# QDOS TCP/IP and Socket functionality

By Martin Head 12/01/16
Based on information by Richard Zidlicky

**Introduction** (Richard Zidlicky)

This document implements TCP/IP as implemented in UQLX. The implementation is due to Jonathan Hudson and is free, the hope is that native QDOS implementations can be kept compatible with it.

**Notes** (Martin Head)

This document is for using the QDOS TCP/IP interface from assembler programming.

A lot of the information is cobbled together from information on using Sockets in the C language (which I don't speak) from the Internet, and by trial and error. As I don't know anything about socket programming, I am learning as I go along, So…

Don't take anything written here as gospel.

**The characteristics of the implementation:**

TCP/IP interface as device drivers.
Most of TCP functionality useable from SBasic. Full functionality with SBasic and some available toolkits.
Implementation of BSD compatible socket library for c68 available

The general design of the interface is chosen so that features more to be used from Assembler/Basic follow QDOS interfacing conventions, those used from C/Unix like applications follow conventions that make it easier to interface for such programs.

**Error Handling**

The IP traps return a normal QDOS error code in D0.
A more useful error code for the last IP error may be obtained from IP_ERRNO. See the end of this document for a list of IP errors.

# Opening IP channels

The Following new devices are available for the Trap#2 Operating system open calls.

**SCK_** A generic socket that can be used for accepting connections, or for netdb access.

<u>Internet Domain</u>

| | |
|---|---|
| **TCP_host:port** | Stream Socket |
| **UDP_host:port** | Datagram Socket |

<u>Unix Domain</u>

| | |
|---|---|
| **UXS_host:port** | Stream Socket |
| **UXD_host:port** | Datagram Socket |

Host and Port parameters are both optional.

Note, UDP and UXD sockets are usable from BASIC

Host and Port, can be both given either by numerical value or name.
E.g. "129.69.1.59:119" or "news.uni-stuttgart.de:nntp"

**Note**

With the exception of **IP_OPEN** and **IP_ACCEPT**. Most of the system calls that expect or return strings, do not use the usual QDOS Word sized length followed by a sequence of characters.

The length of the string is either specified in one of the calls parameters, or the end of the string is terminated in a zero (NULL) byte.

**Open call summary** (standard QDOS Trap#2 calls)

| | | |
|---|---|---|
| IP_OPEN | $01 | |
| IP_ACCEPT | $01 | |
| IO_CLOSE | $02 | Standard QDOS Close |

I do not know the exact rules which govern whether or not the IP_OPEN command succeeds or fails for a given host and port. But here is a list from my observations.

UDP

| IP Address | Open D3=0 | | | Open_in D3=1 | | | Open_new D3=2 | | |
|---|---|---|---|---|---|---|---|---|---|
| 0.0.0.0 | I | X | I | X | X | X | I | X | I |
| 127.0.0.1 | I | X | I | I | X | I | I | X | I |
| 127.0.0.10 | I | X | I | I | X | I | I | X | I |
| 172.16.0.6 | I | X | I | X | X | I | X | X | I |
| 172.16.0.10 | I | X | I | X | X | I | X | X | X |
| 192.168.0.5 | I | X | I | X | X | X | X | X | X |
| 255.255.255.255 | I | X | I | X | X | I | X | X | X |

TCP

| IP Address | Open D3=0 | | | Open_in D3=1 | | | Open_new D3=2 | | |
|---|---|---|---|---|---|---|---|---|---|
| 0.0.0.0 | I | I | I | I | I | I | I | X | I |
| 127.0.0.1 | I | I | I | I | I | I | I | X | I |
| 127.0.0.10 | I | I | I | I | I | I | I | X | I |
| 172.16.0.6 | I | I | I | I | I | I | X | X | I |
| 172.16.0.10 | I | I | I | I | I | I | X | X | X |
| 192.168.0.5 | I | I | I | I | I | I | X | X | X |
| 255.255.255.255 | I | I | I | I | I | I | X | X | X |

I = Succeed
X = Fail

Host IP address of the computer making the tests was 172.16.0.6,
Using port 5900.

| | |
|---|---|
| First column (Black) | QPC2, Not connected to a Network |
| Second column (Red) | Qemulator, Not connected to a Network |
| Third column (Green) | QPC2 connected to a Network with another computer having an IP address of 172.16.0.10 |

Note the way UDP ports don’t seem to ever open in Qemulator, I don’t know if this is a problem in Qemulator, or something I was doing wrong. There are also discrepancies in TCP opens with D3=2

This is the program I used to obtain these results.

```
100 RESTORE
110 READ n
120 port$=":5900"
130 FOR x=1 TO n
140  READ ad$
150  ch=FOP_NEW("udp_" & ad$ & port$)
160  IF ch>0 THEN
170   PRINT ad$;"  Opened OK"
180   CLOSE#ch
190  ELSE
200   PRINT ad$;"  Not OK"
210  END IF
220 END FOR x
230 DATA 7,"0.0.0.0","127.0.0.1","127.0.0.10",
         "172.16.0.6"
240 DATA "172.16.0.10","192.168.0.5",
         "255.255.255.255"
```

Change line 150 for the required Open type, and Socket type.

# IP_OPEN TRAP#2 D0=1

**Opens a channel.**

Input

| | |
|---|---|
| D1.L | Job ID |
| D3.L | code see below |
| A0 | Address of channel name |

Output

| | |
|---|---|
| D0.L | result (0 if OK) |
| D1.L | Job ID |
| A0 | channel ID |

**Description:**

Opens an IP channel for a TCP, UDP, UXS or a UXD connection

The type of the open is defined by the value supplied in D3 where

0 = Creates a socket of requested type/protocol. Host & port not required
(does a C socket() command)

1 = Host and Port must be specified.
Opens a connection for TCP, or sets peer address for UDP sockets.
Returns without error if connection can't be completed within
1-2/50s, internally the connection buildup continues. Every I/O
operation will be blocked until the connection succeeds or fails.
(does a C socket() command, then a C connect() command)

2 = bind TCP or UDP socket to an address. Such sockets can be used for
accepting incoming connections.
(does a C socket() command, then if a Host and Port are supplied,
does a C bind() command)

SuperBASIC equivalents to the D3 values are, 0=**OPEN**, 1=**OPEN_IN**, and 2=**OPEN_NEW**.

# IP_ACCEPT TRAP#2 D0=1

**Provides accept(2) functionality.**

Input

| | |
|---|---|
| D1.L | Job ID |
| D3.L | channel ID, see below |
| A0 | Address of channel name |

Output

| | |
|---|---|
| D0.L | -1 (Not Complete) when no waiting connection |
| D1.L | Job ID |
| A0 | channel ID |

**Description:**

Accept a connection for socket specified by the channel ID supplied in D3.

The channel name pointed to by A0 should be for a socket of the form ‘**SCK_**’

The argument in D3 is a socket that has been previously created with **IP_OPEN**, bound to an address with **IP_BIND**, and is listening with **IP_LISTEN** for connections.

The **IP_ACCEPT** function extracts the first connection request on the Queue, of pending connections, then creates a new socket with the same properties of the supplied channel ID and allocates a new channel ID for the new socket.

**IP_ACCEPT** returns the error ‘Not Complete’ if there are no pending connection requests and can't complete immediately.

To accept a new connection request **IP_ACCEPT** should be in a loop so that it is constantly being called while it returns the QDOS error ‘Not Complete’ (-1).

When **IP_ACCEPT**, returns 0 in D0, then a remote connection has been has been accepted, and A0 will be the channel ID of the new connection.

Use code along the following lines

```
; Accept a new connection.  D7 is the channel ID of the previously
; opened channel

accept  moveq  #$1,d0      ;IP_ACCEPT
        moveq  #-1,d1      ;owned by this job
        move.l d7,d3       ;channel ID
        lea    socket,a0  ;point at SCK_
        trap   #2
        move.l a0,a5       ;A5 is now the possible new socket
                           ;channel ID

        cmp.l  # -1,d0     ;error Not Complete
        beq.s  accept      ;..yes, run round in a loop until open is
                           ;successful, or another error

        tst.l  d0          ;any other error
        beq.s  …….         ;..no, continue

        bra    …….         ;…yes, deal with error

socket  dc.w    4
        dc.b   "SCK_"
```

**Note** the old channel ID that is supplied to D3 should be saved before **IP_ACCEPT** is called. As it may be required for further **IP_ACCEPT** calls, and for closing the channel.

If you require the socket address structure that the C accept(2) function would normally create. After the **IP_ACCEPT** command has completed successfully, use the **IP_GETPEERNAME** function to create it.

The accepted socket may not be used to accept more connections. And the original socket remains open.

This command should be part of the I/O operations, but as it is a Trap #2 instruction, it is included here

# I/O Operations

Many operations typically not regarded as IO were provided by trap#3 calls to gain flexibility.

Basic IO operations (D0=0 - 7) are defined for connected TCP sockets. They may work for UDP sockets when peer address is set, however this use is strongly discouraged. Trap#3,[$48,$49] also work but it is not clear whether they are meaningful and thus may not be supported.

Generally, TCP/IP aware software should probably use the socket specific IO functions - **SEND, RECV, SENDTO, RECVFROM**.

When a trap#3 returns with an error, An additional C confirming error code may be queried by **IP_ERRNO, IP_H_ERRNO** and **IP_H_STRERROR** operations. This code is valid unless -1.

### Basic IO operations

These are compatible to QDOS. The only questionable issue here is whether **IO.FSTRG** should always fill its buffer before returning as it does now, or rather mimic the behaviour of **recv/recvfrom**. Since the number of received characters will be in D1 anyway, this should not disturb any QDOS applications.

# Input/Output Utilisation

**Serial I/O call summary** (standard QDOS Trap#3 calls)

| | |
|---|---|
| IO_PEND | $00 |
| IO_FBYTE | $01 |
| IO_FLINE | $02 |
| IO_FSTRG | $03 |
| IO_SBYTE | $05 |
| IO_SSTRG | $07 |

## IP Trap I/O call summary (Extended Trap #3 calls)

| | |
|---|---|
| IP_LISTEN | $50 |
| IP_ACCEPT | See the Open section |
| IP_SEND | $51 |
| IP_SENDTO | $52 |
| IP_RECV | $53 |
| IP_RECVFM | $54 |
| IP_GETOPT | $55 |
| IP_SETOPT | $56 |
| IP_SHUTDWN | $57 |
| IP_BIND | $58 |
| IP_CONNECT | $59 |
| IP_FCNTL | $5a |
| IP_GETHOSTNAME | $5b |
| IP_GETSOCKNAME | $5c |
| IP_GETPEERNAME | $5d |
| IP_GETHOSTBYNAME | $5e |
| IP_GETHOSTBYADDR | $5f |
| IP_SETHOSTENT | $60 |
| IP_ENDHOSTENT | $61 |
| IP_H_ERRNO | $62 |
| IP_GETSERVENT | $63 |
| IP_GETSERVBYNAME | $64 |
| IP_GETSERVBYPORT | $65 |
| IP_SETSERVENT | $66 |
| IP_ENDSERVENT | $67 |
| IP_GETNETENT | $68 |
| IP_GETNETBYNAME | $69 |
| IP_GETNETBYADDR | $6a |
| IP_SETNETENT | $6b |
| IP_ENDNETENT | $6c |
| IP_GETPROTOENT | $6d |
| IP_GETPROTOBYNAME | $6e |
| IP_GETPROTOBYNUMBER | $6f |

| | |
|---|---|
| IP_SETPROTOENT | $70 |
| IP_ENDPROTOENT | $71 |

| | |
|---|---|
| IP_INET_ATON | $72 |
| IP_INET_ADDR | $73 |
| IP_INET_NETWORK | $74 |
| IP_INET_NTOA | $75 |
| IP_INET_MAKEADDR | $76 |
| IP_INET_LNAOF | $77 |
| IP_INET_NETOF | $78 |

| | |
|---|---|
| IP_IOCTL | $79 |
| IP_GETDOMAIN | $7a |
| IP_H_STRERROR | $7b |
| IP_H_ERRNO | $7c |

The following constants and data types are a mix from AmiTCP/IP and Linux definitions. Not all of them are meaningful or supported on every implementation.

Some definitions may useful for socket(), bind() and connect() calls and their trap#2/#3 equivalents, when trying to convert C code into the QDOS machine code calls.

| | | |
|---|---|---|
| SOCK_STREAM | 1 | stream socket - TCP |
| SOCK_DGRAM | 2 | datagram socket - UDP |
| SOCK_RAW | 3 | raw-protocol interface – SCK ? |
| SOCK_RDM | 4 | reliably-delivered message |
| SOCK_SEQPACKET | 5 | sequenced packet stream |

| | | |
|---|---|---|
| AF_UNSPEC | 0 | unspecified address family |
| AF_INET | 2 | internet: UDP, TCP, etc. |
| PF_UNSPEC | AF_UNSPEC | aliases |
| PF_INET | AF_INET | |

Constants for getsockopt()/setsockopt()

| | | |
|---|---|---|
| SOL_SOCKET | 1 | options for socket level |

| | | |
|---|---|---|
| SO_DEBUG | 1 | |
| SO_REUSEADDR | 2 | |
| SO_TYPE | 3 | |
| SO_ERROR | 4 | |
| SO_DONTROUTE | 5 | |
| SO_BROADCAST | 6 | |
| SO_SNDBUF | 7 | |
| SO_RCVBUF | 8 | |
| SO_KEEPALIVE | 9 | |
| SO_OOBINLINE | 10 | |
| SO_NO_CHECK | 11 | |
| SO_PRIORITY | 12 | |
| SO_LINGER | 13 | ignored, doesn't seem practicable in QDOS |
| SO_BSDCOMPAT | 14 | |

# Data Structures

Many of the Trap #3 commands require, or return data in a particular set organisation, or order.

Parameter Block – For a sockaddr structure

| Offset | Size | Description |
|---|---|---|
| $00 | Long | Pointer to a sockaddr structure |
| $04 | Long | Length of sockaddr structure (usually 16) |

The parameter block may be initialised as follows

```
        lea     parmblk,a2      ;point at start of parameter block
        lea     sockaddr,a1     ;point at sockaddr
        move.l  a1,(a2)         ;set pointer to sockaddr in
                                ;parameter block
        move.l  #16,4(a2)       ;length of socket address in
                                ;parameter block
```

Sockaddr – Socket Address

| Offset | Size | Description |
|---|---|---|
| $00 | Word | Family (usually 2) |
| $02 | Word | Port number |
| $04 | Long | IP address |
| $08 | Long | Zero |
| $0C | Long | Zero |

In_addr

| Offset | Size | Description |
|---|---|---|
| $00 | Long | IP address. |

Hostent – Host Entry

| Offset | Size | Name | Description |
|---|---|---|---|
| $00 | Long | Name | Pointer to Addrlist |
| $04 | Long | Aliases | Pointer to a list of Long IP addresses terminated with a  Null Long word |
| $08 | Long | Addtype | Connection type (usually 2 (AF_INET)) |
| $0C | Long | Length | Number of nodes in IP address (usually 4 (IPV4)) |
| $10 | Long | Addrlist | Pointer to a list of pointers terminated with a Null Long word.<br>Each of these pointers point to a list of Long word IP addresses, terminated with a Null Long word |

For example a hostent structures Addrlist could be -

```
Addrlist----→     pointer 1-----→     IP address
                                      IP address
                                      IP address
                                      Null

                  pointer 2--→        IP address
                                      IP address
                                      Null

                  pointer 3--→        IP address
                                      IP address
                                      IP address
                                      Null

                  Null
```

**Note** – Some of the pointer and addresses returned may not be on Word boundaries (odd addresses). Be careful when reading them

Servent – Server entry

| Offset | Size | Name | Description |
|---|---|---|---|
| $00 | Long | Name | Pointer to a Null terminated string |
| $04 | Long | Aliases | Pointer to a list of Long word pointers terminated with a Null Long word. Each pointer, points to a list of Long word IP addresses terminated with a Null Long word. |
| $08 | Long | Port | Associated port number. |
| $0C | Long | Proto | Pointer to a Null terminated string |

Netent – Network entry

| Offset | Size | Name | Description |
|---|---|---|---|
| $00 | Long | Name | Pointer to a Null terminated string |
| $04 | Long | Aliases | Pointer to a list of Long word pointers terminated with a Null Long word. Each pointer, points to a list of Long word IP addresses terminated with a Null Long word. |

Protoent – Protocol entry

| Offset | Size | Name | Description |
|---|---|---|---|
| $00 | Long | Name | Pointer to a Null terminated string |
| $04 | Long | Aliases | Pointer to a list of Long IP addresses terminated with a Null Long word |
| $08 | Long | Ports | Protocol number. |

# IP_LISTEN TRAP#3 D0=$50

**Provides listen(2) functionality.**

Input

| | |
|---|---|
| D1.L | size of backlog queue – (usually 5) |
| D3.W | timeout |
| A0 | channel ID |

Output

D0 = result (0 if OK)

**Description:**

For a socket that has been bound during open or explicitly with **IP_BIND**, this will set the number of connect requests that are queued for **IP_ACCEPT**. Additional requests will not be handled and clients receive a protocol specific error or retry will be initiated.

The **IP_LISTEN** call applies only to sockets of type TCP_ (stream sockets)

If you don't want to connect to a remote host. You want to wait for incoming connections and handle them in some way. The process is two step: first you **IP_LISTEN**, then you **IP_ACCEPT**

You need to call **IP_BIND** before you can call **IP_LISTEN** so that the server is running on a specific port.

# IP_BIND TRAP#3 D0=$58

**Provides bind(2) functionality**

Input

| | |
|---|---|
| D1.L | length of sockaddr structure |
| D3.W | timeout |
| A0 | channel ID |
| A2 | pointer to a sockaddr structure |

Output

D0 = result

**Description:**

Associates a local address with a socket.

The **IP_BIND** function is required on an unconnected socket before subsequent calls to the **IP_LISTEN** function. It is normally used to bind to either connection-oriented (stream, TCP) or connectionless (datagram UDP) sockets. The **IP_BIND** function may also be used to bind to a raw socket (the socket was created by opening  the channel with "**SCK_**" only?).

The bind function may also be used on an unconnected socket before subsequent calls to the **IP_CONNECT** function before send operations.

**Note** – **IP_BIND** may fail if you use the real IP Address of the local host, when the computer is not connected to a Network.

# IP_CONNECT TRAP#3 D0=$59

**Provides connect(2) functionality**

Input

| | |
|---|---|
| D1.L | length of sockaddr structure |
| D3.W | timeout |
| A0 | channel ID |
| A2 | pointer to a sockaddr structure |

Output

D0 = result

**Description:**

The channel ID isa socket. If it is of type UDP (Datagram), 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 TCP(Stream), this call attempts to make a connection to another socket. The other socket is specified by sockaddr, which is an address in the communications space of the socket. Each communications space interprets the sockaddr, parameter in its own way.

Generally, stream sockets may successfully connect only once; datagram sockets may use connect multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address.

Regardless of the timeout specified, the socket will remain blocked (any IO will timeout or be delayed) until the connection build up succeeded or failed.

**Note** – On UDP connections, **IP_CONNECT** may fail if you use the anything other than IP Address 127.0.0.x, when the computer is not connected to a Network. And when on a Network only the local network IP Addresses, and 255.255.255.255

# IP_FCNTL TRAP#3 D0=$5A

**Provides fcntl(2) (manipulate file descriptor) functionality for IPDEV sockets only.**

Input

| | |
|---|---|
| D1.L | cmd |
| D2.L | arg |
| D3.W | timeout |
| A0 | channel ID |

Output

D0 = result

An awful hack for now don't use it unless you have to.

**Description:**

Performs operations on the open IP channel. The operation is determined by cmd.

This function is typically used to do file locking and other file-oriented stuff, but it also has a couple socket-related functions that you might see or use from time to time.

cmd should be set to F_SETFL (4), and arg can be one of the following commands.

| | |
|---|---|
| O_NONBLOCK (4) | Set the socket to be non-blocking |
| O_ASYNC (64) | Set the socket to do asynchronous I/O. When data is ready to be recv()'d on the socket, the signal SIGIO will be raised. This is rare to see, and beyond the scope of the guide. And I think it's only available on certain systems. |

# IP_GETOPT TRAP#3 D0=$55

**Provides (some) getsockopt functionality**
get options on sockets

Input

| | |
|---|---|
| D1.L | optlen |
| D2.L | level |
| D3.W | timeout |
| | |
| A0 | channel ID |
| A1 | pointer to optval address |
| A2 | optname |

Output

D0 = result
D1.L optlen

**Description:**

Manipulate options for the socket referred to by the IP channel ID. 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 sockets level,Level is specified as 1 (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 **IP_GETPROTOENT**.

The arguments optval and optlen are used to identify a buffer in which the value for the requested option(s) are to be returned.

optlen is a value-result argument, 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. Definitions for socket level options, are described below. Options at other protocol levels vary in format and name.

Most socket-level options utilize an int argument for optval.

The following options are recognized at the socket level. Except as noted, each may be examined with **IP_GETOPT** and set with **IP_SETOPT**.

| | | |
|---|---|---|
| 1 | SO_DEBUG | enables recording of debugging information |
| 2 | SO_REUSEADDR | enables local address reuse |
| 3 | SO_TYPE | get the type of the socket (get only) |
| 4 | SO_ERROR | get and clear error on the socket (get only) |
| 5 | SO_DONTROUTE | enables routing bypass for outgoing messages |
| 6 | SO_BROADCAST | enables permission to transmit broadcast messages |
| 7 | SO_SNDBUF | set buffer size for output |
| 8 | SO_RCVBUF | set buffer size for input |
| 9 | SO_KEEPALIVE | enables keep connections alive |
| 10 | SO_OOBINLINE | enables reception of out-of-band data in band |
| 11 | SO_NO_CHECK | |
| 12 | SO_PRIORITY | |
| 13 | SO_LINGER | linger on close if data present, ignored in QDOS |
| 14 | SO_BSDCOMP | |

# IP_SETOPT TRAP#3 D0=$56

**Provides (some) setsockopt functionality**
set options on sockets

Input

| | |
|---|---|
| D1.L | optlen |
| D2.L | level |
| D3.W | timeout |

| | |
|---|---|
| A0 | channel ID |
| A1 | pointer to optval address |
| A2 | optname |

Output

D0 = result

**Description:**

Manipulate options for the socket referred to by the IP channel ID. 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 sockets API level, level is specified as 1 (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 **IP_GETPROTOENT**.

The arguments optval and optlen are used to access option values for setopt.

Optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. Definitions for socket level options, are described in **IP_GETOPT** above. 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 argument for optval. For **IP_SETOPT**, the argument should be nonzero to enable a boolean option, or zero if the option is to be disabled.

# IP_SHUTDWN TRAP#3 D0=$57

**Provides shutdown(2) functionality**

Input

| | |
|---|---|
| D1.L | how |
| D3.W | timeout |
| A0 | channel ID |

Output

D0 = result

**Description:**

Causes all or part of a full-duplex connection on the socket associated with channel ID to be shut down.

The value how, determines which receptions, or transmissions will be disallowed.

D1= 0, Disable receive
D1= 1, Disable send
D1= 2, Disable send and receive

# Socket specific IO

**IP_SEND** and **IP_RECV** differ from **IO.SSTRG** and **IO.FSTRG** in that they message oriented and allow chunks longer than 32k.

**IP_RECV** and **IP_RECVFM** return immediately when data is available, or after the first message arrives.

**IP_SEND** and **IP_RECV** can be (unlike **IP_SENDTO** and **IP_RECVFM** for UDP) applied only to sockets that have been connected previously.

**Note** – That at the time of writing, I have not been able to get **IP_SENDTO** and **IP_RECVFM** to work

# IP_SEND TRAP#3 D0=$51

**Provides send(2) functionality**

Input

| | |
|---|---|
| D1.L | flag |
| D2.L | len |
| D3.W | timeout |
| A0 | channel ID |
| A1 | pointer to buffer |

Output

D0 = result

| | |
|---|---|
| D1.L | bytes written |
| A1 | buffer address + bytes written |

**Description:**

Used to transmit a message to another socket.

The **IP_SEND** call may be used only when the socket is in a connected state (so that the intended recipient is known).

Also, **IP_SEND** is equivalent to **IP_SENDTO** with the A2 parameter block NULL and 0

The message is found in buffer and has length len.

If the message is too long to pass automically 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.

The flags argument is the bitwise OR of zero or more of the following flags.

$1 MSG_OOB Sends out-of-band data on sockets that support this notion (e.g., of type TCP (SOCK_STREAM)); the underlying protocol must also support out-of-band data.

$4 MSG_DONTROUTE Don't use a gateway to send out the packet, send to hosts only on directly connected networks. This is only usually used by diagnostic or routing programs. This is defined only for protocol families that route; packet sockets don't.

# IP_SENDTO TRAP#3 D0=$52

**Provides sendto(2) functionality**

Input

| | |
|---|---|
| D1.L | flag |
| D2.L | len |
| D3.W | timeout |

| | |
|---|---|
| A0 | channel ID |
| A1 | pointer to buffer |
| A2 | pointer to a parameter block (2 long words)<br>params[0].L = pointer to sockaddr structure, (to)<br>params[1].L = length of sockaddr structure, (tolen) |

Output

D0 = result
+ve => number of bytes sent
-ve => error code

**Description:**

Used to transmit a message to an unconnected Datagram (UDP) socket.

If **IP_SENDTO** is used on a connection-mode (TCP (SOCK_STREAM ) socket, the arguments in the parameter block are ignored (and the error EISCONN may be returned when they are not NULL and 0), and the error ENOTCONN is returned when the socket was not actually connected. Otherwise, the address of the target is given by parameter block values.

The message to send is found in buffer and has length of len.

If the message is too long to pass automically 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 nonblocking I/O mode. In nonblocking mode it would fail with the error EAGAIN or EWOULDBLOCK in this case.

The flags argument is the bitwise OR of zero or more of the following flags.

$1 MSG_OOB Sends out-of-band data on sockets that support this notion (e.g., of type TCP (SOCK_STREAM)); the underlying protocol must also support out-of-band data.

$4 MSG_DONTROUTE Don't use a gateway to send out the packet, send to hosts only on directly connected networks. This is only usually used by diagnostic or routing programs. This is defined only for protocol families that route; packet sockets don't.

# IP_RECV TRAP#3 D0=$53

**Provides recv(2) functionality**

Input

| | |
|---|---|
| D1.L | flag |
| D2.L | buffer size |
| D3.W | timeout |
| | |
| A0 | channel ID |
| A1 | pointer to buffer |

Output

| | |
|---|---|
| D0 = result code | |
| D1.L | bytes read |

**Description:**

Used to receive messages from a socket. Used to receive data on both connectionless (UDP) and connection-oriented (TCP) sockets.

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.

If no messages are available at the socket before D3 times out, **IP_RECV** waits for a message to arrive, unless the socket is nonblocking (see **IP_FCNTL**), in which case the value -1 is returned and the external variable from **IP_ERRNO** is set to EAGAIN or EWOULDBLOCK. The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested.

The flags argument is the bitwise OR of zero or more of the following flags.

$1 MSG_OOB Rquest 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.

$2 MSG_PEEK Cause 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.

$40    MSG_WAITALL    Request 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.

# IP_RECVFM TRAP#3 D0=$54

**Provides recvfrom(2) functionality**

| | |
|---|---|
| D1.L | flag |
| D2.L | buffer size |
| D3.W | timeout |

| | |
|---|---|
| A0 | channel ID |
| A1 | pointer to buffer |
| A2 | pointer to a parameter block (2 long words)<br>params[0].L = pointer to sockaddr structure, (from)<br>params[1].L = length of sockaddr structure, (fromlen) |

Output

D0 = result
+ve => number of bytes sent
-ve => error code
D1.L size of returned sockaddr structure

**Description:**

Used to receive messages from a socket. Used to receive data on both connectionless and connection-oriented sockets.

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.

If no messages are available at the socket before D3 times out, **IP_RECVFM** waits for a message to arrive, unless the socket is nonblocking (see **IP_FCNTL**), in which case the value -1 is returned and the external variable from **IP_ERRNO** is set to EAGAIN or EWOULDBLOCK. The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested.

The flags argument is the bitwise OR of zero or more of the following flags.

$1 MSG_OOB Rquest 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.

$2 MSG_PEEK Cause 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.

$40 MSG_WAITALL Request 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.

# Netdb functions

## IP_GETHOSTNAME TRAP#3 D0=$5B

**Provides gethostname(2) functionality**

Input

| | |
|---|---|
| D2.L | name buffer length |
| D3.W | timeout |
| | |
| A0 | channel ID |
| A1 | pointer to name buffer |

Output

D0 = result

**Description:**

Returns in the name buffer, the name of the host computer as a string terminated with a NULL (0) byte

It returns the name of the computer that your program is running on. The name can then be used by **IP_GETHOSTBYNAME**, below, to determine the IP address of your local machine.

The arguments are simple: name buffer is a pointer to an area of memory that will contain the hostname upon the function's return, and name buffer length, is the length in bytes of the available buffer.

**Note** – The open channel does not need to be connected or bound to anything, just an open “SCK_” will do.

# IP_GETSOCKNAME TRAP#3 D0=$5C

**Provides getsockname(2) functionality**

Input

| | |
|---|---|
| D2.L | len |
| D3.W | timeout |
| | |
| A0 | channel ID |
| A1 | pointer to an empty sockaddr structure |

Output

D0 = result

| | |
|---|---|
| D1.L | length of created, or required sockaddr structure |

**Description:**

Returns a sockaddr structure containing the current IP address and port to which the socket channel ID is bound to. The len argument should be initialised to indicate the amount of space available for the sockaddr structure. On return D1 contains the actual size of the socket address returned.

The returned address is truncated if the buffer provided is too small; in this case, D1 will return a value greater than was supplied to the call.

# IP_GETPEERNAME TRAP#3 D0=$5D

**Provides getpeername(2) functionality**

Input

| | |
|---|---|
| D2.L | len |
| D3.W | timeout |
| A0 | channel ID |
| A1 | pointer to an empty sockaddr structure |

Output

D0 = result

| | |
|---|---|
| D1.L | addrlen |

**Description:**

Returns a sockaddr structure containing the current IP address and port to which the socket channel ID is connected to (the peer). The len argument should be initialised to indicate the amount of space available for the sockaddr structure. On return D1 contains the actual size of the socket address returned.

The returned address is truncated if the buffer provided is too small; in this case, D1 will return a value greater than was supplied to the call in D1.

Once you have either **IP_ACCEPT**ed a remote connection, or **IP_CONNECT**ed to a server, you now have what is known as a peer. The peer is simply the computer you're connected to, identified by an IP address and a port. So...

**IP_GETPEERNAME** simply returns a sockaddr structure filled with information about the machine you're connected to.

# IP_GETHOSTBYNAME TRAP#3 D0=$5E

**Provides gethostbyname(2) functionality**

Input

| | |
|---|---|
| D3.W | timeout |
| A0 | channel ID |
| A1 | pointer to a name buffer containing the host name (terminated with a NULL) |
| A2 | pointer to a hostent structure buffer of 1024 bytes |

The buffer pointed to by A2 must be large enough to hold the largest hostent structure that may be returned (minimum of 500 bytes).

D0 = result

**Description:**

Returns a hostent structure for the given host name. The host name is either a hostname (e.g. “Tower-System”, or “www.google.com”), or an IPv4 address in standard dot notation.

If name is an IPv4 address, no lookup is performed and **IP_GETHOSTBYNAME** simply copies name into the hostent’s Name field and its struct in_addr equivalent into the hostent’s Addlist[0] field.

**IP_GETHOSTBYNAME** and **IP_GETHOSTBYADDR** map back and forth between host names and IP addresses. For instance, if you have "www.example.com", you can use **IP_GETHOSTBYNAME** to get its IP address and store it in a struct in_addr.

**IP_GETHOSTBYNAME** takes a string like "www.yahoo.com", and returns a struct hostent which contains tons of information, including the IP address. (Other information is the official host name, a list of aliases, the address type, the length of the addresses, and the list of addresses—it's a general-purpose structure that's pretty easy to use once you see how.)

**Note** – The open channel does not need to be connected or bound to anything, just an open “SCK_” will do.

# IP_GETHOSTBYADDR TRAP#3 D0=$5F

**Provides gethostbyaddr(2) functionality**

Input

| | |
|---|---|
| D1.L | addrlen |
| D2.L | type (usually 2) |
| D3.W | timeout |
| | |
| A0 | channel ID |
| A1 | pointer to addr buffer |
| A2 | pointer to a hostent structure buffer |

The buffer pointed to by A2 must be large enough to hold the largest hostent structure that may be returned (minimum of 500 bytes).

D0 = result

**Description:**

Returns a hostent structure for the given host address addr of length addrlen and address type type. Valid address type is AF_INET (2).

If you have a struct in_addr or a struct in6_addr, you can use **IP_GETHOSTBYADDR** to get the hostname back.

**IP_GETHOSHBYADDR** takes a struct in_addr or struct in6_addr and brings you up a corresponding host name (if there is one), so it's sort of the reverse of **IP_GETHOSTBYNAME**.

**Note** – The open channel does not need to be connected or bound to anything, just an open “SCK_” will do.

## IP_SETHOSTENT TRAP#3 D0=$60

## IP_SETSERVENT TRAP#3 D0=$66

## IP_SETNETENT TRAP#3 D0=$6B

## IP_SETPROTOENT TRAP#3 D0=$70

**Provides set*ent(2) functionality**

Input

| | |
|---|---|
| D1.L | stayopen |
| D3.W | timeout |
| A0 | channel ID |

Output

D0 = result

**Description:**

The **IP_SETHOSTENT** function specifies, if stayopen is true (1), that a connected TCP socket should be used for the name server queries and that the connection should remain open during successive queries. Otherwise, name server queries will use UDP datagrams

The **IP_SETSERVENT** function opens a connection to the database, and sets the next entry to the first entry. If stayopen is nonzero, then the connection to the database will not be closed between calls to one of the **IP_GETSERV*** functions.

The **IP_SETNETENT** function opens a connection to the database, and sets the next entry to the first entry. If stayopen is nonzero, then the connection to the database will not be closed between calls to one of the **IP_GETNET*** functions.

The **IP_SETPROTOENT** function opens a connection to the database, and sets the next entry to the first entry. If stayopen is nonzero, then the connection to the database will not be closed between calls to one of the **IP_GETPROTO*** functions.

## IP_ENDHOSTENT TRAP#3 D0=$61
## IP_ENDSERVENT TRAP#3 D0=$67
## IP_ENDNETENT TRAP#3 D0=$6C
## IP_ENDPROTOENT TRAP#3 D0=$71

**Provides end*ent(2) functionality**

Input

| | |
|---|---|
| D3.W | timeout |
| A0 | channel ID |

Output

D0 = result

**Description:**

The **IP_ENDHOSTENT** function ends the use of a TCP connection for name server queries.

The **IP_ENDSERVENT** function closes the connection to the database.

The **IP_ENDNETENT** function closes the connection to the database.

The **IP_ENDPROTOENT** function closes the connection to the database.

# IP_GETNETENT TRAP#3 D0=$68

**Provides getnetent(2) functionality**

Input

| | | |
|---|---|---|
| D3.W | timeout | |
| A0 | channel ID | |
| A2 | pointer to a buffer | // cast as necessary |

Output

D0 = result

**Description:**

The **IP_GETNETENT** function reads the next entry from the networks database and returns a netent structure containing the broken-out fields from the entry. A connection is opened to the database if necessary.

# IP_GETNETBYNAME TRAP#3 D0=$69

**Provides getnetbyname(2) functionality**

Input

| | |
|---|---|
| D3.W | timeout |
| A0 | channel ID |
| A1 | pointer to a buffer holding a network name |
| A2 | pointer to a netent structure buffer of 1024 bytes |

Output

D0 = result

**Description:**

Returns a netent structure (or EOF) for the entry from the database that matches the network name pointed to by A1.

**Note** – The open channel does not need to be connected or bound to anything, just an open "SCK_" will do.

# IP_GETNETBYADDR TRAP#3 D0=$6A

**Provides getnetbyname(2) functionality**

Input

| | |
|---|---|
| D1.L | net |
| D2.L | type |
| D3.W | timeout |
| A0 | channel ID |
| A2 | pointer to a netent structure buffer of 1024 bytes |

Output

D0 = result

**Description:**

Returns a netent structure (or EOF) for the entry from the database that matches the network number net. Type should be 2 (AF_INET).

The net argument must be in host byte order.

**Note** – The open channel does not need to be connected or bound to anything, just an open “SCK_” will do.

# IP_GETPROTOENT TRAP#3 D0=$6D

**Provides getprotoent(2) functionality**

Input

| | | |
|---|---|---|
| D3.W | timeout | |
| A0 | channel ID | |
| A2 | pointer to a buffer | // cast as necessary |

Output

D0 = result

**Description:**

The **IP_GETPROTOENT** function reads the next entry from the protocols database and returns a protoent structure containing the broken-out fields from the entry. A connection is opened to the database if necessary.

# IP_GETPROTOBYNAME TRAP#3 D0=$6E

**Provides getprotobyname(2) functionality**

Input

| | |
|---|---|
| D3.W | timeout |
| A0 | channel ID |
| A1 | pointer to a buffer containing a name |
| A2 | pointer to a protoent structure buffer of 1024 bytes |

Output

D0 = result

**Description:**

Returns a protoent structure for the entry from the database that matches the protocol name name. A connection is opened to the database if necessary.

**Note** – The open channel does not need to be connected or bound to anything, just an open "SCK_" will do.

# IP_GETPROTOBYNUMBER  TRAP#3  D0=$6F

**Provides getprotobynumber(2) functionality**

Input

| | |
|---|---|
| D1.L | number |
| D3.W | timeout |
| A0 | channel ID |
| A2 | pointer to a protoent structure buffer of 1024 bytes |

Output

D0 = result

**Description:**

Returns a protoent structure for the entry from the database that matches the protocol number number. A connection is opened to the database if necessary.

**Note** – The open channel does not need to be connected or bound to anything, just an open “SCK_” will do.

# IP_GETSERVENT TRAP#3 D0=$63

**Provides getservent(2) functionality**

Input

| | | |
|---|---|---|
| D1.L | number | |
| D3.W | timeout | |
| A0 | channel ID | |
| A2 | pointer to a buffer | // cast as necessary |

Output

D0 = result

**Description:**

The **IP_GETSERVENT** function reads the next entry from the services database and returns a servent structure containing the broken-out fields from the entry. A connection is opened to the database if necessary.

# IP_GETSERVBYNAME TRAP#3 D0=$64

**Provides getservbyname(2) functionality**

Input

| | |
|---|---|
| D1.L | number |
| D3.W | timeout |
| A0 | channel ID |
| A1 | pointer to a buffer containing a proto |
| A2 | pointer to a buffer of 1024 bytes |

Output

D0 = result

**Description:**

The **IP_GETSERVBYNAME** function returns a servent structure for the entry from the database that matches the service name using protocol proto. If proto is NULL, any protocol will be matched. A connection is opened to the database if necessary.

**Note** – The open channel does not need to be connected or bound to anything, just an open “SCK_” will do.

# IP_GETSERVBYPORT TRAP#3 D0=$65

**Provides getservbyport(2) functionality**

Input

| | |
|---|---|
| D1.L | port |
| D3.W | timeout |
| A0 | channel ID |
| A2 | pointer to a buffer of 1024 bytes |
| A3 | pointer to a buffer containing a proto |

Output

D0 = result

**Description:**

The **IP_GETSERVBYPORT** function returns a servent structure for the entry from the database that matches the port port (given in network byte order) using protocol proto. If proto is NULL, any protocol will be matched. A connection is opened to the database if necessary.

**Note** – The open channel does not need to be connected or bound to anything, just an open “SCK_” will do.

# IP_INET_ATON TRAP#3 D0=$72

**Provides inet_aton(2) functionality**

Input

| | |
|---|---|
| D3.W | timeout |
| A0 | channel ID |
| A1 | pointer to a buffer, name containing an IP address |
| A2 | pointer to a in_addr structure, inaddr |

Output

| | |
|---|---|
| D0 | |
| D1.L | -1 if successful, 0 if not |

**Description:**

Converts the Internet host address pointer at by A1 from the IPv4 numbers-and-dots notation into binary form (in network byte order) and stores it in the structure that inaddr points to. IP_INET_ATON returns nonzero if the address is valid, zero if not. The address supplied in A1 can have one of the following forms:

a.b.c.d Each of the four numeric parts specifies a byte of the address; the bytes are assigned in left-to-right order to produce the binary address.

a.b.c Parts a and b specify the first two bytes of the binary address. Part c is interpreted as a 16-bit value that defines the rightmost two bytes of the binary address. This notation is suitable for specifying (outmoded) Class B network addresses.

a.b Part a specifies the first byte of the binary address. Part b is interpreted as a 24-bit value that defines the rightmost three bytes of the binary address. This notation is suitable for specifying (outmoded) Class A network addresses.

a The value a is interpreted as a 32-bit value that is stored directly into the binary address without any byte rearrangement.

In all of the above forms, components of the dotted address can be specified in decimal, octal (with a leading 0), or hexadecimal, with a leading 0X). Addresses in any of these forms are collectively termed IPV4 numbers-and-dots notation. The form that uses exactly four decimal numbers is referred to as IPv4 dotted-decimal notation (or sometimes: IPv4 dotted-quad notation).

All of these functions convert from a struct in_addr (part of your struct sockaddr_in, most likely) to a string in dots-and-numbers format (e.g. "192.168.5.10") and vice-versa. If you have an IP address passed on the command line or something, this is the easiest way to get a struct in_addr to connect() to, or whatever. If you need more power, try some of the DNS functions like gethostbyname() or attempt a coup d'État in your local country.

The function **IP_INET_ATON** converts from a NULL terminated dots-and-numbers string into a long word in memory pointed to by A2.

**IP_INET_ATON** returns 1 if the supplied string was successfully interpreted, or 0 if the string is invalid (errno is not set on error).

**Note** – The open channel does not need to be connected or bound to anything, just an open “SCK_” will do.

# IP_INET_ADDR TRAP#3 D0=$73

**Provides inet_addr(2) functionality**

Input

| | |
|---|---|
| D3.W | timeout |
| A0 | channel ID |
| A1 | pointer to a buffer, name containing an IP address |

Output

| | |
|---|---|
| D0 | |
| D1.L | IP Address, or -1 if invalid |

**Description:**

Converts the NULL terminated Internet host address pointed to by A1 from IPv4 numbers-and-dots notation into binary data in network byte order in D1.

If the input is invalid, -1 is returned. Use of this function is problematic because -1 is a valid address (255.255.255.255). Avoid its use in favour of **IP_INET_ATON**.

**Note** – The open channel does not need to be connected or bound to anything, just an open "SCK_" will do.

# IP_INET_NETWORK TRAP#3 D0=$74

**Provides inet_network(2) functionality**

Input

| | |
|---|---|
| D3.W | timeout |
| A0 | channel ID |
| A1 | pointer to a buffer, name containing an IP address |

Output

D0 = result

| | |
|---|---|
| D1.L | IP Address, or -1 if invalid |

**Description:**

Converts a NULL terminated string of IPv4 numbers-and-dots notation pointed at by A1, into a number in host byte order suitable for use as an Internet network address. On success, the converted address is returned in D1. If the input is invalid, -1 is returned.

**Note** – The open channel does not need to be connected or bound to anything, just an open “SCK_” will do.

# IP_INET_NTOA TRAP#3 D0=$75

**Provides (2) functionality**

Input

| | |
|---|---|
| D3.W | timeout |
| A0 | channel ID |
| A1 | pointer to a buffer |
| A2 | pointer to a result buffer |

Output

D0 = result

**Description:**

Converts the Internet net address pointed to by A1, given in network byte order, to a string in IPv4 dotted-decimal notation. The NULL terminated string is returned in the buffer pointed to by A2.

The "n" in "ntoa" stands for network, and the "a" stands for ASCII for historical reasons (so it's "Network To ASCII"—the "toa" suffix has an analogous friend in the C library called atoi() which converts an ASCII string to an integer.)

**Note** – The open channel does not need to be connected or bound to anything, just an open “SCK_” will do.

# IP_INET_MAKEADDR TRAP#3 D0=$76

**Provides (2) functionality**

Input

| | |
|---|---|
| D1.L | network number |
| D2.L | host address |
| D3.W | timeout |
| A0 | channel ID |
| A2 | pointer to a result buffer |

Output

D0 = result

**Description:**

The **IP_INET_MAKEADDR** function is the converse of **IP_INET_NETOF** and **IP_INET_LNAOF**. It returns an Internet host address in network byte order, created by combining the network number with the local address host, both in host byte order.

The host address is the computer number, and the network is the number of the network that the computer is on. e.g. a computer with an IP Address of 192.168.0.12 would be computer 12 on the 192.168.0 network.

The exact split, between the network, and the host is determined by the sub-net mask

**Note** – The open channel does not need to be connected or bound to anything, just an open “SCK_” will do.

# IP_INET_LNAOF TRAP#3 D0=$77

**Provides inet_lnaof (2) functionality**

Input

| | |
|---|---|
| D3.W | timeout |
| A0 | channel ID |
| A1 | pointer to a buffer containing a long word IP Address |

Output

D0 = result
D1.L host address

**Description:**

Returns the host address part of the Internet address pointed to by A1. The returned value in D1is in host byte order.

These are legacy functions that assume they are dealing with classful network addresses. Classful networking divides IPv4 network addresses into host and network components at byte boundaries, as follows:

Class A This address type is indicated by the value 0 in the most significant bit of the (network byte ordered) address. The network address is contained in the most significant byte, and the host address occupies the remaining three bytes.

Class B This address type is indicated by the binary value 10 in the most significant two bits of the address. The network address is contained in the two most significant bytes, and the host address occupies the remaining two bytes.

Class C This address type is indicated by the binary value 110 in the most significant three bits of the address. The network address is contained in the three most significant bytes, and the host address occupies the remaining byte.

**Note** – The open channel does not need to be connected or bound to anything, just an open “SCK_” will do.

# IP_INET_NETOF TRAP#3 D0=$78

**Provides inet_netof(2) functionality**

Input

| | |
|---|---|
| D3.W | timeout |
| A0 | channel ID |
| A1 | pointer to a buffer containing a long word IP Address |

Output

D0 = result
D1.L network number

**Description:**

Returns the network number part of the Internet address pointed to by A1. The returned value in D1is in host byte order.

These are legacy functions that assume they are dealing with classful network addresses. Classful networking divides IPv4 network addresses into host and network components at byte boundaries, as follows:

Class A This address type is indicated by the value 0 in the most significant bit of the (network byte ordered) address. The network address is contained in the most significant byte, and the host address occupies the remaining three bytes.

Class B This address type is indicated by the binary value 10 in the most significant two bits of the address. The network address is contained in the two most significant bytes, and the host address occupies the remaining two bytes.

Class C This address type is indicated by the binary value 110 in the most significant three bits of the address. The network address is contained in the three most significant bytes, and the host address occupies the remaining byte.

**Note** – The open channel does not need to be connected or bound to anything, just an open “SCK_” will do.

# IP_IOCTL TRAP#3 D0=$79

**Provides ioctl(2) functionality**

Input

| | |
|---|---|
| D1.L | request, action |
| D3.W | timeout (-1) |
| A0 | Channel ID |
| A1 | pointer to string of character arguments |

Output

D0 = result

**Description:**

The **IP_IOCTL** function manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g. terminals) may be controlled with **IP_IOCTL** requests.

The channel ID supplied in A0 must be an open file descriptor.

Used for device specific input/output operations. request, is a device specific command. e.g. tell a CD ROM drive to open it's tray.

The codes are system specific.

***** Needs further looking into ******

# IP_GETDOMAIN TRAP#3 D0=$7A

**Provides getdomainname(2) functionality**

Input

| | |
|---|---|
| D2.L | len |
| D3.W | timeout |
| A0 | channel ID |
| A1 | pointer to a buffer, name |

Output

D0 = result

**Description:**

Used to access or to change the NIS domain name of the host system.

Returns the null-terminated domain name in the buffer, name, which has a length of len bytes. If the null-terminated domain name requires more than len bytes, **IP_GETDOMAIN** returns the first len bytes

**Note** – The open channel does not need to be connected or bound to anything, just an open “SCK_” will do.

# IP_H_ERRNO TRAP#3 D0=$62

**Provides h_errno (2) functionality**

Input

| | |
|---|---|
| D3.W | timeout |
| A0 | channel ID |

Output

D0 = result
D1.L h_errno

**Description:**

The **IP_GETHOSTBYNAME** and **IP_GETHOSTBYADDR** functions indicate an error condition by returning a null pointer and setting the external integer h_errno to indicate the error return status.

When **IP_GETHOSTBYNAME** or **IP_GETHOSTBYADDR** returns an error status, **IP_H_ERRNO**, which is very similar to **IP_ERRNO**, can be checked to determine whether the error is the result of a temporary failure or an invalid or unknown host.

Use the **IP_H_STRERROR** routine to print the error message describing the failure. If the argument string to herror is not NULL, it is printed, followed by a colon (:) and a space. The error message is printed with a trailing new-line character.

# IP_H_STRERROR TRAP#3 D0=$7B

**Provides special functionality to return the text for h_errno**

Input

| | |
|---|---|
| D1.L | error no |
| D2.L | length of buffer |
| D3.W | timeout |
| A0 | channel ID |
| A1 | pointer to buffer for text |

Output

| | |
|---|---|
| D0 = result | |
| A1 | pointer to buffer with text |

**Description:**

The **IP_H_STRERROR** function returns a pointer to a string that describes the error code passed in the argument error no. (For example, if error no is EINVAL, the returned description will be "Invalid argument".) This string must not be modified by the application, but may be modified by a subsequent call to **IP_H_STRERROR**.

In a nutshell, this function takes an error no values, like ECONNRESET, and prints them nicely, like "Connection reset by peer."

The function **IP_H_STRERROR** returns a pointer to the error message string for a given value (you usually pass in the variable error no.)

**Note** - At least that’s what it’s supposed to do, In testing I have only ever seen “Unknown error” returned.

# IP_ERRNO TRAP#3 D0=$7C

**Provides (2) functionality**

Input

| | |
|---|---|
| D3.W | timeout |
| A0 | channel ID |

Output

D0 = result

| | |
|---|---|
| D1.L | h_errno |

**Description:**

This function will return in D1 the last IP error number (not the QDOS error number), from the last IP command.

**IP_H_STRERROR** may be used to get a human-readable version of the error.

# IP Error codes

This is a list of C Error Codes in Linux, I don’t know how many of them may appear from the QDOS IP calls

| Err no | Error name | Description |
|---|---|---|
| 1 | EPERM | Operation not permitted |
| 2 | ENOENT | No such file or directory |
| 3 | ESRCH | No such process |
| 4 | EINTR | Interrupted system call |
| 5 | EIO | I/O error |
| 6 | ENXIO | No such device or address |
| 7 | E2BIG | Argument list too long |
| 8 | ENOEXEC | Exec format error |
| 9 | EBADF | Bad file number |
| 10 | ECHILD | No child processes |
| 11 | EAGAIN | Try again |
| 12 | ENOMEM | Out of memory |
| 13 | EACCES | Permission denied |
| 14 | EFAULT | Bad address |
| 15 | ENOTBLK | Block device required |
| 16 | EBUSY | Device or resource busy |
| 17 | EEXIST | File exists |
| 18 | EXDEV | Cross-device link |
| 19 | ENODEV | No such device |
| 20 | ENOTDIR | Not a directory |
| 21 | EISDIR | Is a directory |
| 22 | EINVAL | Invalid argument |
| 23 | ENFILE | File table overflow |
| 24 | EMFILE | Too many open files |
| 25 | ENOTTY | Not a typewriter |
| 26 | ETXTBSY | Text file busy |
| 27 | EFBIG | File too large |
| 28 | ENOSPC | No space left on device |
| 29 | ESPIPE | Illegal seek |
| 30 | EROFS | Read-only file system |
| 31 | EMLINK | Too many links |
| 32 | EPIPE | Broken pipe |
| 33 | EDOM | Math argument out of domain of func |
| 34 | ERANGE | Math result not representable |
| 35 | EDEADLK | Resource deadlock would occur |
| 36 | ENAMETOOLONG | File name too long |
| 37 | ENOLCK | No record locks available |

| Err no | Error name | Description |
|---|---|---|
| 38 | ENOSYS | Function not implemented |
| 39 | ENOTEMPTY | Directory not empty |
| 40 | ELOOP | Too many symbolic links encountered |
|  | EWOULDBLOCK | EAGAIN Operation would block |
| 42 | ENOMSG | No message of desired type |
| 43 | EIDRM | Identifier removed |
| 44 | ECHRNG | Channel number out of range |
| 45 | EL2NSYNC | Level 2 not synchronized |
| 46 | EL3HLT | Level 3 halted |
| 47 | EL3RST | Level 3 reset |
| 48 | ELNRNG | Link number out of range |
| 49 | EUNATCH | Protocol driver not attached |
| 50 | ENOCSI | No CSI structure available |
| 51 | EL2HLT | Level 2 halted |
| 52 | EBADE | Invalid exchange |
| 53 | EBADR | Invalid request descriptor |
| 54 | EXFULL | Exchange full |
| 55 | ENOANO | No anode |
| 56 | EBADRQC | Invalid request code |
| 57 | EBADSLT | Invalid slot |
|  | EDEADLOCK | EDEADLK |
| 59 | EBFONT | Bad font file format |
| 60 | ENOSTR | Device not a stream |
| 61 | ENODATA | No data available |
| 62 | ETIME | Timer expired |
| 63 | ENOSR | Out of streams resources |
| 64 | ENONET | Machine is not on the network |
| 65 | ENOPKG | Package not installed |
| 66 | EREMOTE | Object is remote |
| 67 | ENOLINK | Link has been severed |
| 68 | EADV | Advertise error |
| 69 | ESRMNT | Srmount error |
| 70 | ECOMM | Communication error on send |
| 71 | EPROTO | Protocol error |
| 72 | EMULTIHOP | Multihop attempted |
| 73 | EDOTDOT | RFS specific error |
| 74 | EBADMSG | Not a data message |
| 75 | EOVERFLOW | Value too large for defined data type |
| 76 | ENOTUNIQ | Name not unique on network |
| 77 | EBADFD | File descriptor in bad state |
| 78 | EREMCHG | Remote address changed |
| 79 | ELIBACC | Can not access a needed shared library |
| 80 | ELIBBAD | Accessing a corrupted shared library |
| 81 | ELIBSCN | .lib section in a.out corrupted |

| Err no | Error name | Description |
|---|---|---|
| 82 | ELIBMAX | Attempting to link in too many shared libraries |
| 83 | ELIBEXEC | Cannot exec a shared library directly |
| 84 | EILSEQ | Illegal byte sequence |
| 85 | ERESTART | Interrupted system call should be restarted |
| 86 | ESTRPIPE | Streams pipe error |
| 87 | EUSERS | Too many users |
| 88 | ENOTSOCK | Socket operation on non-socket |
| 89 | EDESTADDRREQ | Destination address required |
| 90 | EMSGSIZE | Message too long |
| 91 | EPROTOTYPE | Protocol wrong type for socket |
| 92 | ENOPROTOOPT | Protocol not available |
| 93 | EPROTONOSUPPORT | Protocol not supported |
| 94 | ESOCKTNOSUPPORT | Socket type not supported |
| 95 | EOPNOTSUPP | Operation not supported on transport endpoint |
| 96 | EPFNOSUPPORT | Protocol family not supported |
| 97 | EAFNOSUPPORT | Address family not supported by protocol |
| 98 | EADDRINUSE | Address already in use |
| 99 | EADDRNOTAVAIL | Cannot assign requested address |
| 100 | ENETDOWN | Network is down |
| 101 | ENETUNREACH | Network is unreachable |
| 102 | ENETRESET | Network dropped connection because of reset |
| 103 | ECONNABORTED | Software caused connection abort |
| 104 | ECONNRESET | Connection reset by peer |
| 105 | ENOBUFS | No buffer space available |
| 106 | EISCONN | Transport endpoint is already connected |
| 107 | ENOTCONN | Transport endpoint is not connected |
| 108 | ESHUTDOWN | Cannot send after transport endpoint shutdown |
| 109 | ETOOMANYREFS | Too many references: cannot splice |
| 110 | ETIMEDOUT | Connection timed out |
| 111 | ECONNREFUSED | Connection refused |
| 112 | EHOSTDOWN | Host is down |
| 113 | EHOSTUNREACH | No route to host |
| 114 | EALREADY | Operation already in progress |
| 115 | EINPROGRESS | Operation now in progress |
| 116 | ESTALE | Stale NFS file handle |
| 117 | EUCLEAN | Structure needs cleaning |
| 118 | ENOTNAM | Not a XENIX named type file |
| 119 | ENAVAIL | No XENIX semaphores available |
| 120 | EISNAM | Is a named type file |
| 121 | EREMOTEIO | Remote I/O error |
| 122 | EDQUOT | Quota exceeded |
| 123 | ENOMEDIUM | No medium found |
| 124 | EMEDIUMTYPE | Wrong medium type |

| Err no | Error name | Description |
|---|---|---|
| 125 | ECANCELED | Operation Canceled |
| 126 | ENOKEY | Required key not available |
| 127 | EKEYEXPIRED | Key has expired |
| 128 | EKEYREVOKED | Key has been revoked |
| 129 | EKEYREJECTED | Key was rejected by service |
| For robust mutexes | | |
| 130 | EOWNERDEAD | Owner died |
| 131 | ENOTRECOVERABLE | State not recoverable |