IPBasic

This is a reference guide to IPBasic, Which is a S*Basic interface for the IP device driver by Jonathan Hudson & Richard Zidlicky, as used in UQLX, QPC2, and Qemulator.

The IPBasic interface for the IP device drivers implements most of the functions provided in Qlsocket v1.05 (a socket library for ‘C68’). And has been developed and mainly tested in QPC2. I have highlighted areas where I have found differences between the implementations of the IP device drivers between the different emulators.

Where the IP device driver system calls do not work in QPC2, I have tried to implement them. But I have not been able to test them. So I cannot be sure that these IPBasic commands will work correctly on other systems.

One of the main problems I encountered during my investigations into the IP device drivers, is with UDP connections. In QPC2 they don’t always seem to work as expected, and I have never been able to get the IP_SENDTO and IP_RECVFM functions to work, which are essential for connectionless communications in UDP. Also Qemulator seems to have problems even opening UDP sockets.

These problems may be due to me not understanding how to use them, or doing something incorrectly.

Just about everything I know about using sockets, and the IP device drivers, I have picked up as I have been playing with them. So don’t take anything I say in this document as gospel. I may have got it all wrong!

The Keyword section gives a brief explanation of the keywords function, followed by a loose definition of the syntax and examples of usage.

The Data Structure section details the IPBasic implementation of the data structures as used in the IP device driver.

The Error code section is a list of ‘C’ error codes used in Linux. I don’t know how accurate they are for the IP device driver.

Martin Head

Release V1.00

OPEN, FOPEN
OPEN_IN, FOP_IN
OPEN_NEW, FOP_NEW sockets

OPEN, **OPEN_IN**, and **OPEN_NEW** are used to open IP sockets and link them to SuperBASIC channels.

Each of the three commands will open a socket of the specified type, and may also perform a **IP_CONNECT**, or an **IP_BIND** operation.

OPEN just creates a socket of the requested type/protocol. A host & port not required.

OPEN_IN creates a socket of the requested type/protocol. It opens a connection for TCP, or sets the peer address for UDP sockets by performing an **IP_CONNECT** operation. The Host and Port must be specified.

OPEN_NEW creates a socket of the requested type/protocol. It opens a connection for TCP, or sets host address for UDP sockets. And if a Host and Port are supplied, performs a **IP_BIND** operation.

There are five new devices added to QDOS with the IP drivers, to provide network connections.

SCK	A generic socket that can be used for accepting connections.
UDP	A datagram socket for the Internet domain
TCP	A stream socket for the Internet domain
UXD	A datagram socket for the Unix domain
UXS	A stream socket for the Unix domain

syntax:
channel_number := *numeric_expression*
socket_type := **SCK_ | UDP_ | TCP_ | UXD_ | UXS_**
IP_address := IP Address in IPv4 numbers-and-dots notation
port := Integer between 0 and 65535
url := Internet Universal Resource Locator
IP_specifier := *socket_type_IP_address*:*port* | *socket_type_url*

OPEN#*channel_number,socket_type*
OPEN_IN#*channel_number*,*IP_specifier*
OPEN_NEW#*channel_number*,*IP_specifier*

FOPEN([**#***channel_number,*]*socket_type***)**
FOP_IN([**#***channel_number*,]*IP_specifier***)**
FOP_NEW([**#***channel_number*,]*IP_specifier***)**

example:
i. **OPEN#4,SCK_**
ii. **OPEN_IN#5,”TCP_news.uni-stuttgart.de.nntp”** {same as 129.69.1.59:119}
iii. **OPEN_NEW#ch,”UDP_192.168.0.5:5800”**

Note: I do not know the exact rules which govern whether or not the OPEN commands succeed or fails for a given host and port. But here is a list made from my observations.

<u>UDP</u>

IP Address	**OPEN**	**OPEN_IN**	**OPEN_NEW**
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

<u>TCP</u>

IP Address	**OPEN**	**OPEN_IN**	**OPEN_NEW**
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 = Failed

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 OPEN_NEW

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_LISTEN sockets

IP_LISTEN will set the number of connect requests that are queued for **IP_ACCEPT** on a socket that has been bound during open or explicitly with **IP_BIND**. 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 must wait for incoming connections and handle them in some way. The process is two step: first you use **IP_LISTEN**, then you use **IP_ACCEPT**.

IP_BIND must be used before you can use **IP_LISTEN** so that the server is running on a specific port.

The optional *queue_size* sets the size of the backlog queue, The default being 5.

syntax: *channel_number* := *numeric_expression*
queue_size := *numeric_expression*

IP_LISTEN#*channel_number*[,*queue_size*]

example:
i. **IP_LISTEN#4**
ii. **IP_LISTEN#ch,7**

IP_BIND sockets

IP_BIND is used to associate a local IP address and Port with a socket .

This is required on an unconnected TCP socket before subsequent use of the **IP_LISTEN** command. It is normally used to bind to either connection-oriented (stream, TCP) or connectionless (datagram UDP) sockets. **IP_BIND** may also be used to bind to a raw socket (SCK_).

IP_BIND may also be used on an unconnected socket before subsequent calls to the **IP_CONNECT** command 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.

The optional *family* sets the address family, The default being 2 for Internet.

syntax: *channel_number* := *numeric_expression*
port := *numeric_expression*
IP_address := *string_expression* [in IPv4 numbers-and-dots notation]
family := *numeric_expression*

IP_BIND#*channel_number*,*port*,*IP_address*[,*family*]
IP_BIND#*channel_number*,*sockAddr*

example:
i. **IP_BIND#4,5800,"192.168.0.5"**
ii. **IP_BIND#4,sa$**
iii. **IP_BIND#ch,port,IPAdd$**
iv. **IP_BIND#4,5800,"192.168.0.5",2**

IP_CONNECT sockets

IP_CONNECT is used to attempt to connect to another socket .

If the socket is of type UDP_ (Datagram), this command 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. That is waiting to accept a connection.

Generally, TCP stream sockets may successfully connect only once; UDP datagram sockets may use **IP_CONNECT** multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address.(0.0.0.0)

If **IP_CONNECT** finds no one is listening for a connection on the specified IP address and Port, then **IP_CONNECT** will return with a QDOS 'Transmission error'.

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

The optional *family* sets the address family, The default being 2 for Internet.

syntax:
channel_number := *numeric_expression*
port := *numeric_expression*
IP_address := *string_expression* [in Ipv4 numbers-and-dots notation]
family := *numeric_expression*

IP_CONNECT#*channel_number*,*port*,*IP_address*[,*family*]
IP_CONNECT#*channel_number*,*sockAddr*

example:
i. **IP_ CONNECT #4,5800,"192.168.0.5"**
ii. **IP_ CONNECT #4,sa$**
iii. **IP_ CONNECT #ch,port,IPAdd$**
iv. **IP_ CONNECT #4,5800,"192.168.0.5",2**

Note – In QPC2 with UDP channels. If **OPEN** is used then **IP_CONNECT**, you do not receive any error, but trying to use **IP_SEND** will fail with an 'End of File' error. However if you use **OPEN_IN** instead, then **IP_SEND** will work.

IP_ACCEPT sockets

The function **IP_ACCEPT** is used to accept TCP(stream) connection requests from the specified channel number. And will return a new S*Basic channel number when a connection is accepted.

The new S*Basic channel number should then be used for all further commands relating to this connection.

IP_ACCEPT is used in the Server side of Client/Server connections.

The channel number argument is a socket that has been previously created with **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 listening queue, of pending connections, then creates a new S*Basic channel with the same properties as the supplied channel number, and allocates a new channel number for the new socket.

IP_ACCEPT returns the error ‘Not Complete’ if there are no pending connection requests and can't complete immediately. (See note 1 below)

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 without error, (a non negative number) then a remote connection has been has been accepted, and the returned value will be the channel number of the new connection. That is the next free, S*Basic channel number starting from #3. (See note 2 below)

The new channel number may not be used to accept more connections. However the supplied channel number argument remains open, and can be used to accept further connection requests.

syntax: *channel_number* := *numeric_expression*

IP_ACCEPT(#*channel_number***)**

example: i. **ch=IP_ ACCEPT(#4)**
ii. **ch=IP_ ACCEPT(#channel)**

The following program sample will open a TCP socket, then wait for a connection.
When a successful connection is established, the variable ch will be the S*Basic channel number of a newly created TCP channel.

```
100 OPEN#8,"TCP_"
110 IP_BIND#8,5800,"172.16.0.6"
120 IP_LISTEN#8
130 REPeat loop
140   ch=IP_ACCEPT(#8)
150   IF ch>0 THEN EXIT loop
160   IF ch<>-1 THEN
170     PRINT "Error during ACCEPT - ";
180     STOP
190   END IF
200 END REPeat loop
```

Note 1 – Qemulator does not return immediately with ‘Not Complete’. It waits until a connection request arrives. This means that SuperBASIC will stop when it executes an **IP_ACCEPT** command. And you will not be able to **BREAK** into the program. Also Qemulator tends to ‘hang’ with a ‘Not Responding’ error. The only way to regain control, is to send a connection request so that **IP_ACCEPT** returns, or close Qemulator completely.

Note 2 – IP_ACCEPT will not extend the S*Basic channel table, and will return with an ‘Out of Memory’ error if there is no more room in the table. To avoid this problem ensure there are some unused channels. In the above example **IP_ACCEPT** will be able to choose from between #3 to #7, and #9 to the end of the available channel table.

IP_FCNTL **sockets**

IP_FCTNL is used to perform operations on the open IP channel.

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.

The value argument is the bitwise OR of zero, or more or the following commands.

4 O_NONBLOCK Set the socket to be non-blocking.

64 O_ASYNC 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.

syntax: *channel_number* := *numeric_expression*
value := *numeric_expression*

IP_FCNTL#*channel_number*,*value*

example: i. **IP_ FCNTL#4,68** {set socket to be non-blocking and asynchronous}
ii. **IP_ FCNTL#4,0** {reset socket}

Note 1 –In QPC2 this function gives a 'Not Implemented' error. I have included the function in case it is implemented in other emulators. However I have not been able to test the function, So I don't know if it will work.

Note 2 – To quote Richard Zidlicky "An awful hack for now don't use it unless you have to."

IP_SHUTDWN **sockets**

IP_SHUTDWN is used to shut down all or part of a full-duplex connection on the socket associated with channel number.

The supplied argument determines which receptions, or transmissions will be disallowed.

0	Disable receive
1	Disable send
2	Disable send and receive

syntax: *channel_number* := *numeric_expression*
how := *numeric_expression*

IP_SHUTDWN#*channel_number*,*how*

example: **IP_ SHUTDWN#4,1** {shut down send}

Note – In QPC2, returns QDOS error 'End of file'. IP error 57, 'Invalid slot'

I have included the function in case it is implemented in other emulators. However I have not been able to test the function, So I don't know if it will work.

IP_SEND **sockets**

The function **IP_SEND** is used to send a message, or an area of memory to another socket.

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

IP_SEND differs from **PRINT** in that it is message oriented and allows sending of packets longer than 32k.

It returns the length sent of the message on successful completion. The message is found in the buffer at the start address and has a size of length.

If the message is too long to pass atomically through the underlying protocol, the IP 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 QDOS error return.

The optional flag 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.

The default value of flag is 0 (none).

syntax: *channel_number* := *numeric_expression*
start_address := *numeric_expression*
length := *numeric_expression*
flag := *numeric_expression*

IP_SEND(#*channel_number*,*start_address*,*length*[,*flag*]**)**

example: i. **sent = IP_SEND(#4,start,length)**
ii. **sent = IP_SEND(#ch,start,length,4)** {flag MSG_DONOTROUTE}

IP_SENDTO **sockets**

The function **IP_SENDTO** is used to send a message, or an area of memory to another socket.

IP_SENDTO is used to transmit a message to an unconnected Datagram (UDP) socket.

If **IP_SENDTO** is used on a connection-mode (TCP stream) socket, the sockAddr string is ignored, and IP errors may occur.

The target for the **IP_SENDTO** function is defined in the sockAddr string (which must be supplied), and which must be 16 bytes long.

When the message does not fit into the send buffer of the socket, **IP_SENDTO** normally blocks, unless the socket has been placed in non-blocking I/O mode. In non-blocking mode it would fail with the IP error EAGAIN or EWOULDBLOCK in this case.

It returns the length sent of the message on successful completion. The message is found in the buffer at the start address and has a size of length.

If the message is too long to pass atomically through the underlying protocol, the IP 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 QDOS error return.

The optional flag 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.

The default value of flag is 0 (none).

syntax: *channel_number* := *numeric_expression*
start_address := *numeric_expression*
length := *numeric_expression*
socket_address := *string_expressione*
flag := *numeric_expression*

IP_SENDTO(#*channel_number*,*start_address*,*length,socket_addresss*[,*flag*]**)**

example: i. **sent = IP_SENDTO(#4,start,length,sa$)**
ii. **sent = IP_SENDTO(#ch,start,length,sa$,4)** {flag MSG_DONOTROUTE}

Note – I have never been able to get **IP_SENDTO** to work in QPC2. It returns the QDOS error ‘Bad parameter’, IP error 14 ‘Bad Address’.

I have included the function in case it is implemented in other emulators. However I have not been able to test the function, So I don’t know if it will work.

IP_RECV **sockets**

The function **IP_RECV** is used to receive messages from a socket.

It is used to receive data on both connectionless (UDP) and connection-oriented (TCP) sockets.

IP_RECV differs from **INPUT** in that it is message oriented and allows receving of packets longer than 32k.

It 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, **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 Request 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.

64 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.

The default value of flag is 0 (none).

syntax: *channel_number* := *numeric_expression*
start_address := *numeric_expression*
buffer_size := *numeric_expression*
flag := *numeric_expression*

IP_RECV(#*channel_number*,*start_address*,*buffer_size*[,*flag*]**)**

example: i. **got = IP_RECV(#4,start,length)**
ii. **got = IP_RECV(#ch,start,length,2)** {flag MSG_PEEK}

IP_RECVFM **sockets**

The function **IP_RECVFM** is used to receive messages from a socket.

It is used to receive data on both connectionless (UDP) and connection-oriented (TCP) sockets.

On a successful completion, the sender details are placed in the sockAddr string (which must be supplied), and which must be 16 bytes long.

The function 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, **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 Request 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.

64 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.

The default value of flag is 0 (none).

syntax: *channel_number* := *numeric_expression*
start_address := *numeric_expression*
buffer_size := *numeric_expression*
socket_address := *string_variable*
flag := *numeric_expression*

IP_RECVFM(#*channel_number*,*start_address*,*buffer_size*,*socket_address*[,*flag*]**)**

example: i. **got = IP_RECVFM(#4,start,length,sa$)**
ii. **got = IP_RECVFM(#ch,start,length,sa$,2)** {flag MSG_PEEK}

Note 1 – *socket_address* must be a string variable.

Note 2 – I have never been able to get **IP_RECVFM** to work in QPC2. It returns the QDOS error 'Bad parameter', IP error 14 'Bad Address'.

I have included the function in case it is implemented in other emulators. However I have not been able to test the function, So I don't know if it will work.

IP_GETHOSTNAME$ sockets

The function **IP_GETHOSTNAME$** will return a string containing the name of the host computer that your program is running on.

The name can then be used by **IP_GETHOSTBYNAME** to determine the IP address of your local machine.

syntax: **IP_GETHOSTNAME$**

example: i. **PRINT = IP_GETHOSTNAME$** {prints something like ‘p4-main-system’}
ii. **name$ = IP_GETHOSTNAME$**

Note If this command is used in a daughter, sBasic job in QPC2. It will fail with a ‘Bad parameter’ error.

IP_GETSOCKNAME$ sockets

The function **IP_GETSOCKNAME$** will return a Socket Address structure as a string.

This structure will contain the current IP address and port to which the sockets channel number is bound to. The optional length argument should be set to indicate the size of the Socket Address to return. The default value being 16 characters.

The returned Socket Address structure is truncated if the supplied length argument is too small.

syntax: *channel_number* := *numeric_expression*
length := *numeric_expression*

IP_GETSOCKNAME$(#*channel_number*[,*length*]**)**

example: **sa$ = IP_GETSOCKNAME$(#4)**

Note In Qemulator this function gives a ‘Not Implemented’ error.

IP_GETPEERNAME$ sockets

The function **IP_GETPEERNAME$** will return a Socket Address structure as a string.

This structure will contain the current IP address and port to which the sockets channel number is connected to (the peer). The optional length argument should be set to indicate the size of the Socket Address to return. The default value being 16 characters.

The returned Socket Address structure is truncated if the supplied length argument is too small.

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.

syntax: *channel_number* := *numeric_expression*
length := *numeric_expression*

IP_GETPEERNAME$(#*channel_number*[,*length*]**)**

example: **sa$ = IP_GETPEERNAME$(#4)**

Note If no Peer exists, **IP_GETPEERNAME$** returns the QDOS error ‘End of file’.
In Qemulator this function gives a ‘Not Implemented’ error.

IP_GETHOSTBYNAME$ **sockets**

The function **IP_GETHOSTBYNAME$** will return a Host Entry structure as a string, for the supplied 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 the host 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 Addrlist 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.

IP_GETHOSTBYNAME$ takes a string like "www.yahoo.com", and returns a hostent string which contains 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.)

syntax: *host_name* := *string_expression*
IP_address := *string_expression*

IP_GETHOSTBYNAME$(*host_name***)**
IP_GETHOSTBYNAME$(*IP_address***)**

example: i. **hostEnt$ = IP_GETHOSTBYNAME$(“www.yahoo.com”)**
ii. **hostEnt$ = IP_GETHOSTBYNAME$(“134.16.0.15”)**
iii. **hostEnt$ = IP_GETHOSTBYNAME$(“Tower-System”)**

Note This function may crash Qemulator.

IP_GETHOSTBYADDR$ **sockets**

The function **IP_GETHOSTBYADDR$** will return a Host Entry structure as a string, for the supplied Ipv4 Address in Network byte order.

The optional type argument should be set to indicate the address type. The default being 2, for Internet.

IP_GETHOSTBYADDR$** and **IP_GETHOSTBYNAME$ map back and forth between host names and IP addresses.

syntax: *IP_address* := *numeric_expression*
type := *numeric_expression*

IP_GETHOSTBYADDR$(*IP_address*[,*type*]**)**

example: i. **hostEnt$ = IP_GETHOSTBYNAME$($C0A80005)** {192.168.0.5 in Network byte order}
ii. **hostEnt$ = IP_GETHOSTBYNAME$(ip,2)**

Note In Qemulator this function gives a ‘Not Implemented’ error.

IP_GETNETBYNAME$ **sockets**

The function **IP_GETNETBYNAME$** will return a Net Entry structure from the database as a string, for the supplied network name.

syntax: *network_name* := *string_expression*

IP_GETNETBYNAME$(*network_name***)**

example: **netEnt$ = IP_GETNETBYNAME$("loopback")**

NOTE: In QPC2 and Qemulator, this function gives a 'Not Implemented' error. I have included the function in case it is implemented in other emulators. However I have not been able to test the function, So I don't know if it will work.

IP_GETNETBYADDR$ **sockets**

The function **IP_GETNETBYADDR$** will return a Net Entry structure from the database as a string, for the supplied network number in Network byte order.

The optional type argument should be set to indicate the address type. The default being 2, for Internet.

syntax: *IP_address* := *numeric_expression*
type := *numeric_expression*

IP_GETNETBYADDR$(*IP_address*[,*type*]**)**

example: i. **netEnt$ = IP_GETNETBYADDR$($C0A80000)** {192.168.0.x in Network byte order}
ii. **netEnt$ = IP_GETNETBYADDR$(ip,2)**

NOTE: In QPC2 and Qemulator, this function gives a 'Not Implemented' error

IP_GETPROTOBYNAME$ **sockets**

The function **IP_GETPROTOBYNAME$** will return a Protocol Entry structure from the database as a string, for the supplied protocol name.

The Protocol Entry string will contain information (including it's protocol number) on the supplied protocol name.

A connection is opened to the database if necessary.

syntax: *protocol_name* := *string_expression*

IP_GETPROTOBYNAME$(*protocol_name***)**

example: **netEnt$ = IP_GETPROTOBYNAME$("tcp")** {tcp is protocol number 6}

Note In Qemulator this function gives a 'Not Implemented' error.

IP_GETPROTOBYNUMBER$ sockets

The function **IP_GETPROTOBYNUMBER$** will return a Protocol Entry structure from the database as a string, for the supplied protocol number.

The Protocol Entry string will contain information (including it's protocol name) on the supplied protocol number.

A connection is opened to the database if necessary.

syntax: *protocol_number* := *numeric_expression*

IP_GETPROTOBYNUMBER$(*protocol_number***)**

example: **netEnt$ = IP_GETPROTOBYNUMBER$(6)** {tcp is protocol number 6}

Note In Qemulator this function gives a 'Not Implemented' error.

IP_GETSERVBYNAME$ sockets

The function **IP_GETSERVBYNAME$** will return a Server Entry structure from the database as a string, for the supplied protocol name.

The Server Entry string will contain information (including it's port number and protocol) on the supplied server name.

A connection is opened to the database if necessary.

syntax: *server_name* := *string_expression*
protocol := *string_expression*

IP_GETSERVBYNAME$(*server_name*[,*protocol*]**)**

example:
i. **servEnt$ = IP_GETSERVBYNAME$("pop3")** {pop3 is the server name for port 110, with TCP protocol}
ii. **servEnt$ = IP_GETSERVBYNAME$("http","tcp")**

Note In QPC2 this function gives a 'Bad Parameter' error. And in Qemulator a 'Error in Expression' error.

IP_GETSERVBYPORT$ sockets

The function **IP_GETSERVBYPORT$** will return a Server Entry structure from the database as a string, for the supplied port number.

The Server Entry string will contain information (including it's port number and protocol) on the supplied server name.

A connection is opened to the database if necessary.

syntax: *port_number* := *numeric_expression*
protocol := *string_expression*

IP_GETSERVBYPORT$(*protocol_number*[,*protocol*]**)**

example:
i. **servEnt$ = IP_GETSERVBYPORT$(110)** {pop3 is the server name for port 110, with TCP protocol}
ii. servEnt$ = IP_GETSERVBYPORT$(80,"tcp")

Note In QPC2 this function gives a 'Bad Parameter' error. And in Qemulator a 'Not Implemented' error.

IP_INET_ATON **internet**

The function **IP_INET_ATON** will convert an IPv4, IP Address string in dots and numbers format into a number in network byte order.

The IP Address supplied in 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).

syntax: *IP_address* := *string_expression*

IP_INET_ATON(*IP_address***)**

example:
i. **address = IP_INET_ATON("192.168.0.5")**
ii. **address = IP_INET_ATON("192.168.5")**
iii. **address = IP_INET_ATON("192.11010053")**
iv. **address = IP_INET_ATON("3232235525")**

comment: The above four examples are all the same, showing the four formats.

Note In Qemulator this function gives a 'Not Implemented' error.

IP_INET_NETWORK **internet**

The function **IP_INET_NETWORK** will convert an IPv4, IP Address string in dots and numbers format into a number in network byte order.

syntax: *IP_address* := *string_expression*

IP_INET_NETWORK(*IP_address***)**

example: **address = IP_INET_NETWORK("192.168.0.5")**

Note In Qemulator this function may return an incorrect address.

IP_INET_NTOA$ **internet**

The function **IP_INET_NTOA$** will convert an IP Address in network byte order, to a string in dots and numbers format.

syntax: *IP_address* := *numeric_expression*

IP_INET_NTOA$(*IP_address***)**

example: **address$ = IP_INET_NTOA$($C0A80005)**

Note In Qemulator this function gives a ‘Not Implemented’ error.

IP_INET_MAKEADDR **internet**

The function **IP_INETMAKEADDR** will return 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

The **IP_INET_MAKEADDR** function is the converse of **IP_INET_NETOF** and **IP_INET_LNAOF**.

This is a 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.

syntax: *network_number* := *numeric_expression*
host_number := *numeric_expression*

IP_INET_MAKEADDR(*network_number*,*host_number***)**

example: **address = IP_INET_MAKEADDR($C0A80000,$0000000C)**

comment: $C0A80000 is equivalent to the 192.168.0 and $0000000C is equivalent to the 12

Note In QPC2 and Qemulator, this function gives a ‘Not Implemented’ error.

IP_INET_LNAOF **internet**

The function **IP_INET_LNAOF** will return the host address part of the Internet address supplied in network byte order.

This is a 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.

syntax: *IP_address* := *numeric_expression*

IP_INET_LNAOF(*IP_address***)**

example: **address = IP_INET_LNAOF($C0A8000C)** {will return 12 ($C)}

Note In QPC2 and Qemulator, this function gives a ‘Not Implemented’ error.

IP_INET_NETOF **internet**

The function **IP_INET_NETOF** will return the network number part of the internet address supplied in network byte order.

This is a 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.

syntax: *IP_address* := *numeric_expression*

IP_INET_NETOF(*IP_address***)**

example: **address = IP_INET_NETOF($C0A8000C)** {will return 3232235520 ($C0A80000)}

Note In QPC2 and Qemulator, this function gives a ‘Not Implemented’ error.

IP_GETDOMAIN$ hosts

The function **IP_GETDOMAIN$** will return as a string, the domain name of the host system.

If the domain name is longer than 64 bytes, it will be truncated to 64 bytes.

syntax: **IP_GETDOMAIN$**

example: **domain$ = IP_GETDOMAIN$**

Note In QPC2 this function returns the string ‘unsupported’. And in Qemulator a ‘Not Implemented’ error.

IP_H_STRERROR$ hosts

The function **IP_H_STRERROR$** will return a string that describes the error code passed in the supplied error number argument. (For example, if the IP error number is 22 (EINVAL), the returned description will be "Invalid argument".)

If the error description is longer than 64 bytes, it will be truncated to 64 bytes.

syntax: *error_number* := *numeric_expression*

IP_H_STRERROR$(*error_number***)**

example: i. **errnName$ = IP_H_STRERROR$(22)** {22 is IP error Invalid Argument}
ii. **errnName$ = IP_H_STRERROR$(IP_ERRNO(#ch))**

Note In QPC2 I have only ever seen this function return the string ‘Unknown error’.
In Qemulator this function gives a ‘Not Implemented’ error. I have included the function in case it is implemented correctly in other emulators. See **IPERROR$**

IPERROR$ hosts

The function **IPERROR$** will return a string that describes the error code passed in the supplied error number argument. Using the list of Linux error codes in the Error code section.

This function is intended as a replacement for **IP_H_STRERROR$** when it does not work.

syntax: *error_number* := *numeric_expression*

IPERROR$(*error_number***)**

example: i. **errnName$ = IPERROR$(22)** {22 is IP error Invalid Argument}
ii. **errnName$ = IPERROR$(IP_ERRNO(#ch))**

IP_ERRNO **hosts**

The function **IP_ERRNO** will return the last IP error number to occur. (not a QDOS error number)

The optional channel number should be the channel number that was used by the IPBasic command which failed.

syntax: *channel_number* := *numeric_expression*

IP_ERRNO([**#***channel_number*]**)**

example: i. **errno = IP_ERRNO(#4)**
ii. **errno = IP_ERRNO**

comment: When one of the **IP_** commands, which do not require a channel number, encounters an error. The error is stored, and **IP_ERRNO** picks that up when a channel number is not supplied.

SA_MAKE$ **data structures**

The function **SA_MAKE$** will return a 16 byte socket address string.

The optional family argument will default to 2 for Internet.

syntax: *port* := *numeric_expression*
IP_address := *numeric_expression*
:= *string_expression* [in Ipv4 numbers-and-dots notation]
family := *numeric_expression*

SA_MAKE$(port,*IP*_address[,family]**)**

example: i. **sa$ = SA_MAKE(5800,$C0A80005)** {192.168.0.5 in network byte order}
ii. **sa$ = SA_MAKE(5800,"192.168.0.5")**
iii. **sa$ = SA_MAKE(5800,"192.168.0.5",2)**

SA_PORT **data structures**

The function **SA_PORT** will return the port number from the supplied 16 byte socket address string.

syntax: *sockAddr* := *string_expression*

SA_PORT(*sockAddr***)**

example: **port = SA_PORT(sa$)**

SA_FAMILY **data structures**

The function **SA_FAMILY** will return the family number from the supplied 16 byte socket address string.

syntax: *sockAddr* := *string_expression*

SA_FAMILY(*sockAddr***)**

example: **family = SA_FAMILY(sa$)**

SA_IPADDR **data structures**

The function **SA_IPADDR** will return the IP address in network byte order, from the supplied 16 byte socket address string.

syntax: *sockAddr* := *string_expression*

SA_IPADDR(*sockAddr***)**

example: **family = SA_IPADDR(sa$)**

Data Structures

The IP device driver is implemented in the ‘C’ programming language, and so uses some of those data structures. IPBasic converts these memory based structures into equivalent strings.

Sockaddr – Socket Address – 16 byte string

Index	Size	Description
1	Word	Family (usually 2)
3	Word	Port number
5	Long	IP address
9	Long	Zero
13	Long	Zero

Hostent – Host Entry

Index	Size	Name	Description
1	Long	Name	Pointer to Addrlist index
5	Long	Aliases	Pointer to a list of Long IP addresses terminated with a Null Long word
9	Long	Addtype	Connection type (usually 2 (AF_INET) internet)
13	Long	Length	Number of nodes in IP address (usually 4 (IPV4))
17	Long	Addrlist	Pointer to a list of pointers terminated with a Null Long word. Each of these pointers point to a list of Long word IP ddresses, terminated with a Null Long word

Servent – Service entry

Index	Size	Name	Description
1	Long	Name	Pointer to a Null terminated string of the official service name.
5	Long	Aliases	Pointer to a list of strings terminated with a Null byte. And the list is terminated with a long Null.
9	Long	Port	Associated port number.
13	Long	Proto	Pointer to a Null terminated string

Netent – Network entry

Index	Size	Name	Description
1	Long	Name	Pointer to a Null terminated string of the official network name.
5	Long	Aliases	Pointer to a list of strings terminated with a Null byte. And the list is terminated with a long Null.
9	Long	Addtype	Network address type
13	Long	Net	Network number

Protoent – Protocol entry

Index	Size	Name	Description
1	Long	Name	Pointer to a Null terminated string of the protocol name.
5	Long	Aliases	Pointer to a list of strings terminated with a Null byte. And the list is terminated with a long Null.
9	Long	Ports	Protocol number.

Error Codes

This is a list of the error codes that are returned by **IPERROR$**

Err no	Linux 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
38	ENOSYS	Function not implemented
39	ENOTEMPTY	Directory not empty
40	ELOOP	Too many symbolic links encountered
41	EWOULDBLOCK	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
58	EDEADLOCK	Deadlock

Err no	Linux Error name	Description
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
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

Err no	Linux Error name	Description
121	EREMOTEIO	Remote I/O error
122	EDQUOT	Quota exceeded
123	ENOMEDIUM	No medium found
124	EMEDIUMTYPE	Wrong medium type
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
130	EOWNERDEAD	Owner died
131	ENOTRECOVERABLE	State not recoverable