Networking the QL – Part 1

Martyn Hill, Dec 2020

This series of articles will prove of interest to any QL user wishing to connect their QLs or compatible machines to one another using the in-built network capability (QNET), perhaps to share files or investigate the potential for multi-user gaming, or else just fascinated with this simple but surprisingly effective network technology. There is also material presented here to help those specifically wishing to connect their ZX Spectrum to the QL network.

During my exploration of QNET over several years (and its underlying protocol, which I'll refer to below as 'QLAN') I have come to appreciate the simplicity and elegance of its design, and I would like to call-out the ingenuity of its original designers, Sinclair and Tony Tebby. I'd like also to mention QView (and Lau Reeves, in particular) for the meticulous annotation of the Minerva source-code that aided my early understanding of the network driver, before I then embarked on making-sense of the enhanced TK2 QLAN source-code.

There are some excellent resources describing QNET already available online – notably:

- Roy Wood/Q-Branch's **SuperBASIC/SBASIC Reference Manual Online – Section 17** (https://superbasic-manual.readthedocs.io/en/latest/index.html) - much of which was authored by Rich Mellor
- An article entitled **Network** by David Denham originally published in QL Today, available on Dilwyn Jones' site: http://www.dilwyn.me.uk/docs/articles/network.zip
- The original **QL User Guide**, available again on Dilwyn's site: http://www.dilwyn.me.uk/docs/ebooks/olqlug/index.htm)

Inevitably, there will be some duplication in these articles of information already available, but the aim was to focus on what is *not* already documented, as well as to clarify – and in some cases, to correct – information found elsewhere. The hope is to share something of my own fascination in this area and, with any luck, present something *useful* that you might not have read before and, ultimately to encourage further adoption and even development of new ideas for the QL Network!

The following topics will be covered in this series:

1. Introduction to the QL Network, its capabilities and compatible systems
    - *Useful applications/use-cases for connecting QLs*
    - *Typical problems and troubleshooting*
    - *Extending QNET with TK2*
    - *Considerations when connecting a QL to the ZX Spectrum*
2. A technical deep-dive in to the QLAN protocol and the hardware that it runs over
    - *The basic NET device in hardware and software*
    - *How TK2's FSERVE extends these capabilities*
    - *How the QLAN protocol looks 'on-the-wire'*
3. Development of a simple QL network-game
4. Ideas for the future of networking with QNET

A deliberate decision was made *not* to cover the exciting IPNet/Ethernet projects by Martin Head, nor the SERNET solution (originally developed by Bernd Reinhardt and based on Phil Borman's MidiNet software), except by reference. There are other interesting and related projects such as the recent Retro WiFi project to add wireless IP capability via the QL's serial port and you are encouraged to research these if interested.

Instead, this set of articles will focus on the networking capabilities native to *any QL* and, in any case, you can find plentiful information elsewhere on these other terrific projects.

About the author…

I work as a Technical Customer Success Manager for a global software company based in the UK within their Cyber threat division, assisting customers realise the full value of our advanced, real-time correlation and machine-learning security monitoring solutions within their Security Operations Centres (SOC). Interwoven with my IT career, I also spent 7 years in education, teaching ICT and running the IT department for a secondary school.

I live in West London with my small but wonderful family and am taking some time during the present 'lockdown' conditions to complete a number of long-standing projects – including the write-up of this series of articles.

The subject of heterogeneous computer networking and interfacing diverse electronic and computing devices has always held a fascination for me and I have been experimenting with the QL Network specifically since I acquired a second-hand QL from a generous colleague about 5 years ago. This was many years after having owned my *first* QL as teenager, thanks to my brother who worked at the time for Thorn EMI Datatech, contracted by Sinclair to service the QL and Spectrum hardware. It was also he who gifted-me my first ever home-computer, a ZX Spectrum, at the age of 11 and undoubtedly influenced my subsequent career choices – thanks, brother ☺

Before re-acquiring a real QL, I used various QL emulators running on my PC to develop software and ideas, finally settling on the terrific QPC2 by Marcel Kilgus, which I continue to use as my main QDOS platform – but secretly missing the opportunity for hardware development that is so much more accessible with the original QL than with more modern platforms.

I now own 4 QLs of various states of expansion (one with Tetroid's SGC clone, fitted inside a PC case), a QXL card also installed in the same PC, a couple of Peter Graf's marvelous FPGA-based Q68s as well as two ZX Spectrum/Interface-1s, all of which I have connected together successfully using the QL/ZX Network.

QLAN Development

Enhanced versions of the QLAN device-drivers have been developed for various QL platforms as well as fixing some previously undocumented bugs in the network code of the Spectrum/Interface-1 Shadow-ROM. Using these enhanced drivers, it has been possible to run the network at anything from 4.5x to 8x the original bit-rate with suitably swift QL-like hardware (QL/SGC to Q68 and Q68 to Q68, respectively) and am currently tweaking the basic QL driver to push it to about 1.5x its original speed.

With suitable changes to the Interface-1 ROM routines at the Spectrum end, it is finally possible to achieve *reliable* bi-directional communication between the 7.5MHz QL and a Spectrum, albeit at their original network bit-rate.

Thanks to the clear design and copious source-code comments, it was relatively straight-forward to take the original TK2 network driver (for the S/GC) from the SMSQ/E source-tree and re-engineer it for the Q68 – and its hardware timer/counter - removing most of the dependency on m68k instruction-cycle timings which dictate the design of the original QLAN device driver and are the principal cause of communication problems between different QL machines. The ND-Q68 driver was released to the community via the Sinclair QL Forum and Dilwyn Jones' website last year and it's pleasing to see users start to put their Q68s to use on the network.

An updated version of the ND-Q68 driver will be made available early next year, along with the enhanced drivers for the basic QL and SGC once they're ready for public consumption. If anyone is interested in the Spectrum/Interface-1 network routine enhancements, these could be made available as well, though for most users, it's not a trivial process to update the Interface-1 Shadow-ROM.

My current and long-standing project is the QLAN to USB Bridge 'QLUB' Adapter, designed to interface an emulated QL running on a PC to the native QL network using custom firmware running on an inexpensive microcontroller connected via its USB-port which, in conjunction with new QDOS software and a slightly modified QLAN device driver, is designed to ease the otherwise tedious process of transferring files between the two platforms. With a bit of luck, the QLUB Adapter should also come to fruition later in 2020 and the (simple) hardware design, microcontroller firmware and corresponding QDOS software will be made available to the community as a DIY project. Should one of our community hardware developers wish to pick-up the project and manufacture/package a ready-to-use solution, the author would welcome collaboration!

1. Introduction to the QL Network

*It is recommended that you read this section in conjunction with the **SuperBasic/SBASIC Reference Manual Online** – section 17 'Networking'. I have consciously omitted details that are already well-documented there, duplicating only what is needed to contextualise any additions to the subject, or correct some small errors found in that otherwise very comprehensive resource.*

1.1. QNET Compatible Systems

In addition to the basic QL, the **Aurora** QL mainboard replacement, the **QXL** PC ISA card and the newer **Q68** FPGA-based machine all include most or all of the QNET hardware needed to interconnect – the Q68 will require a few additional components to be retro-fitted (see the Sinclair QL Forum posting at https://www.qlforum.co.uk/viewtopic.php?f=3&t=2881)

A QL fitted with a **Gold** or **SuperGoldCard** will of course run QNET very nicely already – all the required protocol timings being automatically adjusted in software to suit their faster CPUs.

The **Thor** range of machines also have the required QNET hardware and software.

In addition to QL-compatible machines, functionally-similar hardware and the same basic protocol is used by the **ZX Spectrum** when the **Interface-1** is fitted but, due to some subtle bugs in the Interface-1 Shadow ROM, reliable reception at the QL *from* the Spectrum can only take place with a updated Interface-1 firmware in conjunction with a slightly optimised QL NET driver. This is a real pity, as many users transitioned to a QL from the Spectrum and might have benefited from connecting the two, otherwise incompatible machines, without recourse to the slower serial-ports.

Other potential candidates for interconnection with QNET include the **Q40/Q60** and **SAM Coupe**, but these machines would require additional (but relatively simple) hardware interfacing and suitable software to be developed.

1.2. Typical Applications for QNET

Simple file-transfer between two or more stations is perfectly possible with the basic QL network without the enhancements afforded by TK2 (covered later).

Even without TK2, QNET can be put to good use, e.g.:

a) **'Boot-strapping'** – transferring S/Basic extensions and applications to an unexpanded QL at start-up.
b) **Preserving your work** – saving in-memory files to another working QL when you can't get the local microdrives to cooperate.
c) **'Spooling' print files** – using COPY to a machine with an attached printer to get hard-copy.
d) **Multi-player gaming** – streaming real-time game data between two or more machines and players.
e) **Linking with the ZX Spectrum** – whilst code and BASIC files are not directly compatible between QL and Spectrum, preserving Spectrum files on the more reliable QL hardware can prove very useful.

f) **Electronic-interfacing projects** – with a suitably programmed microcontroller to 'speak QLAN', it becomes possible to interface the QL with your electronic breadboarding projects (more in a later article.)

Once a working QNET is established, timing and reliability of the network can actually outperform our ageing MDVs, even though the *raw* bit-rate of the network (c. 89kbps) is much lower than that for microdrives (200kbps nom.) – EXECing large programs over the network can often prove more reliable *and* more rapid than loading from a flaky cartridge, where multiple passes of the tape are often required to correctly read the file-blocks.

Given the interactive nature of the NET device effectively requiring commands to be issued on *both* peer stations simultaneously before file-transfer can begin, use of the basic NET functionality is not especially convenient, but effective none-the-less. As we'll read later, TK2 addresses this inconvenience very effectively with its FSERVE file-server capability…

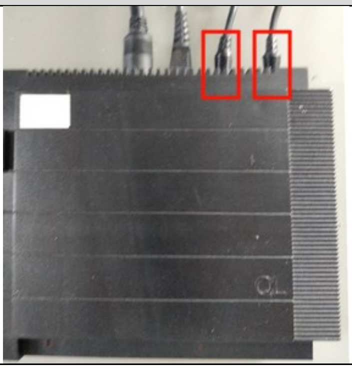
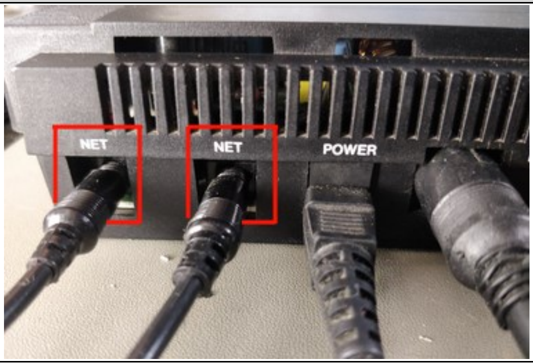
1.3. Connecting Stations

Every QL - and several of its compatible hardware replacements – include the basic hardware and software to allow connection to one another using a simple two-wire cable, such as used for mono-audio applications, terminated with a 3.5mm 'jack' plug of the Tip-Sleeve (TS) varieties at each end.

It is also perfectly possible to use 3-wire stereo cabling with the TRS (Tip-Ring-Sleeve) type plug.

NET Port Connection Plugs	
3.5mm 'TS' Jack-plug (mono)	**3.5mm 'TRS' Jack-plug (stereo)**

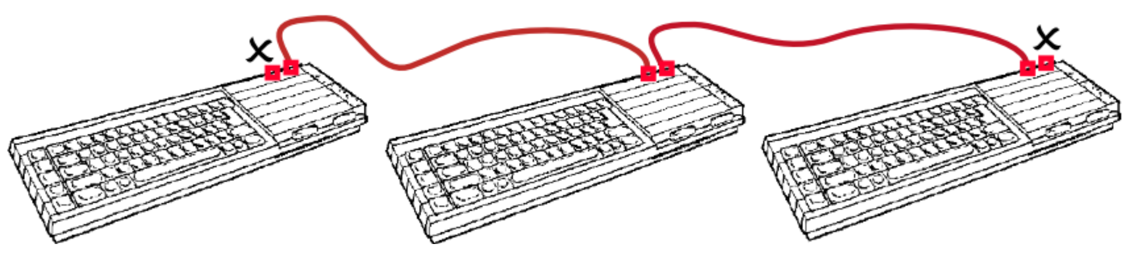
Each station is linked to the next in a daisy-chained/common-bus arrangement using either or both of the two (identical) NET sockets that lurk unobtrusively at the rear-right of your QL; the two stations that sit at the extreme ends of the chain will only have one socket connected, whereas each intervening station will have both occupied.

NET Port Sockets	
	

The most common situation of just two connected stations will therefore require only a **single** 2-core cable, leaving one NET socket disconnected on each station – it does not matter which of the two available sockets are used at each end.

Leaving the two end-stations with a single unoccupied socket on each is part of the QNET hardware design – the sockets are themselves of the 'switched' type, whereby any unconnected socket provides a pull-down termination to the NET line; once a jack-plug is inserted, the termination is effectively disabled on that socket.

QNET Cabling



QL image courtesy of the QL User Guide.

Creating a 'loop' with the two end-stations linked to one another will thus cause poor reception (due to 'signal reflections') and, in all likelihood, render the network unusable.

The simplicity, ready availability, flexibility and robust nature of the cabling required is, in the author's opinion, a key benefit of the QNET design over, say a traditional RS-232/COM port solution.

The maximum *combined* length for the cable-run is quoted at 100m (QL User Guide, p34.) The author has seen success using cable-runs of 30m or more in practice using off-the-shelf and inexpensive mono and stereo-audio cabling.

1.4. Basic Network Usage

The basic NET device driver built-in to QDOS allows for simple, **peer-to-peer** transfer of files using familiar S/Basic commands such as `COPY` and `LOAD`, `LBYTES`, `EXEC` (with their `SAVE` counterparts), as well as sending/receiving arbitrary byte-streams using explicit `OPEN/PRINT` and `OPEN/INPUT/INKEY$` command sequences at the respective ends of the link.

Each station is assigned a unique 'station-ID' between 1 and 63 (or 64, depending upon driver vintage) using the S/Basic `NET` command and a file-transfer is initiated by entering reciprocal commands (load/save) at each station.

If only two machines are connected together, it is possible to successfully communicate *without* changing the station-ID from its default of '1' – ambiguity is avoided by the fact that only one activity (send *or* receive) can take place at any given time on either station. That said, assigning unique station-IDs is recommended even in this situation.

In addition to explicitly specifying the remote peer station-ID when opening a channel, the NET device also recognises a station-ID of `0` (zero) for both input and output - referred-to as a 'network broadcast' - as well as a 'receive from any station' facility, whereby the receiving station NET channel is opened to *its own* station-ID.

The basic QLAN protocol detects the end-of-file (EOF) condition to determine when to close byte-stream type connections as created with an explicit `OPEN` when the NET driver flags the last packet as such when the output channel is `CLOSE`d, suiting 'byte-serial' communications.

However, this method is insufficient for file-transfer type connections initiated with `SAVE`/`SBYTES`/`SEXEC` and received with `LOAD`, etc, where it is necessary for the receiving station to know *from the outset* the expected file-length and its type. To accommodate this, the file-transfer commands (save, etc) prepend a 15-byte 'simple file-header' to the very first packet sent to indicate the number of bytes to follow, along with other meta-data – which is expected and interpreted by the `LOAD`/`LBYTES`/`EXEC` procedures at the receiving end. This will be recognisable to readers familiar with the ZX Spectrum's tape file-header (and equivalent for the Interface-1.)

Some example S/Basic routines might help clarify:

Transferring Data and Files with QNET	
QL-A	***QL-B***
Stream serial data	
NET **1** OPEN #3,**'neto_2'** FOR b%=32 TO 192 PRINT #3, CHR$(b%); END FOR b% : CLOSE #3	NET **2** OPEN #3,**'neti_1'** REPeat recvBytes IF EOF(#3) THEN EXIT recvBytes PRINT INKEY$(#3,-1); END REPeat recvBytes CLOSE #3
Transfer a file (e.g. QL screen-display)	
NET **1** REMark Implicit file-channel automatically opened for output, then closed : SBYTES **'neto_2'**,131072, 32768 : REMark ...Or from a saved file COPY 'win1_test_scr' TO **'neto_2'**	NET **2** REMark Implicit file-channel automatically opened for input, then closed : LBYTES **'neti_1'**,131072 : REMark ...Or, save it to a file COPY **'neti_1'** TO 'win1_test_scr'

Example 'NET Station-ID Server' *(like DHCP)*	

idServer%=1: nextId%=idServer%+1: maxId%=32: keyEsc%=27
:
NET **idServer%**
:
REPeat serveNetIds
  REMark *** (re)Open an input channel **to self** ***
  OPEN #3,**'neti_'**&**idServer%**
  REPeat waitClient
    tempClientId%=CODE(INKEY$(#3,10))
    IF tempClientId%<>0 THEN EXIT waitClient
    IF CODE(INKEY$(#0,10))=keyEsc% THEN EXIT serveNetIDs
  END REPeat waitClient
  :
  OPEN #3,**'neto_'**&**tempClientId%**: PRINT #3,CODE(nextId%);: CLOSE #3
  nextId%=nextId%+1: IF nextId%>maxId% THEN EXIT serveNetIds
END REPeat serveNetIds
:
CLOSE #3
	Server

Client	idServer%=1 REMark *** Generate a random but 'reserved' temporary station-id netId%=RND(33 TO 63) : NET **netId%** : OPEN #3,**'neto_'**&**idServer%**: PRINT #3,CODE(netId%);: CLOSE #3 : OPEN #3,'neti_'&idServer% getId%=CODE(INKEY$(#3,5*50)) CLOSE #3 : IF getId%<>0 THEN NET **getId%** ELSE PRINT #0,"Couldn't get NET ID. Quitting...": STOP END IF

The NET link can also be aborted manually at any time from either end by the user pressing `Ctrl+Space`, triggering the closure of any *implicit* file-transfer channels in the process. An 'EOF' condition will be flagged and detectable at the receiving-end if aborted at the sender station. On the other hand, the sender has no way to know if the user at the receiving station has since aborted the transfer and will instead continue to re-attempt transmission of the latest packet until it is manually aborted itself.

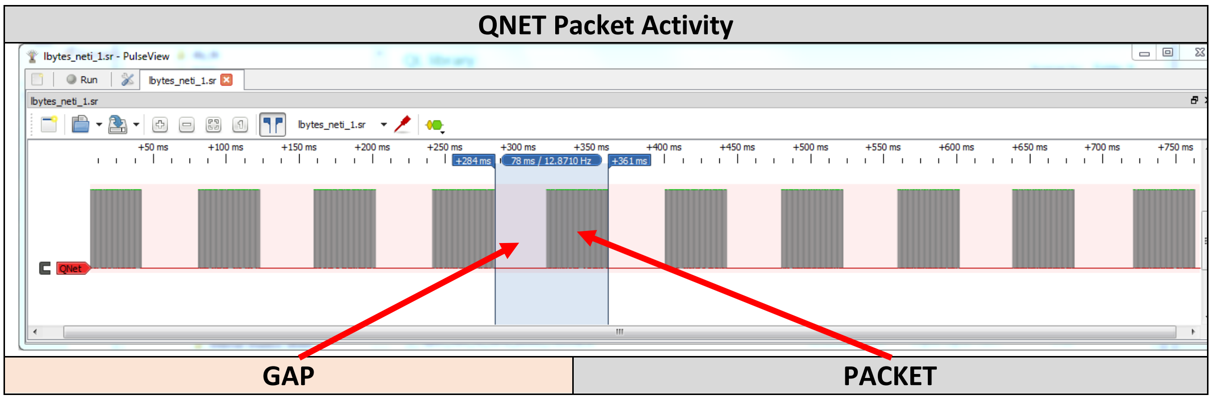
1.5. Packet Anatomy

It may prove interesting to understand how the QLAN protocol arranges for the raw data to appear over the link, how it manages time for other activities in the machine as well as sharing network access-time with other connected stations. This topic will be further expanded in a later article.

All transmitted data is split in to ‘blocks’ or packets of up to 255-bytes, prepended by a packet-header followed by the data-packet itself. For a file consisting of multiple blocks then, all but the last will be the full 255-bytes in length, with the last containing the remaining bytes.

The packet-header includes both the source and destination station IDs, as well as other meta-data used by the protocol including the current ‘block-number’ and checksums (see https://superbasic-manual.readthedocs.io/en/latest/Appendices/Appendix17.html#a17-1-4-qnet-without-toolkit-ii)

A sample signal trace showing several packets of an on-going transmission shows the general link-usage:



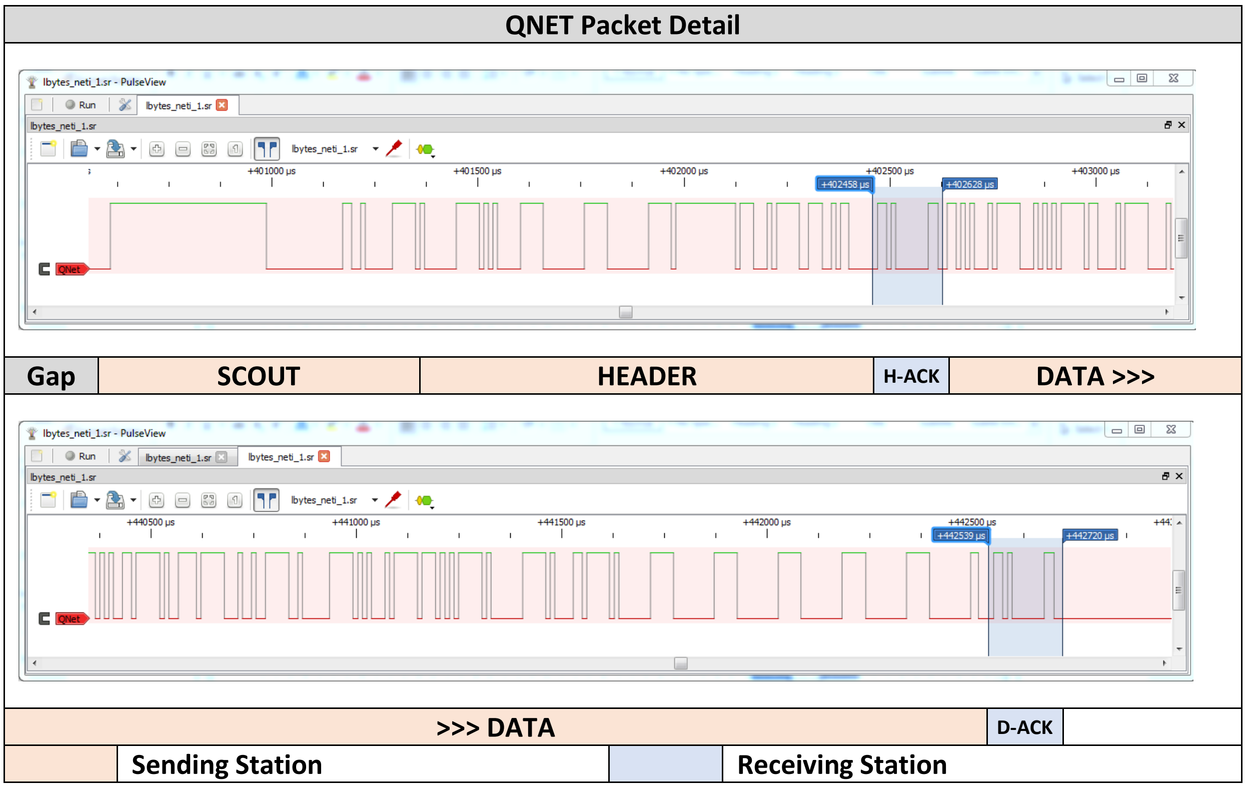
The sending station starts by ‘listening’ for a suitably long silence or ‘gap’ indicating that the link is free, before placing a ‘scout’ bit-pattern on the wire whilst simultaneously reading-back the link state to ensure that no other station is also trying to claim the link.

The bit-pattern used for the scout is carefully computed both to ensure that different sending stations produce a distinct pattern (based principally on their own, unique station-ID) as well as to ensure that a predictable outcome will result – namely, that all stations *except* the one with the *lowest* station-ID will detect the contention and back-off gracefully, whilst the sending station with the lowest station-ID will not even notice the contention and proceed with its transmission.

At the receiving end, once the *start* of the scout is detected, the receiving station actually *ignores* the link state altogether, only starting to pay attention again after a defined time-window has passed designed to skip the scout-phase, but in good time to catch the start of the packet-header that follows.

Following transmission of the header, the sending station listens for an active ‘acknowledge’ signal (a single byte: 0x01) to be received-back before initiating transmission of the data-packet itself and again, listens for a final acknowledgement to determine that the data-packet was successfully received. If successful, the ‘block-number’ is incremented at each end, ready for the next packet to be transmitted.

Zooming-in to the signal trace during the packet transmission shows the following:



If the gap is not detected (*link already in use*), contention found when sending the scout (*another station simultaneously trying to claim the link*), or either acknowledge response-byte is not received (*receiver not listening, or has rejected the packet for some reason*), the sending station will abort the current transmission-cycle and rely on the QDOS IOSS scheduler to re-invoke transmission later. In this case, the current block-number is left alone.

If the receiving station - which is periodically ‘polling’ the link – fails to detect the scout within a defined time-window, determines that the packet is not meant for, or currently expected by, this station or calculates a corrupted check-sum in either of the header or data-packet, it will silently abort the current poll-cycle (thus *not* sending a response byte) and, like transmission, rely on the QDOS IOSS scheduler to re-invoke packet-reception later. The receiver adjusts its ‘expected’ block-number in a similar way to the sender, depending upon success or failure.

Thus, the sender and receiver stay in-sync by referring to the incremental block-number (0-65535) - such that any repeated packets (block N-1) can be detected and discarded at the receiver, whilst continuing to wait for the expected packet (block N). Such a situation can arise when the sending station ‘misses’ the acknowledge-byte sent in response to the last packet sent.

For a ‘repeat’ packet to be discarded in this way, the receiver must still acknowledge it, so that the sender knows to ‘move-on’ to the next block. *One of the bugs found in the Spectrum Interface-1 Shadow ROM relates to this process, whereby the receiving Spectrum would fail to correctly detect a repeat packet (and thus not acknowledge it), leaving the sending station trying to resend an old packet in an infinite loop.*

Network ‘broadcast’ transmissions are much the same except in how acknowledgements are handled and the length of the Scout-phase. On a QL without TK2, *no* acknowledgement handshaking is employed, leading to unreliable broadcasting.

With TK2, an active ACK/Negative ACK is added to the protocol, active only after the end of the data-packet that improves broadcast reliability immensely. Note however that this aspect is *not* entirely compatible with non-TK2 or Spectrum broadcasts.

1.5.1. Mind the Gap…

Gaps are seen interspersed between each attempt to transmit a packet, both between successful and unsuccessful transmissions; the length of the gap between *failed* attempts is typically shorter.

These transmission gaps form an integral part of the QLAN protocol and effectively allow time for the receiver to process the last packet and the sender to prepare the next, as well as to free the line long enough for *another* pair of stations to claim the link during the 'intermission', thus time-sharing the link.

This approach to time-sharing a link is similar to what is referred-to as 'Carrier Sense/Collision Detection, Multiple Access' or 'CS/CDMA' in conventional network terminology.

To give an indication of the impact of these gaps on effective throughput, we observe that, between two 7.5MHz QLs with no other jobs running and no corrupted packets, a 32KB QL display-file will transfer in c. 10.5 seconds, with about 45% of this time spent in the gaps between actual data packets.

Between two Q68 machines on the other hand, the same file is transmitted in about 6.3 seconds with only 15% of the total time spent in the gaps, due in part to the more rapid inter-packet processing, before each station is then ready for the next.

This evaluates to transfer-rates of between 3KB and 5KB per second, depending upon the performance of the machines involved and the minimum length of gap that each can sustain. Thus, higher effective throughput is achieved with an improved link-utilization/efficiency for the very same bit-rate.

If this time spent in gaps seems wasteful, bear in mind that, for an inherently multi-tasking OS such as QDOS, these gaps *also* represent time-slices now available to the CPU to devote to other tasks/Jobs that may be ready to run and thus maintain a higher level of overall responsiveness to the user.

The Spectrum ROM, on the other hand, being inherently mono-tasking, typically uses shorter gaps in its transmission attempts – but only scans the keyboard for 'Break' between each packet with no other user activity possible until the expected packet is successfully received and processed.

In summary, the QLAN and ZXNet protocols are based on a packet re-transmit mechanism (much like Ethernet), whereby it is entirely acceptable for individual packets to be 'missed' when the receiver is not currently listening, and continually re-tried until an active acknowledgement is received and the next packet can be queued for transmission. The IOSS 'retry' mechanism in QDOS is therefore an essential support to the QLAN protocol.

Thus, the effective throughput is highly dependent upon the 'attentiveness' of the receiver – and less on the raw network bit-rate and one reason why the Q68 as an example achieves a better *reception* throughput than the basic QL.

1.6. Trouble-shooting QNET problems

Many QL users have tinkered with connecting their QLs, only to get frustrated and give-up without ever enjoying the benefits of networking. There are several potential reasons for this, the most common of which are described below in terms of either their cause or the observed behaviour. Possible cures are provided in each case:

1. **Unreliable QL hardware** – the earlier issue-5 QL motherboard has the ZX8302 ULA responsible for the network and microdrives connected to the 'contended' data-bus of the QL's main ULA, resulting in irregular access-timing which can cripple QNET. Sometimes they work, another day, not. *Replacing with an Issue-6/7 motherboard, or else gratuitous hacking*

of the Iss-5 board to mimic the later versions resolves this (the author has successfully applied such an invasive mod, but wouldn't recommend it!)

2. **Stuck NET ports** – it is occasionally observed that the voltage present at the NET port gets 'stuck' at one logic-level or the other, indicative of either a failing PNP transistor in the NET output circuit, or a defective/marginal ZX8302 ULA. *Power-cycling the responsible QL can often 'release' the stuck NET port. Replacing the transistor (TR2/ZTX-510 or equivalent) may also help, as well as an obscure mod once documented in a QL magazine of inserting a diode in the Gnd line to the 5V linear-voltage regulator (NB the author has NOT tried this mod!). At the time of writing, replacements for the ZX8302 are still available from RWAP Services/SellMyRetro – the author recommends having a spare one to hand, in any case.*
3. **Corroded NET sockets** – given their typical lack of use, the inner contacts of the NET jack-sockets can become tarnished or even rusted, causing poor or no connection. *An attempt to clean the inner contacts with Isopropyl alcohol and lint-free swab or similar may help (see a user's suggestion at* https://qlforum.co.uk/viewtopic.php?f=12&t=3279*), or else de-soldering the sockets and replacing with a similar variety (if you can still find them) if truly corroded. Remember that the two sockets are ganged together, so the state of one will impact the other, so treat them as a pair.*
4. **Faulty cabling** – if you have made your own – or re-purposed an old - 2-core cable with jack-plugs, the connection failure may be in the soldered plug or within the cable itself. *Check continuity and that the respective 'poles' of the jack-plug have been connected to their counterpart at each end – Tip to Tip, Sleeve to Sleeve.*
5. **Mis-connected cabling** – with more than a couple of machines linked together, it is surprisingly easy to mess-up the cabling, resulting in one or other station being left disconnected, routed back to itself, or creating an unwanted 'loop' between the two extreme ends of the link, thus disabling the NET line-termination. *Recheck the cabling – it is best to abort any on-going transmission at the sending station before unplugging/replugging cables, although the author hasn't experienced any nasty surprises with 'hot-plugging' the net cables as yet!*
6. **Misconfigured Station IDs** – a simple, but common enough occurrence, again when more than two machines are connected, is forgetting to use the `NET` command to set each machine with an unique station-ID. Alternatively, forgetting the station-ID of the machine you are typing-at when entering the respective `NETI_x`/`NETO_x` device name. *Check and re-issue the `NET` command or re-enter the appropriate device name.*
7. **Broadcast failure** – the station-0 network-broadcast feature (using `NETI_0`/`NETO_0`) is not especially reliable due to the lack of active 'acknowledgement' in the broadcast protocol. *Test again using a direct peer-to-peer file-transfer to first validate the link, then retry the broadcast. Upgrading a legacy QDOS ROM to the marvellous Minerva OS can also make network broadcast more reliable due to the re-coding of this feature in Minerva's NET driver – or add TK2, which replaces the broadcast feature entirely with a much enhanced version.*
8. **QL seems to hang** – this may not be a fault, but a normal part of using the network as interrupts are switched-off within the NET 'physical' device-driver during transmission of each packet to ensure consistent timing. In particular, if you abort a transmission in progress, the sending station will repeatedly retry transmitting the current packet up to 2,000x, paying *minimal* attention to the keyboard for the user pressing `Ctrl+Space` between each attempt. The receiving station is usually more responsive to `Ctrl+Space`.
9. **Nothing is received when 'streaming' serial data** – again, this may not be a fault when using an explicit `OPEN`, or even `COPY`ing from another 'serial' type device. The packet/block design of the NET driver means that a full packet of 255 bytes needs to accumulate in the sender's buffer before the physical driver is invoked to transmit down the wire. *As there is no 'FLUSH' capability in the current NET driver, it is necessary to either pad-out the output to fill the 255-byte buffer, or else issue a `CLOSE` at the sending station. When closed, the NET driver will attempt to send whatever is present in the buffer regardless of how full, marking this the*

'last' packet/block. In turn, this will also cause the receiving station to detect an EOF condition.

10. **Bad Parameter (BP) error** – when receiving a file using `LBYTES`/`EXEC` etc, the receiving station expects a simple 'serial file-header' as the first 15-bytes of the very first packet, which it detects when the first byte appears as '0xFF'. If the first byte received is not 0xFF, the command will fail with the QDOS error 'Bad Parameter.' The remaining 14-bytes of the serial header contain the all-important file-length, 'data-type' byte and, if a Job-type file, the 'data-space'. A BP error will also result if the file-type byte in the header does not match what is expected. *Some older versions of TK2 seem to also expect `LOAD` to operate with a serial header, but are not always sent by the corresponding `SAVE`! Upgrade to a later version of TK2.*
11. **Nothing works after loading TK2 from disk** – due to the sensitive software timing-loops used by the NET driver, running TK2 from RAM on a basic QL will fail to allow successful communication because RAM access-contention introduces inconsistent delays running software or reading/writing to RAM. *TK2 running in (inherently uncontended) ROM, or running in RAM on a (Super)GoldCard-equipped QL avoids this issue.*

Having now experimented with QL networking over several years and across diverse QLs and compatible machines, the author rarely experiences problems using QNET today – except those one creates oneself!

1.7. The TK2 enhancements to QNET

Whilst the basic NET device offers some very useful capabilities, almost all of these are enhanced or otherwise augmented with more advanced facilities once TK2 is available. To be frank, anything more than trivial use of the QL requires TK2 to get the best from the machine and this is certainly the case with networking.

TK2 replaces the entire QDOS NET driver once installed, maintaining the QLAN protocol specification to allow basic communication even with QLs *without* TK2; it improves the reliability of some features (e.g. network-broadcast) and adds the all-important `FSERVE` server Job with its corresponding '`Nx`' client-side device-driver.

As mentioned, TK2 requires uncontended RAM or ROM in which to run to avoid the QL's inherent RAM access-contention behaviour that would otherwise mess with the delicate timing of the bit-level driver code.

The basic NET functionality remains as described before, but once the `FSERVE` Job is invoked on one (or more) of the connected stations, it becomes possible to access the file and device resources on the server station as if they were installed locally, through the use of the '`Nx`' client pseudo-device.

`FSERVE` is flexible enough to make almost all the features of the remote device available to the client station (which needs to have TK2 installed, but needn't be running `FSERVE`.) Furthermore, any device-driver linked-in to the serving station can be accessed from the client - not just the obvious file-system devices such as `MDV`, `FLP` or `WIN`.

Thus, `SCR`, `CON`, `SER`, `PAR` and other devices installed on the serving station can all be accessed from the client using the same general form `Nx_<remote_device_spec>`. Some truly clever network programming techniques open-up, especially with the likes of the `MEM` memory-device (thanks to Simon N Goodwin's DIYTK driver.)

Still, most of the value of this client-server capability is in practice found when loading or saving files on the server station. Here are some examples (where `Nx` means `N<server_station-ID>`) and note that *nothing needs else to be typed at the serving-station once FSERVE is invoked*:

a) Backup/archiving of entire MDV cartridges across the network to a folder on the server's WIN device, *with a single command at the client station*:
```
WCOPY 'mdv1_' TO 'Nx_win1_backup_'
```
b) Sending print-files to a quality printer attached to the server's SER port:
```
COPY 'mdv1_print_txt' TO 'Nx_ser1hr'
```
– better still, use TK2's `SPOOL` command in place of COPY.

c) Sending messages to the screen-display of the user sitting at the serving station:
```
OPEN #3,'Nx_scr_100x20a206x118': PRINT #3, "Hello!": PAUSE: CLOSE #3
```
d) Loading EXECutable programs directly from the server's file-system:
```
EX 'Nx_win1_APPS_QED_qed';"mdv1_text_file"
```
e) Directly manipulating the memory (e.g. video-memory) of the server, using MEM:
```
OPEN #3,'Nx_mem': PUT #3\131072: PRINT #3,
FILL$(CHR$(170)&CHR$(0),32767);: CLOSE #3
```
f) Syncing the local real-time clock *(taken from the Minerva docs – who needs NTP?):*
```
f$="Nx_ram1_date_tmp": d%=FOP_NEW(f$)
IF d%>0 THEN CLOSE #d%: SDATE FUPDT(\f$): DELETE f$
```

Some of these examples can *almost* be achieved with the basic NET driver, but bear in mind that, as `FSERVE` runs autonomously on the server station, only the client-side commands need be entered under TK2, as compared to having to enter reciprocal commands on *both* peers without it.

1.7.1. FSERVE's Hidden Gem…

There is a hidden – or at least, not widely documented – feature available when running `FSERVE`, that comes in to its own when 'bridging' between a QL and a QL emulator running on a PC or other QL platforms that lack NET ports.

It turns-out that `FSERVE` running on an *intermediate* station will happily open on the client's behalf a network-type device targeted at a different *target* station such that, assuming that the intermediate server (`Nx`) and the target (`Sy`) stations both also have the SERNET driver loaded and are connected to one-another via a suitable serial-cable, a command like:

```
COPY 'mdv1_file' TO 'Nx_Sy_win1_file'
```

…will result in `mdv1_file` being copied to the emulator's `WIN1_` device, even though the emulated QL is only *indirectly* connected to the client station, and itself only has a SERial port and no QNET link. This overcomes the (current) limitation of QL emulators not being able to connect directly to the QL network.

In practice, this bridging can sometimes prove sluggish. This is because each packet making-up `mdv1_file` must 'hop' across the intermediate station in a store-and forward manner – being re-packaged via `FSERVE` on `Nx` and delivered to the SERNET driver for onward transmission to their final resting-place on the emulator's `WIN` device on station `Sy`.

A further explanation for this performance-drop can be understood when one considers that TK2's network client-server driver employs the common 'command/response' paradigm, introducing additional latency between each data-packet due to requiring 2 or more 'command' packets for each block of data sent and, whilst just noticeable in direct station-to-station communications, delays becomes much more evident with the additional hop introduced with this bridging topology.

Fortunately, as the TK2 client-server leverages extended packet-sizes (up to 1020 bytes, though file-system packets are typically 524 bytes in length), some of the bandwidth inefficiency of the simple NET device is mitigated due to the presence of fewer packet 'gaps' for a given payload (see above.)

All in all, a *really useful* feature of `FSERVE` when combined with its SERNET file-server equivalent and a credit to the designers of each driver!

Just for academic interest (or fun?), you can also observe this feature in action between 3 QLs using just the QLAN each running TK2, the client station 'bouncing' packets off an intermediary station `Nx` to reach the target `Ny`, e.g., with a 32KB QL- display file, try:

```
            LBYTES 'Nx_Ny_win1_ql-display_scr',131072
```

It works, but don't hold your breath!

On a final note, whilst this behaviour might imply otherwise, any attempt to *further* extend the chain by introducing *another* intermediate station, seems to fail completely: e.g.

```
            COPY 'mdv1_file' TO 'Nx_Ny_Nz_win1_file'
```

It is not yet clear why this configuration should fail, but its usefulness is arguable in any case…

1.8. Connecting with the ZX Spectrum

Notwithstanding the known challenges with communications between the QL and Spectrum over QNET/ZXNet, there are some useful activities that become possible when these two machines are linked that are well-worth persevering-with.

During the author's research on connecting the ZX Spectrum, it was read that Sinclair may have envisaged ZXNet – or the "Sinclair Network Standard" - as a core part of the original ZX82/Spectrum design and encouraged its adoption as an 'open-standard', but that it was subsequently dropped from the Spectrum hardware and its base 16K ROM, until it resurfaced in the form of the Interface-1 add-on.

One of the many Spectrum ROM re-writers, Geoff Wearmouth, went as far as implementing the ZXNet routines within his replacement 16K ROM – 'The Sea Change ROM' (see http://zxspectrum.it.omegahg.com/rom/seachange/seachange.pdf) – which uses the Interface-1 hardware, but not its firmware.

In any case, the (basic) QL and Spectrum/Interface-1 share an entirely compatible protocol for their respective network implementations.

How, then, is it that most users were unable to make the Spectrum/QL connection work for them, even though Spectrum to Spectrum network comms seemed relatively usable?

During the author's investigation of this interesting tangent to QL networking, several bugs were identified in the Interface-1 network code, the most significant of which was observation of inconsistent timing when *sending* certain bit-patterns *from* the Spectrum. The cause was found to be 'IO Contention' when writing the next transmission-bit to the Interface-1 ULA register.

This finding aligns with the general consensus that QL *to* Spectrum works reasonably well, but you could rarely receive anything back *from* the Spectrum, severely limiting its usefulness.

This behaviour has since been diagnosed and replacement Z80 routines developed and tested to avoid the ULA Contention issue, among other subtle faults.

Replacing the Interface-1 ROM is not trivial - to put it mildly – due, in part, to the exceptionally tight-spacing inside the elegant interface casing. None-the-less, the author has observed reliable comms between these machines with the modified Interface-1 ROM routines in-place and would be happy to share the code in due course with any interested users willing to hack their precious Interface-1 hardware.

Alternatively, using a slightly revised QNET/TK2 driver on the QL, and tweaking some of the timing-constants, the basic QL can be made to better accommodate the Spectrum's wobbly bit-timings without recourse to hacking the Interface-1. The result is not 100% perfect, but a good step.

Fortunately, the QL fitted with SGC (and possibly, the GC) as well as the QXL can already cope with the Spectrum's timing anomaly without further adjustment, though not always 100% reliably. The Q68 running the ND-Q68 driver, on the other-hand, seems to take the Spectrum's wobbles in its stride…

However it is achieved, once a reliable connection can be established, you can:

a) Backup your ZX microdrive cartridges to reliable QL storage
b) Develop Spectrum software more conveniently within an emulator on the QL (e.g. Speculator or the brilliant ZM/x) and transfer back to a 'real' Spectrum
c) Transfer screen-dumps of your favourite Spectrum games and render them on the QL (with some additional QL software)
d) Explore multi-player/multi-*platform* network games

There are a number of differences that need to be considered when transferring files between these two diverse platforms, some of which are discussed below, before we close the topic on Spectrum/QL networking:

1. Spectrum/Interface-1 vs QL 'file-headers'
2. 'Foreign' programs and Spectrum BASIC tokenisation
3. Screen display file-formats

1.8.1. File-headers

The first challenge to overcome when transferring files between the two platforms is taking-account of differences between their respective 'file-headers.' Such headers are typically only present on files that need them, as opposed to simple 'serial-access' data-files which do not – much like the concepts of 'header' and 'header-less' tape-files, familiar to Spectrum users.

On the QL, there are two forms of file-header, the usual 'directory-device' type headers at 64-bytes each, and the simplified 'serial-file-headers', at 15-bytes (including an initial flag-byte, 0xFF) which include a subset of the 64-byte variety. As the QL's NET device is of the 'serial' type, the shorter file-header is used.

On the 'unexpanded' Spectrum, the tape-system uses 17-byte file-headers, which include the 10-byte/characters of the filename. The Interface-1 introduced a smaller and slightly modified 9-byte file-header used on microdrives and the network, which drops the filename (which on microdrives, is instead stored in the 'sector-header'.)

It proves easier to manage the differences at the QL end, rather than at the Spectrum and, fortunately, the QL can accommodate 'header-less' files easily with the `COPY_N` procedure when needed.

Receiving files from the Spectrum is straightforward, as the QL will treat the incoming file as headerless by default when `COPY` is used with a 'serial' device – only requiring a file-header if `LOAD`/`LBYTES` or `EXEC` (don't!) is used to load the Spectrum file.

Thus, where 'x' and 'y' are the station-IDs of the Spectrum and QL respectively, the following commands will correctly transfer `<filename>` to the QL, storing the Spectrum file, along with its 9-byte header on the QL as if it was part of the file image; the QL sees a file-length 9-bytes larger than the Spectrum records in its header:

Spectrum: `MOVE 'm';1;'<filename>' TO 'n';y` *or*
Spectrum: `SAVE 'n';y SCREEN$` *(or whatever type of file this represents.)*

QL: `COPY 'neti_x' TO 'win1_ZXFiles_<filename>'`

To send the file back again, we must persuade `COPY` *not* to add a QL serial-file-header, so we use the `COPY_N` variant, thus:

QL: `COPY_N 'win1_ZXFiles_<filename>' TO 'neto_x'`

Spectrum*: `MOVE 'n';y TO 'm';1;'<filename>'` *or, better-still*
Spectrum: `LOAD 'n';y SCREEN$` *(or whatever…)*

** The Spectrum `MOVE` command was deliberately 'crippled' in the Interface-1 ROM to limit reading or writing 'non-PRINT' type files from/to microdrive to discourage software piracy. Thus, without a revised ROM, making use of the `*MOVE` extension-command, or else direct manipulation of the Interface-1 microdrive channel-block to force a non-PRINT file-type whilst writing, the saved file will be accessible only through the `OPEN#` command – `LOAD` would fail with 'Wrong file type.'*

Here, any *QL* file-header embedded in the file `win1_ZXFiles_<filename>` is stripped before transmission; it would only confuse the Spectrum. As long as the copied file still includes the Spectrum file-header as the first 9-bytes, it will be recognised correctly by `LOAD` when received on the Spectrum.

In addition to these simple S/Basic commands, you might consider the tools included with the **Speculator** emulator distribution, or else the **QSPEC2** utility suite, by Dave Barker – you may need to hunt them down on the Internet.

1.8.2. Foreign programs and Spectrum BASIC

The QL can't of course run Spectrum (Z80) machine-code programs directly, nor load Spectrum BASIC program-files directly in to the QL's S/Basic interpreter. However, with a suitable Spectrum emulator such as the highly flexible **Speculator** (and its extensive toolkit) by Dave Walker and Simon N Goodwin, or the truly impressive **ZM/x** series of emulators (esp. ZeXceL) by Davide Santachiara & Marco Ternelli of Ergon Development, any program-files transferred to the QL from the Spectrum as described above can, with some additional steps, be loaded in to the emulator and run/developed.

Most of the Spectrum programs you might want to load in the emulator are probably available for download from the various online Spectrum repositories (e.g. World of Spectrum: https://www.worldofspectrum.org/). However, you may have your own code and BASIC programs stored (precariously!) on Spectrum microdrives that you wish to develop further, 'port-over' to QDOS or otherwise enjoy on your QL.

I, for example, developed a number of fairly substantial Spectrum BASIC programs in my youth and would be loathed to have to re-write them from scratch in order to continue their development on my QL.

There are a few utilities available to convert (tokenised) Spectrum BASIC files in to plain ASCII and thus ready for porting to S/Basic, but most of the titles the author finds online are PC-based (ZmakeBAS.exe is a good example.)

However, writing a BASIC de-tokeniser on the QL to do this work would be a relatively straightforward task (using the ZmakeBAS source files as a starting point) – taking care in the process of conversion to S/Basic variants of similar commands. Alternatively, the QSPEC2 utility suite mentioned previously includes a BASIC de-tokeniser/lister command called `SLIST`.

One last note – the Spectrum uses the 'CR' code `CHR$(13)` as its line-terminator rather than the QL's Unix-like approach of using 'LF' `CHR$(10)` and most of the Spectrum word-processor applications seem to stick with CR (or else use fixed line-lengths). Thus, if you plan on transferring *documents* created on the Spectrum, you'll need to patch them to use the QL's line-terminator to make them legible.

1.8.3. Screen Display format

The display-handling between Spectrum and QL is radically different, but there can be a lot of fun to be had in converting the Spectrum's pixel/attribute-based 6912-byte display-file format to the progressive, pixel-based QL 32KB equivalent (best in `Mode 8`) – they make wonderful desktop 'wallpapers' for a QL equipped with the Extended/Pointer Environment and many are available for download from sites such as World Of Spectrum (https://worldofspectrum.org/)

There are a number of such conversion tools available already, including, again QSPEC2's `SVIEW` command - not that this discouraged the author from writing his own from scratch!

For example, I use a QL-rendered screen-dump of Ultimate's iconic "JetPac" as my test-file for exercising the network whilst developing new versions of the NET drivers – it brightens-up an otherwise tedious and repetitive process 😊

In the next part of this series, we shall take a deeper look at the QLAN and TK2 protocols and how they interact with QDOS as well as the NET hardware itself, before going-on to develop the idea of a simple QL game making effective use of the network. We shall close with a presentation about the possible future of networking the QL and its compatibles (and some non-compatibles!)