TCP/IP Manual

Administration and Programming

Fireball-Amoeba 5.6.0

Date: Sun Dec 17 10:33:45 GMT 2000

FIREBALL
DOCUMENTATION
PROJECT

TCP/IP server

Name

ipsvr - the TCP/IP server

Synopsis

Built into the kernel. Add the following lines to the source file list of the kernel Amakefile:

```
conf_gw.c[flags="-I$SRC_ROOT/kernel/server/ip/kernel_dep" +
        "-I$SRC_ROOT/kernel/server/ip/amoeba_dep" +
        "-I$SRC_ROOT/kernel/server/ip"],
$K_SVR_TCPIP[flags="-I$SRC_ROOT/kernel/server/ip/kernel_dep" +
        "-I$SRC_ROOT/kernel/server/ip/amoeba_dep" +
        "-I$SRC_ROOT/kernel/server/ip" +
        '-DAM_KERNEL' + '-D_POSIX_SOURCE' ],
```

Description

The TCP/IP server implements the Internet protocols as described in RFC-791 ‘‘Internet Protocol’’, RFC-792 ‘‘Internet Control Message Protocol’’, RFC-793 ‘‘Transmission Control Protocol’’, RFC-768 ‘‘User Datagram Protocol’’, RFC-1256 ‘‘ICMP router discovery messages’’ and others. The rest of this document assumes familiarity with the basic concepts of TCP/IP networking.

The server runs in the kernel so that it can have access to the Ethernet interfaces. When the kernel is running the various capabilities of the the server are published in the directory *ip* under the kernel directory. A `dir -l` of a kernel with a TCP/IP server typically looks like:

```
bootp:00                          @    69564 KB
floppy:00                         @     1440 KB
ip                                %--1234
printbuf                          printbuf server
printer                           printer server
proc                              process/segment server
ps                                %--1234
random                            random number server
```

```
sys                              system server
tod                              TOD server
tty:00                           +
```

The contents of the *ip* directory is:

```
eth                              To---s--- 1
ip                               To---s--- 2
tcp                              To---s--- 3
udp                              To---s--- 4
```

These are the capabilities for the Ethernet, IP, TCP and UDP interfaces, respectively.
The default TCP/IP server elements for a user are defined in *ampolicy.h*. Typically they are the names in the following table:

Variable Name	Typical default value
ETH_SVR_NAME	/profile/cap/ipsvr/eth
IP_SVR_NAME	/profile/cap/ipsvr/iph
TCP_SVR_NAME	/profile/cap/ipsvr/tcp
UDP_SVR_NAME	/profile/cap/ipsvr/udp

These defaults can be overridden by either the environment variables below or in some cases by the command line options of the various programs that use the TCP/IP server.

Environment Variables

There are four string environment variables, each referring to one of the above server capabilities of the IP server: `ETH_SERVER`, `IP_SERVER`, `TCP_SERVER` and `UDP_SERVER`. The value of these environment variables is interpreted as a path name that refers to a capability of the TCP/IP server. To use the environment variables to override the default server, issue the following commands:

```
ETH_SERVER=/super/hosts/tcpip_host/ip/eth; export ETH_SERVER
IP_SERVER=/super/hosts/tcpip_host/ip/ip; export IP_SERVER
TCP_SERVER=/super/hosts/tcpip_host/ip/tcp; export TCP_SERVER
UDP_SERVER=/super/hosts/tcpip_host/ip/udp; export UDP_SERVER
```

where *tcpip_host* is the name of the host with the TCP/IP server to be used.

Initialization

In order to work properly, the TCP/IP server needs to have an Internet address, a netmask, and routing tables. These can be configured manually and the routing tables can also be configured automatically.
The Internet address and the netmask must configured manually using *ifconfig*(A). The netmask defaults to 255.255.255.0. The program *setupipsvr*(A) can be useful when configuring TCP/IP servers.
The routing tables can be configured using *add_route*(A). If the IP routers support the Internet Router Discovery Protocol (IRDP) and/or the Routing Information Protocol (RIP), for instance UNIXhosts with the *routed* daemon, then the *irdpd*(A) daemon can be used to automatically configure the routing tables.

Warning

Do not choose Internet addresses at random. The addresses should be obtained from the appropriate authority, especially if the Amoeba network is directly connected to the Internet.

Programming Interface Summary

Access to the *IP* server is provided by the library routines described in *ip*(L).

Administration

The TCP/IP server requires very little administration once it is installed. The main thing is to ensure that the *boot*(A) server maintains the capabilities for the default server in */super/cap/ipsvr*. There are commented-out entries in the *Bootfile* provided with the system that show how to do this. See `IP_cap`, `ETH_cap`, etc. for details. Note that the entries in */super/cap/ipsvr* should have Soap column masks ff:80:80. The *boot* server should also be used to keep related servers running that might be required, for example, *rarp*(A), *tftp*(A) and *irdpd*(A).

It will be necessary to maintain the hosts table *(/etc/hosts)* if the Domain Name System (DNS) is not accessible from the Amoeba network. Note that there is currently no DNS server that runs on Amoeba but the Amoeba programs will use any accessible DNS on the network.
N.B. If there is more than one network connected to the IP server host, the server will only talk to the network with the lowest FLIP network number.

Installation

This section describes the installation and configuration of the TCP/IP related kernels and utilities. The first issue is to decide on which host(s) to run TCP/IP servers. This will be determined by the intended use of the TCP/IP server(s).

The TCP/IP server is typically built into a *pool*, *workstation* or *smallbullet* kernel. It tends to take a lot of space, so building it into a *smallbullet* kernel is only a good idea if the Bullet Server host has sufficient memory to support both the Bullet cache and the IP server.

If it is desired to use an X server with a TCP connection to access other sites or UNIXhosts, then it is a good idea to have a TCP/IP server in the *workstation* kernel where the X server runs. This simplifies addressing the display of the X server from UNIXor another site.

The other main issue is booting hosts using TFTP. (This is typically the case with Sun machines.) At least one TCP/IP server must be able to boot without using TFTP, since the TFTP server under Amoeba needs the TCP/IP server. There are two possible solutions:

1. The first TCP/IP server is in an Amoeba kernel which is booted from disk/floppy. This requires only a small disk with one vdisk set up as a boot partition.
2. The kernel with the first TCP/IP server is booted from a TFTP server provided by UNIX

Once the first TCP/IP server is running, *rarp*(A) and *tftp*(A) can be started by the *boot*(A) server and these can boot the remaining Amoeba processors. (The *boot* server is started on the Bullet Server which is normally booted from disk.) Note that the *rarp* and *tftp* servers typically use more than 1.5MB and must run on the only machine guaranteed to be up, i.e., the TCP/IP host. Make sure it has enough memory free.

In the following it is assumed that one of the above mechanisms has been provided to boot the kernel containing the first TCP/IP server.

A TCP/IP kernel needs to to be told its Internet address and a netmask using *ifconfig*(A). This can also be done using *setupipsvr*(A) in combination with the *boot* server. See *setupipsvr*(A) for an example of how to set up the *boot* server to do this.
To check if a server knows its own Internet address use the command

```
ifconfig $IP_SERVER
```

It reports something like:

```
hostaddr= 130.37.24.11 netmask= 255.255.255.0
```

or (after a while):

```
hostaddr not set
```

If the Internet address has not (yet) been set then it can be set using *ifconfig* (as follows:

```
ifconfig −h 130.37.24.11 $IP_SERVER
```

with the appropriate address in place of the one shown).
Take a look at the netmask to see if it matches the netmask of the network the machine is on. The netmask can be changed with

```
ifconfig −n <netmask> $IP_SERVER.
```

If the *boot* server has been set up appropriately, the capabilities for the default TCP/IP server will have been installed in */super/cap/ipsvr* each time *tcpip_host* is rebooted. However, the first time it is necessary to install the capabilities manually. Do this using:

```
get /super/hosts/tcpip_host/ip/eth | put /super/cap/ipsvr/eth
get /super/hosts/tcpip_host/ip/ip | put /super/cap/ipsvr/ip
get /super/hosts/tcpip_host/ip/tcp | put /super/cap/ipsvr/tcp
get /super/hosts/tcpip_host/ip/udp | put /super/cap/ipsvr/udp
```

At this point you can use *ping*(A) to check if the server is working.

```
/super/admin/bin/ping tcpip_host
```

will give something like:

```
PING tcpip_host.am.cs.vu.nl (130.37.24.11): 56 data bytes
64 bytes from 130.37.24.11: icmp_seq=0 ttl=60 time=0 ms
64 bytes from 130.37.24.11: icmp_seq=1 ttl=60 time=0 ms
64 bytes from 130.37.24.11: icmp_seq=2 ttl=60 time=0 ms
```

Interrupt this if it seems to be working. It should now also be possible to use *ttn*(U) to talk to other IP hosts on the Amoeba network. Look up the Internet address of a UNIX machine on the same network and try

```
ttn <Internet address of UNIX machine>
```

this will give you something like:

```
connecting to <Internet address of UNIX machine>
connect succeeded !!!!

SunOS UNIX (xxx.cs.vu.nl)

login:
```

CTRL-D can be used to exit from the login prompt.
At this moment you can use TCP/IP to talk to hosts on the same network as your Amoeba IP server but probably not to talk to machines on other networks. Issue the command to display the routing table of the TCP/IP server:

```
pr_routes
```

If nothing is shown it indicates an empty routing table.
The TCP/IP server needs to know some gateways in order to forward packets that are to be sent to hosts on other networks. There are three ways for the TCP/IP server to get a routing table: built into the kernel, added manually using *add_route*(A), or discovered dynamically using the Gateway Advertisement ICMPs. The last method is the most appropriate one at moment. Unfortunately, current gateways do not generate Gateway Advertisement ICMPs and a daemon is needed to generate these ICMPs. Start the program *irdpd*(A) in the background:

```
irdpd -b -I $IP_SERVER -U $UDP_SERVER &
```

After about one minute rerun *pr_routes*(A). It should show some output. For example,

```
1 DEST= 0.0.0.0, NETMASK= 0.0.0.0, GATEWAY= 130.37.24.2, dist= 1 pref= 1
```

Connections to machines on other networks should now be possible.
Until now Internet addresses have been used instead of host names. There two ways to map a hostname to an Internet address or the other way around. The first one is the */etc/hosts* file. You can add the entry

```
127.0.0.1      localhost
```

to the file */super/unixroot/etc/hosts* and try

```
ttn localhost.
```

This should give the same result as

```
ttn 127.0.0.1
```

The other way is the DNS. The TCP/IP programs are compiled with the so-called resolver library. This means that a DNS server can be used to look up Internet addresses. The resolver library uses the file */etc/resolv.conf*. This file should contain one entry:

```
nameserver 127.0.0.1
```

When that Internet address is changed to the Internet address of a DNS server, and an entry

```
domain <your domain>
```

is added then a command like

```
host uunet.uu.net
```

should report:

```
uunet.uu.net has address 137.39.1.2
uunet.uu.net has address 192.48.96.2
uunet.uu.net mail is handled by relay1.UU.NET
uunet.uu.net mail is handled by relay2.UU.NET
```

Likewise

```
ttn <name of UNIX machine>
```

should work.

See Also

add_route[A] , **boot[A]** , **ftpd[A]** , **ifconfig[A]** , **ip[L]** , **irdpd[A]** , **ping[A]** , **pr_routes[A]** , **rarp[A]** , **setupipsvr[A]** , **telnetd[A]** .

IP general purpose client interface

(in libraries: libamoeba.a)

Name

ip - Internet Protocol server's general-purpose client interface stubs

Synopsis

```
#include "stddef.h"
#include "amoeba.h"

#include "server/ip/hton.h"

u16_t htons(host_word)
u32_t htonl(host_dword)
u16_t ntohs(network_word)
u32_t ntohl(network_dword)
u16_t HTONS(host_word)
u32_t HTONL(host_dword)
u16_t NTOHS(network_word)
u32_t NTOHL(network_dword)

#include "server/ip/gen/oneCsum.h"

u16_t oneC_sum(prev, data, size)

#include "server/ip/tcpip.h"

char *tcpip_why(err)
errstat tcpip_keepalive(chan_cap, respite)
errstat tcpip_keepalive_cap(cap)
errstat tcpip_mkcap(tcpip_cap, obj, cap)
errstat tcpip_open(tcpip_cap, chan_cap)
errstat tcpip_read(chan_cap, buffer, bytes)
errstat tcpip_unkeepalive_cap(cap)
errstat tcpip_write(chan_cap, buffer, bytes)
```

Description

The Internet Protocol (IP) server implements four network protocols, as described in *ipsvr*(A). The four supported protocols are ETH, IP, TCP and UDP. The general-purpose routines described here and the protocol specific routines described in *ip_eth*(L), *ip_ip*(L), *ip_tcp*(L) and *ip_udp*(L) give access to the protocols of the IP server. Access to these services is provided using two types of capabilities: server capabilities and channel capabilities. The server capabilities are called *eth*, *ip*, *tcp*, *udp*, which correspond to the ETH, IP, TCP and UDP interfaces, respectively. The server capabilities can be used to obtain a channel to the corresponding server. This is done with *tcpip_open.* The channel capability can be used to transfer data using the protocol implemented by the server. This can typically be done with the *tcpip_* routines since they are generic to all interfaces. However the semantics of *tcpip_read* and *tcpip_write* vary slightly, depending on the protocol in use. The routines specific for each protocol server and the exact semantics of the *tcpip_* functions are described in the corresponding *ip_*xxx(L) manual page.

The descriptions of the various routines are divided into categories according to whether they are for byte-order conversion or general-purpose (i.e., they are used with all four interfaces). Before they are described, a brief introduction to the various types is given.

Types (general)

server/ip/types.h

> defines *u8_t*, *u16_t*, *u32_t* and *i32_t* (and *U8_t*, *U16_t*, *U32_t* and *I32_t* for use in prototypes).

Rights

The following rights are defined in *server/ip/tcpip.h*:

`IP_RIGHTS_OPEN`
: The right to do a *tcpip_open* (this indicates a server capability).

`IP_RIGHTS_RWIO`
: The right to do I/O (this indicates a channel capability).

`IP_RIGHTS_DESTROY`
: The right to destroy a connection.

`IP_RIGHTS_LINGER`
: The right keeps the server from destroying the connection when it is not used for some time.

`IP_RIGHTS_SUPER`
: The right to get existing capabilities (using *tcpip_mkcap*).

Byte Order Conversion

htons, htonl, ntohs, ntohl

```
u16_t
```

```
htons(host_word)
u16_t host_word;


u32_t
htonl(host_dword)
u32_t host_dword;


u16_t
ntohs(network_word)
u16_t network_word;


u32_t
ntohl(network_word)
u32_t network_word;
```

These macros convert 16-bit and 32-bit quantities to and from the network byte order used by the TCP/IP protocols. The function of the macros is encoded in their name. *H* means host byte order, *n* means network byte order, *s* means a 16-bit quantity and *l* means a 32-bit quantity. Thus *htons* converts a 16-bit quantity from host byte order to network byte order. The difference between the lower case and upper case variants is that the lower case variants evaluate the argument at most once and the upper case variants can be used for constant folding. That is,

```
htonl(f(x))
```

will call f(x) at most once and

```
HTONS(0x10)
```

will be equivalent to 0x10 on a big-endian machine and 0x1000 on a little-endian machine.

General Functions

oneC_sum

```
u16_t
oneC_sum(prev, data, size)
u16_t prev;
u16_t *data;
size_t size;
```

OneC_sum is used to calculate the one's complement checksum needed for IP network packets. The IP checksum is described in RFC-1071 (Computing the Internet checksum).

One_Csum expects three parameters:

prev
: The checksum of previous blocks of data that are to be included in the checksum. The value of prev in first call to *oneC_sum* should be 0.

data
: A pointer to the block of data. The data is interpreted as a series of 16-bit numbers in network byte order, but an odd number of bytes is also allowed.

size
: The size of the data in bytes.

tcpip_why

```
char *
tcpip_why(err)
errstat err;
```

This routine returns a pointer to a statically allocated string describing the error code *err*. If *err* is not one of the errors as described in the *Diagnostics* section below then *tcpip_why* will return the same string as *err_why* (see *error*(L)).

tcpip_keepalive

```
errstat
tcpip_keepalive(chan_cap, respite)
capability *chan_cap;
int *respite;
```

Tcpip_keepalive makes sure that a channel capability is not prematurely destroyed. In general channel capabilities are destroyed when not used for a while. *Tcpip_keepalive* returns the destruction timeout in *respite.* It is sufficient to call *tcpip_keepalive* again within *respite* milliseconds to prevent the channel from being destroyed.
See also *tcpip_keepalive_cap*.

tcpip_keepalive_cap

```
errstat
tcpip_keepalive_cap(cap)
capability *cap;
```

Tcpip_keepalive_cap provides a more convenient interface to *tcpip_keepalive*. Each time a new TCP/IP capability is created, it can be handed to *tcpip_keepalive_cap*. *Tcpip_keepalive_cap* will keep the capability alive until it is either destroyed, *tcpip_unkeepalive_cap* is called for that capability, or the program exits. The first time *tcpip_keepalive_cap* is called, it creates a separate thread to manage all the capabilities that should be kept alive.

tcpip_unkeepalive_cap

```
errstat
tcpip_unkeepalive_cap(cap)
capability *cap;
```

Tcpip_unkeepalive_cap removes the capability *cap* from the list of capabilities that should receive a *tcpip_keepalive*. Normally the IP server will destroy that capability after a while unless some other process keeps the capability alive.

tcpip_mkcap

```
errstat
tcpip_mkcap(tcpip_cap, obj, cap)
capability *tcpip_cap;
objnum obj;
capability *cap;
```

A channel capability can be lost, or can be kept inside a process. *Tcpip_mkcap* recreates the capability which has object number *obj*. This provides a way to obtain an otherwise lost capability. To work out the object number of a particular channel use the *std_status*(U) command.

tcpip_open

```
errstat
tcpip_open(tcpip_cap, chan_cap)
capability *tcpip_cap;
capability *chan_cap;
```

Tcpip_open creates a new channel for a server capability. A pointer to the server capability must be passed via the first argument and the new channel capability is stored in the object pointed to by be the second argument.

tcpip_read

```
errstat
tcpip_read(chan_cap, buffer, bytes)
capability *chan_cap;
char *buffer;
size_t bytes;
```

Tcpip_read transfers data from the TCP/IP server to the client. The call blocks until enough data is available. The semantics of *tcpip_read* are different for each of the servers. See the *tcpip_read* section in the *ip*_xxx(L) manual pages for the exact semantics.

tcpip_write

```
errstat
tcpip_write(chan_cap, buffer, bytes)
capability *chan_cap;
char *buffer;
size_t bytes;
```

Tcpip_write transfers data from the client to the TCP/IP server. The call blocks until enough buffer space is available. The semantics of *tcpip_write* are different for each of the servers. See the *tcpip_write* section in the *ip*_xxx(L) manual pages for the exact semantics.

Diagnostics

The TCP/IP server introduces several new error codes. These are defined in *server/ip/tcpip.h*.

`TCPIP_PACKSIZE`
: This indicates an attempt to read (*tcpip_read*) or write (*tcpip_write*) with a buffer that is too large or too small.

`TCPIP_OUTOFBUFS`
: The TCP/IP server has insufficient memory to execute the request.

`TCPIP_BADIOCTL`
: This indicates an attempt to execute a command the particular server does not understand. For example, a *tcp_ioc_getconf* on an ETH channel.

`TCPIP_BADMODE`
: The request was refused because the channel is not fully configured, in the wrong state or the parameters are invalid.

`TCPIP_BADDEST`
: This indicates an illegal destination address for a packet.

`TCPIP_DSTNORCH`
: The destination was not reachable.

`TCPIP_ISCONN`
: The channel is already connected so a second request is refused.

`TCPIP_ADDRINUSE`
: This address is in use.

`TCPIP_CONNREFUSED`
: The connection was refused by the other side.

`TCPIP_CONNRESET`
: The connection was reset (non-gracefully terminated) by the other side.

`TCPIP_TIMEDOUT`
: The connection was terminated due to an expired timer.

`TCPIP_URG`
: Urgent data is present and the current receive mode does not allow urgent data to be transferred.

`TCPIP_NOURG`
: No urgent data is present and a request came for urgent data.

`TCPIP_NOTCONN`
: The request requires a connected channel and the channel is not connected.

`TCPIP_SHUTDOWN`
: The connection was shutdown. That is, a *tcp_ioc_shutdown* has been executed so no more data can be transmitted.

`TCPIP_NOCONN`
: The connection does not exist.

`TCPIP_ERROR`
: A generic error code for extremely weird cases.

See Also

error(L), **ip_eth[L]** , **ip_ip[L]** , **ip_tcp[L]**, **ip_udp[L]** , **ipsvr[A]** .

IP client interface

(in libraries: libamoeba.a)

Name

ip_ip - Internet Protocol server's IP client interface stubs

Synopsis

```
#include "amoeba.h"
#include "server/ip/ip_io.h"
#include "server/ip/types.h"
#include "server/ip/gen/in.h"
#include "server/ip/gen/ip_io.h"
#include "server/ip/gen/route.h"

errstat ip_ioc_getconf(chan_cap, ipconf)
errstat ip_ioc_getopt(chan_cap, ipopt)
errstat ip_ioc_getroute(chan_cap, route)
errstat ip_ioc_setconf(chan_cap, ipconf)
errstat ip_ioc_setopt(chan_cap, ipopt)
errstat ip_ioc_setroute(chan_cap, route)
```

Description

The Internet Protocol (IP) server implements four network protocols as described in *ipsvr*(A). The routines described below give access to the IP protocol in the IP server. Access to this service is provided using two types of capabilities: a server capability and channel capabilities. The server capability is called *ip*. The server capability is used to obtain a channel to the corresponding server. This is done with *tcpip_open* (see *ip*(L)). The channel capability can be used to transfer data using the protocol implemented by the server. This can also be done with the generic *tcpip_read* and *tcpip_write* routines. Since their semantics vary slightly depending on channel type, these routines are described briefly below. The *ip_ioc_* routines are used to manage the options and status of the IP channel.

Before they are described, a brief introduction to the various types is given. The general-purpose types and rights in the capabilities are described in *ip*(L).

IP Types

server/ip/gen/in.h

defines *ipaddr_t*, *ipproto_t* and struct ip_hdropt (*ip_hdropt_t*).

server/ip/gen/ip_io.h

defines struct nwio_ipconf (*nwio_ipconf_t*) and struct nwio_ipopt (*nwio_ipopt_t*)

server/ip/gen/ip_hdr.h

defines struct ip_hdr (*ip_hdr_t*)

server/ip/gen/route.h

defines struct nwio_route (*nwio_route_t*)

General Functions

tcpip_read

```
errstat
tcpip_read(chan_cap, buffer, nbytes)
capability *chan_cap;
char *buffer;
size_t nbytes;
```

Tcpip_read reads IP packets from the IP channel of a TCP/IP server, specified by *chan_cap*, into *buffer*, which has length *nbytes*. If the packet is larger than *nbytes* the packet is truncated to the buffer size and the error `TCPIP_PACKSIZE` will be returned. It blocks until it has a complete IP packet. If successful it returns the number of bytes read. Otherwise it returns a negative error status. If the read is interrupted the server returns `STD_INTR`.

It is possible to read the data only or the data plus IP header. See *ip_ioc_setconf* below for details.

`Error Conditions:`

See *ip*(L) for a description of the error codes.

```
TCPIP_BADMODE
TCPIP_PACKSIZE
```

```
STD_INTR
```

tcpip_write

```
errstat
tcpip_write(chan_cap, buffer, nbytes)
capability *chan_cap;
char *buffer;
size_t nbytes;
```

Tcpip_write transfers the IP packet in *buffer* from the client to the TCP/IP server's IP channel specified by *chan_cap*. The IP packet to be sent is *nbytes* long. It accepts only complete packets. It computes the checksum for the packet and inserts it at the appropriate place. The call blocks until enough buffer space for the whole packet is available in the TCP/IP server. On success it returns the number of bytes written. On failure it returns a negative error code.

```
Error Conditions:
```

See *ip*(L) for a description of the error codes.

```
TCPIP_BADMODE
TCPIP_PACKSIZE
STD_INTR
```

IP Functions

ip_ioc_getconf

```
errstat
ip_ioc_getconf(chan_cap, ipconf)
capability *chan_cap;
struct nwio_ipconf *ipconf;
```

Ip_ioc_getconf reports the Internet address and the netmask. For the *nwio_ipconf* structure see *ip_ioc_setconf* below.

ip_ioc_getopt

```
errstat
```

```
ip_ioc_getopt(chan_cap, ipopt)
capability *chan_cap;
struct nwio_ipopt *ipopt;
```

This call returns in *ipopt* the current options of the IP channel specified by *chan_cap*. See *ip_ioc_setopt* for a description of the options.

ip_ioc_getroute

```
errstat
ip_ioc_getroute(chan_cap, route)
capability *chan_cap;
struct nwio_route *route;
```

Ip_ioc_getroute can be used to query an IP server about it is routing table. The structure *nwio_route* is defined in *server/ip/gen/route.h*:

```
typedef struct nwio_route
{
        u32_t nwr_ent_no;
        u32_t nwr_ent_count;
        ipaddr_t nwr_dest;
        ipaddr_t nwr_netmask;
        ipaddr_t nwr_gateway;
        u32_t nwr_dist;
        u32_t nwr_flags;
        u32_t nwr_pref;
        ipaddr_t nwr_ifaddr;
} nwio_route_t;
```

```
#define NWRF_EMPTY        0
#define NWRF_INUSE        1
#define NWRF_STATIC       2
#define NWRF_UNREACHABLE  4
```

The requested entry is taken from *nwr_ent_no*. Entries are counted from 0, so the value 0 can be used for an initial query. The size of the routing table is returned in *nwr_ent_count*. The *nwr_flags* field indicates if the entry is in use (`NWRF_INUSE`) and if the entry is static (`NWRF_STATIC` using *ip_ioc_setroute*) or generated by the IP server itself. A destination can be marked unreachable using `NWRF_UNREACHABLE`. The route is described by *nwr_dest*, *nwr_netmask*, *nwr_gateway*, *nwr_dist*, and *nwr_pref*. *Nwr_dest* and *nwr_netmask* select the destination addresses. A value of 0.0.0.0 (0x0) in both *Nwr_dest* and *nwr_netmask* means every host. A value of 255.255.255.255 (0xffffffff) in *nwr_netmask* means a single host. Other values of *nwr_netmask* are netmasks for the network specified by *nwr_dest*. *Nwr_gateway* is the gateway that should be used. *Nwr_dist* is the minimal distance (i.e. if the specified network is subnetted, destinations can be at varying distances.) Packets with a ‘‘time to live’’ smaller than *nwr_dist* will not reach the destination. If two routes have equal netmask and distance fields but different gateways then the gateway with highest value in *nwr_pref* is used.

ip_ioc_setconf

```
errstat
ip_ioc_setconf(chan_cap, ipconf)
capability *chan_cap;
struct nwio_ipconf *ipconf;
```

Ip_ioc_setconf can be used to inform the IP server about its Internet address and/or its netmask. Normally an IP server will discover its Internet address using the RARP protocol. *Ip_ioc_setconf* can be used in the case that the RARP failed, or the netmask has to be changed. Note that higher level protocols (TCP and UDP) assume that the Internet address of an IP device does not change, therefore TCP and UDP stop functioning if the Internet address is changed.

The structure *nwio_ipconf* is defined in *server/ip/gen/ip_io.h*:

```
typedef struct nwio_ipconf
{
        u32_t   nwic_flags;
        ipaddr_t nwic_ipaddr;
        ipaddr_t nwic_netmask;
} nwio_ipconf_t;
```

```
#define NWIC_NOFLAGS            0x0
#define NWIC_FLAGS              0x3
#       define NWIC_IPADDR_SET          0x1
#       define NWIC_NETMASK_SET         0x2
```

The function of *nwio_ipconf* depends on the value of *nwic_flags*. If `NWIC_IPADDR_SET` is set then the Internet address will be set to *nwic_ipaddr*. If `NWIC_NETMASK_SET` is set then the Internet address will be set to *nwic_netmask*.

ip_ioc_setopt

```
errstat
ip_ioc_setopt(chan_cap, ipopt)
```

```
capability *chan_cap;
struct nwio_ipopt *ipopt;
```

Before an IP channel can be used, it has to be configured using *ip_ioc_setopt*. The structure *nwio_ipopt* is defined in *server/ip/gen/ip_io.h*:

```
typedef struct nwio_ipopt
{
        u32_t nwio_flags;
        ipaddr_t nwio_rem;
        ip_hdropt_t nwio_hdropt;
        u8_t nwio_tos;
        u8_t nwio_ttl;
        u8_t nwio_df;
        ipproto_t nwio_proto;
} nwio_ipopt_t;
```

```
#define NWIO_NOFLAGS    0x0000L
#define NWIO_ACC_MASK   0x0003L
#       define NWIO_EXCL        0x00000001L
#       define NWIO_SHARED      0x00000002L
#       define NWIO_COPY        0x00000003L
#define NWIO_LOC_MASK   0x0010L
#       define NWIO_EN_LOC      0x00000010L
#       define NWIO_DI_LOC      0x00100000L
#define NWIO_BROAD_MASK 0x0020L
#       define NWIO_EN_BROAD    0x00000020L
#       define NWIO_DI_BROAD    0x00200000L
#define NWIO_REM_MASK   0x0100L
#       define NWIO_REMSPEC     0x00000100L
#       define NWIO_REMANY      0x01000000L
#define NWIO_PROTO_MASK 0x0200L
#       define NWIO_PROTOSPEC   0x00000200L
#       define NWIO_PROTOANY    0x02000000L
#define NWIO_HDR_O_MASK 0x0400L
#       define NWIO_HDR_O_SPEC  0x00000400L
#       define NWIO_HDR_O_ANY   0x04000000L
#define NWIO_RW_MASK    0x1000L
#       define NWIO_RWDATONLY   0x00001000L
#       define NWIO_RWDATALL    0x10000000L
```

The options are divided into several categories: `NWIO_ACC_MASK`, `NWIO_LOC_MASK`, `NWIO_BROAD_MASK`, `NWIO_REM_MASK`, `NWIO_PROTO_MASK`, `NWIO_HDR_O_MASK` and `NWIO_RW_MASK`. A channel is configured when one option of each category is set.

The options covered by `NWIO_ACC_MASK` control the number of channels that can use one IP next-level protocol. If `NWIO_EXCL` is specified then only that channel can use a certain IP next-level protocol. If `NWIO_SHARED` is set then multiple channels that all have to specify `NWIO_SHARED` can use the same IP next-level protocol, but incoming packets will be delivered to at most one channel. `NWIO_COPY` does not impose any restrictions. Every channel gets a copy of an incoming packet.

NWIO_LOC_MASK and NWIO_BROAD_MASK control the delivery of packets. If NWIO_EN_LOC is specified then packets that are explicitly sent to the IP server are delivered. If NWIO_EN_BROAD is specified then broadcast packets are delivered. Either one or both of them can be disabled with NWIO_DI_LOC and NWIO_DI_BROAD.
NWIO_REMSPEC can be used to restrict communication to one remote host. This host is taken from the *nwio_rem* field. If any remote host is to be allowed then NWIO_REMANY can be used.
NWIO_PROTOSPEC restricts communication to one IP next-level protocol, specified in *nwio_proto*. NWIO_PROTOANY allows any protocol to be sent or received.

NWIO_HDR_O_SPEC specifies all IP header options in advance. The values are taken from *nwio_hdropt*, *nwio_tos*, *nwio_ttl*, and *nwio_df*. *Nwio_hdropt* specifies the IP options that should be present in an outgoing packet. *Ip_hdropt_t* is defined in *server/ip/gen/in.h*:

```
typedef struct ip_hdropt
{
        u8_t iho_opt_siz;
        u8_t iho_data[IP_MAX_HDR_SIZE-IP_MIN_HDR_SIZE];
} ip_hdropt_t;
```

The *iho_opt_siz* bytes in *iho_data* are appended to the IP header. *Nwio_tos* specifies the value of the ‘‘type of service’’ bits, *nwio_ttl* gives the value of the ‘‘time to live’’ field and *nwio_df* specifies whether fragmentation is disallowed or not. NWIO_HDR_O_ANY specifies that the header options must be specified with each write request.

NWIO_RWDATONLY specifies that the header should be omitted from a write request. This option can only be used when all header fields are specified in previous options: NWIO_EN_LOC, NWIO_DI_BROAD, NWIO_REMSPEC, NWIO_PROTOSPEC and NWIO_HDR_O_SPEC. A read operation will also only return the data part, so the IP options will be lost.

ip_ioc_setroute

```
errstat
ip_ioc_setroute(chan_cap, route)
capability *chan_cap;
struct nwio_route *route;
```

Ip_ioc_setroute adds a route to the routing table. See *ip_ioc_getroute* above for a description of the *nwio_route* structure. The fields *nwr_ent_no* and *nwr_ent_count* are ignored.

Diagnostics

The complete set of error codes is described in *ip*(L).

See Also

error(L), **ip[L]** , **ip_eth[L]** , **ip_tcp[L]** , **ip_udp[L]** , **ipsvr[A]** .

Ethernet client interface

(in libraries: libamoeba.a)

Name

ip_eth - Internet Protocol server's Ethernet client interface stubs

Synopsis

```
#include "amoeba.h"
#include "server/ip/eth_io.h"
#include "server/ip/types.h"
#include "server/ip/gen/ether.h"
#include "server/ip/gen/eth_io.h"

errstat eth_ioc_getstat(chan_cap, ethstat)
errstat eth_ioc_getopt(chan_cap, ethopt)
errstat eth_ioc_setopt(chan_cap, ethopt)
```

Description

The Internet Protocol (IP) server implements four network protocols, as described in *ipsvr*(A). The routines described below give access to the Ethernet protocol of the IP server. This allows reading and writing of raw Ethernet packets. Access to this service is provided using two types of capabilities: a server capability and channel capabilities. The server capability is called *eth*. It is used to obtain a channel to the corresponding server. This is done with *tcpip_open* (see *ip*(L)). The channel capability can be used to transfer data using the protocol implemented by the server. This is also done with the generic *tcpip_read* and *tcpip_write* routines. Since their semantics vary slightly depending on channel type, these routines are described briefly below. The *eth_ioc_* routines are used to manage the options and status of the ETH channel.

Before they are described, a brief introduction to the various types is given. The general-purpose types and rights in the capabilities are described in *ip*(L).

Eth Types

server/ip/gen/ether.h

defines struct ether_addr (*ether_addr_t*), *ether_type_t* and *Ether_type_t* for use in prototypes.

server/ip/gen/eth_io.h

defines struct nwio_ethopt (*nwio_ethopt_t*) and struct nwio_ethstat (*nwio_ethstat_t*)

server/ip/gen/eth_hdr.h

defines struct eth_hdr (*eth_hdr_t*)

General Functions

tcpip_read

```
errstat
tcpip_read(chan_cap, buffer, nbytes)
capability *chan_cap;
char *buffer;
size_t nbytes;
```

Tcpip_read transfers Ethernet packets from the Ethernet channel specified by *chan_cap* to the client. The data is returned in *buffer* which has size *nbytes*. The call blocks until a packet is available. If successful it returns the number of bytes read. If unsuccessful the function returns an error status. It is possible to read just the data of the Ethernet packets or both the header and the data. See *eth_ioc_setopt* below for details. Only reads with a buffer size greater than or equal to the maximum packet size are allowed.

```
Error Conditions:
```

See *ip*(L) for a description of the `TCPIP_` error codes.

```
TCPIP_BADMODE
TCPIP_PACKSIZE
STD_INTR
```

tcpip_write

```
errstat
```

```
tcpip_write(chan_cap, buffer, nbytes)
capability *chan_cap;
char *buffer;
size_t nbytes;
```

Tcpip_write transfers the *nbytes* of data in *buffer* from the client to the Ethernet channel specified by *chan_cap*. The call blocks until enough buffer space is available. The buffer must contain a complete packet and be at least of minimum size (60 bytes). On success this function returns the number of bytes written. On error it returns a negative error status.

Error Conditions:

See *ip*(L) for a description of the `TCPIP_` error codes.

```
TCPIP_BADMODE
TCPIP_PACKSIZE
STD_INTR
```

ETH Functions

eth_ioc_getop

```
errstat
eth_ioc_getopt(chan_cap, ethopt)
capability *chan_cap;
struct nwio_ethopt *ethopt;
```

This call returns in *ethopt* the current options of an Ethernet channel specified by *chan_cap*. See *eth_ioc_setopt* for a description of the options.

eth_ioc_getstat

```
errstat
eth_ioc_getstat(chan_cap, ethstat)
capability *chan_cap;
struct nwio_ethstat *ethstat;
```

Eth_ioc_getstat returns the Ethernet address and some statistics about the channel specified by the capability *chan_cap*. The result is returned in the structure to which *ethstat* points. The *struct nwio_ethstat* defined in *server/ip/gen/eth_io.h* is used to pass the configuration description to the server.

```
typedef struct nwio_ethstat
{
        ether_addr_t nwes_addr;
        eth_stat_t nwes_stat;
} nwio_ethstat_t;
```

```
typedef struct eth_stat
{
    unsigned long
        ets_recvErr,        /* # receive errors */
        ets_sendErr,        /* # send error */
        ets_OVW,            /* # buffer overwrite warnings,
                                (packets arrive faster than
                                 can be processed) */
        ets_CRCerr,         /* # crc errors of read */
        ets_frameAll,       /* # frames not aligned (# bits
                               not a multiple of 8) */
        ets_missedP,        /* # packets missed due to too
                               slow packet processing */
        ets_packetR,        /* # packets received */
        ets_packetT,        /* # packets transmitted */
        ets_transDef,       /* # transmission deferred (there
                               was a transmission of an
                               other station in progress */
        ets_collision,      /* # collisions */
        ets_transAb,        /* # transmissions aborted due
                               to excessive collisions */
        ets_carrSense,      /* # carrier sense lost */
        ets_fifoUnder,      /* # fifo underruns (processor
                               is too busy) */
        ets_fifoOver,       /* # fifo overruns (processor is
                               too busy) */
        ets_CDheartbeat,    /* # times unable to transmit
                               collision signal */
        ets_OWC;            /* # times out of window
                               collision */
} eth_stat_t;
```

eth_ioc_setopt

```
errstat
eth_ioc_setopt(chan_cap, ethopt)
capability *chan_cap;
struct nwio_ethopt *ethopt;
```

Before a capability of an Ethernet channel can be used to send or receive Ethernet packets, it has to be configured using *eth_ioc_setopt*. The structure *nwio_ethopt* is defined in *server/ip/gen/eth_io.h*:

```
typedef struct nwio_ethopt
{
        u32_t nweo_flags;
        ether_addr_t nweo_multi, nweo_rem;
        ether_type_t nweo_type;
} nwio_ethopt_t;
```

```
#define NWEO_NOFLAGS    0x0000L
#define NWEO_ACC_MASK   0x0003L
#       define NWEO_EXCL        0x00000001L
#       define NWEO_SHARED      0x00000002L
#       define NWEO_COPY        0x00000003L
#define NWEO_LOC_MASK   0x0010L
#       define NWEO_EN_LOC      0x00000010L
#       define NWEO_DI_LOC      0x00100000L
#define NWEO_BROAD_MASK 0x0020L
#       define NWEO_EN_BROAD    0x00000020L
#       define NWEO_DI_BROAD    0x00200000L
#define NWEO_MULTI_MASK 0x0040L
#       define NWEO_EN_MULTI    0x00000040L
#       define NWEO_DI_MULTI    0x00400000L
#define NWEO_PROMISC_MASK 0x0080L
#       define NWEO_EN_PROMISC  0x00000080L
#       define NWEO_DI_PROMISC  0x00800000L
#define NWEO_REM_MASK   0x0100L
#       define NWEO_REMSPEC     0x00000100L
#       define NWEO_REMANY      0x01000000L
#define NWEO_TYPE_MASK  0x0200L
#       define NWEO_TYPESPEC    0x00000200L
#       define NWEO_TYPEANY     0x02000000L
#define NWEO_RW_MASK    0x1000L
#       define NWEO_RWDATONLY   0x00001000L
#       define NWEO_RWDATALL    0x10000000L
```

The configuration is divided in a number of sections (covered by the xx_MASK macros). Options can be set in the *nweo_flags* field. The first section (`NWEO_ACC_MASK`) controls the access to a certain Ethernet packet type. If `NWEO_EXCL` is selected then this is the only channel that can send or receive Ethernet packets of the selected type. If `NWEO_SHARED` is selected then multiple channels (which all have to select `NWEO_SHARED`) can use the same Ethernet type. They all can send packets but incoming packets will be delivered to at most one of them. If `NWEO_COPY` is selected then multiple channels have access to the same Ethernet type and all receive a copy of an incoming packet.
The `NWEO_LOC_MASK` flags control the delivery of local packets, i.e., with a destination address equal to the Ethernet address of the machine where the TCP/IP server is running. If `NWEO_EN_LOC` is selected then these packets will be delivered and with `NWEO_DI_LOC` they will be discarded.

`NWEO_BROAD_MASK`, `NWEO_MULTI_MASK`, and `NWEO_PROMISC_MASK` do the same to broadcast packets, multicast packets and promiscuous mode packets as `NWEO_LOC_MASK` does for local packets, except that the precise multicast address is taken from the *nweo_multi* field.

The `NWEO_REM_MASK` flags control whether communication is restricted to a single destination or not. `NWEO_REMSPEC` restricts sending and receiving of packets to the single remote computer specified in the *nweo_rem* field. `NWEO_REMANY` allows sending to and receiving from any remote computer.

`NWEO_TYPESPEC` restricts sending and receiving of Ethernet packets to the protocol type specified in *nweo_type*. (For example, the RARP protocol has type number 0x8035.) The type has to be in network byte order (see *htons* in *ip*(L)). `NWEO_TYPEANY` allows any type.

If the Ethernet header is completely specified by the *nweo_flags* i.e., all of `NWEO_EN_LOC`, `NWEO_DI_BROAD`, `NWEO_DI_MULTI`, `NWEO_DI_PROMISC`, `NWEO_REMSPEC` and `NWEO_TYPESPEC` are specified, then `NWEO_RWDATONLY` can be used to send and receive just the data part of an Ethernet packet. The header information is filled in by the server from the pre-specified information. Otherwise `NWEO_RWDATALL` must be used. If `NWEO_RWDATALL` is specified then both Ethernet header and data must be provided for each packet.

Diagnostics

The complete set of error codes is described in *ip*(L).

See Also

error(L), **ip[L]** , **ip_ip[L]** , **ip_tcp[L]** , **ip_udp[L]** , **ipsvr[A]** .

UDP client interface

(in libraries: libamoeba.a)

Name

ip_udp - Internet Protocol server's UDP client interface stubs

Synopsis

```
#include "amoeba.h"
#include "server/ip/udp_io.h"
#include "server/ip/types.h"
#include "server/ip/gen/in.h"
#include "server/ip/gen/udp.h"
#include "server/ip/gen/udp_io.h"
```

```
errstat udp_close(chan_cap, flags)
errstat udp_connect(udp_cap, chan_cap, srcport, dstport, dstaddr, flags)
errstat udp_destroy(chan_cap, flags)
errstat udp_ioc_getopt(cap, udpopt)
errstat udp_ioc_setopt(cap, udpopt)
errstat udp_read_msg(chan_cap, msg, msglen, actlen, flags)
errstat udp_reconnect(chan_cap, srcport, dstport, dstaddr, flags)
errstat udp_write_msg(chan_cap, msg, msglen, flags)
```

Description

These routines give access to the UDP protocol of the IP server (see *ipsvr*(A)). Access to this service is provided using two types of capabilities: a server capability and channel capabilities. The server capability is called *udp*. The server capability is used to obtain a channel to the corresponding server. This is done with *tcpip_open* (see *ip*(L)). The channel capability can be used to transfer data using the protocol implemented by the server. This is also done with the generic *tcpip_read* and *tcpip_write* routines. Since their semantics vary slightly depending on channel type, these routines are described briefly below. There is also a more convenient interface for reading and writing described in the *UDP Library Functions* section below.

The *udp_* routines are used to manage the options and status of the UDP channel.
Before they are described, a brief introduction to the various types is given. The general-purpose types and rights in the capabilities are described in *ip*(L).

UDP Types

server/ip/gen/udp.h

defines *udpport_t* and *Udpport_t* for use in prototypes.

server/ip/gen/udp_io.h

defines struct nwio_udpopt (*nwio_udpopt_t*).

server/ip/gen/udp_hdr.h

defines struct udp_hdr (*udp_hdr_t*) and struct udp_io_hdr (*udp_io_hdr_t*).

General Functions

A more convenient interface to read and write UDP packets is described in the *UDP Library Functions* section below. The following interface is described for completeness.

NB. The TCP/IP server sends and receives the *udp_io_hdr* in network byte-order. This includes the *uih_ip_opt_len* and *uih_data_len* fields. The following two *tcpip_* routines do not decode these or any other fields to the local byte-order. The *udp_read_msg* and *udp_write_msg* functions do decode the length fields of the header making them simpler and safer to use.

tcpip_read

```
errstat
tcpip_read(chan_cap, buffer, nbytes)
capability *chan_cap;
char *buffer;
size_t nbytes;
```

Tcpip_read reads UDP packets from the UDP channel specified by *chan_cap* into *buffer*, which is of size *nbytes*. The call blocks until a complete packet is available. If successful it returns the number of bytes read.

It is possible to obtain just the data from the UDP packets or the data plus UDP headers. See *udp_ioc_setopt* below for details.

`Error Conditions:`

See *ip*(L) for a description of the error codes.

```
STD_INTR
TCPIP_BADMODE
```

tcpip_write

```
errstat
tcpip_write(chan_cap, buffer, nbytes)
capability *chan_cap;
char *buffer;
size_t nbytes;
```

Tcpip_write writes the UDP packet in *buffer*, of size *nbytes* to the UDP channel specified by *chan_cap*.

`Error Conditions:`

See *ip*(L) for a description of the error codes.

```
TCPIP_BADMODE
```

UDP Functions

udp_ioc_getopt

```
errstat
udp_ioc_getopt(cap, conf)
capability *cap;
struct nwio_udpopt *conf;
```

Udp_ioc_getopt returns the current options that result from the default options and the options set with *udp_ioc_setopt* When `NWUO_LP_SEL` or `NWUO_LP_SET` is selected the local port is returned in *nwuo_locport*. When `NWUO_RP_SET` is selected the remote port is returned in *nwuo_remport*. The local address is always returned in *nwuo_locaddr*, and when `NWUO_RA_SET` is selected the remote address is returned in *nwuo_remaddr*.

udp_ioc_setopt

```
errstat
udp_ioc_setopt(cap, conf)
capability *cap;
struct nwio_udpopt *conf;
```

A UDP channel must be configured using *udp_ioc_setopt* before any data can be read or written.

Udp_ioc_setopt takes two parameters, a pointer to the capability of the UDP channel and pointer to a *nwio_udpopt* structure that describes the requested configuration. The *nwio_udpopt* structure is defined in *server/ip/gen/udp_io.h* as:

```
typedef struct nwio_udpopt
{
        unsigned long nwuo_flags;
        udpport_t nwuo_locport;
        udpport_t nwuo_remport;
        ipaddr_t nwuo_locaddr;
        ipaddr_t nwuo_remaddr;
} nwio_udpopt_t;
```

```
#define NWUO_NOFLAGS            0x0000L
#define NWUO_ACC_MASK           0x0003L
#define         NWUO_EXCL               0x00000001L
#define         NWUO_SHARED             0x00000002L
#define         NWUO_COPY               0x00000003L
#define NWUO_LOCPORT_MASK       0x000CL
#define         NWUO_LP_SEL             0x00000004L
#define         NWUO_LP_SET             0x00000008L
#define         NWUO_LP_ANY             0x0000000CL
#define NWUO_LOCADDR_MASK       0x0010L
#define         NWUO_EN_LOC             0x00000010L
#define         NWUO_DI_LOC             0x00100000L
```

```
#define NWUO_BROAD_MASK         0x0020L
#define         NWUO_EN_BROAD           0x00000020L
#define         NWUO_DI_BROAD           0x00200000L
#define NWUO_REMPORT_MASK       0x0100L
#define         NWUO_RP_SET             0x00000100L
#define         NWUO_RP_ANY             0x01000000L
#define NWUO_REMADDR_MASK       0x0200L
#define         NWUO_RA_SET             0x00000200L
#define         NWUO_RA_ANY             0x02000000L
#define NWUO_RW_MASK            0x1000L
#define         NWUO_RWDATONLY          0x00001000L
#define         NWUO_RWDATALL           0x10000000L
#define NWUO_IPOPT_MASK         0x2000L
#define         NWUO_EN_IPOPT           0x00002000L
#define         NWUO_DI_IPOPT           0x20000000L
```

A UDP channel is considered configured when one flag in each category has been selected. Thus one of NWUO_EXCL, NWUO_SHARED or NWUO_COPY, one of NWUO_LP_SEL, NWUO_LP_SET or NWUO_LP_ANY, one of NWUO_EN_LOC or NWUO_DI_LOC, one of NWUO_EN_BROAD, or NWUO_DI_BROAD, one of NWUO_RP_SET, or NWUO_RP_ANY, one of NWUO_RA_SET, or NWUO_RA_ANY, one of NWUO_RWDATONLY, or NWUO_RWDATALL, and one of NWUO_EN_IPOPT, or NWUO_DI_IPOPT. The NWUO_ACC flags control the access to a certain UDP port. NWUO_EXCL means exclusive access: no other channel can use this port. NWUO_SHARED means shared access: only channels that specify shared access can use this port, and all packets that are received are handed to at most one channel. NWUO_COPY imposes no access restriction and all channels get a copy of every received packet for that port.

The NWUO_LOCPORT flags control the selection of the UDP port for this channel. NWUO_LP_SEL requests the server to pick a port. This port will be in the range from 32768 to 65535 and it will be unique. NWUO_LP_SET sets the local port to the value of the *nwuo_locport* field. NWUO_LP_ANY does not select a port. Reception of data is therefore not possible but it is possible to send data. (See *tcpip_read,* and *tcpip_write* above).

The NWUO_LOCADDR flags control the reception of packets. NWUO_EN_LOC enables the reception of packets with the local IP address as destination. NWUO_DI_LOC disables the reception of packets for the local IP address.
The NWUO_BROAD flags control the reception of broadcast packets. NWUO_EN_BROAD enables the reception of broadcast packets and NWUO_DI_BROAD disables the reception of broadcast packets.

The NWUO_REMPORT flags let the client specify one specific remote UDP port or allow any remote port. NWUO_RP_SET sets the remote UDP port to the value of *nwuo_remport*. Only packets with a matching remote port will be delivered and all packets will be sent to that port. NWUO_RP_ANY allows reception of packets from any port and when transmitting packets the remote port must be specified.

The NWUO_REMADDR flags control the remote IP address. NWUO_RA_SET sets the remote IP address to the value of *nwuo_remaddr.* Only packets from that address will be delivered and all packets will be sent to that address. NWUO_RA_ANY allows reception of packets from any host and when transmitting packets the remote host has to be specified.

The NWUO_RW flags control the format of the data to be sent or received. With NWUO_RWDATONLY only the data part of a UDP packet is sent to the server and only the data part is received from the server. This option can only be set if the header information is completely specified by the other options. Otherwise NWUO_RWDATALL must be used. The NWUO_RWDATALL mode presents the data part of a UDP packet with a header that contains the source and destination IP address, source and destination UDP ports, the IP options, etc. The server expects such a header to be present in front of the data to be transmitted.

The NWUO_IPOPT flags control the delivery and transmission of IP options. When NWUO_EN_IPOPT is set, IP options will be delivered and sent. When NWUO_DI_IPOPT is set, IP options will be stripped from received packets and no IP options will be sent.

UDP Library Functions

The following routines provide a somewhat easier to use interface to UDP than the routines described above (*tcpip_open*, *udp_ioc_setopt*, *tcpip_read* and *tcpip_write*).

udp_connect

```
errstat
udp_connect(udp_cap, chan_cap, srcport, dstport, dstaddr, flags)
capability *udp_cap;
capability *chan_cap;
udpport_t srcport;
udpport_t dstport;
ipaddr_t dstaddr;
```

```
int flags;
```

Udp_connect combines the functionality of *tcpip_open* and *udp_ioc_setopt*. A pointer to a UDP server capability should be passed in *udp_cap*, and the channel capability will be returned in the capability pointed to by *chan_cap*. If *srcport* is 0 then an unused port will be selected, otherwise the local port will be set to *srcport*. If *dstport* is non-zero then communication will be restricted to remote ports equal to *dstport*, otherwise any data can be sent to or received from any remote port. The same applies to *dstaddr*: if *dstaddr* is non-zero then only *dstaddr* can be reached. Currently no flags are defined so *flags* should be 0.

udp_reconnect

```
errstat
udp_reconnect(chan_cap, srcport, dstport, dstaddr, flags)
capability *chan_cap;
udpport_t srcport;
udpport_t dstport;
ipaddr_t dstaddr;
int flags;
```

Udp_reconnect is the same as *udp_connect* except that an existing channel capability is (re)used.

udp_read_msg

```
errstat
udp_read_msg(chan_cap, msg, msglen, actlen, flags)
capability *chan_cap;
char *msg;
int msglen;
int *actlen;
int flags;
```

Udp_read_msg delivers a UDP packet. The data part of the UDP packet is prepended with a *udp_io_hdr*. The actual length of the (possibly truncated) packet is returned in *actlen*. No flags are defined so *flags* should be 0.

udp_write_msg

```
errstat
udp_write_msg(chan_cap, msg, msglen, flags)
capability *chan_cap;
char *msg;
int msglen;
int flags;
```

A UDP packet can be sent with *udp_write_msg*. *Msg* should point to a *udp_io_hdr* followed by the data part of the UDP packet. The *uih_dst_addr* and *uih_dst_port* fields of the *udp_io_hdr* should be filled in if no values were specified in the *udp_connect*, or *udp_reconnect* call.

udp_close

```
errstat
udp_close(chan_cap, flags)
capability *chan_cap;
int flags;
```

Udp_close cleans up the administration kept by the UDP library but does not destroy the capability. This function should be used if the capability was passed to another process and should continue to exist. No flags are defined so *flags* should be 0.

udp_destroy

```
errstat
udp_destroy(chan_cap, flags)
capability *chan_cap;
int flags;
```

Udp_destroy not only cleans up the administration kept by the UDP library but also destroys the channel capability.

Diagnostics

The complete set of error codes is described in *ip*(L).

See Also

error(L), **ip[L]** , **ip_eth[L]** , **ip_ip[L]** , **ip_tcp[L]** , **ipsvr[A]** .

TCP client interface

(in libraries: libamoeba.a)

Name

ip_tcp - Internet Protocol server's TCP client interface stubs

Synopsis

```
#include "amoeba.h"
#include "server/ip/tcp_io.h"
#include "server/ip/types.h"
#include "server/ip/gen/in.h"
#include "server/ip/gen/tcp.h"
#include "server/ip/gen/tcp_io.h"

errstat tcp_ioc_connect(chan_cap, tcpcl)
errstat tcp_ioc_getconf(chan_cap, tcpconf)
errstat tcp_ioc_getopt(chan_cap, tcpopt)
errstat tcp_ioc_listen(chan_cap, tcpcl)
errstat tcp_ioc_setconf(chan_cap, tcpconf)
errstat tcp_ioc_setopt(chan_cap, tcpopt)
errstat tcp_ioc_shutdown(chan_cap)
```

Description

The Internet Protocol (IP) server implements four network protocols, as described in *ipsvr*(A). The routines described below give access to the ‘‘Transmission Control Protocol’’ (TCP) of the IP server. TCP implements a connection-oriented, byte-stream protocol on top of IP (see RFC-793).

Access to this service is provided using two types of capabilities: a server capability and channel capabilities. The server capability is called *tcp*. The server capability is used to obtain a channel to the corresponding server. This is done with *tcpip_open* (see *ip*(L)). The channel capability can be used to transfer data using the protocol implemented by the server. This is also done with the generic *tcpip_read* and *tcpip_write* routines. Since the semantics of the *tcpip_* functions vary slightly depending on channel type, these routines are described briefly below. The *tcp_ioc_* routines are used to manage the options and status of the TCP channel.

Before these routines are described, a brief introduction to the various types is given. The general-purpose types and rights in the capabilities are described in *ip*(L).

TCP Types

server/ip/gen/tcp.h

defines *tcpport_t* and *Tcpport_t* for use in prototypes.

server/ip/gen/tcp_io.h

defines struct nwio_tcpconf (*nwio_tcpconf_t*), struct nwio_tcpcl (*nwio_tcpcl_t*), struct nwio_tcpatt (*nwio_tcpatt_t*) and struct nwio_tcpopt (*nwio_tcpopt_t*).

server/ip/gen/tcp_hdr.h

defines struct tcp_hdr (*tcp_hdr_t*) and struct tcp_hdropt (*tcp_hdropt_t*).

General Functions

tcpip_read

```
errstat
tcpip_read(chan_cap, buffer, nbytes)
capability *chan_cap;
char *buffer;
size_t nbytes;
```

Tcpip_read reads data from the TCP channel specified by *chan_cap* into *buffer*. The call blocks until either *nbytes* of data are available, or an error condition is detected. On success it returns the number of bytes read. If the connection has been closed then it returns 0. If urgent data is pending and the channel is not configured for urgent data then the error code `TCPIP_URG` is returned. The channel should then be reconfigured for urgent data using *tcp_ioc_setopt*. Further *tcpip_read*s should then be done until the error code `TCPIP_NOURG` is returned, indicating that there is no more urgent data. At this point the channel can be reconfigured for normal operation.

```
Error Conditions:
```

See *ip*(L) for a description of the error codes.

```
STD_INTR
TCPIP_NOCONN
TCPIP_URG
TCPIP_NOURG
```

tcpip_write

```
errstat
tcpip_write(chan_cap, buffer, nbytes)
capability *chan_cap;
char *buffer;
size_t nbytes;
```

Tcpip_write writes the *nbytes* of data in *buffer* to the TCP channel specified by *chan_cap*. The call blocks until enough buffer space is available to hold the entire write. Urgent data can be sent by setting the `NWTO_SND_URG` option using *tcp_ioc_setopt* and then doing the write.

```
Error Conditions:
```

See *ip*(L) for a description of the error codes.

```
STD_INTR
TCPIP_NOTCONN
TCPIP_SHUTDOWN
```

TCP Functions

tcp_ioc_connect

```
errstat
tcp_ioc_connect(chan_cap, clopt)
capability *chan_cap;
struct nwio_tcpcl *clopt;
```

Tcp_ioc_connect tries to setup a connection with a remote TCP/IP server. The channel must be fully configured (see *tcp_ioc_setconf*) and values for the local port, the remote port and the remote address must have been specified using `NWTC_LP_SET` or `NWTC_LP_SEL`, `NWTC_SET_RA` and `NWTC_SET_RP`.

tcp_ioc_getconf

```
errstat
tcp_ioc_getconf(chan_cap, tcpconf)
capability *chan_cap;
struct nwio_tcpconf *tcpconf;
```

This call reports the current configuration of a TCP channel. See *tcp_ioc_setconf*for *struct nwio_tcpconf*.

The *nwtc_flags* field shows the status of the *access*, *locport*, *remaddr* and *remport* fields. *Nwtc_locaddr* contains the Internet address of the TCP/IP server. *Remaddr* contains the Internet address of the remote TCP/IP server, once it has been set with `NWTC_SET_RA` or after a successful connect or listen (see *tcp_ioc_connect* and *tcp_ioc_listen*). *Nwio_locport* contains the local TCP/IP port, set with `NWTC_LP_SET` or the selected port set with `NWTC_LP_SEL`. *Nwtc_remport* contains the TCP port of the remote TCP/IP server as set with `NWIO_SET_RP` or after a successful connect or listen (see *tcp_ioc_connect* or *tcp_ioc_listen*).

A value of 0 (zero) is reported for *nwtc_remaddr, nwtc_locport* or *nwtc_remport* when no value is set, either explicitly or implicitly.

tcp_ioc_getopt

```
errstat
tcp_ioc_getopt(chan_cap, tcpopt)
capability * chan_cap;
struct nwio_tcpopt *tcpopt;
```

This returns in *tcpopt* the current options for a TCP channel specified by *chan_cap*. See *tcp_ioc_setopt* for a description of the various options.

tcp_ioc_listen

```
errstat
tcp_ioc_listen(chan_cap, clopt)
capability *chan_cap;
struct nwio_tcpcl *clopt;
```

Tcp_ioc_listen waits until a remote TCP/IP server tries to connect to this channel. The channel has to be configured (see *tcp_ioc_setconf*). An additional restriction is that the local port must be set (with `NWTC_LP_SET)` or selected (with `NWTC_LP_SEL`). When a remote address is set, only connections for that host are accepted, and when a remote port is set, only connections from that port are accepted. After a successful listen, *tcp_ioc_getconf* can be used to find out the address and port of the other side.

tcp_ioc_setconf

```
errstat
tcp_ioc_setconf(chan_cap, conf)
capability *chan_cap;
struct nwio_tcpconf *conf;
```

Before a TCP channel (created by calling *tcpip_open* with a TCP server capability) can be used or options set, it must configured using *tcp_ioc_setconf*. The parameters to *tcp_ioc_setconf* are the channel capability and a *struct nwio_tcpconf* as defined in *server/ip/gen/tcp_io.h*:

```
typedef struct nwio_tcpconf
{
        unsigned long nwtc_flags;
        ipaddr_t nwtc_locaddr;
        ipaddr_t nwtc_remaddr;
        tcpport_t nwtc_locport;
        tcpport_t nwtc_remport;
} nwio_tcpconf_t;


#define NWTC_NOFLAGS    0x0000L
#define NWTC_ACC_MASK   0x0003L
#       define NWTC_EXCL        0x00000001L
#       define NWTC_SHARED      0x00000002L
#       define NWTC_COPY        0x00000003L
#define NWTC_LOCPORT_MASK       0x0030L
#       define NWTC_LP_UNSET    0x00000010L
#       define NWTC_LP_SET      0x00000020L
#       define NWTC_LP_SEL      0x00000030L
#define NWTC_REMADDR_MASK       0x0100L
#       define NWTC_SET_RA      0x00000100L
#       define NWTC_UNSET_RA    0x01000000L
#define NWTC_REMPORT_MASK       0x0200L
#       define NWTC_SET_RP      0x00000200L
#       define NWTC_UNSET_RP    0x02000000L
```

A TCP channel is considered configured when one flag in each category has been selected. Thus one of `NWTC_EXCL`, `NWTC_SHARED` or `NWTC_COPY`, one of `NWTC_LP_UNSET`, `NWTC_LP_SET` or `NWTC_LP_SEL`, one of `NWTC_SET_RA` or `NWTC_UNSET_RA`, and one of `NWTC_SET_RP` or `NWTC_UNSET_RP`.

The `NWTC_ACC_` flags control the access to a TCP port. `NWTC_EXCL` means exclusive access. An attempt to configure a channel will be denied if the same port is specified as that of a channel that requested exclusive access. `NWTC_SHARED` indicates that several channels use the same port but cooperate. If the shared mode is specified for one channel then all other channels that use the same port should also be configured with the `NWTC_SHARED` flag. `NWTC_COPY` is specified when the programmer does not care about other channels. This is the default.

The locport flags control which TCP port is used for communication. `NWTC_LP_UNSET` indicates the absence of a local port. This is the default. `NWTC_LP_SET` means that the *nwtc_locport* field contains the local port to be used by TCP. This value must be in network byte order (see *Byte Order Conversion* in *ip*(L)). `NWTC_LP_SEL` requests the TCP server to pick a port. This port will be in the range from 32768 to 65535 and will be unique.

The *remaddr* flags specify which hosts are acceptable for connections. `NWTC_SET_RA` indicates that only connections to the host specified in *nwtc_remaddr* are acceptable. *Nwtc_remaddr* should be in network byte order (see *Byte Order Conversion* in *ip*(L)). `NWTC_UNSET_RA` allows any host on the other side of a connection. This is the default.

The *remport* flags specify which remote ports are acceptable for connections. `NWTC_SET_RP` indicates that only the port specified in *nwtc_remport* is acceptable. `NWTC_UNSET_RP` allows any port on the other side of a connection. This is the default.

tcp_ioc_setopt

```
errstat
tcp_ioc_setopt(chan_cap, tcpopt)
capability * chan_cap;
struct nwio_tcpopt *tcpopt;
```

The behavior of a TCP channel can be changed using *tcp_ioc_setopt*. The parameters to *tcp_ioc_setopt* are a channel capability and a *struct nwio_tcopt* as defined in *server/ip/gen/tcp_io.h*:

```
typedef struct nwio_tcpopt
{
	u32_t nwto_flags;
} nwio_tcpopt_t;
```

The *nwto_flags* field takes the following flags:

```
#define NWTO_NOFLAG             0x0000L
#define NWTO_SND_URG_MASK       0x0001L
#       define NWTO_SND_URG     0x00000001L
#       define NWTO_SND_NOTURG  0x00010000L
#define NWTO_RCV_URG_MASK       0x0002L
#       define NWTO_RCV_URG     0x00000002L
#       define NWTO_RCV_NOTURG  0x00020000L
#define NWTO_BSD_URG_MASK       0x0004L
#       define NWTO_BSD_URG     0x00000004L
#       define NWTO_NOTBSD_URG  0x00040000L
#define NWTO_DEL_RST_MASK       0x0008L
#       define NWTO_DEL_RST     0x00000008L
#       define NWTO_NOTDEL_RST  0x00080000L
```

`NWTO_SND_URG` and `NWTO_SND_NOTURG` control the sending of urgent data. If *NWTO_SND_URG* is set, data written with *tcpip_write* will be sent as urgent data. If `NWTO_SND_NOTURG` is set, data will be sent as normal data. Similarly, `NWTO_RCV_URG` and `NWTO_RCV_NOTURG` control the reception of urgent data. If the wrong kind of data is present *tcpip_read* returns the error `TCPIP_URG` or `TCPIP_NOURG`, as appropriate.

`NWTO_BSD_URG` and `NWTO_NOTBSD_URG` control whether data is sent and received according to the BSD semantics or according to the semantics described in the RFC. These options can only be specified if a successful *tcp_ioc_connect* or *tcp_ioc_listen* has been done.

The `NWTO_DEL_RST` can be used to delay the sending of an `RST` packet in response to a `SYN` packet. Normally, when a `SYN` arrives for a port and there is no listen for that particular port, the TCP server will send an `RST` packet. This results in a “Connection Refused” error at the sending host. If a server repeatedly accepts connections for the same port, it can use this bit tell the TCP server not to send an `RST` immediately. When a `SYN` packet arrives and the TCP server is about to send an `RST` packet, it will look for existing channels to the same port with the bit set. If such a channel exists, the TCP server will delay the sending of the `RST` packet to give the server an opportunity to open a new channel and start a listen.

tcp_ioc_shutdown

```
errstat
tcp_ioc_shutdown(chan_cap)
capability *chan_cap;
```

Tcp_ioc_shutdown tells the TCP/IP server that no more data will be sent over the channel specified by *chan_cap*. This command can be issued when the channel is connected to a remote TCP/IP server. The TCP/IP server will tell the remote TCP/IP server and the client of the remote TCP/IP server will receive an end-of-file indication.

Diagnostics

The complete set of error codes is described in *ip*(L).

See Also

error(L), **ip[L]** , **ip_eth[L]** , **ip_ip[L]** , **ip_udp[L]** , **ipsvr[A]** .

TCP example program

```
/*
 * This util reads with a minimal ftp interface files
 * from a remote UNIX machine. On the UNIX side,
 * the filed server handles the file requests. All received
 * data will be written to stdout.
 */

#define TCPIPCAPNAME "/profile/cap/ipsvr/tcp"

/* some internal msg and status constants */

#define TAPEWRITE      100
#define     TAPEREAD   101
#define TAPEPORT       10000
#define TAPE_EOF       -128
#define MSG_OK  128

/* Forward declarations */
void    tcp_shutdown();
int     tcp_readn();
void    getremipaddr();
void  convert_remipaddr();

/* some global data */
capability     TcpipCap;
capability     ChanCap;
errstat err;
struct  nwio_tcpconf   tcpconf;
struct  nwio_tcpcl     tcpclopt;
struct  nwio_tcpopt    tcpopt;

unsigned  char RemIpAddr[4];



int main(argc,argv)
int argc;
char *argv[];
{

char    buffer[8192];
int     command;
int     byteswrite,writtenbytes;
unsigned long  buffercount;
char    *filename;

int     n,m;

if (argc == 1)
        usage();

if (argc == 2 || argc == 3)
        filename=argv[1];

if (argc == 3)
{
        /* Convert the IP-Addr. ...*/
```

```
        convert_remipaddr(argv[2]);
}
else
{
        /* Now its time to determine the remote IP-Addr... */
        getremipaddr();
}

fprintf(stderr,"Remote IP-
Address:%d.%d.%d.%d\n",RemIpAddr[0],RemIpAddr[1],RemIpAddr[2],RemIpAddr[3]);


/* Get the tcpip server capability */

err=name_lookup(TCPIPCAPNAME,&TcpipCap);

if (err != STD_OK)
{
        fprintf(stderr,"Error:Can't get TCPIPCAP.\n");
        return(-1);
}

/* open a tcpip channel */

err=tcpip_open(&TcpipCap,&ChanCap);

if (err != STD_OK)
{
        fprintf(stderr,"Error:tcpip_open.\n");
        fprintf(stderr,"%s\n",tcpip_why(err));
        return(-1);
}


/* get the default configuration for this tcpip channel */

err=tcp_ioc_getconf(&ChanCap,&tcpconf);

if (err != STD_OK)
{
        fprintf(stderr,"Error tcp_iocgetconf.\n");
        fprintf(stderr,"%s\n",tcpip_why(err));
        tcp_shutdown();
}

/* set the remote IP address */

tcpconf.nwtc_remaddr=(u32_t)((RemIpAddr[3] << 24)
                            + (RemIpAddr[2] << 16)
                            + (RemIpAddr[1] << 8)
                            + (RemIpAddr[0] ));


/* and the remote server port */

tcpconf.nwtc_remport=(tcpport_t) htons(TAPEPORT);

tcpconf.nwtc_flags=NWTC_EXCL
                | NWTC_LP_SEL
                | NWTC_SET_RA
                | NWTC_SET_RP;



/* set the new configuration */
```

```
err=tcp_ioc_setconf(&ChanCap,&tcpconf);

if (err != STD_OK)
{
        fprintf(stderr,"Error:tcp_ioc_setconf.\n");
        fprintf(stderr,"%s\n",tcpip_why(err));
        tcp_shutdown();
}


tcpclopt.nwtcl_flags=0;
tcpclopt.nwtcl_ttl=0xffff;

/* connect to the remote server */

err=tcp_ioc_connect(&ChanCap,&tcpclopt);

if (err != STD_OK)
{
        fprintf(stderr,"Error:tcp_ioc_connect.\n");
        fprintf(stderr,"%s\n",tcpip_why(err));
        tcp_shutdown();
}

/* No Time-destroy of ChanCap... */

err=tcpip_keepalive_cap(&ChanCap);

if (err != STD_OK)
{
        fprintf(stderr,"Error:tcpip_keepalive_cap.\n");
        fprintf(stderr,"%s\n",tcpip_why(err));
        tcp_shutdown();
}


/* Send the read request to the Server: */

command=htons(TAPEREAD);

err=tcpip_write(&ChanCap,
                (char *)&command,
                (size_t)sizeof(command));

if (err < 0)
{
        fprintf(stderr,"Error:tcpip_write.\n");
        fprintf(stderr,"%s\n",tcpip_why(err));
        /* shutdown the failed connection */
        tcp_shutdown();
}


/* Wait, until SERVER is ready to receive... */
command = 0;

err=tcpip_read(&ChanCap,
        (char *)&command,
        (size_t)sizeof(command));

if (err < 0)
{
```

```
        fprintf(stderr,"Error:tcpip_read.\n");
        fprintf(stderr,"%s\n",tcpip_why(err));
        tcp_shutdown();
}

/* check the answer ... */
command=htons(command);

if (command != MSG_OK)
{
        fprintf(stderr,"Error:Server is not ready.\n");
        tcp_shutdown();
}


/* Now send the remote Path and file name...*/

command=htons(strlen(filename));

err=tcpip_write(&ChanCap,
                (char *)&command,
                (size_t)sizeof(command));

if (err < 0)
{
        fprintf(stderr,"Error:tcpip_write.\n");
        fprintf(stderr,"%s\n",tcpip_why(err));
        tcp_shutdown();
}


/* Wait, until SERVER ack. ... */
command = 0;

err=tcpip_read(&ChanCap,
        (char *)&command,
        (size_t)sizeof(command));

if (err < 0)
{
        fprintf(stderr,"Error:tcpip_read.\n");
        fprintf(stderr,"%s\n",tcpip_why(err));
        tcp_shutdown();
}

/* check again the answer from the remote server ... */

command=htons(command);

if (command != MSG_OK)
{
        fprintf(stderr,"Error:Server is not ready.\n");
        tcp_shutdown();
}

/* and the string... */

err = tcpip_write(&ChanCap,
                filename,
                (size_t)strlen(filename));

if (err < 0)
```

```
{
        fprintf(stderr,"Error:tcpip_write.\n");
        fprintf(stderr,"%s\n",tcpip_why(err));
        tcp_shutdown();
}


command=0;

/* Wait, until SERVER ack. ... */

err=tcpip_read(&ChanCap,
                (char *)&command,
                (size_t)sizeof(command));
if (err < 0)
{
        fprintf(stderr,"Error:tcpip_read.\n");
        fprintf(stderr,"%s\n",tcpip_why(err));
        tcp_shutdown();
}


command=htons(command);

if (command != MSG_OK)
{
        fprintf(stderr,"Error:Server is not ready.\n");
        tcp_shutdown();
}

/* Ok, now receive the beast... */

buffercount=0;

do
{
        /* How many Bytes to receive ... */


        err = tcpip_read(&ChanCap,
                (char *)&command,
                (size_t)sizeof(command));

        if (err < 0)
        {
                fprintf(stderr,"Error:tcpip_read.\n");
                fprintf(stderr,"%s\n",tcpip_why(err));
                tcp_shutdown();
        }

        byteswrite=(signed int)(short)htons(command);

        /* we must read the data in fragments ... */

        if((byteswrite > 0) && (byteswrite <= sizeof(buffer)))
        {


                err=tcp_readn(buffer,(size_t) byteswrite);
                if (err < 0)
                {
                        fprintf(stderr,"Error:tcpip_read.\n");
```

```
                        fprintf(stderr,"%s\n",tcpip_why(err));
                        tcp_shutdown();
                }

                if (err != byteswrite)
                {
                        fprintf(stderr,"Received not all promised
bytes     [promised/received]:%d,%d.\n",byteswrite,err);
                        tcp_shutdown();
                }

                /* write the data to STDOUT */
                writtenbytes=write(1,buffer,byteswrite);

                if ((writtenbytes != byteswrite) && (writtenbytes > 0))
                {
                        fprintf(stderr,"Cant write enough Bytes to
stdout(%d).\n",byteswrite);
                        tcp_shutdown();
                }

                if (writtenbytes < 0)
                {
                        fprintf(stderr,"Local Write error. Shutdown conn.\n");
                        {
                                tcp_shutdown();
                        };
                        byteswrite=TAPE_EOF;
                        break;

                }
                else
                {
                        buffercount++;
                        /* send that we are ready ... */
                        command=htons(MSG_OK); /* Client have received DATA, send
it. */
                        err=tcpip_write(&ChanCap,
                                (char *)&command,
                                (size_t)sizeof(command));

                        if (err < 0)
                        {
                                fprintf(stderr,"Error:tcpip_write.\n");
                                fprintf(stderr,"%s\n",tcpip_why(err));
                                tcp_shutdown();
                        }
                }


        }

} while(byteswrite > 0);

fprintf(stderr,"FILEREAD:Received %d Blocks.\n",buffercount);
if(byteswrite != TAPE_EOF)
{
        fprintf(stderr,"Server aborted.\n");

}

/* shutwdown the connection */
```

```
err=tcp_ioc_shutdown(&ChanCap);
if (err != STD_OK)
{
        fprintf(stderr,"Error:tcp_ioc_shutdown.\n");
        fprintf(stderr,"%s\n",tcpip_why(err));
        return(-1);
}


}

/* Do the shutdown */

void tcp_shutdown()
{
        err=tcp_ioc_shutdown(&ChanCap);
        if (err != STD_OK)
        {
                fprintf(stderr,"Error:tcp_ioc_shutdown.\n");
                fprintf(stderr,"%s\n",tcpip_why(err));

        }
        exit(-1);
}

/* Read size bytes from the tcpip channel */

int tcp_readn(char *ptr,int size)
{
int     no_left,no_read;
        no_left=size;
        while(no_left > 0)
        {
                no_read=tcpip_read(&ChanCap,ptr,(size_t) no_left);
                if (no_read < 0) return(no_read);
                if (no_read==0) break;
                no_left -= no_read;
                ptr += no_read;
        }
        return(size-no_left);
}
```

Table of Contents