrieve detailed information  about
       the error; see IP_RECVERR in ip(4).

       The  operation  of  sockets  is controlled by socket level
       options.  These options  are  defined  in  .
       Setsockopt(2)  and  getsockopt(2)  are used to set and get
       options, respectively.

RETURN VALUES
       -1 is returned if an error occurs;  otherwise  the  return
       value is a descriptor referencing the socket.

ERRORS
       EPROTONOSUPPORT
                     The protocol type or the specified protocol is not
               supported within this domain.

       EMFILE  There are too many open files.

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

       ENOBUFS or ENOMEM
                 Insufficient memory is available.  The socket can-
                      not  be  created  until  sufficient  resources are
               freed.

        EINVAL   Unknown protocol, or protocol  family  not  avail-
               able.
```

write

```
NAME
       write - write to a file descriptor

SYNOPSIS
       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
       write  writes  up to count bytes to the file referenced by
       the file descriptor fd from the buffer  starting  at  buf.

RETURN VALUE
       On success, the number of bytes written are returned (zero
       indicates nothing was written).  On error, -1 is returned,
      and  errno is set appropriately.

ERRORS
       EBADF  fd is not a valid file descriptor or  is  not  open
             for writing.

       EINVAL fd is attached to an object which is unsuitable for
             writing.

      EPIPE  fd is connected to a socket  whose  reading
             end  is closed.

       EAGAIN  Non-blocking I/O has been selected using O_NONBLOCK
              and  there  was  no room in the pipe or socket con-
             nected to fd to write the data immediately.

      EINTR  The call was interrupted before  any
             data was written.

       ENOSPC The  device  containing  the file referred to by fd
             has no room for the data.

      EIO   A low-level I/O error occurred.
```

SNMP for eCos

Release notes

Version

This is a port of UCD-SNMP-4.1.2

See http://ucd-snmp.ucdavis.edu/ for details. And send them a postcard.

Package contents

The SNMP/eCos package consists of two eCos packages; the SNMP library and the SNMP agent.

The sources are arranged this way partly for consistency with the original release from UCD, and so as to accommodate possible future use of the SNMP library without having an agent present. That could be used to build an eCos-based SNMP client application.

The library contains support code for talking SNMP over the net - the SNMP protocol itself - and a MIB file parser (ASN-1) which is not used in the agent case.

The agent contains the application specific handler files to get information about the system into the SNMP world, together with the SNMP agent thread (snmpd in UNIX terms).

Additional requirements

These packages are intended for use with eCos release 1.4.4 or later; an updated TCP/IP stack which exposes additional statistical information to SNMP; and an updated ethernet device driver where applicable.

Current release contents

The release consists of two EPKs:

- Two SNMP packages; versions are to accompany the 1.4.*x* release:

  `snmp-10by.epk` (i.e. 1.0beta*y*) and

  `/1_0by/` for ‘`/VERSION/`’

  in source repository pathnames, where y is the minor vesion number of the current release).
  305kB

- Updated TCP/IP networking package plus common ethernet support. Both versions are to accompany the 1.4.*x* release:

  `net-10by.epk` (i.e. 1.0beta*y*) and

  `/1_0by/` for ‘`/VERSION/`’

  in source repository pathnames, where y is the minor vesion number of the current release).
  494kB

The EPKs can be added to the current eCos release. See below for further instructions on selecting these new package versions for your build; be sure to get the versions above first.

To add the new EPKs to the eCos 1.4.*x* repository, run up the GUI Package Administration Tool, click ‘**Add...**’ and select all three .epk files (hold down the **Ctrl** key to extend the selection).

In the config tool, most packages should be version “v1_4_*x*”. The EBSA ethernet driver should be version “v1_4_*x*”. When you add the “Networking” and “Common ethernet support” packages to the config, they should be “v1_0b*y*”. When you add the “SNMP agent” and “SNMP library” packages to the config, they should be “v1_0b*y*”.

Please be sure to choose these versions in the config tool to build a fully working SNMP-capable eCos library.

If you have an existing configuration customized to your application which already uses the network and ethernet drivers, you can add the SNMP packages to it and switch over to the newer versions of the network and ether driver packages using the **Build**->**Packages** dialog. NB: SNMP requires the later versions of the network and ether drivers.

For command-line use (eg. under Linux), use the script ecosadmin.tcl which will be found in the packages directory. Ensure that either the environment variable ECOS_REPOSITORY is set correctly, or unset it and run the script from within the packages directory itself.

```
cd packages
./ecosadmin.tcl add ebsa285eth-1.4.x.epk
./ecosadmin.tcl add net-1.0by.epk
./ecosadmin.tcl add snmp-1.0by.epk
```

After this, use ecosconfig as usual, but remember that you may now have a choice of versions for the net and ether driver packages if you added this distribution to an existing repository rather than to a pure 1.4.*x* installation. Starting from scratch in a new empty build directory with “ecosconfig new ebsa285” will collect the newest versions of all the packages. Re-building in an existing build tree will not; to continue with an existing configuration, add the two SNMP packages, and change the versions of the networking, common ethernet, and ebsa ethernet driver to the new ones using the “ecosconfig version VERSION PACKAGE”, like this:

```
  % ecosconfig version v1_0by     net    net_drivers
  % ecosconfig version v1_4_x  ebsa285_eth_driver
```

(the VERSION part is in version-directory-name style, as in the repository)

MIBs supported

The standard set in MIB-II, together with the Ether-Like MIB. The MIB files used to compile the handlers in the agent and to “drive” the testing (`snmpwalk` *et al* under LINUX) are those acquired from that same UCD distribution.

These are the supported MIBs; all are below mib2 == 1.3.6.1.2.1:

```
system        { mib2 1 }
interfaces    { mib2 2 }
              [ address-translation “at” { mib2 3 } is deprecated ]
ip            { mib2 4 }
icmp          { mib2 5 }
tcp           { mib2 6 }
udp           { mib2 7 }
              [ exterior gateway protocol “egp” { mib2 8 } not
                  supported ]
              [ cmot { mib2 9 } is “historic”, just a placeholder ]
dot3          { mib2 10 7 } == { transmission 7 } “EtherLike MIB”
snmp          { mib2 11 }
```

eCos changes

Small changes have been made in two areas:

1) the ARM/EBSA-285 Ethernet driver
2) the OpenBSD TCP/IP networking package

These changes were made in order to export information about the driver and the network that the SNMP agent must report. The changes were trivial in the case of the network stack, since it was already SNMP-friendly. The ethernet device driver was re-organized to have an extensive header file and to add a couple of APIs to extract statistics that the i82559 keeps within itself.

There is a performance hit for recording that data; disabling config option CYGDBG_DEVS_ETH_ARM_EBSA285_KEEP_STATISTICS in the device driver will prevent that.

The other platform ethernet device drivers have not been changed to match at this stage; if the exported information is missing, SNMP will report zero values for such data (in the dot3 MIB).

The interface chipset has an ID which is an OID; that which means Intel i82558 is returned since none is yet defined for the 82559. It may need to be added to the client MIB, it was not defined in those from UCD.

Starting the SNMP Agent

A routine to instantiate and start the SNMP agent thread is provided in

```
PACKAGES/net/snmp/agent/VERSION/src/snmptask.c
```

It starts the snmpd thread at priority CYGPKG_NET_THREAD_PRIORITY-2, ie. two below the TCP/IP stack service thread. To use that convenience function, this code fragment may be copied (in plain C).

```
#ifdef CYGPKG_SNMPAGENT  {
                extern void cyg_net_snmp_init(void);
                cyg_net_snmp_init();
}
#endif
```

The entry point to the SNMP agent is simply `externC void snmpd(void)`; so you can of course easily start it in a thread of your choice at another priority instead if required, after performing whatever other initialization your SNMP MIBs need. A larger than default stacksize is required.

Configuring eCos

Apart from adding the networking, common ethernet device drivers, snmp library and snmp agent packages, there is no configuration required.

Test cases

Currently only one test program is provided which uses SNMP.

“snmpping” in the SNMP agent package runs the ping test from the TCPIP package, essentially, for 10 times, with the snmpd running also, so you can interrogate it using host tools of your choice. It supports MIBs as documented above, so eg. “`snmpwalk <hostname> public dot3`” under Linux/UNIX should have the desired effect.

Starting the SNMP agent is not integrated into other network tests, nor is it started automatically in normal eCos startup - it is up to the application to start the agent when it is ready, at least after the network interfaces are both “up”.

SNMP clients and package use

SNMP clients may use these packages, but this usage is currently untested: the reason why this port to eCos exists is to acquire the SNMP agent. The fact that that the SNMP API (for clients) exists is a side-effect. See the standard man page SNMP_API(3) for details. There are further caveats below about client-side use of the SNMP library.

All of the SNMP header files are installed beneath .../include/ucd-snmp in the install tree. The SNMP code itself assumes that directory is on its include path, so we recommend that client code does the same. Further, like the TCP/IP stack, compiling SNMP code requires definition of _KERNEL and __ECOS, and additionally IN_UCD_SNMP_SOURCE.

Therefore, add all of these to your compile lines if you wish to include SNMP header files:

```
-D_KERNEL
-D__ECOS
-DIN_UCD_SNMP_SOURCE=1
-I$(PREFIX)/include/ucd-snmp
```

Unimplemented features

Currently, the filesystem and persistent storage areas are left undone, to be implemented by the application.

The SNMP library package is intended to support client and agent code alike. It therefore contains lots of assumptions about the presence of persistent storage ie. a filesystem. Currently, by default, eCos has no such thing, so those areas have been simply commented out and made to return empty lists or say “no data here.”

Specifically,

```
PACKAGES/net/snmp/lib/VERSION/src/parse.c
```

contains code to enumerate MIB files discovered in the system MIB directories (“/usr/share/snmp/mibs”), and read them all in, building data structures that are used by client programs to interrogate an agent. This is not required in an agent, so the routine which enumerates the directories returns an empty list.

```
PACKAGES/net/snmp/lib/VERSION/src/read_config.c
```

contains two systems, the first reads various configuration files (“/usr/share/snmp/snmp.conf”, “/usr/share/snmp/snmpd.conf”, “/usr/local/share/snmp/snmpd.local.conf” and the like) to control both the SNMP applications and the agent in the usual UNIX fashion. The second system contains code to record persistent data as files in a directory (typically “/var/ucd-snmp/”) thus preserving state permanently. Neither of these is supported because there is no filesystem present; as required, a cleaner interface to permit application code to manage persistent data will be developed in consultation with customers.

MIB Compiler

In the directory net/snmp/agent/VERSION/utils/mib2c, there are the following files:

```
README-eCos             notes about running with a nonstandard
                        perl path.
README.mib2c            the README from UCD; full instructions on
                        using mib2c
mib2c                   the perl program
mib2c.conf              a configuration file altered to include the
                        eCos/UCD
mib2c.conf-ORIG         copyright and better #include paths; and
                        the ORIGinal.
mib2c.storage.conf      other config files, not modified.
mib2c.vartypes.conf
```

mib2c is provided BUT it requires the SNMP perl package SNMP-3.1.0, and that in turn requires perl nsPerl5.005_03 (part of Red Hat Linux from 6.0, April 1999).

These are available from the CPAN (“the Comprehensive Perl Archive Network”) as usual; http://www.cpan.org and links from there. Specifically:

PERL itself:

http://people.netscape.com/kristian/nsPerl/
http://people.netscape.com/richm/nsPerl/nsPerl5.005_03-11-i686-linux.tar.gz SNMP.pl
http://www.cpan.org/modules/01modules.index.html
http://cpan.valueclick.com/modules/by-category/05_Networking_Devices_IPC/SNMP/
http://www.cpan.org/authors/id/G/GS/GSM/SNMP.tar.gz

(note that the .tar.gz files are not browsable)

For documentation on the files produced, see the documentation available at http://ucd-snmp.ucdavis.edu/ in general, and file AGENT.txt in particular.

It is likely that the output of mib2c will be further customized depending on eCos customer needs; it’s easy to do this by editing the mib2c.conf file to add or remove whatever you need with the resulting C sources.

The UCD autoconf-style configuration does not apply to eCos. So if you add a completely new MIB to the agent, and support it using mib2c so that the my_new_mib.c file contains a init_my_new_mib() routine to register the MIB handler, you will also need to edit a couple of control files; these claim to be auto-generated, but in the eCos release, they’re not, don’t worry.

```
PACKAGES/net/snmp/agent/VERSION/include/mib_module_includes.h
```

contains a number of lines like

```
#include “mibgroup/mibII/interfaces.h”
```

so add your new MIB thus:

```
#include “mibgroup/mibII/my_new_mib.h”
```

```
PACKAGES/net/snmp/agent/VERSION/include/mib_module_inits.h
```

contains a number of lines like

```
init_interfaces();
init_dot3();
```

and so on; add your new MIB as follows:

```
init_my_new_mib();
```

and this should work correctly.