A simple UDP driver for the Q68

Before you do anything else, please configure the driver with MenuConfig. Once this is done, please physically connect the machines before loading and initializing the driver. Once the driver is loaded, you may supply it with certain information. You should read this entire manual. If you are really impatient and know what UDP ports, ARP, DHCP etc are, and what binding and connecting etc mean, you can have a look at the quick start guide.

Table of Contents

1 Configuring the driver

The driver needs to be configured using the standard MenuConfig program. You should configure:

1.1 Initialize driver automatically

Once loaded and initialized, the driver uses up quite some memory (for buffering etc.). You might not want this to happen when you do not use the driver, yet not have to load it separately when you do need it.

Configuring the driver not to initialize immediately means that the driver itself is loaded into memory but, apart from that, it uses no memory (the UDP device isn't even there). You just get the keyword "`STK_INIT`". Once you use that, the driver will be fully initialized. Please see below for more details on this keyword.

1.2 Duplex mode

The Ethernet chip in the Q68 can operate in full duplex or half duplex mode (hint: full is better than half). It can also try to find out what mode the peer supports, and thus "auto-negotiate" the duplex mode.

When auto-negotiating, the two connected machines try to find out how best to communicate. This is the best choice if you don't know what speed the peer is capable of, in which case, you should leave this option at auto-negotiation. However, auto-negotiation takes time (a few seconds), during which the machine will seem to hang. With modern equipment, using duplex mode is a pretty safe bet, which is why the driver is pre-configured to that mode.

If you connect two Q68s with each other directly (not through a hub or router) with auto-negotiation, please be sure to initialize the drivers within very few seconds (3 or 4) of each other, else auto-negotiation will fail.

1.3 DHCP

Set whether the Q68 should get its IP address from a DHCP server (see below) or have a fixed IP address.

1.4 IP address for the Q68

If you prefer a fixed IP address, here you can set the IP address the Q68 should have. If you do use DHCP, the Q68 indicates to the DHCP server that this is the address it would like to have. There is no guarantee that this is the address the DHCP server will give it, though. Note that if the Q68 is connected to a router (which generally servers as a DHCP server), you will probably have to configure the router to reserve that address for the Q68.

1.5 Flush size and Linefeed

When you send data to a channel opened to a UDP device, the data doesn't necessarily get sent out right away. The UDP device may buffer the data up to a maximum of 1468 bytes (which corresponds roughly to one Ethernet frame *sans* headers). You may however, have the buffer send data out earlier by flushing the buffer when a certain size is reached. Here you set the size in bytes, from 0 (no buffering) to 1468.

Note that not all data is buffered - for more information see below.

You can also set whether, when a single LF is sent, the data in the buffer should be flushed immediately. You will probably want to leave this at "yes" - for more information see below.

1.6 Network name

This is the name (known as "host name") that the Q68 will indicate to the router (DHCP server) as its own name on the network when using DHCP. Some routers do use that name when so requested, some don't.

1.7 Gateway IP

If you intend to use the driver for connections not only with a direct connection or on your LAN, but also via the internet, then you must, at least, set the gateway IP (and the netmask, see below).

1.8 Gateway MAC

If you intend to use the driver for connections not only with a direct connection or on your LAN, but also via the internet, and in the (probably very unlikely) case that your gateway doesn't reply to ARP requests for its MAC, then you must set the gateway's MAC here, as a string of type "aa-bb-cc-dd-ee-ff". In most cases this will not be necessary – I don't know of any gateway that wouldn't reply to ARP requests (for an explanation of what ARP requests are, see below).

1.9 Subnet mask

If you intend to use the driver for connections not only with a direct connection or on your LAN, but also via the internet, you must set the "netmask" (as an IP address) here. In most cases this will be "255.255.255.0".

NOTE : If you only communicate via LAN with machines on the same subnet, you may leave the gateway IP and netmask at 0.

2 Loading and initializing the driver.

The driver is loaded with the normal LRESPR command. **This must be done from job 0** (else you get an error not implemented).

If the driver is configured not to initialize itself automatically, LRESPRing the driver only loads it into memory, but does not initialize it yet in any way, except for providing the STK_INIT command.

The command **STK_INIT** will initialize the driver. ***This must also be used from the main basic (job 0)*** – trying to use it from any other basic will result in an error "not implemented"**.**

Syntax: **STK_INIT** *[type]*

The optional type parameter can be used to select the duplex type and whether DHCP should be used. If this parameter is present, the configured values will be ignored.

- 0 = use auto-negotiation for duplex
- 1 = use full duplex
- 2 = use half duplex.
- Any of the above +4 = use DHCP

If this parameter is absent, the configured values will be used.

During initialization, a certain number of new Basic keywords are added to the system (for a non-exhaustive list, see below).

Note: remember that any new keywords added to the system will not appear in a daughter SBasic job until a NEW or LOAD command is issued there.

3 Setting the necessary information

To communicate over UDP, the system needs to know its own IP address and have access to several other pieces of information: a port for incoming data and, for outgoing data, a port to send out the data as well as the peer's port, IP address and MAC address.

Note: the *"peer"* is always the other computer or device to which the one you are working with is connected.

3.1 Setting the IP address of the Q68

Before doing anything else, you now need to set the IP address which the Q68 should use for itself. This can be done either manually, or automatically with "DHCP".

3.1.1 Setting the IP address manually

This can bee done by configuring the driver with the normal MenuConfig, as mentioned above. Just fill in the IP address you want the set for the Q68 and also set "Use DHCP to get IP" to "no". Alternatively, you can use the SET_MY_IP command.

3.1.2 Setting the IP address automatically with DHCP

If the Q68 is connected to a router (to fit into your LAN) or any other system that acts as DHCP server, then you may also obtain an IP via DHCP. To do this, configure the driver to "Use DHCP to get IP" (yes), or use the `STK_INIT` command with the optional *type* parameter.

When the driver is started, this will automatically start a background job called "DHCP daemon". This job will contact the DHCP server and get an IP address. Once this is done the DHCP daemon will be terminated.

You can then find out some information, including the IP address assigned to the Q68 by the DHCP server, with the **`DHCP_INFO$`** function:

`PRINT DHCP_INFO$` prints out the relevant information.

At any time you may use the `GET_MY_IP$` function to get the IP of your Q68:

"`PRINT GET_MY_IP$`" will print, for example, "172.16.0.1" if that is the address of your Q68.

3.2 The peer IP address

For outgoing data, you need to set the IP address of the peer, this is done via the OPEN_IN command, see below.

3.3 Handling MAC addresses

When data is to be sent to a peer over Ethernet, the system needs to know how to reach that peer. For this, the system needs to know not only the peer's IP address, but also its *media access control* (MAC) address (there is supposed to be one unique MAC address for every device worldwide that has a network interface controller…). MAC addresses are 6 bytes long and are generally displayed as hexadecimal bytes separated by hyphens or colons:

11-22-33-44-55-66 or 11:22:33:44:55:66

You may either supply the system manually with the MAC address for the peer to which you wish to connect, or ask the system to try to find it automatically via an "ARP" request. Note that you need the MAC address of each peer to which you wish to connect the Q68.

3.3.1 Manual MAC resolution

To supply a MAC address to the system directly, use the "**ARP_SET**" command:

ARP_SET *ip_address$, mac_address$*

Where ip_address$ is the IP address in the form of "aaa.bbb.ccc.ddd", e.g. "172.16.0.1". The *mac_address$* parameter is given in the way MAC addresses are customarily given, as set out above. These parameters need to be within quotes (if they are not variables). The hexadecimal letters for the MAC address may be given in upper or lower case.

So, for example

```
ARP_SET "172.16.0.1","AF-de-12-B8-98-cF"
```

would set the MAC address of the peer reachable at IP address 172.16.0.1 to AF-DE-12-B8-98-CF.

The IP/Mac combination is stored in the ARP cache (see below).

3.3.2 Automatic MAC resolution via ARP

The system is also able to try to find out the MAC address of a peer via the "Address Resolution Protocol" (ARP). To do this, the system will generate an ARP "request" and wait for the ARP "reply" coming from the peer. This of course presumes that the peer is able to handle such an ARP request.

The system will notably generate an ARP request when, for the first time, you open a channel to a peer, and also when you try to send something to the peer over a UDP channel and the peer's MAC is not yet known. During the time taken for the request and reply to be generated and acted upon, the job that does try to send the data itself is suspended (more or less).

You may also use the **ARP_REQ** command to force the system to try to get the MAC address of a peer via the ARP.

Syntax: **ARP_REQ** *ip_address$*

where *ip_address$* is the address of the peer. If the peer replies and communicates its MAC address, that is added to the ARP cache.

3.3.3 The ARP cache

The system maintains a "cache" of IP/MAC combinations. Currently, the cache can hold entries for up to 50 MAC addresses. This is handled transparently by the system and does not require user intervention. For, example, when - after issuing an ARP request- a reply is obtained by the peer, the corresponding IP/MAC combination is stored in the ARP cache. IP/MAC combinations may also be added to the cache via the **ARP_SET** command described above. The cache will always only contain one MAC address for a specific IP.

The cache is not permanent – it is lost when the machine is switched off, and rebuilt whenever necessary. There is no mechanism (yet?) to let the cache entries age, go stale and fall out of the cache, but the **ARP_REMV** function will remove an address from the cache.

Syntax: **ARP_REMV** ip_address$

where *ip_address$* is the IP address be the removed from the cache.

3.3.4 Supplying the MAC of the Q68 to the peer

Just like the Q68 needs to know the MAC address of a peer to which is wished to connect, a peer also needs to know the MAC address of the Q68 if it wishes to speak with the Q68. So, if you need to connect to the Q68 from a peer, you will need to tell the peer the Q68's MAC address, or let the peer try to find out that MAC address itself.

Note that, in most cases, once you have sent something to the peer from the Q68 (which automatically includes sending the Q68 IP and MAC addresses), the peer will note and remember the IP/MAC combination. If you want to initiate communications from the peer's side, it needs to obtain this information beforehand.

3.3.4.1 Manually setting the Q68 MAC address for the peer

The MAC address of the Q68 is stored in its Ethernet controller, the CP2200. You can find out the Q68's Ethernet controller MAC address with the **CTR_MACADDR$** function:

Syntax: res$= **CTR_MACADDR$**

This is a function that returns a formatted string in the format described above, so something like "PRINT CTR_MACADDR$" will print your Q68 MAC address.

Once you have the Q68's MAC address you can tell the peer that address. How you set the MAC address for the Q68 on the peer depends on the operating system of the peer. It generally is some variation of an "ARP" command.

Under Linux this is generally something like “arp -s IP_address MAC_address”, such as
`arp -s 178.89.89.85 0A0203040506`
(note there is no separator between the MAC address hex digits).

Under windows it is generally something like “arp -s IP_address MAC_address”, such as
arp -s 178.89.87.85 0A-02-03-04-05-06

3.3.4.2 Letting ARP handle the Q68 MAC address

If the peer is itself aware of the ARP mechanism (most modern computers are), there is no need to set the Q68's MAC address manually. When trying to connect to a Q68 for which it doesn't have the MAC address, the peer will send out an ARP request. The Q68 will receive and reply to such a request (this is part of what the ARP daemon does, see below), so there is no need to set the MAC address manually.

3.3.5 The ARP daemon

This is a small background program that will reply to ARP requests sent to the Q68 to inquire its MAC address. It is also responsible to emit ARP requests itself for an as yet unknown MAC address of a peer. You should not remove the ARP daemon job.

4 The channel buffer

4.1 A simple buffer

The device implements a simple buffer so that, when you send data to a channel opened to it, the data doesn't necessarily get sent out over the wire right away. This is to avoid having to send out packets containing only one single character, which isn't really very efficient. The device may buffer up to a maximum of 1468 bytes (which corresponds roughly to one Ethernet frame *sans* headers).

4.2 Data that is not buffered

Not all data is buffered: the device only buffers data sent to it via the *iob.sbyt* trap, where only a single byte is sent to the channel. This means that:

- data sent through the iob.smul trap (where multiple bytes are sent);
- data sent via the IP specific commands (sendto, send);

does not get buffered, but is sent out immediately. Of course, any data that was buffered until that time will get sent out first (in a separate packet).

4.3 Flushing the buffer

As mentioned in the configuration section you may however set a flush size – once this is reached, the data in the buffer is flushed.

You can also cause the buffered data of a channel to be flushed out immediately by using the UDP_FLUSH command:

UDP_FLUSH #*channel* will flush out the data in that channel.

4.4 Beware of the PRINT command

There is one possible pitfall with the PRINT command: When you PRINT something from S-Basic, there may actually be two I/O operations involved. Thus, a **`PRINT "Q68"`** will actually first print the string "Q68" via the iob.smul trap and then print the LF (chr$(10)) via the iob.sbyt trap. This means that the "Q68" string is sent immediately (no buffering), whereas the LF character is buffered. And that would be a bit of a bother if, at the other end, the peer is waiting for data via the INPUT command, which waits until it gets the LF…

There are three ways to cope with that:

1. Configure the driver so that single LFs are always flushed.
2. Append the LF at the end of the PRINT command:
   PRINT "Q68"&chr$(10);
   Note the ";" at the end, which avoids sending a second LF character.
3. Use the UDP_FLUSH command after the PRINT.

5 Using the driver

Much like a serial device, the driver lets you connect to another computer and send it data via UDP - but like the QL network, data is sent in “packets”. So you have one computer waiting for input and the other sending something. Traditionally and though I find the lingo incorrect, the computer that is waiting for input is called the “server” and the one that is going to send data is the “client”.

5.1 Using the Q68 as a server

To let the Q68 be a server, you need to tell it to open a channel on which it will expect input. This is done with the “normal” OPEN_NEW command (or the TK2 equivalent FOP_NEW function).
OPEN_NEW#chan% ; ”UDP_yourIP:port”

The ‘*yourIP*’ part of the name is **the IP of your Q68, NOT that of the peer.** Exceptionally, It may also be the “loopback” address (see **below**).

This is then separated by a colon from the *port*.

The *port* is simply an imaginary endpoint on which the Q68 will expect its input – it doesn’t exist physically. Ports are used to distinguish different communication channels.

There are 65536 possible ports (0 – 65535), you may choose any one you like. However, many ports are, conventionally, reserved for certain services. For example, the DHCP service uses ports 67 and 68. It is generally recommended to use ports as of 50000. The important thing to remember about ports is simply that for computers to talk to each other, they must agree about the port numbers.

Due to the presence of decimal points and colons, the name needs to be within quotes.

Example : `OPEN_NEW#3, ”UDP_172.16.0.1:50000”`. This will expect input on port 50000.
Instead of typing the IP address of your Q68, you could also use the `GET_MY_IP$` function explained above:

```
OPEN_NEW#3,”UDP_” & GET_MY_IP$ & ”:50000”
```

5.2 Using the Q68 as a client

If you want to use the Q68 as a client, use the `OPEN_IN` keyword (or the TK2 equivalent FOP_IN function).

```
OPEN_IN #chan% , ”UDP_peerIP’:port”
```

The ‘peerIP’ part of the name is the IP of the peer, not that of your Q68. This is then separated by a colon from the *port.*

The *port* is the port number on which the peer will expect its input. You must choose the one used by the peer for its input.

Example: `OPEN_IN#3 ; ”UDP_172.16.0.2:50002”` (if the peer has IP address “172.16.0.2“ and is listening on port 50002).

5.3 Using the loopback address

One IP address is very special : 127.0.01. This is the “loopback” address : anything sent from this address stays within the computer and is NOT sent out over the Ethernet port. So if you open a channel to a peer with `OPEN_IN` and use the loopback IP address, any packet you sent through the channel will be sent to the IP address 127.0.0.1 on your Q68.

There is no other loopback address, and no other 127.xx.xx.xx addresses are allowed.

Using the loopback address is s-l-o-w - it‘s mainly there to test the driver. Moreover, the loopback address doesn’t handle fragmented packets.

5.4 Establishing a connection

A connection is thus established by opening the relevant channels on the Q68 and the peer.

For example, let’s assume that the Q68 is the client and the peer is the server. The Q68 has IP address 172.16.0.1 and the peer, 172.16.0.2

On the Q68:

```
chan% = FOP_IN( ”UDP_172.16.0.2:50000”)
```

on the peer:

```
chan%= FOP_NEW( ”UDP_172.16.0.2:50000”)
```

You can now send a message to the peer from the Q68:

```
PRINT#chan%,”Hello  from the Q68”&chr$(10);
```

and receive it on the peer:

```
a$ = INPUT(#chan%)
PRINT a$
```

Once a connection is established between the client and the server, **and the client has sent some data to the server**, the connection becomes bilateral: the server may also send data to the client who may receive it.

6 The UDD driver

6.1 Confusing use of OPEN_IN and OPEN_NEW

The UDP driver tries to respect the convention for such drivers as initially set out for the UQLX emulator. I find the choice of opening types extremely confusing: OPEN_IN should be used for the “server” – after all, it is waiting for input sent by the client. OPEN_NEW, or OPEN, should be used by the client…

There is no way that you could use an existing program to use the UDP driver, unless you are be able to reprogram it.

6.2 The UDD driver

The UDD driver was conceived to get around this problem. This is pretty much the same as the UDP driver, except that the channel open keys are the right way round. So suppose your peer at IP address 172.16.0.2 is listening on port 50000, you may write something like this:

`SAVE “UDD_172.16.0.2:50000”` to save the current Basic program.

7 Overview of the supplied new SBasic keywords

Once the driver is initialized, a slew of new keywords becomes available. Most will not be necessary for everyday use – those that are have most likely already been set out in the explanations above.

7.1 ARP related keywords

ARP_REQ

Add an IP to the ARP request queue. The system will issue an ARP request for that IP.
Syntax:
ARP_REQ *ip$*

ARP_GET$

Get a MAC address from the ARP cache.
Syntax:
mac$=**ARP_GET$** (ip$)
where
ip$ = IP as decimal string (aaa.bbb.ccc.ddd), separator must be '.'
mac$ is the MAC address as string AA-BB-CC-DD-EE-FF, or empty string if none.

ARP_SET

Set an entry in the ARP cache (associate IP with mac).
Syntax:
ARP_SET ip$,mac$
where
ip$ is the IP as decimal string (aaa.bbb.ccc.ddd), separator must be '.'
mac$ is the MAC address to set, as string AA-BB-CC-DD-EE-FF, separator is indifferent

ARP_REMV

remove an entry from the ARP cache
Syntax:
ARP_REMV ip$
where
ip$ = IP as decimal string (aaa.bbb.ccc.ddd), separator must be '.'

7.2 Low level keywords

STK_INIT

This initializes the controller, the interrupt routines, the UDP device and general working area.
Syntax:
STK_INIT *[type]*
where the optional *type* parameter may have the following values

- 0 = use auto-negotiation for duplex
- 1 = use full duplex
- 2 = use half duplex.
- Any of the above +4 = use DHCP, else it will be ignored

If this parameter is absent, the configured values will be used.

CTR_WRTPKT

Write a packet directly to the controller and send it.
Syntax:
ret = **CTR_WRTPKT** (package_size,address)
where
package_size = length of packet, should be <= 1470 bytes
address = memory address of packet to the sent
ret is 0 or <0 if error

CTR_RDPKT

Read a packet directly from the controller into memory.
Syntax:
ret = **CTR_RDPKT** (buffer_length,address)
where
buffer length = max length of packet, should be >=1540
address = memory address whither the received packet will be copied
ret <0 if error, else length of packet read

CTR_MACADDR

Put the controller's MAC address at memory address.
Syntax:
ret = **CTR_MACADDR** (dummy,address)
address is a 6 bytes space where the MAC address will be deposited
ret = 0

7.3 Protocol keywords

UDP_MKHDR

Make a UDP packet header
Syntax:
UDP_MKHDR buffer, data_struct, payloadend
where
buffer is the address for the header AND the payload
the payload MUST start at buffer + 8
data_struct is a structure with data about the connection
(see separate document – inexistent as yet)
payloadend points to the end of the payload (1 byte after end)

IP4_MKHDR

Make an IPv4 packet header
Syntax:
IP4_MKHDR buffer, data, payloadend
where
buffer is the address for the header AND the payload
the payload MUST start at buffer + 20
data_struct is a structure with data about the connection
(see separate document – inexistent as yet)
payloadend points to the end of the payload (1 byte after end)

ETH_MKHDR

Make an Ethernet packet header
Syntax:
ETH_MKHDR buffer,data,payloadend
where
buffer is the address for the header AND the payload
the payload MUST start at buffer + 14
data_struct is a structure with data about the connection
(see separate document – inexistent as yet)
payloadend points to the end of the payload (1 byte after end)

ETH_LAST_ERROR$

This as yet imperfectly implemented function returns the last error generated by the driver.
Syntax:
error = **ETH_LAST_ERROR**
where
error is the error returned.

7.4 Getting IPs, MACs, DHCP and server info

CTR_MACADDR$

Get the Q68 ethernet controller's MAC address as string of type “”AA-BB-CC-DD-EE-FF”.
Syntax:
mac$ = **GET_MACADDR$**
where
mac$ is the MAC address as string of type “AA-BB-CC-DD-EE-FF”.

GET_MY_IP$

Get the IP set for the Q68
Syntax:
ip$=**GET_MY_IP$**
where
ip$ is the returned IP as a decimal string (aaa.bbb.ccc.ddd).

DHCP_INFO$

Get information about various IPs, masks, names and the connection state of the Q68.
Syntax:
res$ = **DHCP_INFO$** or PRINT **DHCP_INFO$**
res$ will be formatted string (with line feeds to separate the elements) containing the information.
The information supplied is as follows:

```
IP address : The IP the Q68 currently has
Subnet IP  : The subnet as the Q68 understands it
Netmask    : The netmask used to calculate the subnet
DHCP SRV IP: The IP of the DHCP server
Router IP  : The IP of the router/gateway
DNS SRV IP : The IP of a DNS server
Duplex mode: The current duplex mode, will be “full” or “half”
Domain name: The domain name
Host name  : The Q68 network name
```

GET_GW_IP$

Get the IP set for the gateway.
Syntax:
ip$=**GET_GW_IP$**

GET_GW_MAC$

Get the MAC for the gateway, as a string of type AA-BB-CC-DD-EE-FF. This may still be 0 if no attempt has been made to contact the outside world yet and if no gateway MAC address was configured.
Syntax:
ip$=**GET_GW_MAC$**

GET_NETMASK$

Get the netmask (as an IP) .
Syntax:
ip$=**GET_NETMASK$**

GET_MACADDR$

Get a text representation of a MAC address lying as a 6 bytes number at an address in memory.
Syntax:
mac$ = **GET_MACADDR$** (address)
where
address is the pointer to an address where the MAC lies.
mac$ is the MAC address as string of type “AA-BB-CC-DD-EE-FF”.

7.5 Setting IPs, MACs and server info

CTR_MACADDR

set the Q68 controller's MAC address. **YOU SHOULD NEVER USE THIS COMMAND**!
Syntax:
CTR_MACADDR mac$
where
mac$ Is the address as a string of type “AA-BB-CC-DD-EE-FF”.

SET_MY_IP$

Set the IP for the Q68.
Syntax:
**SET_MY_IP$** ip$
where
ip$ = IP as decimal string (aaa.bbb.ccc.ddd), separator must be '.'.

SET_GW_IP$

Set the IP for the gateway.
Syntax:
**SET_GW_IP$** ip$

where
ip$ = IP as decimal string (aaa.bbb.ccc.ddd), separator must be '.'.

SET_GW_MAC$

set the MAC for the gateway. iI this isn't set, and the Q68 needs to connect with the outside world, it will question the gateway for its mac address via ARP.
Syntax
**SET_GW_MAC$** mac$
where
mac$ Is the address as a string of type "AA-BB-CC-DD-EE-FF".

SET_NETMASK$

Set the netmask (as an IP) .
Syntax:
**SET_NETMASK$** ip$
where
ip$ = IP as decimal string (aaa.bbb.ccc.ddd), separator must be '.'.

HOST_NAME

Tries to set the host name of the Q68. The host name is the name under which the Q68 is known at the network level.
Syntax:
HPST_NAME
where
name$ is a string of max 64 characers.

7.6 Conversions

IP2LONG

Convert an IP string (aaa.bbb.ccc.ddd) to a long.
Syntax:
res = **IP2LONG(**string$)

LONG2IP$

Convert a long to an IP string (aaa.bbb.ccc.ddd).
Syntax:
res$ = IP2LONG(ip_number)

7.7 Misc

PING

“Pings” an IP address and shows the result.
Syntax:
PING ip_address$ [, nbr_of_times]
where
ip_address$ is the IP of the peer to ping
nbr_of_times is the (optional) number of times an attempt to ping is made, default is 10.

PEEK_LW

Little helper function to get a long integer even if it lies at an odd address.
Syntax:
val = **PEEK_LW** (address)

POKE_LW

Little helper function to set a long integer even if it lies at an odd address.
Syntax:
POKE_LW (address, value)

CLR_SPC

Clear a memory range.
Syntax:
CLR_SPC (size,address)
address is start of memory to be cleared
size if length of memory to be cleared. ATTENTION, will be extended to a multiple of 4 (clr.l)

UDP_FLUSH

Flushes the buffer for a UDP channel.
Syntax:
UDP_FLUSH #*channel*

DO_TRAP3

Makes a trap #3 call with the supplied parameters and returns the changed one(s).

This function simply calls the corresponding TRAP#3 on the channel in question. This is an easy way to use any of the IP_xxxx traps. You fill in the registers with the information needed and use the function. On return, the new values (except for d0) will be filled in. The return of the function is the error code of the trap (i.e. register d0).
Syntax:

result = **DO_TRAP3** *(#channel, reg_D0, reg_D1, reg_D2, reg_D3, reg_A1, reg_A2)*
Syntax:

reg_D0 to *reg_A2* are the values these registers should have when entering the trap - they are set (except for D0) to the return values from the trap.

result is the error code from the trap.

(Note use of this keyword isn't restricted to the UDP channel, you can use it on any channel).

8 IP specific trap calls currently supported

RECV
RECVFROM
SEND
SENDTO
INET_NTOA [1]
INET_ATON [1]
IP_CONNECT
IP_GETHOSTNAME
IP_GETPEERNAME
IP_ERRNO [2]
IP_H_ERRNO [2]

1 handle only strings expressed in decimal
2 none of the functions set these error codes yet

9 Quick start guide

If you know what DHCP, ARP, MAC addresses and IP addresses are what they are used for, here is a quick start guide:

- Configure the driver.
- Physically connect the machine to its peer.
- LRESPR the driver from job 0.
- Maybe initialize the driver with the command STK_INIT from job 0. This will attempt to auto-negotiate the duplex mode.
- If you are connected to a DHCP server (probably your internet router) and have configured the driver accordingly, it should get its IP from the DHCP server.
- Alternatively, unless already configured, set your IP with the SET_MY_IP command and, if you're not only on the LAN, set the gateway IP and the netmask, and also, if the gateway doesn't reply to ARP requests; the gateway MAC. If you only communicate via LAN with machines on the same subnet, you may leave the gateway info and netmask at 0.
- Check the info with the DHCP_INFO$ function.
- If the peer is not an ARP enabled machine, set the MAC address of the peer with the ARP_SET ip$,mac$ command.
- Now open connections as explained in the example. If you have trouble PRINTing single characters, you might want to look at the buffer mechanism.
- Most importantly – have fun.

10 Sockets, binding, connecting and disconnecting

UDP traditionally works with “sockets”, though this driver here tries to hide that and (doesn’t really use sockets as independent structures).

A socket is a data structure containing information about where information is supposed to be sent to or received from. Normally, when one wants to send information via UDP, one would open a socket and set some information in the socket – one can “bind” and/or “connect” a socket. In the context of this driver, socket and channel are the same.

10.1 Binding

10.1.1 Definition

Binding means that one (deliberately) sets the port through which the socket/channel sends and receives information – this is the LOCAL port, on the machine on which the socket/channel is opened. The socket (and thus the channel opened) can only receive data that arrives at the port it is bound to. An (as yet) unbound socket/channel cannot receive any data.

This is what happens implicitly if you **`OPEN_NEW`** a UDP socket:
`OPEN_NEW#3, ”UDP_172.16.0.1:50000”`.
This will expect input on port 50000 on a machine whose IP address is 172.16.0.1. You MUST use the IP address of that machine.

Once a channel/socket is bound to a port **it cannot be unbound.** You must close the channel and open a new one to another port.

Moreover, once a channel/socket is bound to a port, **no other channel/socket may be bound to the same port** (this is fact is pretty logical – the port is what drivers to use determine to what channel/socket a packet arriving at the Ethernet port belongs and should be sent). This also means that to receive data, a channel/socket must be bound.

10.1.2 Automatic binding

At times, a channel/socket is NOT bound yet (e.g. when a channel is opened and connected via OPEN_IN, see below). You may then either bind it explicitly, via the corresponding IP_BIND trap, or, more comfortably, by simply sending some data through it: when you try to send data through an unbound channel/socket, the driver will automatically bind it to a random port, chosen between numbers 50176 and 65535.

10.2 Connecting

When a channel is opened and bound, it still has no idea where any packet to be sent through it should be sent to. It may thus be important to “connect” the socket to the peer.

For UDP, when a socket is “connected”, it means that the socket has been told to what peer (IP address and port) it should send the UDP packets – but also from what peer (IP address and port) it should accept input. **Once a socket is connected, it can only send and receive data to and from that peer.**

Automatic connection

The driver automatically “connects” when you open it to the address of a peer with the **OPEN_IN** command :

```
chan% = FOP_IN( ”UDP_172.16.0.2:50000”)
```

This opens a channel to port 50000 on the peer having the IP address 172.16.0.2, and automatically connects the channel/socket to it.

11 Limitations

11.1 No local host apart from loopback

Apart from the usual loopback address 127.0.0.1, no other local host addresses (0.0.0.0 or any of the 127.xx.xx.xx) are allowed.

11.2 Not all traps supported

Not all of the traps defined for uqlx are supported ATM. See above for those that are.

11.3 Names of channels

The names of the UDP channels must be given in decimal “dot” notation (127.0.0.1) – nothing else is accepted.

11.4 Loopback address

This is slow and does not handle fragmented packets.