Software-Defined NIC

10baseT full duplex, Mark 1
as distributed in arrayForth® Rev 2b

A software-defined full duplex 10baseT Network Interface Controller is implemented as a team of nodes controlling transmit and receive signal pins directly. The signals are conditioned by minimal electrical interface circuitry. External transmit timing is used. The team is designed to function as a memory-mapped device but may be stripped down for direct use by other node teams. This NIC is supported by the polyFORTH® TCP/IP package on the host chip of the EVB001 Evaluation Board.

Related Documents

IEEE Standard 802.3-2002® or later
DB001: F18A Technology Reference
DB002: G144A12 Chip Reference
DB003: EVB001 Eval Board Reference
DB004: arrayForth User's Guide
DB006: polyFORTH Supplement for G144A12
DB008: polyFORTH TCP/IP Package
AN003: SRAM Control Cluster Mark 1
AB002: 10baseT Line Receiver & Manchester Decoder

Contents

1. Problem Statement

Our chips needed an economical way to communicate with other systems (or people) at usefully high speeds. We chose 10baseT Ethernet because it is ubiquitous and versatile. IPv4 based protocols over Ethernet are usable across a desk or across a solar system, and this suite has proven to be a great enabler of applications.

We chose a goal of 10 Mbit/s communication rates for three main reasons. First, with 100 ns between bits and 50 ns minimum between signal edges, it appeared nontrivial, yet feasible, to "bit-bang" Ethernet using F18 nodes in pipelined processing. Second, we believed it would showcase the unique characteristics of our chips, representing an achievement that is probably infeasible to match on anyone else's chip. Third, our team's experience with implementing critical real-time systems has taught us not to use higher speed network interfaces than are necessary for the application. Using the lowest speed practical network interface minimizes the exposure of such systems to compromise of their real-time functions by having to receive and process excessive packet traffic from other systems. 10baseT is the lowest speed that is supported by virtually all Ethernet networking equipment at this time.

The decision to "bit-bang" rather than to use an external NIC was one that seemed economical. NICs are complex things and we thought controlling the wires should be simpler than controlling a NIC. The NIC chips are expensive and are not as energy-efficient as are ours. In addition, most NICs require many-pin parallel interfaces; we did not wish to commit large numbers of GPIO pins to a solution for this problem.

1.1 Results

This initiative consumed 6 days for designing and building two prototype electrical interfaces, followed by 20 days of software design, implementation and testing over a period of six weeks in early 2012. This is actually not much longer than it has taken to program some commercial NICs, and several of those days would not have been necessary had we chosen to use oscillators instead of ceramic resonators in the initial prototypes.

At the end of this effort we proved that our goals were, in fact, feasible. We were able to demonstrate the software-defined NIC running 24x7 in concert with the polyFORTH networking package and communicating properly with remote systems on a publicly accessible IPv4 network.

To validate integration with high level protocols and application code in a single chip, the NIC was configured to operate as a memory mastering device, communicating with the polyFORTH system and its TCP/IP package using data structures in external SRAM as do typical commercial NICs.

The resulting system is robust; for a matter of years, multiple boards have been run 24x7 without reset except when moving them or intentionally disrupting power, and the longest periods of continuous operation have exceeded one year. We present this system as a comprehensive example of using our array of computers to solve a problem demanding pipelined, parallel processing.

The next section discusses the implementation of the NIC team. The F18 software necessary to compile and load this team is included in the current arrayForth release. It also makes suggestions for adapting the software to other environments in which other native F18 node teams might directly transmit and receive extremely simple protocols without using polyFORTH or external SRAM.

The third section discusses the E-NET piggy-back board, available in limited quantities, as an example of electrical and physical interface to a 10baseT line. It shows how to install this board on an EVB001 evaluation kit.

The fourth section shows how to configure the software to use the E-NET board. This is necessarily a simplified discussion. Please refer to DB008 (polyFORTH TCP/IP Package) for complete configuration and operation procedures.

2. Implementation

This App Note documents the implementation of a software-defined full duplex 10baseT Network Interface Controller. Single transmit and receive pins are bit-banged by software. External hardware is minimal: A conventional transformer with integrated filters is required. The receive line has a termination resistor across the twisted pair and a series current limiting resistor connected to a GPIO pin for input. The transmit pin drives a pair of op-amps to provide the necessary drive voltage for the transformer, designed to minimize power when the transmitter is idle. The implementation is a team of 26 nodes, of which four are simply wire. Various means of transmit timing are supported. The team is designed to function as a memory-mapped device but may be stripped down for direct use by other node teams. This NIC is supported by polyFORTH on the host chip of the EVB001 Evaluation Board.

2.1 High Level Interface using Shared Memory Structure

When this team is fully implemented, it communicates with the SRAM Control Cluster Mark 1 as a memory master. The primary interface with high level code uses a table in external SRAM organized as follows:

Name	Content	
t.cm	Command to TX pipeline, zero when taken.	
t.pf	^ next descr to process, = t.rx if none	Rx Descriptor Pool
t.rx	^ next descr to be filled by RX, = t.ep if none	
t.ep	^ next empty descriptor for freed buffer	
t.lk	Latest Link Status Word	
t.dp	32-bit count of packets dropped for no buffers	
t.sk	^ USER AREA TO AWAKEN for TX/RX completions	
t.wk	Value of WAKE to store into user areas	
t.xa	20-bit TX buffer address	
t.xn	Length of TX buffer in octets; negative to force link down; 0 when done	
t.pp	Poll period for commands	
t.tt	^ USER AREA TO AWAKEN for timer prodding	

The addresses of the elements in this table are set by mutual agreement in the source code for both the firmware (in the source code for node 108) and the high level polyFORTH system (in block 25 where it is allocated in high memory.) This method dramatically simplifies the firmware by eliminating an entire layer of indirection.

The Rx Descriptor pool is an array of 4-cell descriptors with address and size set by polyFORTH such that modulo addressing within it may be achieved simply by ANDing an incremented address with the mutually agreed mask value **rxdm** defined in the firmware source code for node 108 and in polyFORTH block 583. Another simplification is the rule that there must exist at least one more descriptor than the number of physical buffer structures, thus eliminating a classical problem with distinguishing a full descriptor table from an empty one. A descriptor looks like this:

Ofs	Content
+0	High 4 bits of 20-bit packet store address
+1	Low 16 bits of 20-bit packet store address
+2	16-bit address of TCP/IP package Buffer Structure
+3	Not used

The use of these structures to control the NIC is explained in the next section.

2.1.1 Control Procedures

The NIC is **disabled** on boot and may also be disabled under host control. In this state, the NIC refrains from any DMA operations and discards all record of whatever it may have previously seen in the shared memory structure until it is enabled again. Therefore, it is safe for the host system to do anything it wishes in memory, such as rebooting.

2.1.1.1 Enabling the NIC

First, initialize the shared memory structure and Rx Descriptor array. Everything should be zero except:

t.pf **t.rx** and **t.ep** should point at valid positions in the descriptor array. For example if there is one less buffer than the number of descriptors *n* , descriptors 0 through *n-2* might describe the *n-1* extant buffers and descriptor *n-1* would be empty. t.pf and t.rx would in this case point at descriptor 0, and t.ep would point at descriptor *n-1* . If no buffers have been described yet, you might set all three to descriptor 0.

t.sk should point to the STATUS cell for the task to awaken on transmit / receive completion.

t.wk should hold the WAKE value to be stored into STATUS cells for awakening tasks.

t.pp sets the polling period for commands; 100 is the smallest recommended value, 10000 is a relaxed value that generates little polling overhead.

t.tt should point to the STATUS cell for the TCP timer task which will typically be awakened at the frequency of SLP heartbeats when the link is up, or at roughly 10 Hz when the link is down.

Then unmask the port used by the NIC in the SRAM cluster using !MMASK and pass it a stimulus using !MMSTIM . This will start the DMA Nexus and slaves working normally, polling for commands but preventing packet reception.

When packet reception is enabled, incoming packets are counted in t.dp if no buffers are available (t.rx = t.ep). Otherwise a packet is stored into the buffer descriptor at t.rx and when finished the NIC increments t.rx and awakens the task t.sk . If you find t.pf not equal to t.rx you may process the t.pf descriptor and increment t.pf . When you wish to make another buffer available to the NIC, you may use the descriptor at t.ep and increment t.ep . You own t.pf and t.ep ; the NIC owns t.rx . As long as the number of buffers is less than the number of descriptors there are no other special cases you need to worry about.

2.1.1.2 Host Commands

Once the NIC has been enabled, t.cm zero indicates that it's ready for a command to be stored into t.cm . The enabled NIC polls t.cm for nonzero command values and upon finding one sets t.cm back to zero and executes the command. *Only* the following exact values may be used:

x8011 (+RX) Fully enables the NIC for packet reception.

x0011 (-RX) Disables packet reception but leaves the NIC enabled.

x0015 (-DMA) Completely disables the NIC.

x0032 (+tx) May be used any time t.xn is zero. Sends t.xa and t.xn as a command to the Tx Control. Command is executed and when done t.xn is set to zero and the task t.sk is awakened. Three commands are defined:

- **t.xn = -1** Forces link down and autonegotiation.
- **t.xn = 0** Completely ignored (task is not awakened)
- **t.xn > 0** Transmits a packet starting at t.xa whose length, excluding CRC, is t.xn octets.

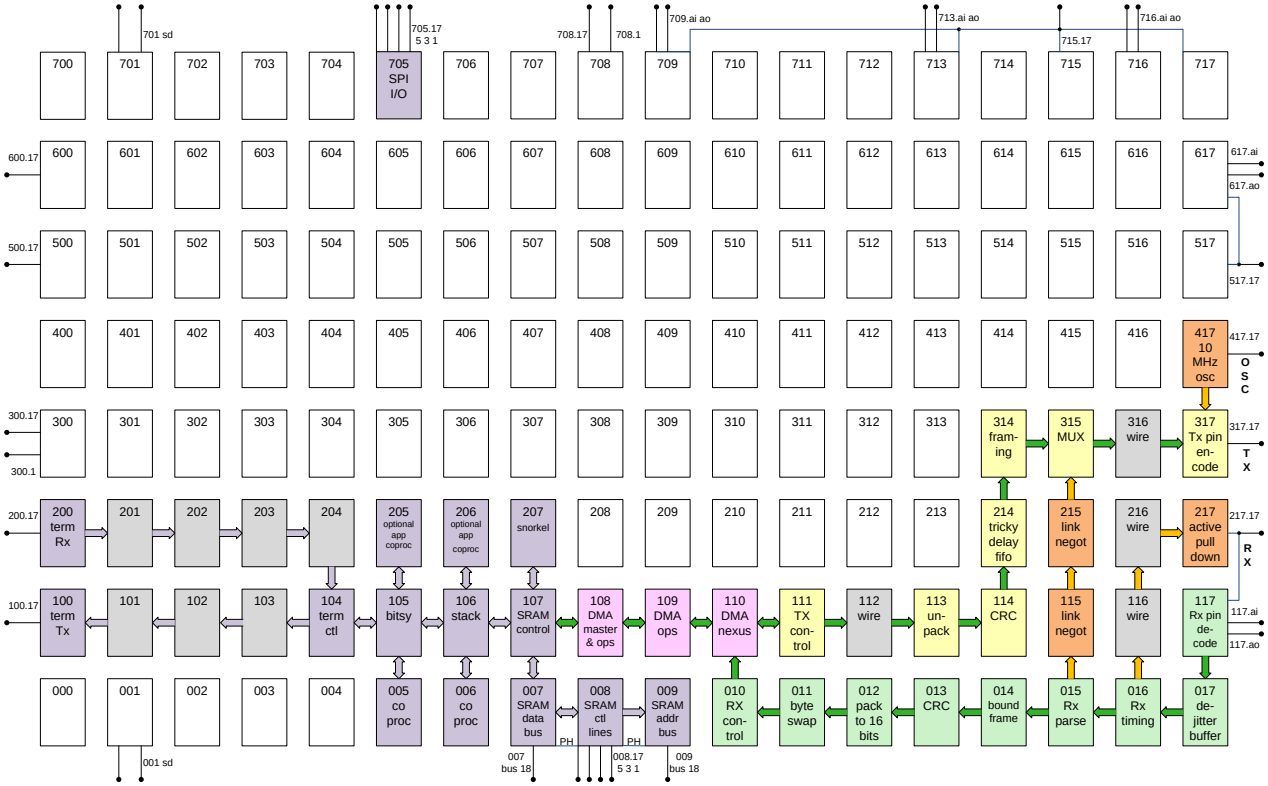
To bring the NIC the rest of the way up after enabling it, send +RX and force link down to autonegotiate a fresh link.

To shut the NIC down in an orderly fashion, for example to reboot a system, use the following sequence:

1. Stop sending packets.
2. Use -RX and wait long enough to fully receive a packet (one second is more than enough)
3. Use -DMA and proceed.

2.2 Chip Floorplan

The diagram below shows our employment of the nodes in a G144A12 when the NIC is connected to the SRAM Control Cluster (light purple) and is used by polyFORTH Virtual Machine (also light purple) for this purpose.



10baseT NIC Mark 1 with polyFORTH

Three nodes (108, 109, 110) shown in pink handle bidirectional DMA operations with the shared memory control structure and with the packet buffers. Because the NIC runs full duplex, these nodes allow interleaved operations on behalf of the outbound and inbound data paths.

The outbound path is shown in yellow, with two wire nodes. Transmit signal element timing comes from a 10 MHz oscillator connected to node 417. Node 315 multiplexes signals from the outbound data path and automatic link negotiation signals produced and delivered by nodes 115 and 215.

The inbound path is shown in green, using nodes 117 and 017 through 010. The two wire nodes 116 and 216 and the active pull-down node 217 are necessary because, while the standard suggests that link pulses involve a positive and negative signal transition, most link partners (including us) do not actually produce the negative phase; this active pull-down ensures that node 117 sees the signal going low so that it may again wait for a positive edge.

Although the polyFORTH timing signal is not shown, when this NIC is active the system is normally configured to use a 10 MHz clock obtained by having nodes 516 and 517 witness the 10 MHz signal on pin 417.17. This clock is normally interrogated by high level code using the Snorkel/Ganglion mechanism. Nothing prevents the system from using some other timing signal if it serves the application better, but the TCP/IP RFCs require that the system have *some* sort of timing signal to meet timing requirements.

The power used by the NIC is dominated by three nodes that must run continuously. These are 110 and 315, which must poll **io** because they receive unsynchronized stimuli from multiple places, and 016 which must continually measure the time between incoming bits.

2.3 Inbound Data Path

Reception begins with the node that listens to the Rx data pin and ends with the DMA Nexus in node 110. Its primary task is recognition of incoming Ethernet frames and either storing them in external SRAM or disposing of them when there are no buffers available. Packet length is checked for legality and CRC is calculated to detect line errors. In addition link pulses are recognized and interpreted, generating stimuli for a side path (nodes 115 and 215) that handle link negotiation, always insisting on 10baseT Full Duplex with no flow control. Unlike most if not all other Ethernet receivers, this one does not require a clock.

This section follows the incoming signals from the pin, describing what each node in the pipeline does, what signals are moved between the nodes, and why the implementation was done in this manner. Note that this is generally the reverse of the compilation order, because in many cases we pass messages in the direction of data flow via port execution, which requires that the recipient be compiled before the sender so that the sender has symbolic access to the desired places in recipient's code.

Refer to these load descriptors for initial conditions in each node of the receiver:

```
722 list
- load descriptor,
,
rx 117 +node 117 /ram io /b left /a,
....5555 5D55 left down 5555 5D55 left down,
....5555 5D55 10 /stack 5 /p,
..17 +node 17 /ram 200 /p,
..116 +node 116 /ram down /a up /b 0 /p,
..216 +node 116 /ram up /a right /b 0 /p,
..217 +node 217 /ram right /a io /b 0 /p,
tim 16 +node 16 /ram right /a io /b 3C /p,
prs 15 +node 15 /ram left /b right /a left /p,
frm 14 +node 14 /ram right /a left /b 4 /p,
crc 13 +node 13 /ram left /a right /b 212 /p,
pak 12 +node 12 /ram right /a left /b 0 /p,
swp 11 +node 11 /ram left /a right /b 0 /p,
ctl 10 +node 10 /ram right /a down /b 23 /p,
(...)
```

2.3.1 Node 117: Rx Pin Manchester Decode

A single pin (217.17) is programmed as a high-impedance input (see the code for node 217 below.) The receiver itself lives in node 117, which is actually an analog node; 117.17 is a shared, read-only pin mirroring what's on 217.17. The analog output pin 117.ao is used, in the default configuration, as an enable signal for the 10 MHz oscillator; this code leaves that signal at "zero" meaning that no current is being sourced, letting the oscillator's pull-up resistor take that pin high. To actually turn off the oscillator would require the addition of a pull-down resistor. The code for node 117 is shown below, combined with the code for node 017:

```
the ethernet rx pin node acts as a continuous
manchester decoder, recovering and passing,
each data bit to an adjacent buffering node,
to eliminate jitter in its timing sequence.,
this code automatically acquires bit sync,
during frame preambles. when a link pulse is,
positive only it decodes as a single '1' in,
which case another node must drive the pin low
momentarily to reset the schmitts; this may,
produce a spurious '0'.,
,
edge samples pin and waits for opposite state.
run waits for the edge at the center of a bit
cell, writes the value of io with the data bit
in position 17 to the buffering node, then,
delays long enough that the next edge call,
will be well centered in the first half of the
next bit cell, maximizing margins for sampling
and waiting. it is paramount that the delay,
never be too long since that means loss of bit
sync. duty cycle is always lt 100 pct.
```

```
772 list
117/017 rx pin 117 node 0 org reclaim
edge rise fall 00 push @b -if,
.....pop !b ! . drop . . ;,
then over !b ! . pop drop drop ;,
,
run 05 edge a! @b ! orig 4,
...best-42 5 ..best-145 9 push a!,
...begin unext run ; 0A,
,
17 buffers for 16 and converts bits to 1/0,
17 node 0 org +cy,
wire 200 down a! right b! 20000 clc
pass 206 @ over and dup . + dup . + !b pass ;
20A reclaim
```

Functionally, this node decodes incoming bits and passes them to node 017 in the high order (sign) bit of a word that is otherwise garbage (the entire content of **io**).

Note that initially **B**=**io** **A**=**left** **P**=**5**(run) and the stack holds a repeating pattern of **left down 5555 5d55** . Both of these values disable the ADC VCO to save energy and set the DAC to high impedance (zero value). 5555 sets wakeup direction for pin high, 5D55 for pin low.

Manchester coding involves a stream of bits represented as edges occurring at a given interval between edges (100 ns in this case). A positive going edge means 1 while a negative going edge means 0. These are the edges that "matter". In order to encode two consecutive 1s it is necessary to insert an "overhead" negative going edge between the two positive going edges that "matter"... 50 ns after the first that matters, and 50 ns before the second. To correctly determine which edges matter, the Ethernet preamble consists of 62 bits of alternating 1s and 0s. During the preamble *there are only edges that matter*. This permits the receiver to lock on using one means or another. Most Ethernet receivers use this period to finely adjust their internal oscillators and use the output of such oscillator for sampling the waveform, potentially drifting on long packets. We do it much more simply.

The behavior of the routines **run** and **edge** is described in the shadows. Thanks to the nicely defined preamble and framing of an Ethernet packet, synchronization is automatic and does not require any additional statefulness in the code. This simple, 10-word program is a complete Manchester decoder that does not depend on a precision time base to receive a long packet. The packet length actually has no effect on its ability to stay synchronized, because it locks onto the incoming signal at each bit-encoding edge. *Note that we use ! instead of @ to wait for the shared pin. This is necessary because analog nodes are set to wait on pin write rather than pin read, as in a GPIO node.*

The default time delay value 4 in run may need adjustment. The three values shown (4, 5, 9) correspond with three different prototypes of the external electrical interface, and the first two of these had long wires connecting the Rx signal to the pin, with much parasitic LC. More experience with the E-NET board should yield a single value. This is the only really critical time delay in this application.

Please refer to AB002 for additional information about this code including scope traces.

2.3.2 Node 017: De-Jitter Buffer

It's imperative that, whenever node 017 decodes a bit and writes it to a COM port, the recipient is already reading the port so that the time delay to dispose of the value does not vary other than by PVT. There's a good deal of decision making involved in the next steps of processing each incoming bit. If any part of that were to "back up" enough that the first node was not reading before node 017 wrote, this would cause jitter in node 017's timing, and compromise bit synchronization. In such a case, inserting one buffering node in the timing sequence relieves the tight timing pressure on the following nodes. Node 017 is *almost* a wire node; however, because it has 100 ns to work with, we gave it a modest task to perform by masking off the sign bit received from 117 and shifting it into bit position 0. Note that the node runs in extended arithmetic mode to facilitate this circular shift.

Functionally, node 017 passes a sequence of bits to node 016 as numeric values (0 and 1).

2.3.3 Node 016: Rx Timing

The next step in analysis of the incoming bit stream is recognition of link pulses, start-of-packet, data bits, and end-of-packet. Discrimination between these things must be based on the time following each. Zero (low going) bits are only interesting as data; all the other events occur on 1 (high going) bits.

```
node 16 times events from the receiver, dis-,
criminating data, flp, slp and link-down,
timeout. events passed to node 15 are,
...ldown call link down, always sent first.,
...-flp call start of flp word,
......flp0 call flp 0 bit,
......flp1 call flp 1 bit,
......slp call slow link pulse or end word,
...sdata call packet follows...,
......0 data bit zero,
......1 data bit 1,
......negative end of packet.,
yank messages node 116 to enable 217 yank-down
..of pin if it stays high several bit times.
?lp waits for a link pulse up to given number
..of usec. returns us waited, 0 if no pulse.
send passes an event word to node 015.
pkt sends sdata followed by 0/1 events till,
..end of packet.
idle is the non-stateful observer.
!data has used 100 ns by the time,
..loop starts so 5 iterations gets,
..us 2 bit times, plenty for idle,
..w/o dribbling.
```

```
768 list
016 rx timing 16 node 0 org,
00 0 org
yank 00 a down a! dup ! a! ;
?lp us-us 03 dup for 1us 48 for 20.7ns,
..@b 2* 2* -if drop @ 09/2 if drop,
....pop drop pop - . + ; then then,
drop next next nada dup or ;
send n 0F a left a! over ! a! drop ;
pkt 13 @p .. sdata 15
!data send 1bt 4 for @b 2* 2* -if @ !data ;,
..then next - yank idle ahead
lkp us-us 1E ?lp if 0 ?lp if pkt,
..then drop then yank ;
idle 26 then send 100000 lkp if @p .. -flp
flps 2C send 150 lkp if -100 . + -if,
..33 100 lkp drop .. @p flps ; flp1,
..38 then @p flps ; flp0,
3A then @p idle ; slp,
ent 3C then yank @p idle ; ldown,
3F reclaim
```

Functionally, node 016 receives 0 and 1 bits from node 017, using internal time delays to discriminate between fast link pulses, slow link pulses, link down, start-of-packet, data bits, and end-of-packet. "Events" are passed to node 015 as instructions calling routines in node 015, or as 0/1/neg for data bits and end packet. In addition, node 116 is messaged with a word of garbage when active pull-down should be considered. The following comments expand upon the shadows above and assume you've read the shadows already.

`ent` is the entry point and link down routine. It pulls the Rx line down, sends the **`ldown`** event and goes **`idle`** .

`idle` waits for line activity (packets or link pulses). The link pulses must occur periodically during idle time on a link that is validly connected to a partner. When a link pulse or data is not seen for approximately 100 ms, it falls through to the **`ent`** routine to take the link down on our side (the other side, if not already down, will go down when we have stopped sending link pulses ourselves.)

`lkp` is used by **`idle`** to wait for a 1-bit, signifying either a link pulse or start-of-packet, within the given number of microseconds. If another 1-bit is seen within about one microsecond, this must be a data packet so we in effect jump off to **`pkt`** which ends by jumping to **`idle`** for sending the end-of-packet value. Otherwise it sends the **`-flp`** event and enters **`flps`** ...

`flps` sends the starting **`-flp`** event and waits to see if more pulses follow. Pulses occurring within the approximate timing margins for fast link pulses (111 to 139 µs between clock pulses, data pulse 55.5 to 69.5 µs after clock pulse, total of 17 clock pulses) are sent as **`flp0`** or **`flp1`** events. When there is no pulse within about 150 µs, this marks the end of a sequence of zero or more fast link pulses so it sends the **`slp`** event and goes **`idle`** . Thus, a fast link pulse message sends **`-flp`** followed by one or more of **`flp0`** or **`flp1`** and **`slp`**. An isolated slow link pulse is simply **`-flp`** followed by **`slp`** .

The timings employed by this node are not critical because the intervals distinguishing data bits, fast and slow link pulses are matters of scale, not of fine values. Because this node must detect an idle line, it is logically necessary that it must spin; so it is running, and using power, continuously.

2.3.3.1 Nodes 116, 216, 217: Active Pull-Down

The standards require a link pulse to drive the differential line high but then allow the line to simply be relaxed rather than being actively driven low. Because our GPIO pins have hysteresis due to their Schmitt triggers (see DB001 and

DB002 for details), we require a good solid negative going pulse to reset the Schmitt so we can once again perceive a positive going edge. Therefore we must be prepared to actively pull the Rx line down after any link pulse, including an end-of-packet.

```
when a device sends only the positive going,
half of a link pulse, we must yank the pin low
ourselves to reset the schmitt so we can see,
the next positive going link pulse.,
,
node 016 sends a word of garbage 116-216-217 a
fter seeing a link or end packet pulse.,
,
node 217 responds by waiting 2 bit times to,
let an idle pulse finish, then yanking the pin
if it has remained high all that time. the,
delay loop is nominally 18.1 ns/cycle.,
,
note that 15555 is used in init to leave the,
pin in weak pulldown as a measure to reduce,
incoming noise. ideally that is taken care of
outside and more symmetrically so that we can
leave the pin at high z.
```

```
770 list
217/216/116 rx active pull-down,
,
217 node 0 org reclaim
init 00 25555 5555 15555 over over over over,
..over over over over hi z !b
yanker 06 @,
..9 for 08 drop @b -while . . next 0A,
drop !b !b yanker ;,
then drop yanker ; 0D,
,
minimal latency a-b wire,
116 node 0 org
wir 00 begin begin @ !b unext unext wir ;,
02 reclaim
```

Nodes 116 and 216 use the same code (above) with different initial registers, simply passing the signal from 016 (a word of garbage) into node 217 which is running **yanker** .

yanker waits for the stimulus from node 216 and then samples the pin 10 times in a fairly slow loop. If the pin is still high then we conclude it has been left floating and pull it down with a very short drive signal ... enough to change the voltage of the pad and reset the Schmitts, but not necessarily long enough to even be visible as a pulse outside the chip due to RLC parasitic loading on the pin.

We'd originally intended to leave the pin at high impedance but learned empirically that the slight loading of the weak pull-down resistor reduced incoming noise, so the system is shipped using the weak pull-down.

2.3.4 Node 015: Rx Parsing

The next step in this pipeline parses the events detected by node 016 to appropriately handle slow link pulses, fast link pulses, the start and end of Ethernet packets and the data bits comprising them. All timing has been dealt with so this node only needs to execute when actually processing an event. Messages are sent by node 016 as **call** instructions; therefore this node's starting address must be **left** so that it will execute whatever node 016 sends.

```
node 15 parses incoming events from node 15,,
maintaining link state and sending refined,
events to 115 for negotiation and 014 for,
packet data and state changes.,
b left for incoming, a right for outgoing...,
..nz state new link state 0 down, 1 10hd,,
....else last active code word received.,
..zero packet follows...,
....0 data bit zero,
....1 data bit 1,
....negative end of packet.,
link most recent link state, changed when
prv shows two consecutive same states. we need
..a better noise filter than this some day.
negot sends link state value to node 115
report reports link change to host.
?link updates link, sends to negot and sends,
..to node 14 if state has changed. prods timer
..task on 16ms link pulses or 100ms timeouts.,
,
ldown*-flp*flp0*flp1*slp and*sdata,
are called thru the port by node 16
```

```
766 list
015 rx parse 15 node 0 org reclaim,
,
!prv n 00 @p drop !p ;*prv -n 01 0 ;
!link n 03 @p drop !p ;*link -n 04 FFFF ;,
,
negot n-n 06 a down a! over ! a! ;
report n 09 a io a! @ 2* -if drop a! drop ;,
..then drop a! -1 ! ! ;,
,
ldown 10 1 0*slp nw
?link nw over if 2/ if 2* 2* .. -while then,
..wx 17 drop prv over or if drop !prv ; then,
..wx drop link over or .. if drop dup !link,
....negot -1 ! ! report ;,
..then - report then then drop ;
-flp -nw 23 10000 link if ; then drop 1 ;
flp1 n-n 29 10000 or
flp0 n-n 2B 2/ push 2/ pop ;,
,
sdata 2D dup or begin ! @b -until ! ;,
30
```

Functionally, node 015 mainly sends packets, verbatim, to node 014 including preamble and transmitted CRC. These are denoted by a word of zero to indicate start-of-packet, one and zero values to indicate bits, and a negative value to indicate end-of-packet. In addition, incoming link status changes (link words, or link-down) are passed as a nonzero

word followed by the new link state (0 down, 1 10hd, otherwise link code word). The new link state values are passed to node 115 as well for use in negotiation.

The most straightforward message from node 016 is the call to **sdata** indicating the start of a packet. We respond by passing a word of 0 to node 014 and then pass consecutive bits received until we have passed the negative value indicating end-of-packet.

For link pulses we receive two kinds of message sequence: **-flp slp** indicating a slow link pulse, and **-flp [flp0|flp1]*n slp** indicating a code word of *n* bits, where *n* should be 16. The 26 words of code starting with slp down through flp0 decode and process all these combinations as follows:

-flp is guaranteed to start each sequence. It initializes the stack with what is shown as nw in stack effect comments. n starts as the value 10000 in hex and is used to count code word bits received. w is a 16-bit link state that starts as either the current link state from the variable link or as 1 (the value for 10 HD) if the link is presently down. If this is a slow link pulse, these are the values that will be processed by slp if the event was a true slow link pulse. Otherwise we will receive one or more fast link pulses as follows:

flp0 and **flp1** receive consecutive bits of a 16-bit link code word. These are sent LSB first and for a legitimate code word there must be exactly 16 bits. When a bit is received, the value w is shifted right and the new bit goes into its most significant bit position, so that after receiving 16 bits the new code word will be correctly aligned in w. In addition n is shifted right for each bit received. Thus its value will be exactly 1 after 16 bits have been received, or hex 10000 if none were received. Any other values indicate an invalid code word.

slp ends each link pulse sequence; its function is to inspect n and w updating link state variables and generating messages accordingly. The state variables are:

prv holds the value of the last valid link word received (0, 1, or a code word).

link holds the current filtered link state (0, 1, or a code word).

There are four valid inputs to ?link as follows (values of n w are shown):

1 0 indicates link down message received (timed out waiting for link pulses).

10000 1 indicates that an SLP was received when the current link state is "down". This is the expected behavior when connecting with a non-negotiating partner (i.e. 10 HD only).

10000 w indicates that an SLP was received when the current link state is w. This is the expected behavior when a "heartbeat" SLP is received on a successfully negotiated link.

1 w indicates that a 16-bit FLP sequence was received with a code word value of w.

The logic in slp is as follows:

If the length n is invalid (not hex 10000 or 1), the event is ignored. Otherwise, the new value of w is deemed valid...

If it differs from prv the new value is stored into prv and no further action is taken. If it's the same, this means we have seen the value w at least two consecutive times...

If it differs from link the new value is stored into link and it's passed to node 115 for negotiation and reported to node 014 as a new state. Otherwise, there's been no change in link state, so this is taken to be a heartbeat event and is reported to node 014 as such (link state -1).

report is used to pass link status messages to node 014. In some envisioned usage scenarios, link status messages could "back up" in later nodes of the pipeline; to prevent loss of autonegotiation in some of these scenarios, we skip delivering the link status message if node 014 is not reading its COM port when we are ready to send the message.

2.3.5 Node 014: Rx Framing

The next thing to do is to frame the packet by eliminating the preamble and determining its actual data length.

```
node 14 frames packets by stripping preamble .
appending two words at end.,
output messages...,
,
..-1 state new link state 0 down, 1 10hd,,
....else last active code word received.,
,
..0 packet as follows,
....0/1 data bits, multiple of 8, 1 per word,
....10 8000 0000 0000 0000,
....length in bits after preamble including,
......any dribble bits.
```

```
764 list
014 rx frame 14 node 0 org reclaim,
,
runt 00 28040 28000 !b 0 !b
idl 04 @ -if !b @ !b idl ; then !b,
pre 7 for @ -if runt then next,
..55 for @ -if runt then,
....over over and 1-1 if end preamble dup or
pkt @ -if 28000 !b drop !b idl then,
..!b 1 . + pkt ;
-pkt then drop,
..over over or while drop next then runt ;,
1F reclaim
```

Functionally, this node passes 2-word link state messages unchanged. Packets are altered by removing the preamble. Two new words are added to the end of each packet, replacing the single negative end-of-packet marker used as input to this node. The first word is always sent from this node as x28000 and other status bits are turned on, as appropriate, later on in the pipeline. Their definitions are as follows:

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Function
1	0	1	0	0	0	0	0	0	0	0	c	r	l	0	d	d	d	Packet status eventually delivered to SRAM. c = CRC error r = runt (<64 octets in packet) l = too long (>1518 octets) d = no. of dribble bits
Actual packet length in bits after preamble																		Even if too long. Includes any dribble bits.

idl is the main program of this node, waiting for data from node 015. Link state messages are detected and passed to node 013 unchanged. When a start of frame is seen, it is also passed along and we fall through into `pre` .

pre adaptively strips the preamble. A standard preamble consists of 62 consecutive bits of alternating 1 and 0, starting with 1. It's followed by two consecutive 1-bits indicating start of frame delimiter. Noise, weak signals, common mode offset and other factors can shorten the effective length of the preamble and can also delay our capture of synchronization with the bit stream. What we do is to receive 8 bits, ignoring their states, and then receive a maximum of 56 more bits, watching for two consecutive 1-bits. During both of these loops we abort processing and send the status and count words if end-of-packet occurs. If we do not find them within that maximum of 64 bits, the packet is delivered as zero length. If we find the two consecutive ones within that sequence of 56, we fall through into `pkt` which passes data bits, counting them, and appends status and bit count after the last data bit has been copied. Otherwise we jump ahead to `-pkt` where we check for, and stop preamble processing on, two consecutive 0's. ***In the exceptional cases of two 0's in the preamble or of no start of frame delimiter within 64 bits, this code simply jumps to*** `runt` ***which is unsafe; instead it should continue absorbing bits from node 015 until the negative end-of-packet word appears, and only then jump to*** `runt` ***. Until this is fixed in a release later than 02c, node 016 is capable of confusing node 015. It's never caused a problem so far simply because preamble garbage is vanishingly rare in full duplex connections.***

2.3.6 Node 013: Rx CRC

Next we calculate, and check, the incoming CRC.

```
node 13 passes framed data, calculating crc on
packets and setting crc error bit as needed.,
output messages...,
,
..-1 state new link state 0 down, 1 10hd,,
....else last active code word received.,
,
..0 packet as follows,
....0/1 data bits, multiple of 8, 1 per word,
....10 8000 0000 0c00 0000 where,
......c crc-error,
....length in bits after preamble including,
......any dribble bits.,
,
+crc inserts one bit. hl is 32-bit crc packed
into a 36 bit number, 14/18. m tests the bit,
shifted out of high order.
check sets c status if crc is not all 1's.,
debug and variables are for testing.,
** then swap comments to force worst,
case when measuring timings.
```

```
762 list
013 rx crc reclaim 13 node 0 org +cy,
,
+crc mhlb-mhl 200 push dup . + pop or push,
..dup . + over over and .. 204 if ** then,
....205 or 130 or pop 11DB7 or ;,
..209 ** then drop pop ;,
check mhls 20A push push 3C000 or pop,
..and - if 40 then pop or !b @ !b ...
run 212 @ -if !b @ !b run ; then !b,
....4000 clc dup dup or dup,
..219 31 for @ -if check,
....21D then dup !b 1 or +crc next ahead,
..222 begin dup !b +crc swap then @ -until,
done check ;,
226 reclaim exit,
,
wire 231 @ !b wire ; 232,
!hi 20A @p drop !p ;*high 31416 ;
!lo 20D @p drop !p ;*low 31416 ;,
debug mhls push push dup !hi pop dup !lo pop
```

Functionally, this node verifies each packet's CRC and sets the `c` bit in the status word if it is not all 1s. Note that it all runs in extended arithmetic mode.

run is the main program of this node, waiting for data from node 014. Link state messages are detected and passed to node 012 unchanged. When a start of frame is seen, it is also passed along and we initialize the stack with the values `m h l` after which each bit received is passed to node 012 and is then used to update the CRC polynomial one bit at a time via `+crc` . When end-of-packet is received (now this is the status word), it is passed to `check` which sets the `c` bit in that status word, if the CRC is not all 1s, and then it copies the length word from node 014 to 012. The first 32 bits of the packet are inverted which is one way to get the desired answer out of the CRC algorithm.

2.3.7 Node 012: Rx Packing

After the CRC has been calculated we no longer need to process one bit at a time; we can now pack the bits into 16-bit octet pairs. This reduces the operating cycle of successive nodes in the pipeline from 100 ns to 1.6 µs, a reasonable interval for interleaving transmit and receive DMA operations with high level Virtual Machine program execution.

```
node 12 packs framed data into 16-bit pairs of
octets, little-endian, last word padded if odd
output messages...,
,
..-1 state new link state 0 down, 1 10hd,,
....else last active code word received.,
,
..0 packet as follows,
....00 bbbb bbbb aaaa aaaa a 1st octet b 2nd,
....10 8000 0000 0c00 0000 where,
......c crc-error,
....length in bits after preamble including,
......any dribble bits.
```

```
760 list
012 rx pack 12 node 0 org reclaim,
,
idl 00 @ -if !b @ !b idl ; then !b
pkt 04 10000 dup
bit w 06 @ -if push ahead
flush begin drop 2/,
....swap then dup 1 and until drop 2/ !b,
..pop !b @ !b idl then,
shift push dup 1 and if,
....drop 2/ !b dup dup then drop,
..pop if drop 2/ over or bit ;,
....then drop 2/ bit ;,
19 reclaim
```

Functionally, this node converts the packet representation from a bitstream to a stream of little-endian octet pairs.

run is the main program of this node, and waits for data from node 014. Link state messages are detected and passed to node 012 unchanged. When a start of frame is seen, it is also passed along. We initialize the stack with two copies of the value x10000 which is used both as the initial value of the shift register, for counting to 16, and also as the bit set in that register when a 1 is inserted. Consecutive data bits are inserted in the high order position of the register and shifted down. When 1 appears in the LSB of the shift register, 16 bits have now been inserted and after one more right shift there is a little endian octet pair transmitted to node 012. At end-of-packet the register shifts down as needed so that the alignment is correct (the transmission order has LSB first). The status and count words are copied across to 012 as received from 014.

2.3.8 Node 011: Rx Byteswap

Because the packets will be stored in memory in standard Internet order (big-endian) the next step is to swap bytes.

```
node 11 byteswaps packet data from node 12,
to avoid inter-bit latency. also checks length
and framing, enforcing stored size limits.,
output messages...,
,
..-1 state new link state 0 down, 1 10hd,,
....else last active code word received.,
,
..0 packet as follows,
....00 aaaa aaaa bbbb bbbb a 1st octet b 2nd,
....10 8000 0000 0crl 0ddd where,
......c crc-error,
......r runt; lt 64 octets.,
......l too long; gt 1518 octets.,
......d number of dribble bits.,
....length received in octets without dribble,
,
759 limits packet size stored to 1518 octets,
plus one word of dribble.
```

```
758 list
011 rx byteswap 11 node 0 org reclaim,
,
idl 00 @ -if !b @ !b idl ; then !b
pkt 04 FFFF dup dup or 759 push begin
word msk 0 08 @ -if
sts w 09 push @ dup push 2/ 2/ 2/,
..pop 7 and pop or,
..over -64 . + -if drop 20 or dup then drop,
..over -1519 . + - .. -if drop 10 or dup then,
....drop !b !b idl then,
bswap 1B a push dup 2* 2* a!,
....9 for +* unext drop over a and,
..!b pop a! next,
lng 23 begin @ -until sts ;,
25 reclaim exit
```

Functionally, this node delivers packets to node 010 byte-swapped into big-endian octet order.

In addition it checks the length field and uses it to determine the `r` (runt) and `l` (too long) status bits as well as the `d` field, but does not change the value of the length word. In passing and byte-swapping words, only a maximum of 1520 octets are actually transferred to node 010, to limit the size of DMA transfer and avoid overrunning buffers in the event of jumbo packets. Remaining data are silently discarded, in which case the actual CRC field of the packet will not be transferred to memory.

2.3.9 Node 010: Rx Control

Finally we have complete events ready to present to the host system via the shared memory structures ... or not, if you do not wish to operate that way. *By design, the inbound data path can end with node 011. Instead of feeding events to the Rx Control node to interact with the host using shared memory structures, you may have it feed events (link status and incoming packets) directly to application code running in nodes; aim initial register* ***b*** *at the node you wish to use for this purpose and leave out our code for node 010 (and presumably the three DMA mastering nodes as well.) Please make sure the code you place in that node will be able to keep up with the incoming stimuli; congesting the incoming data path at the end can cause loss of bit synchronization with the line and in extreme cases can disrupt link status. Build FIFOs or parallel / pipelined node teams as necessary to avoid congestion.*

```
10 is xmit dma control. incoming...,
,
..-1 status msg, next word one of two formats,
....0xxxx new link state x is 0 down, 1 10hd,,
......else last active code word received.,
....-1 signal to prod timer task.,
,
..0 packet as follows,
....00 aaaa aaaa bbbb bbbb a 1st octet b 2nd,
....10 8000 0000 0crl 0ddd where,
......c crc-error,
......r runt; lt 64 octets.,
......l too long; gt 1518 octets.,
......d number of dribble bits.,
....length received in octets without dribble
```

```
756 list
010 rx ctl 10 node 0 org,
x! wa 00 dup dup or
ex! wap 01 @p !b !b .. /ex! .. !b !b ;
x@ a-w 04 dup dup or
ex@ ap-w 05 @p !b !b .. /ex@ .. !b @b ;
?live -n 08 @p !b @b ; /live ;,
?rxb a,p-ap 0A @p !b .. ,rxb .. @b @b ;
2tal a 0D @p !b !b ; ,2t
+des 0F @p !b ; .. ,des,
,
!pg 11 @p drop !p ;*pg 0 ;
x!+ an-a' 14 over pg ex! 1 . + ;
!lk 18 @ ?live -if drop -if,
..t.wk x@ t.tt x@ x! ahead swap,
....then t.lk x! then then ...
run 23 @ if !lk ; then ?live -if,
..?rxb if !pg dup 2 . +,
....begin @ -if push drop pop x!+ @ x!+,
......+des run then x!+ end,
..then no buffs t.dp 2tal then
toss 36 begin @ -until @ run ;,
38 reclaim exit
```

Functionally, this is where we process incoming events and use the DMA Nexus to interact with shared host memory.

`?live` interrogates the DMA Nexus to determine how far the NIC has been enabled. 0 means no DMA operations are allowed, 1 means no packets are to be stored, and -1 means the NIC is fully enabled.

`run` is the main program of this node, and waits for an event from node 011.

On a link status event, there's a jump to **`!lk`** to process it. This routine checks NIC status and, iff fully enabled, then it does one of two things: For link status changes it stores the new link word into the **`t.lk`** field of the shared memory structure in external SRAM, while for heartbeats, it awakens the polyFORTH TCP timer task by storing the wake value provided in the **`t.wk`** field into the task whose STATUS address is provided in the **`t.tt`** field. This latter function provides a reliable way of cycling that task at a reasonable frequency in an energy-efficient manner.

On an incoming packet check NIC status. If not fully enabled, it does nothing ... the incoming packet is silently discarded. However, when enabled, it uses **`?rxb`** to find the next inbound buffer, if any, from the descriptor queue and its pointers in shared memory. Assuming that such a buffer is available (p nonzero), the value p is saved using **`!pg`** and the buffer address is incremented by 32 bits to save space for the status and length values. Then the text of the received packet is stored in consecutive 16-bit octet pairs using **`x!+`** . When the end of the packet is reached, the status and length values are stored into the first 32 bits of the buffer skipped earlier, and the descriptor is posted as complete for processing using **`+des`** (which maintains the descriptor queue pointers and awakens the polyFORTH task that's responsible for completions of transmit and receive operations). If no buffer was available, increments the 32-bit counter **`t.dp`** in shared memory using **`2tal`** .

It's possible to accomplish this much relatively high-level work in 56 words of code because much of the detailed work of these high level functions is done by port executed code in the three DMA nodes.

2.4 Outbound Data Path

Transmission begins with the Tx Control node that is stimulated by the DMA Nexus when high level code wants to transmit a frame, and ends with the Tx pin node which drives the pin with Manchester coded data. Along the way the CRC is calculated and framing is added. In addition, link negotiation signals are merged into the outbound stream when needed. Transmission is, as one would expect, considerably simpler than is reception.

This section follows the outbound packets from memory to the pin, in the same manner as was done with the receive pipeline in the preceding section. Refer to these load descriptors for initial conditions in each node of the transmitter:

```
726 list
- tx load descriptor,
,
tx 417 +node 417 /ram 2 /p,
..317 +node 317 /ram 20000 30000 over over,
....over over over over over over 10 /stack,
....io /b up /a 4 /p,
..316 +node 316 /ram left /a right /b 0 /p,
..315 +node 315 /ram left /b C 10 /p,
..215 +node 215 /ram up /a down /b 9 /p,
..115 +node 115 /ram down /a io /b 7 /p,
..314 +node 314 /ram down /a right /b A /p,
..214 +node 214 /ram up /a down /b 27 /p,
..114 +node 114 /ram left /a up /b 20A /p,
..113 +node 113 /ram right /a left /b A /p,
..112 +node 112 /ram left /a right /b 0 /p,
..111 +node 111 /ram right /b left /a 16 /p,
exit,
mon 517 +node 517 /ram io /b 200 /p,
..516 +node 516 /ram left dup /a /p right /b
```

2.4.1 Node 111: Tx Control

This node controls the memory operations to process Tx commands. *By design, the outbound data path can begin with node 112 when interfacing directly to node code; in that case, it is absolutely essential that you never let the pipeline run dry downstream by introducing undue delays within the data stream. There is only the time buffering of one wire node (112) available; as soon as the first bit moves from 113 to 114, the crystal starts pulling bits and you must sustain a rate of 1.6 µs per word.*

```
111 is xmit dma control. note that b points to
dma and a to tx chain, reverse of normal.,
,
+tx command from pf alerts us to new function
found in t.xn ...,
,
..-1 force autonegotiation,
..0 invalid, ignored,
..+n transmit packet of n octets starting at,
....double address in t.xa. data guaranteed,
....not to cross page boundary.,
,
when any of these functions complete, t.xn is
set zero to show done and pf task is awakened.
a new tx command may not be stored and +tx cmd
sent until t.xn is zero. dma controller will,
lock up if this protocol is violated
```

```
754 list
111 tx ctl 111 node 0 org
x! wa 00 dup dup or
ex! wap 01 @p !b !b .. /ex! .. !b !b ;
x@ a-w 04 dup dup or
ex@ ap-w 05 @p !b !b .. /ex@ .. !b @b ;,
,
!pg 08 @p drop !p ;*pg 31416 ;
inc n-n 0B 1 . + ;
done 0D 0 t.xn x!,
..wake 11 t.wk x@ t.sk x@ x! ...
run 16 begin @b drop t.xn x@ until,
....1A dup 2* 2* -if ! drop done then,
..1E drop dup ! - inc - 2/ push,
..t.xa dup x@ !pg inc x@ begin,
....26 dup pg ex@ ! inc next drop done ;,
2B ...
```

run is the main program of this node, and waits for signal from the Nexus that a command has been sent. The command is retrieved from `t.xn` , processed, and then `t.xn` is zeroed and the task is stimulated. For `t.xn` negative it sends a negative number onward to force link-down and autonegotiation. For `t.xn` positive it sends that value (octet count) followed by (`t.xn`+1)/2 16-bit big-endian octet pairs. For `t.xn` zero it assumes an anomaly and does nothing and the task is not stimulated.

2.4.2 Node 112: Wire

This node is necessary geometrically, however it also serves a useful purpose as a 1-word buffer, allowing an additional 1.6 µs of time buffering which may simplify the code feeding the pipeline.

```
112 is just a wire.
```

```
744 list
112 tx wire 112 node 0 org reclaim,
wire 00 @ !b wire ; 01,
,
01 reclaim
```

2.4.3 Node 113: Unpack

The next step is to convert 16-bit big-endian octet pairs into a bit stream with the LSB of each octet sent first.

```
113 passes force-down and packets. unpacks,
packets to bit strings.,
,
incoming data...,
..-1 force renegotiation of link.,
,
..n packet of n bytes follows in n/2 words,
....00 aaaa aaaa bbbb bbbb a 1st octet b 2nd,
,
outgoing data...,
..-1 force renegotiation of link.,
,
..n packet of n bytes follows in n*8 words,
....0/1 data bits, multiple of 8, 1 per word,
....1x xxxx xxxx xxxx xxxx end marker
```

```
742 list
113 tx unpack 113 node 0 org reclaim,
hi 1n-1 00 2/ 2/ 2/ 2/ 2/ 2/ 2/ 2/ 03/2 ...
lo 1n-1 7 for over over and !b 2/ next drop ;
done 08 20000 !b ...
run 0A begin @ dup !b - -until,
..0C - push 1 zif then,
..0F begin @ over over hi drop zif done,
....13 then lo drop next done ;,
,
16 reclaim
```

Functionally, we send onward link-down messages or packets consisting of a nonzero positive octet count, a string of data bits as 0s and 1s, and a negative end marker.

2.4.4 Node 114: CRC

Now that the data represent a sequence of bits, the CRC for the outgoing packet can be calculated.

```
114 calculates and attaches crc.,
,
+crc inserts one bit. hl is 32-bit crc packed
..into a 36 bit number, 14/18. m tests the bit
..shifted out of high order.
run passes link-down msg and starts crc when,
..packet arrives. the packet must be longer,
..than 32 bits due to loop structure.
trail ends pkt by running 32 zeroes into the,
..crc and then clocking it out, whole word,
..msb first.,
..trail takes on the order of 2.5us or 25 bit,
..times before it sends the first crc bit down
..the pipeline. hence the special delay node.,
,
outgoing data...,
..-1 force renegotiation of link.,
..n packet of n bytes follows,
....0/1 data bits, n*8 words,
....20000 end data marker,
....0/1 crc bits, 32 words
```

```
740 list
114 tx crc reclaim 114 node 0 org +cy,
+crc mhlb-mhl 200 push dup . + pop or push,
..dup . + over over and .. 204 if ** then,
....205 or 130 or pop 11DB7 or ;,
..209 ** then drop pop ;
run 20A begin @ dup !b - -until,
..20C 4000 clc dup dup or dup 31 for,
....211 @ dup !b 1 or +crc next,
..ahead begin dup !b +crc swap then @ -until
trail 219 !b 31 for 0 +crc next,
..21E 3 for dup . + push dup . + pop next,
..223 31 for dup . + push dup . + pop,
....dup dup or dup . + 1 or !b next run ;,
22D reclaim
```

run is the main program, and waits for a link-down (-1) which is passed along, or a packet which is passed and accumulated into the CRC. When the negative end-packet word is received 32 zeroes must be pushed into the CRC calculation before the first CRC bit is ready to go (in yet another interesting bit order). This introduces a 25 bit-time delay between the last data bit and the first CRC bit leaving this node, a serious problem that is solved by the next node in the pipeline. To facilitate this solution, the end data marker is sent between the last data bit and the first CRC bit.

2.4.5 Node 214: Special Delay FIFO

Here we solve the timing problem created by that 2.5 µs gap before the first CRC bit from the preceding node.

```
214 buffers bits out from crc node, holding 32
bit times of data to keep the packet flowing,
while calculating final crc. see crc node for
incoming stimuli.,
,
outgoing data...,
..-1 force renegotiation of link.,
,
..n packet of n bytes follows in n*8 words,
....0/1 data bits, multiple of 8, 1 per word,
....0/1 crc bits, 32 words,
....1x xxxx xxxx xxxx xxxx,
,
we start a packet by notifying framer to send
preamble. this gives us 64 bit times during,
which we prime our 32 bit fifo. thereafter we
send one bit for each new bit received. upon,
receiving end packet indicator we empty the,
fifo while crc node generates the final 32,
bits that we pass through.
```

```
738 list
214 tx delay 214 node 0 org reclaim,
,
00 buffer 32 words 20 org,
,
@!+ an-a'n' 20 a push push dup a!,
..1 . + 31 and @ pop ! pop a! ;,
,
run 27 begin @ dup !b - -until dup or
fill a-a 2A 31 for @ @!+ drop next,
pass a-a 2E @ -if*,
..purge a-a 2F push 31 for dup @!+ !b next*,
..crc a-a 33 31 for @ !b unext pop !b run ;
body a-a 38 then @!+ !b pass ;,
3A reclaim
```

Functionally this node simply passes along the outbound event (link down or start packet); however for packets, the first data bit is delayed while it fills a 32-bit (3.2 µs) FIFO. Functionally this node is a special 32-bit FIFO. One special aspect is that the first data bit is not sent onward until the FIFO has filled; at which point for each bit coming in from the CRC node, a bit is pushed out to the next. When the end data marker is received, it pushes out the remaining 32 bits in the FIFO, giving the CRC node plenty of time to be ready with the first CRC bit. At that point it copies 32 bits across from the CRC node and follows it up with the negative value that now indicates end-of-packet, including CRC.

2.4.6 Node 314: Framing

The next step is to frame the packet with a 64-bit preamble and an ending idle pulse, expressing the resulting sequence of events as subroutine calls for port execution in the multiplexor node.

```
315 passes commands and packets, converting to
bit commands, adding preamble and idle pulse,
to frame packets. incoming stimuli begin with
the t.xn field value as follows...,
,
..-1 force renegotiation of link.,
,
..n packet of n bytes, in n*8 words.,
....0/1 data bits, multiple of 8, 1 per word,
....1x xxxx xxxx xxxx xxxx end marker,
,
pre generates ethernet preamble of 61 bits,
alternating 1/0, start frame delimiter of two
consecutive 1's, all bits of packet including
crc, and the end of frame signal.
```

```
736 list
314 tx framing 314 node 0 org,
,
-attn 00 @p !b ; .. ;,
-lnk 02 @p !b ; .. /-lnk ;,
,
+one 04 @p !b ; .. /one
+zer 06 @p !b ; .. /zer
+idle 08 @p !b .. /idle ; ...
run 0A @ -if attn -lnk run ; then attn
pre 0D 30 for +one +zer next +one +one
pkt 14 @ -if +idle then,
..16 if +one pkt then,
..19 +zer pkt,
1B reclaim
```

run is the main program of this node. A link down message gets the multiplexor's attention, then has it do a **/-lnk** to kill the link, releasing the mux by allowing a return to its polling loop. For packets we fall through to **pre** ...

pre requests the multiplexor's attention and as soon as its first command is accepted it's critical to have each bit ready no more than 100 ns after its predecessor was accepted. There is actually a little more slack than that due to the time buffering in the next two nodes, but not much. Uses **/one** and **/zer** in the mux node to transmit 31 pairs of 1/0 bits (the preamble) and two consecutive 1s (the Start of Frame Delimiter) and then falls through to **pkt** ...

pkt receives the entire packet, translating it to **/one** and **/zer** calls in the mux node. The packet is ended and the mux is released with a jump to **/idle** .

2.4.7 Node 315: Packet / Link Pulse Multiplexor

This is the perfect place to select between packets and link negotiation / link pulses coming up another path from the early nodes of the Rx pipeline.

```
multiplexor for command of node node 315,
by 314 or 215.,
spins, will use smart io read when available.
this loop is costing 3.4-3.5 ma.,
,
to claim the node start writing code to its,
port; no focusing call needed, and a will be,
pointed at the port in node 315. you then own
it until you send it a return instruction or,
equivalent. code may call these routines...,
,
/slp generates a slow link pulse
/flp delays 62.5us
/-lnk delays 1.25 sec with no link pulses,,
..thus forcing renegotiation
/one sends a manchester '1'
/zer sends a manchester '0'
/idle sends end of frame
```

```
732 list
315 tx mux 315 node,
host*'rd-- 12195 lit ;*'-d-- 12115 lit ;,
*....'r--- 121D5 lit ; target 0 org,
,
!who n 00 @p drop !p ;*who -n 01 -1 ;
/slp 03 @p !b ; .. slp ;
/flp 05 @p !b ; .. flp ;
/-lnk 07 1.25 sec 20000 for /flp next ;,
,
kall 0C a dup a! !who push ex pop dup push !wh
o ;
run 10 'rd-- a! @ drop io a! @ dup push 13/1,
..2000 and if '-d-- kall then 17,
..pop 2* 2* -if 'r--- kall then run ; 1C
/wd n 15 for /slp /flp,
...dup 1 and if /slp then drop,
/flp 2/ next /slp 223 for /flp next ;,
,
/idle 2C @p !b ; .. idle ;
/one 2E @p !b ; .. 1bit
/zer 30 @p !b ; .. 0bit,
32
```

run is the main program of this node. It continuously polls **io** looking for writes by nodes 314 (packet framing) or 215 (link pulses). When one is found to be writing it gets our attention; we call the port and execute code in that port until we're given an effective return. During that interval **a** points to the port that is executing and the other port is completely ignored. What we actually do for the node in question depends on what code it calls from the port. The supported functions, mechanized by port execution sent onward to the Tx pin node, are these:

/one **/zer** and **/idle** generate the Manchester coded components of a packet (ones, zeroes, and the link-idle pulse marking the end of the packet.

/slp generates a link pulse. No time delay is included, either before or after the pulse.

/flp produces a precise 62.5 µs time delay, measured by the Tx pin node using the reference oscillator.

/wd transmits a 16-bit code word using a sequence of 17 clocking link pulses at 130 us intervals with 1s being represented as additional link pulses halfway between these clock pulses. The Tx pin is forced to be silent for 14 ms after each code word.

/-lnk forces the link down by making the Tx pin silent for a measured 1.25 seconds.

2.4.7.1 Nodes 115 and 116: Link Negotiation

All link negotiation is done in response to events observed on the Rx line. Each time the link state (expressed as a code word, zero meaning down, 1 meaning we are seeing NLPs implying the partner doesn't negotiate) has changed, the new value is passed to the negotiation nodes in such a way that the latest received will be the next processed. As long as the negotiation nodes do not see a latest code word indicating the partner has acknowledged 10 Mbit FD, they continue to force the link down then attempt negotiation with a fixed sequence of code words.

```
node 115 passes new link states from 015 to,
215 without blocking 015 ever. suspends till,
new state is available then spins on io to,
deliver it to 215. new state may be received,
while spinning and this simply replaces the,
state we are trying to deliver, so that when,
215 gets around to reading it will see the,
most recent link state received.,
,
node 215 takes latest state from 115 and check
s it for ack of 802.3 10fd mode. if so, we are
happy and do nothing. otherwise we perform one
negotiation cycle, open loop, by sending 5,
code words of 10fd, 8 of ack 10fd, and then,
ignoring new link states for six seconds to,
let the other end make up its mind. at the end
of that time we expect to receive a good ack,
from node 115. 215 is not required to be timel
y in reading from 115.
```

```
734 list
115/215 autonegot 215 node 0 org,
,
!link n 00 @p drop !p ;*link -n 01 -1 ;
-attn 03 @p !b ; .. ;,
-lnk 05 @p !b ; .. /-lnk
wd n 07 @p !b !b ; .. @p /wd
auto 09 begin @ dup !link,
....dup 41 4041 and 41 4041 or until,
neg 0F -lnk 4 for 41 wd next,
....7 for 4041 wd next -attn,
..60000 20000 n.n ms for,
....41664 for unext next auto ; 21,
,
115 node 0 org
!link n 00 @p drop !p ;*link -n 01 -1 ;
pass a up a! link ! a! ;
done 07 @ !link begin,
..@b 400 or dup 400 and if pass done ;,
..then drop 2000 and until done ;,
12
```

These shadows accurately describe the behavior of these two nodes.

Node 115's behavior is designed to always preserve the most recent new link state whenever multiple state changes occur while node 215 is busy, which can be a period measured in seconds.

Node 215 receives each new, latest link state, and is happy when it sees an acknowledgment of a 802.3, 10 Mbit, full duplex link. This is the last expected incoming code word during a successful negotiation. Thereafter, the negotiation nodes are inactive because the Tx pin node automatically generates NLPs frequently enough to meet link integrity check rules, unless it is being forced not to.

If the most recent state is not what it's looking for, then node 215 gets the multiplexor's attention and keeps it for quite a while, preventing the transmitter from sending packets while node 215 embarks on an open loop negotiation strategy. The first thing it does is to force the link down by preventing NLP generation for 1.25 seconds. It then sends five requesting code words and eight acknowledging words as indicated in the shadows, releases the multiplexor, and waits for about six seconds before considering a new link state if one's present. Assuming that the open loop negotiation was successful, there will be just one new state waiting at that point, and it will be the acknowledgment of our requested link configuration so that the process ends.

2.4.8 Node 316: Wire

This node's code is in the same block as that of node 317, below. It's a unidirectional wire and is only there because the topology of the cluster requires it. However, as usual its presence presents a timing buffer on the order of 100 ns which may relieve demands on earlier nodes in the pipeline.

2.4.9 Node 317: Tx Pin

The last node must drive the transmitter pin with compliant timing, synchronized with the reference oscillator, producing link pulses when appropriate, and managing the pin in such a way that transmitter power is minimized when the line is idle.

```
manchester encoding transmitter. instructions
executed in right port.,
,
bit sends a zero hi-lo.,
over bit sends one lo-hi. once transmission is
started bits must be ready before needed.,
idle ends transmit with tp-idl waveform, high
for 3 bit times then sil-ent.,
sil -ent sets hi-z, forces silence for 1/2 the
number of bit times given, then polls for work
to do while keeping the oscillator happy.,
counts cycles, generating a slow link pulse if
there have been 16ms / 320,000 stimuli since,
last transmit or link pulse.,
slp sends the link pulse and resumes timing.,
called thru the right port for flp sequences,
during auto-negotiation.,
flp delays 62.5us to time interbit interval.
```

```
730 list
317/316 tx pin 317 node 0 org,
,
0bit 00 @ drop !b @ drop !b ;
1bit 02 drop 0bit drop ;
init 04 0 dup ! ...
sil 1/2bt 06 push 0 .. @ drop !b ..,
....09 begin @ drop unext ..,
..0A dup 15 for 0C 19999 for,
....0E @ drop drop .,
....0F @b 2* 2* . -if drop r---,
..13 then next next drop ... slp,
slp 16 @ drop dup !b @ drop 81 sil ;
goose 1A begin drop @ drop dup !b
idle 1C @b -until,
...drop 4 for @ drop unext 83 sil ;
flp 22 1249 sil ; 24,
,
316 wires instructions from 315 to 317,
316 node 0 org
wir 00 begin begin @ !b unext unext wir ; 02
```

This node's stack is filled with io values for low and high pin drive, with high on the top of the stack. **b** addresses **io** and **a** addresses the oscillator node. All operations of this node are synchronized tightly with the output of the oscillator node which feeds one word across the COM port for each phase of the 10 MHz clock, i.e. one word comes across the port each 50 ns. The node tries to start each new thing it does immediately after receiving a clock edge so that as much of that 50 ns interval is available as possible; in some cases the code will delay until the next clock edge just to achieve this condition. Being late for the next clock edge would be a Very Bad Thing.

`init` begins the node's execution by writing a word to the oscillator node, giving permission to start the oscillator and promising that this node will faithfully receive words from the oscillator at 20 MHz until the next time the chip is reset. This falls through to the main program **`sil`** requesting 1/2 bit time of silence...

`sil` is the main program and takes a stack argument which is a time interval, in clock edges, during which the pin should be silent before resuming normal idle behavior. It waits for the next clock edge before setting the pin to high impedance (thus conserving power, see circuit description), after which it absorbs one more than the number of clock edges given while maintaining silence on the pin and ignoring any other stimuli. Thereafter the routine begins an idle loop lasting 320,000 clock edges (16 ms) during which, after each clock edge, it checks for a write from the Tx pipeline. When one is seen, the routine calls its **right** port to execute whatever code is sent through the port. Port sequences are not expected to return, but rather to end at `sil` . If the loop ends without any work from the pipeline, it means that the line has been silent for 16 ms; it will generate a slow link pulse, then resume another 16 ms idle loop.

The routines that may be called through the port from the pipeline are these:

`flp` delays 62.5 us with the pin at high impedance and resumes idle loop.

`slp` generates a slow link pulse by driving the pin high for one bit time then releasing it to high impedance for 162 clock edges, or 8.1 µs. No time delay is included before or after the pulse.

`0bit` Manchester codes a zero by driving the pin high at the next clock edge then low at the following edge.

`1bit` Manchester codes a one by driving the pin low at the next clock edge then high at the following edge.

`idle` produces the "end of transmission delimiter". If the pin is not already high, waits for a clock edge and then drives it high. Leaves the pin high for three bit times then releases it to high impedance and resumes the idle loop.

2.4.9.1 Node 417: Tx Timing (Oscillator)

The Tx Pin node requires stimuli at 20 MHz. The default version of node 417 derives this signal from a 10 MHz square wave input on pin 417.17.

```
fox 819a-10 10 mhz oscillator input.,
each 50 ns edge sends a return instruction,
thru up port to tx pin node 317 which waits,
for the edge by calling or reading the port.,
,
node 317 must wait faithfully for every edge,
for compatibility with soft oscillators.,
,
note node 317 starts us when it is ready to go
by writing to the port.,
```

```
788 list
417 tx osc 417 node 2 org,
,
init 02 up a! @ drop .io b! left a!,
..left 15555 up 0 10000,
..left 15555 up 800 10800,
...dup dup drop drop 13/2
go 13 . @ drop . !b a! ! a! go ;,
16 reclaim exit
```

As noted above, this routine waits for a word from the Tx Pin node before starting its operations, waiting for each edge of the input signal and transmitting a word to the Tx Pin node on each. The code listed above leaves the pin terminated by weak pull-down, which helps with noise on prototype electrical interfaces. We have not explored whether or not this is necessary when using the E-NET board.

2.5 DMA and Function Control

Three nodes are used for this purpose. Although it might have been possible to use only two, the geometry of the cluster allows for three to work with so we took advantage of the resulting flexibility. These are shown in compilation order for clarity in symbols. The high level functions of these nodes support the filling, posting and management of the pool of receive buffers and the shared memory structure described at the beginning of this section.

The DMA Nexus node 110 controls DMA operations on behalf of the Tx and Rx control nodes 111 and 010, each of which provides small, atomic units of work via port execution. It is logically necessary that commands from these nodes are completely unsynchronized so in the F18A, the Nexus must poll its **io** register to sequence stimuli.

Refer to these load descriptors for initial conditions in each of these nodes:

```
                                                    722 list
                                                    - load descriptor,
                                                    ,
                                                    (...)

                                                    ..110 +node 110 /ram down /a left /b 15 /p,
                                                    ..109 +node 109 /ram left dup /p /a right /b,
                                                    ..108 +node 1801 /ram right dup /p /a left /b,
```

2.5.1 Node 108: Nexus Slave II

This node requests DMA operations of the SRAM control node (107) under the direction of the Nexus.

```
node 108 is the actual sram master.,                746 list
it supports passthru memory ops and is,             108 sram master reclaim host host's memory...,
capable of being programmed for other slave,        t.cm 8000 lit ;*t.pf 8001 lit ;
operations. registers are,                          t.rx 8002 lit ;*t.ep 8003 lit ;
,                                                   t.lk 8004 lit ;*t.dp 8005 lit ; double
a and p right for instructions from 109,            t.sk 8007 lit ;*t.wk 8008 lit ;
b left for sram controller.,                        t.xa 8009 lit ;*t.xn 800B lit ;
,                                                   t.pp 800C lit ;*t.tt 800D lit ;
.ex@ and*.ex! are port called to effect our,        rxdm FF7F lit ; target,
instructor's x@ x! etc words.                       ,
stm waits for stimulus and informs master.          108 node 39 org 278 load 40 0 org
inc increments a number.                            .ex! paw 00 push @ push @ pop pop ex! ;
+tsk awakens our task.                              .ex@ pa-w 03 push @ pop ex@ ! ;,
+des circularly increments next rxd pointer.       ,
?rxb returns next rx buffer address, p zero if      stm 06 @b ! ;
none available for use.                             inc n-n 07 1 . + ;
2tal increments a double number at given addr.      +des 09 4 t.rx x@ + rxdm and t.rx x! temp ;
                                                    +tsk 11 t.wk x@ t.sk x@ x! ;
                                                    ?rxb -p,a 16 t.rx x@ t.ep x@ over or if,
                                                    ..drop dup inc x@ over x@ then ! ! ;
                                                    2tal a 20 @ dup inc dup x@ inc dup push,
                                                    ..FFFF and over x! pop 2* -if,
                                                    ....drop drop dup x@ inc over x!,
                                                    then ;,
                                                    2E 1801 bin ...
```

Start address is **right** to execute commands from node 109. The literal values defined at the start of the block provide agreed-upon addresses for fields in the shared memory structure and are visible to the other DMA nodes and to the Tx and Rx Control nodes.

stm is used to shut down DMA operations until a stimulus is received from the SRAM control node and passed upward to the Nexus, enabling the NIC after which node 108 resumes normal operation as the slave of 109.

.ex@ .ex! +des +tsk ?rxb and **2tal** are the routines that may be called from node 109 to perform the functions identified in the shadows.

2.5.2 Node 109: DMA Nexus Slave

This node serves mainly as an intermediary between the Nexus and node 108, although it can and does add more high level functions.

```
node 109 connects dma nexus with 108, the,
actual sram master.,
,
supports passthru memory ops and supports the,
nexus with some other overflow operations.,
registers are,
,
a and p left for instructions from 110,
b right for instructions to 108.,
,
x@*x! etc command slave node's .ex@ and .ex!
.ex@ and*.ex! are port called to effect our,
instructor's x@ x! etc words.
/stm stim wait by 108
/?cm poll cmd
```

```
748 list
109 dma nexus slave 109 node 0 org,
x! wa 00 dup dup or
ex! wap 01 @p !b !b .. @p .ex! .. !b !b ;
x@ a-w 04 dup dup or
ex@ ap-w 05 @p !b !b .. @p .ex@ .. !b @b ;
.ex! p.aw 08 push @ push @ pop pop ex! ;
.ex@ p.a-w 0B push @ pop ex@ ! ;,
,
,
/stm 0E @p !b .. stm .. @b ! ;
/?cm -n 11 t.cm x@ if,
..dup dup or t.cm x! then ! ;,
/rxb -a,p 18 @p !b @b .. ?rxb .. @b ! ! ;
/2t a 1B @p !b .. 2tal .. @ !b ;
/des 1E @p !b ; .. +des,
20 ...,
```

.ex@ **.ex!** **/stm** **/des** **/rxb** and **2t** are used to invoke the corresponding functions by node 108 on command from the Nexus.

?cm shows how to define a high level function in this node; it's the operation to poll for a new command from the host.

2.5.3 Node 110: DMA Nexus

The Nexus serves as the slave of Rx and Tx control nodes to perform memory operations. The DMA state variable lives here and is interrogated by port commands. DMA operations are actually performed by nodes 109 and 108 as directed by port commands coming from 110.

```
node 110 is continually active, polling its,
clients 010 and 111 as well as monitoring 109
for stimuli.
poll polling dispatcher for rx/tx nodes and,
stimulus from pf. these functions must protect
the top of stack. dma functions are...
/ex@ and*/ex! sram ops.
/live returns state.,
,
polls at t.pp period for t.cm commands from pf
pf commands are...
!live sets 0 no dma, 1 no traffic, -1 all up.
-dma cmd to state 0 don't use !live for this!
+tx alert tx node of new function in t.xn,
```

```
750 list
110 dma nexus 110 node 0 org,
x! wa 00 dup dup or
ex! wap 01 @p !b !b .. @p .ex! .. !b !b ;
x@ a-w 04 dup dup or
ex@ ap-w 05 @p !b !b .. @p .ex@ .. !b @b ;
/ex! paw 08 @ push @ push @ pop pop ex! ;
/ex@ pa-w 0B @ push @ pop ex@ ! ;,
,
stm 0E 0 @p !b .. /stm fallthru...
!live 11 @p drop !p ;*live -1 ;
sel 14 a! @ push ;
-dma 15 stm
?cmd 16 live if @p !b @b .. /?cm .. if,
..1B dup push 2* 2* ex then,
..1D t.pp x@ 2* 2* push then
poll 20 begin io a! A800 dup,
23 begin drop zif ?cmd then,
25 ..@ over and until,
27 2* 2* .. -if tx right sel 2A then,
2A 2* 2* .. -if rx down sel poll ; 2E then,
2E 2* 2* -until @b !live ?cmd ;,
+tx x 32 right a! ! ;,
34 ...
752 list
110 continued,
/live -n 34 live ! ;
,rxb -a,p 36 @p !b @b .. /rxb .. ! @b ! ;
,2t a 39 @p !b .. /2t .. @ !b ;
,des 3C @p !b ; .. /des,
3E ...
```

live is the major NIC state variable: 0 means disabled, no DMA operations are allowed; positive nonzero means DMA is permitted but no packets are to be stored; and negative means the NIC is fully enabled.

-dma is the start address of this node, establishing the initial state of the DMA mechanism. The **stm** command is used to make node 108 wait for a stimulus from the SRAM control node 107. Until this stimulus is received, nodes 108 and 109 will not accept further commands from the Nexus, which guarantees that it won't try by setting `live` to 0. Having done this we fall through into the polling loop at `?cmd` ...

?cmd If NIC is disabled, skips ahead to `poll` . Otherwise, checks for, and clears, a command from `t.cm` and if one was sent executes it. The low 10 bits of the command word are an address to call in node 110; the command word, shifted left 2 bits, is available as an argument on the stack. After the command, if any, is executed, fetches the `t.pp` command polling interval, multiplies it by 4 and uses it as the loop count for io register polling in poll before the next time `?cmd` is performed.

poll loops on testing **io** for writes from nodes 109, 111 or 010, returning to `?cmd` when the loop count is exhausted. On seeing anyting from 111 (Tx) or 010 (Rx Control), we set **a** to the port, read a word from the port expecting it to be a call instruction, and effectively call the routine whose address was given in that instruction with **a** available for the routine to communicate with the node that called it. If node 109 is trying to write, it means the NIC is disabled and we have received a stimulus from the SRAM controller; the value received is a mask that's guaranteed to be positive nonzero, so we store it into `live` making our state enabled with no packets and immediately go make the test in `?cmd` .

3. Hardware Interface

This section presents an example for interfacing the above software directly with a 10baseT line. The example uses the E-NET piggyback board and attaches this to an EVB001 Evaluation Board. When set up as instructed here, the polyFORTH TCP/IP package may be configured to communicate with the Internet using this hardware and software.

3.1 The E-NET Board

This board is available from GreenArrays and is designed as a simple electrical interface for the EVB001. The schematic for this, and the bill of materials with parts values, are on following pages.

J3 connects the board with the GPIO and analog nodes on the right side of the chip; other headers supply 1.8V, 5V and ground. U2 is a transformer/filter, specialized for 10baseT, which AC couples our circuitry to the twisted pairs.

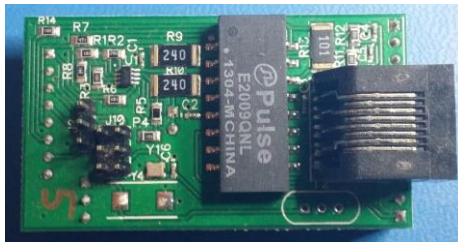
The passive receive circuit has the 100 ohm load prescribed by the IEEE 802.3 standard (R13), a voltage divider (R11,R12) to place the center of the signal's swing at 0.9V, and capacitors (C4, C5) to reduce noise. Its only other component is a current limiting resistor (R14) to prevent excessive current being shunted by the GPIO pin's protection diodes.

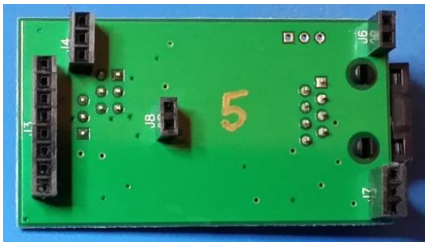
The transmit circuit requires the dual op-amps (U1 and R1-R6, R9-R10) to amplify our 1.8V GPIO pin to swing far enough to achieve the voltage range required by the standard. The op-amps are set to turn on only when the signal from the GPIO pin is near one or the other rail and well away from the high impedance resting voltage set by the divider (R7, R8). This allows us to minimize DC current through the primary of the transformer when our transmitter is idle, by setting the pin at high impedance.

The board was designed mainly to provide the 10 MHz signal for transmitter timing from an onboard oscillator Y1 but to permit experimentation with other methods.

3.1.1 Jumpers

The E-NET board is shipped with jumpers set to the default positions indicated by the red arcs in the tables below; because pin 1 placement is not per convention on this board, the tables are aligned to match the photo of the top side. J2 pin 1 is at the top of the 3-pin header; J10 pin 1 is in the lower right corner of the 6-pin header.





As shipped, the board is set to present its 10 MHz oscillator output on host pins 517.17 and 417.17, and to use 117.ao as the enable signal for the oscillator.

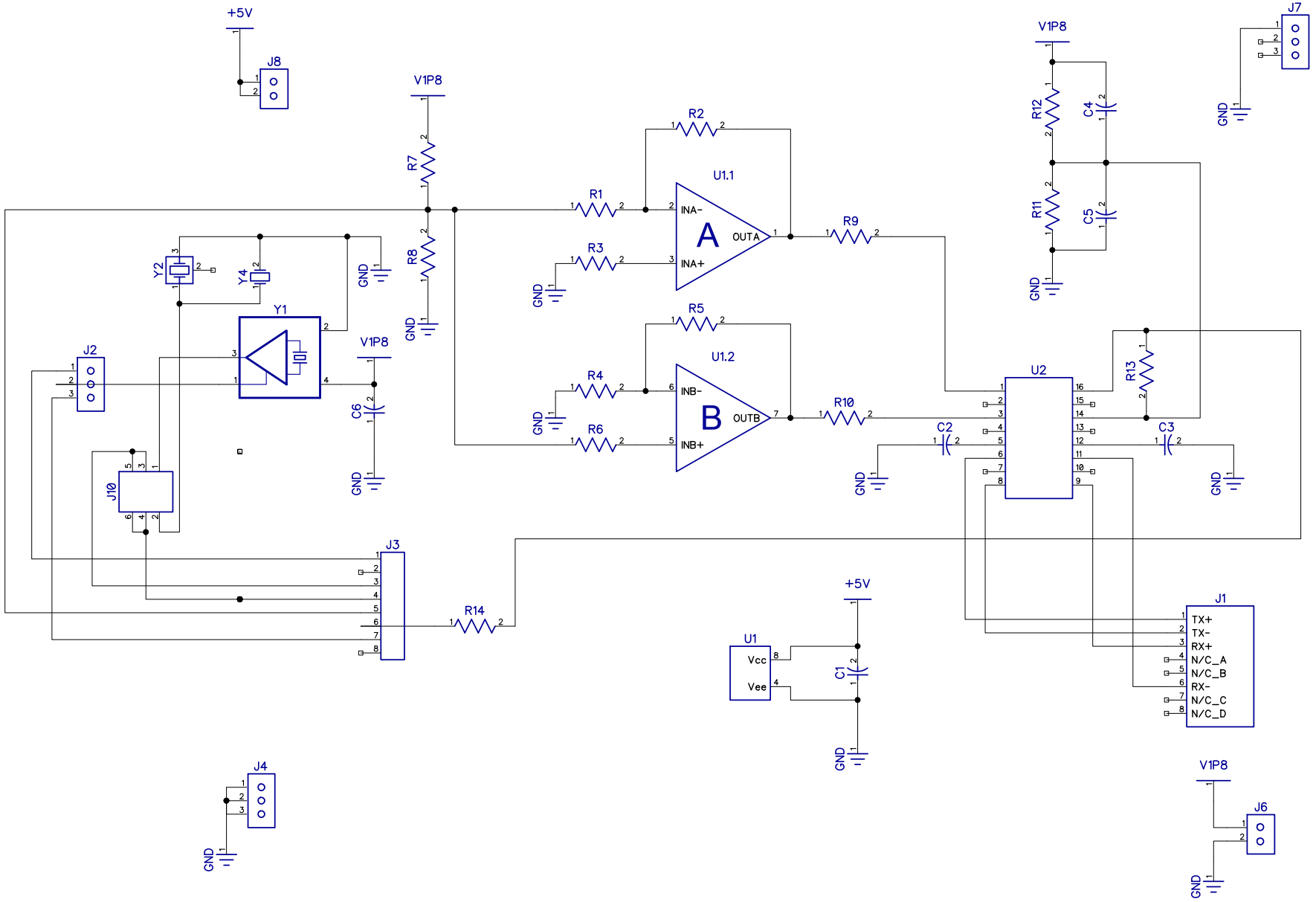
J2

1	Host 617.ao
2	Osc Ena Signal
3	Host 117.ao

J10

Host 417.17	6	5	Host 517.17
Host 417.17	4	3	Host 517.17
Y2/Y4 Devices	2	1	Osc Out

The Y2/4 devices are designed for ceramic resonators and/or crystals. Code in the package supports the ceramic resonator but our experience is that most lack the absolute accuracy demanded by most link partners. We have considered using the onboard oscillator only to start a crystal and then maintain the crystal at lower power by "dribbling" it at the node; this board is designed to permit such experimentation.

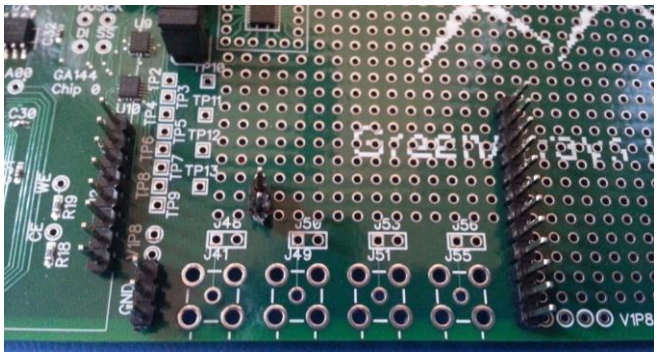

+5V
J8
V1P8
R7
R8
R2
R1
R3
U1.1
INA-
INA+
OUTA
A
R9
GND
Y2
Y4
Y1
J2
R5
R4
R6
U1.2
INB-
INB+
OUTB
B
R10
C6
J10
J3
R14
U1
Vcc
Vee
C1
J4
V1P8
R12
C4
R11
C5
J7
U2
R13
C2
C3
J1
TX+
TX-
RX+
N/C_A
N/C_B
RX-
N/C_C
N/C_D
J6

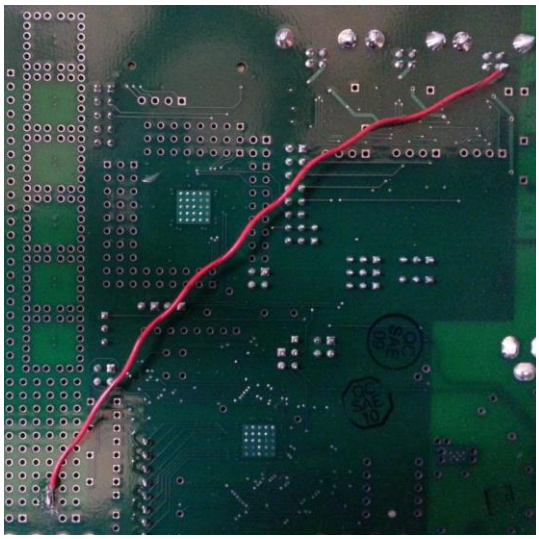
#	RefDes	Value	Type	Part
1	C1	0.1uF	CAP_0402	Part 1
2	C2	0.1uF	CAP_0402	Part 1
3	C3	0.1uF	CAP_0402	Part 1
4	C4	0.1uF	CAP_0402	Part 1
5	C5	0.1uF	CAP_0402	Part 1
6	C6	0.1uF	CAP_0402	Part 1
7	J1		Ethernet	Part 1
8	J2		CON3	Part 1
9	J3		CON8	Part 1
10	J4		CON3	Part 1
11	J6		CON2	Part 1
12	J7		CON3	Part 1
13	J8		CON2	Part 1
14	J10		HEADER_2X3	Part 1
15	NetPort1		GND	Part 1
16	NetPort2		GND	Part 1
17	NetPort3		GND	Part 1
18	NetPort4		V1P8	Part 1
19	NetPort5		+5V	Part 1
20	NetPort6		GND	Part 1
21	NetPort7		GND	Part 1
22	NetPort8		GND	Part 1
23	NetPort9		GND	Part 1
24	NetPort10		V1P8	Part 1
25	NetPort11		+5V	Part 1
26	NetPort12		GND	Part 1
27	NetPort13		V1P8	Part 1
28	NetPort14		GND	Part 1
29	NetPort15		V1P8	Part 1
30	NetPort16		GND	Part 1
31	NetPort18		GND	Part 1
32	NetPort20		GND	Part 1
33	R1	1K	RES_0805	Part 1
34	R2	2.7K	RES_0805	Part 1
35	R3	1K	RES_0805	Part 1
36	R4	1K	RES_0805	Part 1
37	R5	1.7	RES_0805	Part 1
38	R6	1K	RES_0805	Part 1
39	R7	470	RES_0805	Part 1
40	R8	390	RES_0805	Part 1
41	R9	24	RES_2512	Part 1
42	R10	24	RES_2512	Part 1
43	R11	2k	RES_0805	Part 1
44	R12	2k	RES_0805	Part 1
45	R13	100	RES_2512	Part 1
46	R14	2.2k	RES_0805	Part 1
47	U1		MAX4413EKA-T	Part 1
48	U2		pulse E2009QNL	Part 1
49	Y1		ASF	Part 1
50	Y2		ECS-110-S-4-3IL	Part 1
51	Y4		12SMXA	Part 1

3.2 Installation on EVB001

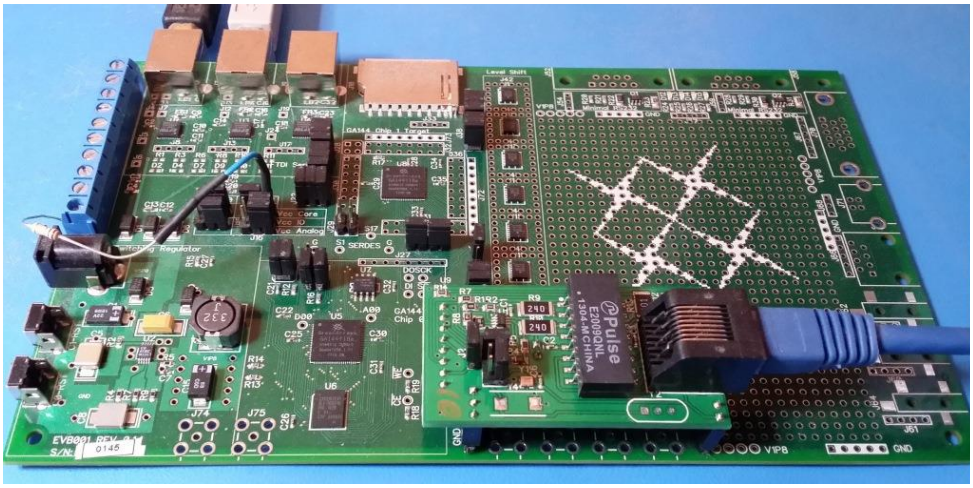
Installation requires a soldering station, some hook-up wire, and single-row male headers as supplied with the EVB001 kit. When installing the headers, be careful to keep them perpendicular to the board surface; there are many pins on four separate headers that must line up with the female sockets on the bottom of the E-NET board. One possibility for doing the soldering might be to insert the headers into the bottom of the E-NET and use it as a jig while tack-soldering the headers to the board. The E-NET could then be unplugged and the rest of the pins soldered with good flow, keeping the alignment of the pins unchanged.

You will need to break the single-row header into four smaller headers of 12, 8, 3 and 2 pins respectively; it is possible to replace the 12-pin header with two smaller headers (2 and 3 pins respectively) if you wish. The headers are installed as shown here, with the 8-pin header occupying all of the J21 holes which connect with the analog and GPIO pins on the right side of the host chip. Note that the 3-pin header occupies three ground holes and provides ground for the board. The 12-pin header on the right side is used to power the low voltage part of the circuit from one of the five V1P8 holes at the board edge; the remainder of the pins on this header are used only for mechanical support. Finally the 2-pin header out in the middle is used to supply 5V; we will get it from an USB interface in this case.





Once all these headers have been soldered in place with good electrical and mechanical connection, flip the board over and connect a source of 5VDC to the 2-pin header. The example to the right shows 5V acquired from USB port A. This requires port A to be connected to a USB host that is willing and able to supply power. You may wish to use another source, depending on the intended use of the resulting system. The Ethernet transmitter depends on having this voltage available. The exact voltage is not critical but it should not vary too far from 5VDC.



This image shows an EVB001 with E-NET board installed and running. Note that the E-NET jumpers are in their default positions.

4. Using the NIC with arrayForth Release 2b

All the code for basic operations up through UDP (but not TCP) are included in the arrayForth 2b release. This section introduces the steps necessary to configure that system to use the NIC code and physical E-NET board installation discussed above.

4.1 arrayForth Configuration

4.1.1 Block 362: polyFORTH Load Descriptors

As shipped, the load descriptors for the NIC are commented in the load descriptor block. You will need to make changes in block 362. These are discussed in section 5.2.1 of DB006 but the revision of 120923 shows an obsolete block number (470).

```
this block describes entire chip's loading for
polyforth boot environment.,
,
suitable for use with ide loader, streamer,,
and softsim.,
,
all nodes not otherwise programmed are loaded
with ganglia for ad hoc routing.,
,
the sram cluster is invoked here for complete
documentation although it has been loaded,
earlier and is excluded from the path used by
the main stream.,
,
enable loading of clock nodes if you have ftdi
or ether clock connected to 517.,
,
enable loading of ether cluster if you have,
nic hardware attached to right side of chip.
```

```
362 list
- load descriptors,
1, nn dup +node 100 /mod 1 and 2* swap 1 and +
1714 + push 32 32 pop /part warm A9 /p ;
ganglia nns for i -1 + n-nn 1, next ;,
,
sea of mk1 ganglia,
snorkel mk1 207 +node 1605 /ram up /b 37 /p,
,
sram cluster mk1 sram 2 + load,
virtual machine 2 fh load,
serial terminal 4 fh load,
additional i/o 6 fh load,
clock nodes exit,
..517 +node 517 /ram io /b 200 /p,
..516 +node 516 /ram left dup /a /p right /b,
ethernet cluster mk1 exit,
..ether 2 + load
```

To fully enable this function, comment **`exit`** on the lines reading `clock nodes` and `Ethernet cluster mk1` above. This assumes the E-NET board has default jumpers, using the 10 MHz oscillator on the board and connecting the output of that oscillator to pins 517.17 and 417.17. If for some reason you do not wish to use the clock provided by nodes 516 and 517, the load descriptors for those nodes will need to be commented but those for the Ethernet cluster will still have to be active.

Having made this change, next time you load the chip for polyForth by any of the supported means the NIC will come up and should negotiate a 10baseT Full Duplex connection if the link partner is capable of negotiation.

Procedures for loading polyFORTH F18 support using the IDE, or installing it in SPI flash, are documented in section 5.2.2 of DB006. In addition, software release 02b includes a much faster one-time load procedure that builds a serial boot stream and injects it into the chip; this is done with **`430 load`** .

Review block 200 to make sure you are not compiling any code for any of the nodes used by the NIC. If you did, your code would replace the NIC code and of course nothing would work.

4.1.1 Block 772: Rx Node Internal Delay Setting

As noted in the description of the F18 NIC code, there is an internal delay in node 117 that may need adjustment for reliable reception. The default value, 4, may require lengthening. Contact our hotline if you need assistance.

4.2 polyFORTH Configuration

As shipped, polyFORTH does not load any networking code, and including the F18 code for the NIC as described above has no effect on the operation or performance of the virtual machine. The NIC will negotiate with its link partner but no outgoing packets are transmitted and all incoming packets are silently discarded. In order to use the NIC we recommend the following sequential steps, to make sure things are working properly.

First, if you have not done so already, configure the EVB001 COM ports A, B and C in accordance with DB003 so that they will move data efficiently.

Second, if you have not done so already, load and run polyFORTH successfully as distributed (see DB006 section 5.2), configure and verify proper operation of the Terminal Emulator program per DB006 section 5.5, boot polyFORTH using HI and exercise it enough, if you haven't already done so, to be satisfied that you have a working system.

Third, physically install the E-NET board as described above, with default jumpers. If you load the standard polyFORTH in this state, the E-NET board will do nothing; if connected to a link partner, the partner will see no negotiation and will assume a 10 Mbit Half Duplex connection.

Fourth, make the arrayForth changes described above to load the NIC and repeat the second step to make sure you have not broken polyFORTH. If you have connected the E-NET board to a negotiable link partner, the link should be up and set for 10 Mbit full duplex operation. If this does not happen, take note so you can report this if you contact us for help, but go ahead with the remaining steps.

Fifth, configure and test the polyFORTH Networking support as indicated in the next section.

4.2.1 Block 9: Select 10 MHz time clock.

Regardless of whether you plan to use the 10 MHz clock for polyFORTH, set it up now to verify that the oscillator on the E-NET board is running. Boot polyFORTH and say **`COUNTER TIMER`** . You should see a time interval in microseconds.

```
2409
 0                        N O T I C E
 1 This material is copyright (c) 2011-2012 by GreenArrays, Inc.
 2 (GAI).  All rights reserved.  This notice must appear on all
 3 authorized copies of this software. GreenArrays & arrayForth are
 4 are (r) trademarks of GAI and polyFORTH is a (r) TM of FORTH Inc
 5
 6 This software has been provided to facilitate use of GreenArrays
 7 chips only.  It is provided at no charge subject to Conditions
 8 with which you agreed when downloading and installing it.
 9 While we believe this software to be bug free, GreenArrays makes
10 no representations or warranties, including but not limited to
11 the implied warranties of merchantability or fitness for a
12 particular purpose.
13
14 This block is loaded when, after booting, the operator types HI
15    It extends the system with resident code (under  EMPTY ).
```

```
9
 0 ( Electives)   DECIMAL
 1 : <CREATE>   BLK @ DUP IF  OFFSET @ +  THEN ,  (CREATE) ;
 2    ' <CREATE> 'CREATE !  1 CONSTANT B/H
 3
 4 ( Aids)  12 LOAD  13 LOAD  14 LOAD  10 LOAD
 5 ( RAMdisk  45 46 THRU)  ( Errors)  33 LOAD  ( Buffers)  34 LOAD
 6 ( 32-bit & extens)  15 18 THRU  21 23 THRU  28 LOAD
 7 ( Calendar)  30 LOAD   ( Select one of these clocks:)
 8    ( Serial 47 47)   ( FTDI 31 32)   ( Ether) 121 122  THRU
 9 : SYSTEM   29 HELPS  DATE  ."  Time " TIME ;
10
11 ( Compat)  35 LOAD   ( Tasks)  25 26 THRU
12 ( Edit)  19 20 THRU   ( Tasks)  27 LOAD   ( Ext)  120 LOAD
13    FORTH GILD   VPT   SYSTEM   ' ?CREATE 'CREATE !
```

As shipped, block 9 is configured to use a serial protocol, provided by the Terminal Emulator, for interval timing and time-of-day; see line 8 which as shipped reads `( Serial) 47 47` . Change that line as shown above by commenting the two block numbers `47 47` and uncommenting the values `121 122` for the Ether clock. polyFORTH users learn to be very careful when editing block 9; embrace this practice here by making sure line 6 reads ***exactly*** like this:

```
8    ( Serial 47 47)   ( FTDI 31 32)   ( Ether) 121 122  THRU
```

When you are done editing block 9, type **`FLUSH RELOAD`** . Assuming all is well, the system should do a warm reboot to show a bogus date and time under the system help screen. Verify that you have a working 10 MHz clock by typing **`COUNTER TIMER`** again. This time you will get a time in microseconds with one digit to the right of the decimal place. Scale is confirmed by typing **`COUNTER 1000 MS TIMER`** .

4.2.2 Block 126: Set System Time-zone Offset.

Although not required, many TCP/IP protocols require that computers know time of day for UTC and some encourage knowledge of local time. The constant TZONE in block 126 is the local offset from UTC in hours. As shipped the value is -7 representing MST in the US (UTC-7).

```
2526
 0 [time]  given @time and date, returns seconds since 0000Z 1/1/00
 1    This is the time format used by TCP and UDP TIME function.
 2    sF's belief that 1900 is leap is corrected.
 3 [now]  returns present time in those units.
```

```
126
 0       ( Unix time)
 1 -7 CONSTANT TZONE   @TIME TODAY @ CONSTANT Bdt 2CONSTANT Btim
 2 : Booted ( - d n)   Btim Bdt ;
 3 : [time] ( t d - s)   1-  43200 U* D2*
 4    2SWAP @T/SEC SWAP M*/ D+  TZONE 3600 M* D- ;
 5 : [now] ( - d)   @TIME TODAY @ [time] ;
```

Change this constant as appropriate (It is permissible to set it to zero and simply operate on UTC if you wish, but it will be reflected in all time of day data displayed by polyFORTH.)

4.2.3 Block 559: Set Network Configuration

As shipped, block 559 contains settings intended to facilitate use of multiple EVB001 boards from a common source base provided by the Terminal Emulator. This is why line 7 loads block 24009 (from the SPI flash) and the remaining configuration phrases are commented. Unless you have this problem, comment line 7, then edit and un-comment those lines that are necessary. The discussion here covers the minimum to get the system up; see DB008 for discussion in depth of all the configuration capabilities in this package.

```
2959
 0 As hardware interfaces are loaded here, it is crucial that like
 1    hardware instances be grouped together.
 2 Lan data links are configured as well.  Each physical interface
 3    must have at least one LAN declaration, which must then be
 4    referenced appropriately elsewhere, or no outbound traffic
 5    will ever occur on it.
 6 INET  defines a connected IP network on the preceding LAN given
 7    max no. of local hosts to cache for ARP (0 if no ARP),
 8       ( maxhosts INET d.d.d.d hexmask)
 9       (          IPADR .d or d or d.d ... )
10 IPREF  selects primary address for n'th INET declaration to use
11    as preferred source address on a (slip) connection.
12 Set GATE with IP address of the only gateway we know about, or
13    leave it zero if none is known.
14 Set eROUTE nonzero if this box is to act as an IP router.
```

```
559
 0 ( Configuration)
 1 ( Ether Cluster Mark 1)  IBUFS 31  0  582 tLOAD
 2 ( LAN route)  OU' 1_ETHER 0 rte !  OU' IEEE802.3 1 rte !
 3    OU' ETHERTYPE 2 rte !
 4
 5 CREATE TEMP  ( Config aids)  554 tLOAD
 6
 7 ( Which board)  24009 LOAD
 8
 9  ( HEX 0001 S/N  DECIMAL)
10  ( INET 76.252.160.88 FF.FF.FF.F8 IPADR .89 ( creek)
11  ( INET 192.168.77.0  FF.FF.FF.00 IPADR .14 ( h-ga144)
12  (    IP: 192.168.77.1  GATE 2! )
13  (    IP: 76.252.160.94  GATE 2! )
14
15 FORGET TEMP
```

Line 7: Comment it for now.

Line 9: Uncomment by removing both parentheses and edit. The definition of S/N provides the low order 16 bits of a MAC address to be presented by this interface. Its value should be the serial number of your EVB001. We write it in hex so that it can be easily recognized in packets and in ARP tables for debugging.

Line 10: Remove the first left parenthesis and edit. The first string following INET is the IP subnet origin (host zero) for the primary network on which this system will operate. The second string is the netmask and must be in hex. The string following IPADR in the next phrase is the host address portion of the IP address and may consist of one or more octets as appropriate for the netmask in use. By our convention, the following comment is the host name.

Line 12: Uncomment by removing both parentheses and edit. The string following IP: is the full IP address for a router or NAT box on the same network as identified in line 10 that may be used as a gateway for sending IP packets to hosts not on this network.

Leave the rest alone until you have studied the TCP/IP package documentation.

4.2.4 Block 540: Set IP address of TIME host

As shipped, the networking package tries to set its clock using the date and time information obtained from some host that supports the UDP TIME service on port 37. This protocol returns a "unix time", the time of day in seconds since 0000 UTC on 1 Jan 1900. If the query is successful, the polyFORTH date and time are set to the equivalent in whatever local time zone was specified by TZONE and will be maintained thereafter using the 10 MHz clock.

```
2940
 0 Main load block for TCP/IP.
 1 Considerably simplified from ATHENA x86 model.
 2
 3 - No routing support
 4 - No IPSec support
 5 - Only one net, one IP addr.
```

```
540
 0 ( TCP/IP)   EMPTY   DECIMAL
 1 ( MSG .IPV  34 STRING IP.5b 12/22/10")
 2 : +t ( n - n)   [ BLK @ 540 - ] LITERAL + ;
 3 : tLOAD   +t LOAD ;   : tTHRU   +t SWAP +t SWAP THRU ;
 4 ( Extensions  480 LOAD)   ( Sys dep's)  543 544 tTHRU
 5 ( Parms)  549 553 tTHRU   ( Standard)  541 tLOAD
 6 ( Hosts  555 557 tTHRU)   ( Config)  559 560 tTHRU
 7 ( Fire up)  542 tLOAD
 8
 9 CR .( Size ) HERE ' +t - .  FORTH GILD  3000 MS
10 ( Clock)  760 761 THRU  CR IP: 192.88.236.6 SYNCLK-ONE   EMPTY
11    @TIME TODAY @  ' Bdt 1+ !  ' Btim 1+ 2!  FORTH GILD
12 EXIT  FORTH CAP NETWORK   ( FTP printing  736 tLOAD)
13    ( Export API  netroot EXECUTE  6 FH 6 FH THRU)
14    ( Net print  981 540 - FH LOAD)
15 FORTH 'CAPSULE @ (CAP)  NETWORK
```

The address shown on line 10, 192.88.236.6, is that of a system at a GreenArrays office that provides this service. Although we recommend that you select a service provider within your own network infrastructure, you may leave this unchanged initially; if GATE has been set for a router or NAT box that has access to the global Internet, you should be able to get a usable time setting from our system

4.2.5 Loading the Networking Package

After making the above changes, cleanly boot polyForth and say **HI** .

Then type **ETHER LOAD** to compile the networking package, activate the NIC, and attempt to set the time. If the latter fails, you will get an error message but the package will still have been loaded and activated, assuming all of the steps above have been taken correctly.

You may verify that something is happening by using **.SUMM** to display the table of counts for incoming and outgoing traffic.

You may PING other hosts by typing **PING 192.88.236.1** (substitute desired IP address) and may view the hops taken enroute by typing **TRACERT 192.88.236.1** (again substitute desired IP address.)

Please contact our hotline for further assistance if needed, and please refer to DB008 for more information about use of the networking package.

4.3 polyFORTH Driver

As an example of controlling this NIC from high level Virtual Machine code, here's one version of the "driver" code from the polyFORTH TCP/IP Networking Package as of release 2b. This code will not be discussed here, nor will it be kept current; please read it for a general understanding only; please refer to the current arrayForth distribution for the current source code, and to DB008 for a discussion of the driver and its usage.

```
2982
 0 Interface to GreenArrays 10baseT-fd Ethernet Cluster, Mark 1.
 1    P_X=mode; 1 )x=# buffers to take from IBUFS
 2 For now we use MAC address 00-44-44-EE-ss-ss where ssss is the
 3    decimal serial number of the eval board.  OUI 00-44-44 is not
 4    registered with IEEE.
 5
 6 All packets provided by the cluster are processed.  Multicasts
 7    may be inhibited within the cluster.  We may get unicast
 8    packets addressed to others until/unless the switch knows
 9    who we are.
10 This version uses two tasks.  OETH is the one awakened by the
11    cluster; it recognizes both inbound and outbound completion.
12
13 RELOAD  is extended to shut down receiver, wait long enough for
14    both tx and rx to complete, and sending ether cluster dormant
```

```
582
 0 ( G144 10baseTFD Mk1 Cluster)
 1 ( q nb md ix ox typ n)  2 0 1 lHDW PORTAL 1_ETHER  IN: -BUF
 2 ( Agreed with cluster)  32 CONSTANT #rxd
 3
 4 ( Mode)  P_X !   ( Bufs)  #rxd 1- MIN 1 )x !  P_HOME !
 5 HEX HERE  0044 , 44EE , ( Serial) 0000 ,   DECIMAL
 6 : S/N ( n)   LITERAL SWAP OVER 2 + !  1_ETHER
 7    OUT: DUP P_ADR 3 MOVE  IN: P_ADR 3 MOVE  OWN: ;
 8
 9 UP  64 #R BACKGROUND IETH  IETH @ IN: P_task !
10 ( Set task)  OWN: 'portal @  IN: P_task 'portal 1+ HIS !
11 UP  64 #R BACKGROUND OETH  OETH @ OUT: P_task !
12 ( Set task)  OWN: 'portal @  IN: P_task 'portal 1+ HIS !
13
14 ( Rest)  DECIMAL 1 FH 5 FH THRU   OWN:
15 FORTH HERE ]  -RX  20000 FOR NEXT  -DMA  +RELOAD [
```

```
2983
 0 t.cm  is command, zeroed when taken.  Rx descriptor pool:
 1   t.pf  adr of next descr to process, =t.rx if none.  We own.
 2   t.rx  adr of next descr to be used, =t.ep if none.  Nodes own.
 3   t.ep  adr of next empty descr for freed buffer.  We own.
 4 t.lk  link status; t.dp dbl count of drop no bufs
 5 t.sk  ^user area to awaken for tx/rx; t.wk value of  wake .
 6 t.xa  dbl xmit buf adr, t.xn count (neg force link down, 0=done)
 7 t.pp  Poll period for commands.  100 gives 4.5% overhead and
 8    target works; 10 gives 23.7% and makes serial disk fail.
 9 t.tt  ^user area to awaken for timing prodding.
10 t.rxd  descriptor tbl:  +0 dbl store adr, +2 'bufr, +3 unused.
11
12 wdbp  incoming origin offset from B_IP.
```

```
583
 0    ( Data base)   t.cm 1+
 1    1 OFS t.pf   1 OFS t.rx   1 OFS t.ep
 2    1 OFS t.lk   2 OFS t.dp   1 OFS t.sk   1 OFS t.wk
 3    2 OFS t.xa   1 OFS t.xn   1 OFS t.pp   1 OFS t.tt   DROP
 4
 5 #rxd 4 * -1 XOR CONSTANT rxdm   HEX HIGH U. DECIMAL
 6 ( Align to size)  HIGH 1- #rxd 8 * NEGATE AND #rxd 8 * + 'HI !
 7 HEX HIGH U. DECIMAL #rxd 4 * HTABLE t.rxd  HEX HIGH U. DECIMAL
 8
 9 CREATE etb0   0 ,  t.rxd DUP , DUP , ,  0 , 0 , 0 ,
10    ( tsk) OUT: P_task @ ,  WAKE ,  0 , 0 ,  0 ,  10000 ,
11    TMON @ ,
12 etb0 t.cm t.tt OVER - 1+ MOVE   FORGET etb0
```

```
2984
 0 Cluster input buffers are dedicated.  The number of such buffers
 1    MUST be less than the size of the descriptor pool.
 2 iHQ  is a pseudoqueue to which completed input buffers are
 3    returned.  The appending method stores DMA and buffer struc
 4    addresses into the next available descriptor for rx use.
 5 <ibufs  scarfs buffers from general pool.
 6
 7 CMD  gives DMA active ether a command and waits till it's taken.
 8 +RX  enables receiver operation
 9 -RX  disables after current op completed (if any active)
10 -DMA  deactivates all DMA; cluster waits for a stimulus.
11 +TX  starts transmission of the given packet, sans CRC.
12
13 +ETHER  unmasks and stimulates the cluster, enables receiver ops
14    forces link autonegotiate and marks both sides of the portal
15    active.
```

```
584
 0    ( Operations)   HEX
 1 : !pq ( b q)   DROP  t.ep @ >R  DUP I 2 + !  'bufr @!
 2    B_PLATE t.ep @ 2!  R> 4 + rxdm AND t.ep !  'bufr ! ;
 3 QUEUE iHQ  8000 HERE 1- OR!  ' !pq ,
 4
 5 : <ibufs   IN: 1 )x @ 1- FOR  P_HOME @ ?QUE IF  P_HOME @ deq
 6       'bufr !  iHQ B_HOME !  Notify  THEN NEXT ;
 7 <ibufs   FORGET <ibufs
 8
 9 : CMD ( n)   t.cm !  BEGIN PAUSE  t.cm @ 0= UNTIL ;
10 : -DMA  15 CMD ;  : -RX  11 CMD ;  : +RX  8011 CMD ;
11 : +tx  32 CMD ;
12 : +TX ( d n)   BEGIN t.xn @ 0= UNTIL  t.xn ! t.xa 2!  +tx ;
13 : +ETHER   MMASK @ MMBITS @ OR !MMASK  MMBITS @ !MMSTIM
14       +RX  -1 DUP DUP +TX
15    83 OUT: P_FLAG -&!  83 IN: P_FLAG -&! ;
```

```
2985
 0 Assumptions:  Buffer handed from lMAC or equivalent via >OUTER.
 1    Logging, if any, will be handled by inbound task for this
 2    portal when transmission is complete (or canned).  Maximum
 3    permitted buffer on way in to us is 1514 bytes (1500 ether
 4    data plus 6 each dest/source addresses & ether type field.)
 5    Of ethernet header, source address has not been filled yet.
 6
 7 Submit  adds trailing pad boilerplate as needed so the packet
 8    is at least 60 decimal bytes long.  Inserts source address
 9    for our board in the packet header, appends to outbound queue
10    If int code inactive, starts transmission with an SWI.
11 Packets submitted for transmission while portal's OUTBOUND side
12    is marked SHUTDOWN or INHIBIT are notified as undeliverable.
13
14 +mrq  requests a Multicast Setup command.  Used to restart hung
15    receivers per Intel published errata.
```

```
585
 0 ( Tx submit)   HEX   VARIABLE txact
 1 : Txgo   B_LAYERS 2@  B_TEXT 2@ t.xa !  + t.xa 1+ !
 2    t.xn !  +tx  tCLK 2@ B_TIME 2!  -BUF ;
 3
 4 : Submit   0 3C B_LNG -  0 MAX +BP   0 )t 2TALLY
 5    P_ADR 0 B_DD SWAP 3 + SWAP 3 XMOVE
 6    'bufr @  P_FLAG @ 3 AND IF  3 )t 2TALLY
 7          -BUF -notify EXIT  THEN  P_QUE >tail
 8       txact @ 0= IF  1 txact !  Txgo  THEN -BUF ;
 9 ' Submit  OUT: P_SUB !
10
11 : Txdone   OUT: P_QUE deq 'bufr !  1 )t 2TALLY
12    B_EVENT @ Up 80 OR B_EVENT !  B_STATUS @ Up 1+ B_STATUS !
13    P_FLAG 1+ @ 81 AND IF  B_LINKS IN: P_QUE >tail  OUT:
14       ELSE Notify  THEN  P_QUE ?QUE IF  P_QUE @ 'bufr !
15          Txgo EXIT  THEN 0 txact ! ;
```

```
2986
 0 OETH  receives stimuli from the cluster.
 1
 2 Inbound packets are processed and queued to IETH.
 3 Completed transmits are queued to IETH for potential logging.
 4    Status from cluster is combined with broadcast, multicast and
 5    snoop.
 6 When a transmit completes, queue is checked and the next is
 7    started if necessary.
 8
 9 This task never waits for anything else, has minimal stack re-
10    quirements, and can accept cluster stimuli at any time.
```

```
586
 0    ( OETH Task)   HEX
 1 : Og   BEGIN  t.pf 2@ XOR WHILE  t.pf @ DUP 2 + @ DUP 'bufr !
 2       B_TEXT @ b_data !  tCLK 2@ B_TIME 2!
 3    1 0 B_LYR 2!  0 B_LAYERS !  IN: P_ORG B_LAYERS 2 + !
 4    B_TEXT 1+ @  DUP 1+ B@  4 + ABS 5F2 MIN B_LAYERS 1+ !
 5       DUP 2 + B@ >R  DUP 3 + B@  OVER 4 + B@ AND  I AND 1+ 0= IF
 6       R> DROP 40  ELSE  R> 1 AND 1 = 20 AND  THEN DUP >R 0= IF
 7          DUP 2 + B@ P_ADR @ -  OVER 3 + B@ P_ADR 1+ @ - OR
 8          OVER 4 + B@ P_ADR 2 + @ - OR IF  R> 10 OR >R  THEN THEN
 9       B@ 77 AND IF 2  ELSE 1  THEN R> OR B_STATUS !  0 )t 2TALLY
10       FF01 B_EVENT !  P_QUE >TAIL  4 + rxdm AND t.pf ! REPEAT
11    txact @ IF  t.xn @ 0= IF  Txdone  THEN THEN ;
12
13 VARIABLE ARGH   VARIABLE NST
14 : +OETH   OETH ACTIVATE  OWN:  BEGIN NST TALLY STOP
15       Og  DEPTH UNTIL  ARGH TALLY NOD ;  RECOVER
```

```
2987
 0 Inbound packets are preceded by the 32 bit value RDES0 which we
 1    shift down for convenience.
 2
 3 Up  is a kludge till the flags really work
 4 Dn  is a kludge, should be handled with a message and a way
 5    to determine when everyone's finished.
 6
 7 More messages needed such as change mode.
```

```
587
 0    ( Control)
 1 IN' IEEE802.3 CONSTANT Pal   OWN:
 2 : Ib   4 0 -BP  ?LOG  ?OK IF  Pal >INNER  ELSE Notify  THEN ;
 3 : Ob   B_LINKS lHDW !BUFR  OUT: ?LOG  Notify ;
 4
 5 VARIABLE OOPS
 6 0 ARRAY Evec   ' Ob , ' Ib , ' Notify , ' Notify ,
 7    ' Notify DUP , DUP , DUP , ,
 8 : 0In   OWN: IN: P_ORG 'portal 1+ !  +OETH  +ETHER
 9    BEGIN  PAUSE  OWN: P_QUE deq  !IN  7ev Evec @EXECUTE
10       DEPTH UNTIL  OOPS TALLY NOD ;
11
12 : Uu   IN: P_FLAG @ 1 AND IF  1 P_FLAG -&!  2 P_FLAG OR!
13    IN: P_task ACTIVATE  0In  THEN ;   ' Uu IN: P_UP !
```

GreenArrays®

Application Note AN007 *Revised 11/05/14*

IMPORTANT NOTICE

GreenArrays Incorporated (GAI) reserves the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to GAI's terms and conditions of sale supplied at the time of order acknowledgment.

GAI disclaims any express or implied warranty relating to the sale and/or use of GAI products, including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright, or other intellectual property right.

GAI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using GAI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

GAI does not warrant or represent that any license, either express or implied, is granted under any GAI patent right, copyright, mask work right, or other GAI intellectual property right relating to any combination, machine, or process in which GAI products or services are used. Information published by GAI regarding third-party products or services does not constitute a license from GAI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from GAI under the patents or other intellectual property of GAI.

Reproduction of GAI information in GAI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. GAI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of GAI products or services with statements different from or beyond the parameters stated by GAI for that product or service voids all express and any implied warranties for the associated GAI product or service and is an unfair and deceptive business practice. GAI is not responsible or liable for any such statements.

GAI products are not authorized for use in safety-critical applications (such as life support) where a failure of the GAI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of GAI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by GAI. Further, Buyers must fully indemnify GAI and its representatives against any damages arising out of the use of GAI products in such safety-critical applications.

GAI products are neither designed nor intended for use in military/aerospace applications or environments unless the GAI products are specifically designated by GAI as military-grade or "enhanced plastic." Only products designated by GAI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of GAI products which GAI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

GAI products are neither designed nor intended for use in automotive applications or environments unless the specific GAI products are designated by GAI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, GAI will not be responsible for any failure to meet such requirements.

The following are trademarks or registered trademarks of GreenArrays, Inc., a Nevada Corporation: GreenArrays, GreenArray Chips, arrayForth, and the GreenArrays logo. polyFORTH is a registered trademark of FORTH, Inc. (www.forth.com) and is used by permission. All other trademarks or registered trademarks are the property of their respective owners.

For current information on GreenArrays products and application solutions, see www.GreenArrayChips.com

Mailing Address: GreenArrays, Inc., 774 Mays Blvd #10 PMB 320, Incline Village, Nevada 89451
Printed in the United States of America
Phone (775) 298-4748 fax (775) 548-8547 email Sales@GreenArrayChips.com