Acorn Unix Econet Device Driver and Network Device

RISC-iX Group

Acorn Computers, Cambridge, UK

ABSTRACT

This document describes a device driver which controls the Acorn Econet, and a Unix Networking Device which uses Econet, in Acorn's implementation of Unix 4.3BSD, from both the user's view and from that of a programmer who wishes to understand and maintain the code.

1. Assumptions

Familiarity with Unix and C is assumed, and with Unix Ethernet Networking if you are concerned with the Econet implementation thereof, and you should have the file "econet.h" in front of you too. A maintenance programmer will also require "econet.c", "econetmc.h" and "econetmc.s", all of which should be in "src/sys/dev/econet" where "src/sys/GENERIC" is your kernel making directory. See also man (4) eco.

2. General Description of the Econet Device

The econet is a low cost, moderate speed (50-300kBit/S on-net) network for connecting together microcomputers. It allows one machine to transmit a *packet* of data to another which has a suitable *reception* enabled to receive that data. A packet contains at least one byte, and at most 1280 bytes, of data.

2.1. Other reading.

For another treatment of some of the information here, at least what econet is and what it does, see the Acorn documents about Econets with BBC Microcomputers, such as "Econet Advanced User Guide" (the most relevant), "Econet Installation Guide", "Econet Level II (or III) File Server Manager's Guide", "Econet Level II (or III) File Server User Guide", "Econet Bridge Installation Guide" et al.

2.2. What's in a packet?

Associated with a packet, as well as the data, there is a *port number* and a *control byte*, and the addresses of the sender and destination machines. The port number is from 1 to 255, and a reception can select a port to receive packets on. The control byte is from 0 to 127 and is merely a 'comment' field which is available to someone when examining receive blocks without looking in the data. The addresses are each two bytes, a net number from 0 to 127 and a station number from 1 to 254. Receive blocks can specify a particular address, or any address or even 'this net any station' or 'any net this station', though the latter two are not actually thought particularly useful.

Transmission is synchronous in the calling process, using (a copy of) its store to transmit from, and reception is asynchronous into kernel heap store buffers, with a wakeup() of an associated condition when a reception completes. When a client examines the reception to confirm that a

reception did indeed happen, the buffer is marked as reserved so that when he actually reads the data, he is sure to get that which is associated with the reception whose status he read. If a marked or unmarked buffer is not touched for a long time, it will be discarded.

2.3. What else can we do?

There are a small number of other, special operations which the econet can do:

* *Broadcast:* This a special type of packet which can be received by many machines at once; it is addressed to all machines. The receivers see the packet just as if it were a standard transmission just for them. There can be at most eight (8) bytes of data in a broadcast packet.
* *Machine type:* A special operation, directed at a particular machine, which returns data identifying the machine type, version of econet software present and the like. A machine will always respond to this if it is up and running, so it is commonly used as a fast check that a machine exists before trying to send a packet to it.

3. User Interface

Your Unix system should be configured with up to 32 character devices with names of the form "/dev/eco0", "/dev/eco1",.... "/dev/ecou", "/dev/ecov" using 0..9 then a..v as the "digits". These should have major device number 9, and minor device numbers from 0 to 31. The mapping from device names to minor device numbers is utterly insignificant, except that it should be one-to-one.

In order to use the econet from an application, you must open one of these devices in the usual way, using

```
int handle = open(char *name,0,0);
```

These devices are exclusive access, ie each one can only be opened once, and so you must try each in turn until you succeed. Having opened the device, all use of it is via

```
int result = ioctl(handle, /* constant */ OPCODE, &struct_xxx);
```

where struct_xxx is either an integer, struct eco_op or struct eco_stats.

When you have finished using the econet, just close the device and all resources you claimed or were allocated will be disposed of.

3.1. The ioctl() operations

These are described in reasonable detail in the file "econet.h", and the following should be read in conjunction with it.

3.1.1. ECOINITIALISE

This is used by the econetup program at system start time to enable the econet system, specifying the numbers of small and large buffers to allocate. It cannot be used again.

3.1.2. ECOCLAIMPORT, ECOCHOOSEPORT, ECOFREEPORT

These claim a specific port for our use, ask for any old port for our use and free one we obtained already respectively. The "we" means for this minor device, so forked processes with the same handle on just one device have the same view of the econet system's state. You must own a port to be able to enable reception on that port. The argument is just an integer. When freeing a port, any receptions enabled on that port are discarded, as is any buffered data associated therewith, and

if such a reception is the master of a group, the grouping is dissolved (see below).

If you try to claim a port which is already allocated, or ask to choose a port when there are no free ports at all, or try to free a port which you don't own, the error code is EBUSY.

If you attempt any operation with a port number outside the legal range of 1 to 255, the error code is EINVAL.

3.1.3. ECOENABLERECEPTION

This enables a new reception with the station, net and port number you specified. If you are enabling reception on any port, use port number zero, which you cannot and need not claim. Beware that chaos can ensue if you enable reception on any port while other applications are using more specific receptions. To enable reception from any machine, use net and station numbers of 255.

It is possible to "group" receptions together so that you can wait for activity on any one of the group. This is done by quoting, when opening subsequent receptions, the group you got back on opening the first reception of the group. Initially you should supply a group number of zero, ie no group. When the call returns the group will be filled in with the group of this reception. To attach other receptions to the group, keep the number supplied in place, otherwise set it to zero again.

The buffers parameter sets a limit on the number of packets the system will buffer up for you which are attached to this reception. One or two is usually sufficient. Buffered data is not kept forever, but is discarded after some time to allow the system to decongest if a client process stops but doesn't actually die.

If you try to use a port you don't own (zero excepted), the error code is EBUSY.

If you give invalid parameters of any sort, or there already exists a similar but not identical reception which is ambiguous with the one requested, the error is EINVAL. Note that repeated identical requests are not faulted, they are idempotent.

3.1.4. ECODISABLERECEPTION

This disables a reception. You must quote the same parameters you quoted when enabling the reception. It also throws away any data which is still in buffers associated with this reception, and terminates cleanly such a reception going on at the moment, and if the reception is the master of a group, the grouping is dissolved. Further, any other processes which are waiting for activity on any reception in the same group are awakened.

If you quote a port you don't own (zero excepted) the error code is EBUSY.

If no matching reception can be found, the error code is EINVAL.

3.1.5. ECOBUFFERSTATUS

This finds out whether any data have been received associated with the reception specified, and if so it returns specific information about the actual packet received, and a handle to quote to actually read or discard the data. Note that the control byte returned has the top bit set; the actual number quoted when transmitting was this & 0x7F.

If the reception specified doesn't match any currently enabled the error code is EINVAL.

If there is no data buffered on this reception, the data_length and the handle fields will be set to zero, and no error status is set.

3.1.6. ECOBUFFERREAD, ECOBUFFERKILL

These read or discard a buffered packet, which you specify with the handle obtained in an ECOBUFFERSTATUS call. You know the length of the data expected because you were told it above when you did an ECOBUFFERSTATUS to discover this reception.

If the handle quoted doesn't match a buffered packet, the error code is EINVAL.

The error code may also be anything which can be returned by the kernel routine copyout() if the your buffer was deficient in any way. In this case the buffered data will be lost.

3.1.7. ECOAWAITRECEPTION, ECOAWAITRECEPTIONTIMED

These wait, efficiently (via sleep() and wakeup()), for some activity on (any reception in the same group as) the reception specified by station, net and port. The call returns when either the timeout has expired or something *might* have happened to (one of) your reception(s). When it returns you should poll each of your receptions via an ECOBUFFERSTATUS call to see which one(s) had received a packet.

If the reception specified doesn't exist, the error code is EINVAL.

3.1.8. ECOTRANSMIT

You specify the station, net, control byte and port, together with the address and length of the data to transmit. The call will return when either the call has succeeded or it has failed after performing retries, intended to be enough to overcome random effects such as noise, net congestion and the like. If some other process is using the transmitting equipment, the process will block until it becomes free. To perform a broadcast, specify station and net numbers of 255, and a data length of <= 8. Note that the data length parameter is not examined specially for a broadcast, it is just that the transmission simply won't work, possibly in ways that you won't be told about, because of the behaviour of other machines on the network.

If any parameter is bad, the error code is EINVAL.

If there is no network clock signal reaching our machine, the error code is ENETDOWN.

If the destination station is not listening, ie has no reception enabled which is suitable for receiving our packet, or its buffers are congested, or it is simply not switched on or attached to the net, the error code is ENETUNREACH.

If there is an unusual error, such as a packet breaking part way through, the error code is ECONNABORTED (coincidence that this starts with ECO), and more specific information is retained for examination via the ECOSTATS call.

The error code can also be any returnable by the kernel routine copyin(), if your description of the address and length of the data to transmit is deficient in any way.

3.1.9. ECOMACHINETYPE

You quote just the station and net numbers of the machine to interrogate. You are returned one word of data describing that machine as low byte = machine type, 0x0E is Acorn Unix, midlo byte = 0, midhi byte = version number low, high byte = version number high. E.g. 0x0801000E = "Acorn Unix version 8.01". (Version of the econet code that is.) The same error codes as transmit apply.

3.1.10. ECOSTATS, ECOSTATSCLEAR

These return statistics about the network's performance, optionally clearing the counts to zero, and a very specific error code about the last unusual transmission error this device experienced, in the form ((byte offset into driver code) << 16) + ((register code) << 8) + (register value). Register codes are: 1 and 2, status register one and two of the 68B54, code 0 is an unexpected machine address at the start of a packet, code 0xF is an unexpected FIQ.

4. Source Code Structure

There are two modules in the econet system in Unix.

4.1. econet.c

This module is the interface as seen by applications. It implements open(), close() and ioctl() procedures as required by a character device. Everything except open() and close() goes via the ioctl() interface. Transmit and read-received-data operations could have been contrived into read() and write() calls, but it would have been so contrived as to be unpleasant. Therefore read() and write() for this device are both nodev(). It happens that the major device number currently chosen is 9, but this is irrelevent to this code. Minor device numbers must be in the range 0 to 31 - others will be faulted on open.

This module also implements the Unix Econet Networking system, using the device code. The Unix Econet Networking system is documented later.

Exclusive access to each of the 32 minor devices implemented is enforced, so that the minor device number can be used to control use of port numbers. It manages the reception buffer and reception enable structure which the FIQ code crawls over, and contains a timer tick routine which performs wakeup() as required by foreground processes on completion of a FIQ-based operation. It also looks after copying of store on transmit so that it is safely accessible to low level interrupt code. The file "econet.h" details the interface to this world.

The ioctl routine, eco_ioctl() can be called directly by other parts of the kernel; this is the means used to implement Unix networking via the econet. Minor device numbers between 128 and 253 should be quoted in this case; these are required for port allocation and the like, BUT the use of different numbers by different customers is not enforced, as obviously no device open has been performed. It is a matter for documentation and convention to allocate these pseudo-device numbers to such kernel clients as there may be. Minor device 254 is reserved for use by the Unix Networking system, and 255 is the rogue value used to flag a port as unclaimed, so don't use it.

4.2. econetmc.s

This module contains a great deal of relatively simple assembler code which implements the various state machines associated with the passage of an econet packet, and talks to the hardware appropriately. Most of this module is a large number of brief FIQ handlers, which are installed in sequence, each by the previous handler, to perform the econet operation. The file "econetmc.h"

describes the structures common to econetmc.s and econet.c, in forms for both "cc" and "as".

5. A Note about the Hardware

The econet is a bussed network - it connects in parallel to all machines on a particular net, and connected machines need not be powered on for the rest of the net to work. The econet connection medium is two twisted pairs, one for differential clock and one for differential data. These are decoded into a clock and data signal using differential driver/receivers, together with a clock detection circuit and a collision detection circuit. The controller chip is the 68B54 ADLC, which provides support for a checksummed labelled data transfer unit called a *frame*.

Basically what the software sees is as follows:

5.1. When transmitting

The processor enables the transmitter by setting various flags in control registers in the controller. The chip starts to drive the line in an 'active idle' state, so that other machines see that it is in use, and it then causes FIQs to request that two bytes of data be supplied for transmission. The chip will start to transmit the data, requesting further data as its FIFO becomes sufficiently empty. After putting the final byte of the frame in the chip, the CPU can set a flag to indicate that this is the end of the frame. The chip will transmit the remaining data from its FIFO, append two bytes of checksum information and then return to driving the line idle while causing a FIQ to signify successful transmission of the frame. The code which handles this FIQ turns off the transmitter to enable a responder to use the network.

5.2. When receiving

The processor enables the receiver by setting various flags in control registers in the controller. The chip then 'listens' to the line until a frame of data appears on it. When the first byte of the frame has been loaded into the FIFO, the chip causes a FIQ, asserting a flag called 'address present' to tell the CPU that we have a data byte, and it's the first one of a frame. After this, as more data come in, it causes further FIQs to inform the CPU that two bytes of data can be removed from the FIFO. After the last byte of data is removed, the chip sets a flag to tell the CPU as much, having checked the checksum information at the end of the frame.

5.3. When ignoring

If, while receiving, a station decides that it isn't at all interested in the frame, the controller chip can be programmed to ignore all the rest of the frame, and following frames, until the line goes inactive idle. Only then does it cause an interrupt, so that the controller can be reinitialised for another reception.

5.4. Errors

If, during either of the above, the CPU doesn't respond to a FIQ in time and fails to either supply or remove data to/from the FIFO, or another station drives the line at the same time as the transmitter, or the checksum is wrong (receiving end only) due to line noise, the CPU(s) are alerted via a FIQ with an error status flag set in the chip.

5.5. More information

Much more information can be found in the data sheets for the 68B54 from Motorola, particularly a nice green dedicated book (with bigger writing and more pictures than the standard data book) called "An Introduction to Data Communication" MC6854-ADLC Motorola Peripherals. If you wish to maintain or alter the econet code in any substantial way, I call this essential reading.

6. An Econet Packet in Detail

A standard econet packet consists of four frames, exchanged alternately by the transmitter and receiver of the packet. Note the different, more global use of the words 'transmitter' and 'receiver' here. The receiver does indeed pass short frames back to the transmitter to confirm that all is well during the four-way handshake protocol, which goes as follows:

(i) The transmitter sends a scout frame down the net. This contains the destination station address, the source address, the port and control byte.

(ii) The receiving machine, having buffered the scout frame, drives the line actively idle while it examines its enabled receptions and reception buffers. If there are suitable candidates, it remembers them and then sends back a scout acknowledge frame, consisting only of the two stations' addresses.

(iii) The transmitter sends a data frame. This consists of the two addresses and all the data in the packet, in a single frame.

(iv) The receiving machine sends back a final acknowledge frame, consisting only of the two stations' addresses.

Each machine only proceeds to the next stage in this protocol if the previous frame was correct, otherwise it goes right back to its idle state, tidying up suitably, not driving the line at all. The other partner notices this and knows to abort the protocol and tidy up himself. Causes of failures include line noise, other stations erroneously driving the line, or the data being too big to fit in the allocated buffer (which is only detected as the data arrive during the data frame), and there simply being no suitable receive block open in the destination or the destination just being absent or turned off (both of which manifest as no returned scout acknowledge).

The other machines on the net, to whom the packet is not addressed, know it after receiving the first one or two bytes of the scout frame - the destination address - and program their controller chip to ignore all frames until the line is unused. Thus they do nothing until the end of the handshake, after the final acknowledge, when the chip makes a FIQ to tell them the line is now available.

It can be seen that if the transmitter gets the final acknowledge frame successfully, it is guaranteed that the data have been received correctly.

A failure at any time during the protocol is detected by both partners, and so they both know to retry/reset their state as necessary. There is one exception to this rule, which is an unavoidable problem known as 'guarding the last frame'. If the final acknowledge frame gets stomped, say by line noise causing a checksum error, the transmitter will believe there was an error, while the receiver is quite happy. This could be made safe by adding another frame to the end of the protocol, an acknowledge for the final acknowledge, but then that last frame would be unguarded and a similar situation could arise with the roles reversed. We stick to the four-way version because it gives the invariants:

* If a transmitter says it worked, then it is guaranteed that the receiver got the data successfully.

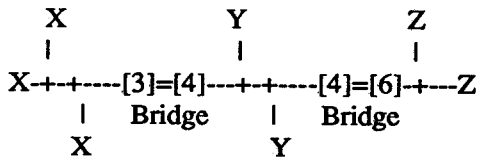
* If a receiver says it received a packet successfully, then the transmitter does not necessarily know that it did so.

This means that, if transmission retries are employed in the event of failure (a generally sound practice), the protocol can only fail in a way that gives rise to repeated packets from a receiver's point of view, and such packets can be detected by associating a simple sequence number with each packet, effectively imposing a higher level protocol atop the reliable datagram service the econet provides (this is the use of the control byte in some Acorn File Server protocols and the reason for its existence).

Conversely, any odd-numbered-way protocol can have the transmitter believing it has succeeded, while the receiver should throw the data away if the final acknowledge acknowledge is corrupt, leading to inaction from both parties and the need for some timeout driven system of concluding that the transmission was actually ignored and trying again. The four-way handshake is preferred.

7. Econet Addressing and Bridges

Having said that an econet address is two bytes, a station number from 1 to 254 and a net number from 0 to 127, a note of explanation about net numbers is in order. Only the station number is bound into a machine (using either NVRAM or links), the net number is determined externally by the *bridge* that links this network to the rest of the world. A bridge is really a small computer with two econet interfaces, one for each net it connects to. One net can have many bridges on it, but they had better agree about its net number, and had better not link to any nets which clash. In other words simply connected webs of networks are OK, loops are prohibited. For example:



In this diagram, net 3 has three stations X on it, net 4 has two stations Y and net 6 has two stations Z. Stations on net 3 *can* talk to stations on net 6, despite it being two 'hops' away, simply by quoting an address with 6 in the net number. It should not be inferred from this diagram that bridges need to be at the ends of a net, nor anything else related to physical layout of the nets.

Now, consider the simplest case of a network with no bridge at all, just one net with a number of stations. It has no net number, so we choose to use zero for the net number. So machines respond to scout frames with their station number and net number zero when receiving, and send net number zero as their address when transmitting.

Suppose you then add a bridge to this network, say denoting it network 3. Traffic from one machine to another on this net will, as before, have net 0 in the addresses. We don't want to change all the software in the every machine on the net, or have it responding to 0 or 3 in a scout, as this would slow down the response time, and require the machine to always know what net it is on. So we keep the rule that addresses referring to this net have net number zero in them, *while on this net.* For this reason, zero is invalid as an actual network number, it means 'this net' wherever you are.

When quoting a station address, you can of course use either the net number of this net or zero - the code notices that you are referring to this net and actually uses zero for the net number in the lower levels. The system determines what net we are on the first time it needs to know by broadcasting to the bridge(s) and waiting for the result. The procedure realnet() does this.

Bridges are store-and-forward devices on a per-frame basis. I.e., first the scout is buffered by the bridge, modified in its store if necessary and then sent out the other side, holding the line active idle on the transmitter's network. It then buffers the scout acknowledge from the receiver's network, modifies that, then sends it back to the transmitter, and so on.

The bridge remembers both the net numbers of its two sides, and the numbers of all the nets which are reachable by further 'hops' on each side. Thus the [3]=[4] bridge above knows that it can also get to net 6 via net 4, and the [4]=[6] bridge knows that net 3 is via net 4.

The modification the bridge performs is as follows, say to a packet received on the left hand side of a bridge:

a) Destination address: if the net number is one reachable via the right hand side of the bridge then buffer the packet for forwarding, and if the net number is that of the actual net on our right, alter it to zero.

b) Source address: if the net number is zero, it came from the very net on our left so alter it to the number of that net.

Thus, for example, a scout from 3.189 (Net 3, station 189) to 3.190 will contain the addresses 0.190,0.189 (Dest,Srce) and the bridge will correctly ignore it.

A scout from 3.189 to 4.254 will contain 4.254,0.189 and the [3]=[4] bridge will take it and modify it to 0.254,3.189 and send it on to net 4 where station 254 will respond to it. The reply, containing 3.189,0.254 will similarly be changed back to 0.189,4.254 when it is echoed back to net 3, and so on.

A scout from 3.189 to 6.254 will contain 6.254,0.189 and the bridge will take it, and forward it, leaving the 6 alone and altering the 0 to 3, ie addresses 6.254,3.189. The other bridge, [4]=[6] will grab this and alter it to 0.254,3.189 and forward it onto net 6, where station 254 will respond.

There is a limit of eight on largest number of "hops" a frame could make on a connected set of networks. If there are more than 8 nets in any possible route, the protocol used by a bridge to ask all the other bridges where nets are falls over, because it uses broadcast packets which have a limit of 8 bytes of data associated with them.

In practice it has been found to be a good plan to have one extremely fast, short network with only bridges on it, connecting a relatively large number of networks together, say all those in one building, in our case nets 1 to 16 (bar a couple which are both connected to net 4) are all connected to backbone net 102, so any route between nets from 1 to 16 is only 3 hops.

(Stn -(1)-> Bridge -(101)-> Bridge -(4)-> Stn).

8. An Example in Detail - Transmitter

This description deals with the internal behaviour of econet_do_transmit() and eco_ticker(), the procedures which do all the housekeeping needed to transmit a packet, and the FIQ code in econetmc which does all the real work of talking to the net.

8.1. The ticker routine and the transmit state machine

Basically, a transmission goes through three states:

(i) Not yet started

(ii) Started

(iii) Finished

When a process makes a call to transmit a packet, single-threaded access to the hardware and static variables is obtained using a busy flag in the usual way, then a static record is initialised to describe the transmission. The transmit code makes one attempt to start the transmit by calling econet_transmit_low_level() (implemented in econetmc.s) and then either marks the current status as started or still waiting, then sleep()s on the transmit record, leaving the ticker code (which is called once per centisecond) to sort it out and wakeup() on the record when it's all over.

The ticker code examines the status of the transmit and, if it has not yet started it makes repeated attempts to start it, with a limit on the number of tries (10 seconds' worth) before we give up, reset the whole low-level system (in desperation) and treat it like any other error. If the net is working this should never happen. When the transmit does start, the next time the ticker code activates it will see this and merely count down another limit (again 10 seconds' worth) while testing whether the transmit has completed. If the transmit doesn't complete in this time, we reset the whole low-level system and treat it like any other error. Again this shouldn't actually happen. After the transmit has completed (as far as the FIQ code is concerned) we either have successful completion or an error of some sort. In the successful case, the transmit state flags are cleared down to idle and we wakeup() the transmit record, thus causing the calling process to restart and continue its work. In the event of an error, we backoff for a short while (1 to 31 centiseconds) and try the whole thing again a number of times, up to 63 for transmit a standard packet and not at all for a machine type enquiry. If we haven't succeeded after all this, the result is returned in the same way as a successful completion.

8.2. The FIQ code: Line access

First of all a transmitter must gain control of the econet itself, without destroying anyone else's traffic, or an incoming reception. The first step in this process is to determine that this station is not receiving a packet at the moment. This is done by examining the value of a variable *rxMode* internal to the asynchronous reception code in econetmc.s, in the procedure econet_transmit_low_level(). When it is zero, our system is 'quiet'. Having waited for the system to be quiet by virtue of the repeated calls to this procedure by the ticker routine, we know we can use the econet hardware on our machine, and can proceed to examine the line, using the controller chip, to see whether it is busy. At this point the 'No Clock' error can be detected, and returned as status immediately. If the network is not busy, we can start the transmission immediately. If the network is busy, i.e. other machines are using it, then we return just as if we were receiving ourselves, and the ticker code will see to trying again later.

8.3. Some macros

To understand this part of the code it is essential to grasp the meaning of three macros:

a) *SetFIQRoutine*. This macro sets the FIQ handler to branch to the label given as its argument.

b) *MakeNextFIQRoutine*. This macro sets up a 'pending' FIQ handler branch to its argument in a store location. The current handler set by SetFIQRoutine is unchanged.

c) *SetNextFIQRoutine*. This macro sets the FIQ handler to branch to the label used in the previous MakeNextFIQRoutine.

These macros, together with a small number of special calls in econet.c, handle the interface with the fiq handling system which permits multiple fiq devices to be active simultaneously.

These are useful for causing 'subroutine' like behaviour in a stream of FIQs. For example (each label a different FIQ):

```
A:  SetFIQRoutine B
    MakeNextFIQRoutine D    ; D is a "return address"

B:  SetFIQRoutine C

C:  SetNextFIQRoutine   ; "pop" the "return address"

D:  SetFIQRoutine E
```

It can be seen that the code in routines B and C can also be used in a different stream of FIQs, without code changes, by, say:

```
X:  SetFIQRoutine B
    MakeNextFIQRoutine Y    ; Y is a "return address"

B:  SetFIQRoutine C

C:  SetNextFIQRoutine   ; "pop" the "return address"

Y:  etc...
```

The hidden 'next FIQ routine' variable is like a short stack (only one entry!), used to allow routines to be shared between many calling contexts. An example of this is code which sends the address of the other machine, then our address out onto the econet. This is the form of the first four bytes of every frame in the four-way handshake, and so the same code is used each time, whether we are transmitting the packet ourself, or replying to scout or data frames during a reception.

To understand the macros fully, read the source.

8.4. The actual transmission

Having decided to start the transmission, the code enables the line driver in the controller chip, sets the FIQ routine to one called TxFromMe, and the next FIQ routine to TxControlAndPort, and then returns. The subsequent FIQs cause transmission of a frame containing the destination address, our address, then in TxControlAndPort, the control and port bytes. The code then enables the receiver, disables the transmitter and waits for the scout acknowledge to come back. The scout acknowledge frame is handled by routines WaitForScoutAck et al, which check that it came from the correct place and then enable the transmitter again to send the data. The transmitter FIQs are handled again by TxFromMe, and then a next FIQ routine called TxStartData et al which transmit the data frame. The code then listens for the final acknowledge, with FIQ handlers WaitForFinalAck et al, which finally reset the system to its quiet state and write status into the transmit control block for the timer tick to notice.

Throughout the code mentioned above there are branches off to other routines and FIQ handler sequences to transmit the other types of packet that are possible (viz. broadcast, machine type operations). These are not documented in any detail, as they are not greatly different from the standard ones, and if you can follow the above with reference to the code, you will be able to understand how the code works in the other cases with reference to the section below dealing therewith.

This may seem hard to follow, but I believe that as there are many simple little routines called one after the other, each handling a couple of bytes of the protocol, it would be worthless to

document each in detail when the code is really quite readable itself.

9. An Example in Detail - Receiver

9.1. Data Structures

There are three data structures used for the reception of data.

9.1.1. The aliveChain

The aliveChain holds a singly linked list of records describing receptions which are currently enabled. The FIQ code looks down this list to determine whether we should respond to a packet from a particular station on a particular port.

9.1.2. largeBuffers and smallBuffers

These are two very similar linked lists of buffers for holding received data. LargeBuffers is a number (obtained at boot time in "/etc/rc.net") of 1280 byte buffers, and smallBuffers is a different number of smaller, 200 byte buffers. The default is 20 small buffers and 5 large ones. During a reception the data are buffered twice, in one buffer from each list (assuming such are available) as long as they fit, and on completion the best fit is kept and marked as received with a pointer to the associated enable record on the aliveChain.

9.2. Specifying a receive block

A receive block has associated with it a number of parameters which are used to select whether a packet can be received into it. These are the 'from' station and the port number of the packet. The 'from' station can be either a specific machine or any machine at all. The port can specify a particular port or any port (in a storm) at all. The usual forms are, for a server type machine, any station, this particular port (the port number being pre-allocated by software designers for use only for a particular service) or, having established communication between two machines, this machine, this port or any port. It can be seen that the reception FIQ code should examine receive blocks in a particular order, more specific receive blocks first, to prevent a malicious/erroneous application opening a receive block for any station, any port, thus denying packets to any other receive blocks in the machine.

The FIQ code, when comparing a scout frame it has buffered, looks down the alive chain for a suitable receive block. Therefore we choose to put blocks which are 'wild' (not completely specified) on the end of the chain, while totally specific (this machine, this port) ones are added at the head of the chain. This prevents packets for specific blocks being 'stolen' by less specific ones. Currently the scheme is very simple, either a block is wild or completely specific. A more complex scheme could be implemented with an insertion point in the middle of the chain for 'partially wild' blocks, after specific ones but before totally wild ones, but this is thought unnecessary and over elaborate, and it would complicate the chain update algorithms excessively.

9.3. The actual reception

By default, when the net is quiet, our machine has the reception FIQ handler installed and the controller chip initialised to receive and make a FIQ when it does. So, when the scout is driven out onto the net by the transmitter, our controller (and everybody else's) sees it and FIQs the CPU to tell it. The standard FIQ handler, StartReception, checks that it is the first byte of a frame, and then checks the station number in the first byte of the frame against our station number. If they are different, it sets the chip to ignore the rest of the packet and FIQ again when the line is free. If they are the same we install a new FIQ handler, CheckNetOfScout.

If the packet may be a broadcast (station number = &FF), a different new FIQ handler is installed, and we go on to receive the broadcast. The train of events for receiving a broadcast is not documented here in any detail, for if you can follow this section, you will be able to work out the function of the code in the others with reference to the section describing the other packet types below.

The next FIQ calls CheckNetOfScout, which does just that, examining the next byte of the scout. Note that all code which deals with our net number (which is always zero to our code, of course) actually compares it to a RAM variable. This is put in so as to facilitate a possible future attempt to build a network bridge with an ARM in it, based on this code. Provided that the net is OK, it then arranges to use BufferScout to place the rest of the scout in a buffer where we can examine it. BufferScout may also used later to receive the actual data portion of the packet. After buffering the data, CheckNetOfScout is called, which 'turns the line around' driving the network actively idle, while looking down the alive chain of receptions to find whether we are enabled to receive such a packet. If so, it then searches the two buffer chains for unused buffers in which to place the data. If the reception isn't enabled, or we can't find any buffer at all, the routine ReceptionError is called, which resets the hardware thus dropping the line, so the transmitter gets a 'Not Listening' error. The selected buffers are remembered and mostly set up for completion, with the actual station, net, port and control bytes written in, and a pointer to the reception block written in too.

Having found a good receive block and one or two buffers, it sends back a scout acknowledge frame using TxFromMe, which then goes to WaitForData et al to get the data into store using either BufferData if both a small and a large buffer are in use, or just BufferScout if there was only one buffer to use. Either BufferData then goes to EndOfDataAnyBuffers, or BufferScout goes to EndOfData1Buffer, which once more uses TxFromMe to send the final acknowledge frame, which then calls EndOfData2 to end the transmitted frame and arrange to reset the world when the frame has completed. It also updates the status in the buffer we are actually going to keep, depending on the length of the data, to RxStatus_Signalling, to indicate that this block has received, but it has not yet signalled its associated condition, if any. The ticker code will notice this as it runs down each buffer chain every tick, and update the status to indicate complete reception and wakeup() anyone waiting if necessary. The reception is now complete.

An error at any stage during this is handled by simply resetting the chip and the receive block and going back to our quiet state, as if the packet weren't actually addressed to us at all.

The same comments apply here as do to the transmission code; this may seem tricky, but the steps taken are really quite simple, it's just that there are rather a lot of them. Also, the re-use of quite a lot of code results in tricky use of flag bits in rxMode to tell common code what's going on so it can do the right thing, but this is again quite simple, there's just rather a lot of cases.

10. What to change to add Econet drivers to a kernel

The device driver has to be included in the table of procedures for character devices in "sys/arm/conf.c", as major device number 9.

The procedure eco_initialise() (or ecprobe()) has to be called from machdep_main() in "sys/arm/machdep.c" after initialisation of the kernel heap is performed via kmem_init().

11. Other Packet Types

11.1. Standard packet

This is only included as an example and for completeness.

I will depict a packet in the form below, which is much like that used on the screen of a net monitor, a program you should obtain and run (on a BBC machine) and learn to love before attempting modification of the unix econet code. It displays the traffic on the net as is, and is a very useful debugging aid. It also gives further status information such as a small letter v before the last byte of a frame indicating frame valid, an i indicating net idle, and others for various kinds of fault. I won't show the letters below except for this first example.

BD00BE00C0v7F BE00BDv00 BD00BE00aabbccdd...xxvyyzz BE00BDv00 i

contains actual data (four frames):

BD00BE00C07F BE00BD00 BD00BE00aabbccdd...xxyyzz BE00BD00

This shows a scout from station 0.190 (&BE) to 0.189 (&BD), with control byte 64 (&40) (the top bit is set in the actual frame, the control byte value is the byte value - &80) and port number 127 (&7F). From now on I will deal only in Hex numbers, as it makes things a great deal easier to picture.

The scout frame is answered by a scout acknowledge containing only the addresses. Next the transmitter sends the actual data frame, the data is aabbccdd....xxyyzz, however many bytes of it there are. The receiver then sends the final acknowledge, identical to the scout acknowledge. The line goes idle and the packet is complete.

11.2. Broadcast

A broadcast is simply an overlong scout with the data in the next (up to) eight bytes (or more on future versions of econet). This shows a broadcast from station 0.190 (&BE) with control byte 1 and port number 7. As more than one station can receive the packet at once, there is no handshaking at all.

FFFFBE008107aabbccddeeffgghh

11.2.1. Implementation note

To receive a broadcast, the data are buffered in the same piece of store as the rest of the scout frame, then copied into place if a suitable receive block is found.

To transmit one, the standard TxFromMe, TxControlAndPort sequence is used followed by TxData to splurge out the data, then we've finished.

11.3. Machine Type

Machine type an extended scout with port number 0, control byte 8, and further data in the scout which is ignored. The returned frame just contains the data we want after the machines' addresses.

BD00BE008800FFFFFFFF BE00BD00aabbccdd

This shows 0.190 asking 0.189 what its machine type is. 0.189 returns four bytes of data, aabbccdd, signifying its type in the format: MachineType, 0, VersionLow, VersionHigh.

12. Unix Networking over Econet

This section assumes familiarity with system management of Unix Networking systems, including ifconfig, arp et al. Read SMM:15 (Unix System Manager's Manual, "4.3BSD Networking Implementation Notes"), section 6.3 describes where Econet fits into the system. See man arp, man 4 arp, man ifconfig etc. etc..

12.1. Where the code attaches to the Unix Kernel

Basically the "ec" device pretends to be a low-level ethernet interface used by a higher ethernet networking layer "if_ether.c", which makes ethernet headers and implements ARP (Address Resolution Protocol) over (it believes) ethernet. "if_ether.c" can control many 'ethernet' interfaces, one of which may in fact be implemented over econet, via the code in "econet.c" rather than the ethernet hardware specific code in "if_et.c".

The initialisation procedure ecprobe() calls if_attach() to add a new interface to if_ether's tables, declaring its output and ioctl procedures. If_ether chooses to call these according to the INET address family of the econet network, which is set up by an ifconfig call in the "/etc/rc.net" file when the system goes multi-user.

The ioctl() call deals with bringing the interface up or taking it down, and that's about all.

12.2. Transmitting a Packet

The ecoutput() procedure basically takes a socket address containing a packet type and address, and a list of mbufs containing data to transmit. For Internet packets, it retrieves the INET address of the destination from the socket address and calls arpresolve() (implemented in if_ether) to turn it into a hardware address via the arp table. Arp packets contain a hardware address already. These hardware addresses are six byte quantities, like ethernet hardware addresses, but econet only uses bytes 4 and 5 as station and net numbers respectively. Bytes 0 to 3 should be zero. Real net numbers rather than zero for local net are used, and translated to zero and back if necessary. It then queues the transmission to be started in turn, when the net is free, and calls trytx() to attempt starting a transmit from the queue. If the INET address wasn't resolved by arpresolve(), we just return, as arpresolve() will sort out retransmission after address resolution has occurred via transmission and reception of arp packets. Arp packets which are recognised as being of the standard form are compressed into only 8 bytes of data so that they will fit into an econet broadcast. Broadcasts of other types are carried out by broadcasting a very small packet flagging that we have broadcast data available, while we place the data in a static buffer to hand out to stations which request it as a result of the flag broadcast. There is a limit to the frequency with which a broadcast can be 'transmitted', to avoid congesting the network with the resulting handshaking packets.

12.3. Reception

A ticker routine, ec_ticker(), called once every centisecond, examines our reception, and if there is some data, it ascertains the type of the packet, copies the data into mbufs and makes calls to add it to the correct queue for that packet type. The ticker routine also tries to start any queued transmissions via the trytx() code. It also carries out the unpacking required for the specially handled standard form arp packets, and also responds to broadcast flag packets by queueing a broadcast request packet, and responds to broadcast request packets by queueing a point-to-point packet

containing the broadcast data.

12.4. Detailed Packet Formats

12.4.1. General

All Econet packets used for Unix networking use the same econet port number, int ec_port = EC_PORT = 0xD2, which is one of those reserved for Acorn use. All calls to the econet device code use minor device number EC_MINOR_DEVICE = 254. The packet type is distinguished using the control byte. Values for the control byte are as follows:

```
/* things I put in the control byte */
/* standard ip packet */
#define ECOTYPE_IP        (0x01)
/* this is for "unusual" arp packets that I can't shorthand */
#define ECOTYPE_ARP     (0x0A)
#define ECOTYPE_REVARP     (0x09)
/* these are for shorthand arp packets */
#define ECOTYPE_ARP_SPECIALS   (0x20)
#define ECOTYPE_ARP_REQUEST   (0x20 + ARPOP_REQUEST)
#define ECOTYPE_ARP_REPLY       (0x20 + ARPOP_REPLY)
#define ECOTYPE_REVARP_REQUEST    (0x20 + REVARP_REQUEST)
#define ECOTYPE_REVARP_REPLY        (0x20 + REVARP_REPLY)
/* these are for handshaking out a cheated broadcast */
#define ECOTYPE_BCFLAG      (0x0F)
#define ECOTYPE_BCREQ       (0x0E)
#define ECOTYPE_BC      (0x0B) /* says get type from broadcast_type */
```

12.4.2. IP, ARP, REVARP

All the data found on the mbuf chain I was given is transmitted, unaltered, in a single packet if the address is a specific machine, with the control byte set to ECOTYPE_IP, ECOTYPE_ARP or ECOTYPE_REVARP as appropriate. If the address is the broadcast address, the data is copied into the broadcast_buffer, if it is free, and a broadcast flag packet is transmitted instead.

12.4.3. BCFLAG

An econet broadcast with a single irrelevent byte of data is transmitted with control byte ECOTYPE_BCFLAG to indicate that I have broadcast data available.

12.4.4. BCREQ

When I receive a Broadcast Flag (BCFLAG) packet, I send a point to point packet back with a single irrelevent byte of data and the control byte set to ECOTYPE_BCREQ. The hardware address of the sender of the flag is available, so I know where to send the request.

12.4.5. BC

ECOTYPE_BC never appears in any packet on the network. It is used to flag to the transmission dequeueing code that the broadcast_buffer should be transmitted, with packet type obtained from the static int broadcast_type. The ticker code queues such a request on reception of a Broadcast Request (BCREQ) packet, causing the actual broadcast data to be transmitted point-to-point to a station which indicated a desire to receive it. When that station receives the data, it is indistinguishable from a standard packet from the point of view of the econet code, and is demultiplexed accordingly.

12.4.6. ARP_REQUEST, ARP_REPLY, REVARP_REQUEST, REVARP_REPLY

These packets all denote a shorthand arp packet. An arp packet as seen by if_ether, the layer above the econet code, consists of an 8 byte header, whose final byte is the arp opcode, followed by six bytes of sender hardware address, four bytes of sender protocol (INET) address, six bytes of target hardware address, four bytes of target protocol (INET) address. All the six byte hardware addresses used by the econet system consist of four zero bytes, then the econet station number, then the econet net number. When an econet packet of any type is received, both hardware address fields may be synthesized from information available in the reception record, as we always know who sent the packet, and we know who we are. Therefore, provided that the first seven bytes of the 8 byte header match a proforma header that I know about, I need only send the two four-byte INET address fields, and the recipient can make up the rest of the packet to hand to the upper layer itself. The eighth byte of the header, the arp op, is used to choose the control byte value used in the econet packet, and on reception it is used to reconstruct a header including the same opcode.

Note that in the broadcast case (type ARP_REQUEST), the reconstructed target hardware address does not necessarily match the target protocol address, but this does not matter as the layer above either discards the packet if is does not already know the hardware address of the target, or fills in the hardware address itself in the reply, as the point of the ARP_REQUEST broadcast is for the transmitter to discover the hardware address for a given INET address. I believe that this means that replies describing 'published' hardware addresses (see man 4 arp, man 8 arp) will substitute the wrong hardware address on reception, but I don't know whether this requires attention. It is also unclear whether this system supports reverse arp, but I believe the code as is, does.

13. What has been Achieved

13.1. A summary:

13.1.1. The Device

We have an econet device with a complete set of ioctl() operations to allow implementation of any existing econet protocol, suitable for communication with existing Acorn FileServers, PrintServers and so on, and suitable for allowing other Acorn machines to communicate with applications running on Acorn Unix machines to emulate any of these existing Acorn Servers.

13.1.2. Unix Networking

We have a Unix network device which is equivalent to a 10MBit/s ethernet device except for the performance, particularly with internet broadcasts. It implements standard internet packets and 'normal' types of arp packets efficiently, and internet broadcast packets less efficiently via a flag and request mechanism.

I have a Unix machine with no ethernet interface at all which boots, NFS mounts the usual set of file systems from our VAX, and then operates exactly as does any machine on the ethernet. The performance is slower than the ethernet, but not by the expected factor of 50-100 (from the on-net data rates) but by between 1 to 5 depending on the econet load and application. See the document "Acorn Unix Networking over Econet: Performance" for further details.

The econet only machine communicates with the VAX (which has no econet interface) via a third machine (of which there may be more than one) which has both types of interface and is nominated for this duty in both machines, by

route add net <VAX name> <gateway name on econet> 1

in the econet only machine and

route add net <econet machine name> <gateway name on ethernet> 1

in the VAX. The gateway machine automatically forwards the packets to the right place.

13.2. Testing

The main test vehicle is the 'basher', a program consisting of a number of transmitting processes which transmit packets to a partner bashing machine and wait to receive them back subtlely modified, and one echoing process which waits for packets of the types sent by the other machine's transmitters and echoes them back modified as necessary. The basher is fully symmetrical, and typically runs with 3 transmitters each machine, ie three full duplex channels in each direction. Packet sizes are cycled from 1 to 256 or 1 to 1280 depending on whether a 'big' basher is requested, the control byte is independently cycled, each transmitter has its own sending port and different receiving port, and every byte of the echoed data is checked in the transmitter for correctness, the data being a pattern depending on the port, control byte, packet length and position within the data (to detect repetition, transposition or omission). This tests all the bandwidth of the transmitting interface. The echoing process tests the reception code by enabling a number (typically larger than necessary) of any station, specific port receptions, grouped together and then waiting for a reception on each reception of the group in turn. The transmitters cancel and reenable their reception, which is station-specific and port-specific, each packet to test for storage leaks and so on.

There is also a program called 'machlist' which does a machine type packet for machines from 1 to 254, printing the answer if the machine exists, thus generating a machine list.

The basher will run for days.

A shell script which runs the basher in background while repeatedly running machlists in foreground and background simultaneously, to test safe update of structures when many processes are using the system, will run for days.

A shell script which does the above, plus repeatedly formatting a floppy disc, thus using the fiq code heavily, will run for hours.

Running any of the above on a machine or pair of machines via an rlogin over econet, thus bashing the Unix networking simultaneously with device access works for days (the basher prints a lot of text).

Running any of the above on a machine whose main disc space is on the VAX via NFS mounts over econet, so it has to fetch all the objects it runs over econet, works for hours.

In all these cases the time limits are because I stopped the machines because I wanted to do something else, not because the system panicked or hung or failed in any way.

14. Limits to the Implementation

14.1. Features

14.1.1. Internet Broadcasts

The implementation of internet broadcasts via the handshaking mechanism described above is inefficient as it requires the broadcast data to be sent individually to each machine which would receive it were it a proper broadcast. As a response to each machine requires two econet packets (one from each machine), and an econet transmission is started on centisecond timer ticks if there is no other activity, we can respond to at 50-100 machines per second. The time limit before another internet broadcast can start is currently 5 seconds, giving enough time for 250-500 machines to be serviced before another broadcast comes along, and in practice ensuring that one machine cannot swamp the econet with repeated internet broadcasts.

The reason econet broadcasts are limited to 8 bytes is that BBC Computers drive the econet ('turn the line around' - see above) after the 8th byte of a broadcast packet, which is after all an elongated scout frame. It has been found that broadcasts with up to 14 bytes can be made to work on econets even if BBC machines are present by using net 0xFF, station 0 as the destination address, causing BBC machines to ignore the packet, but econet bridges stamp on them at the 20th byte of the frame just like a BBC machine, but 6 bytes later. It is also not thought to be the case that bridges forward such packets correctly even if their total length is less than 20 bytes.

Archimedes machines running RISC OS support longer broadcasts (up to 1k of data), as does the Unix econet device driver. If the bridge code were rewritten to support long broadcasts of both types (0xFFFF and 0xFF00), Unix could use econet broadcasts directly for internet broadcasting, with a consequent improvement in performance.

14.2. Bugs

14.2.1. Mounting

Experiment has shown that the NFS mount code fails if there is more than one network interface active when it is invoked. Thus it works fine on an econet only or ethernet only machine, but if both are active it hangs badly. The workaround is to bring up econet and ifconfig it ready for use in /etc/rc.net, then immediately ifconfig it down:

```
# start unix networking over econet
ifconfig ec0 inet arp -trailers netmask ${NETMASK} ${HOSTNAME}_e
# take it down so that mounting works!!! BUG....
ifconfig ec0 inet down
#
```

and then bring it up again at the end of /etc/rc by

```
ifconfig ec0 inet up
```

The reason for this bug is unknown. If you have attempted a mount with both networks up, which has hung, if you take the econet down it gets better. On a machine which mounts via econet, the same is true of ethernet being up or not.

14.2.2. Reverse ARP and 'Published' addresses

It is not known whether the compressing of ARP packets prevents these from working, as it assumes that the hardware addresses in the arp packet are always those of the transmitter and recipient of the packet. See above.

This has been fixed, at least to the degree that published addresses are known to work, from 10/11/88, by checking that the hardware addresses in the outgoing packet are what is expected, ie those of the two stations participating, or, in the case of a broadcast packet, all zero in the target hardware address field. Only in these cases are the packets compressed, otherwise full size arp packets are used. I believe this allows all types of arp to work efficiently.

14.2.3. Rebooting

If you get back to single user state from having been multi-user, with the econet interface still ifconfiged 'up', trying to restart the system (with the standard rc.net et al) results in a panic because the port we use is already allocated and the reception we use is already enabled etc.. You must take the econet interface down first, freeing the port so you can claim it again later.

This has been fixed from 10/11/88, by first doing a call to free the port, ignoring the result, before claiming it again. This retains the check that it is the unix networking which has this port, but prevents the panic if we already had it. Freeing the port also deletes our reception and any data buffered up on that port, so the world gets a clean start.

14.2.4. Route

Having told the VAX that it had a choice of routes to the econet via ethernet, both with metric 1 (number of hops), when one of those machines was taken away, the system failed to work with "nfs server <VAX> not responding". As soon as I did a "route delete net blah blah" to remove the dead path from the VAX's route table it recovered. It should have sorted itself out without my help. Cause unknown.

15. Finally

The final word is: this code is fairly tricky, involving as it does asynchronous updates of many interesting structures. Do not meddle with it unless you are convinced that you thoroughly grasp the principles and detail involved.