A Really Long Extension Cord: Final Report

Miles Silva
Department of Brain & Cognitive Sciences
Massachusetts Institute of Technology
Cambridge, MA, USA
mbsilva@mit.edu

Jordan Wilke
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
wilke18@mit.edu

***Abstract*—We present a system that enables remote gameplay of the retro game console the *Nintendo Entertainment System (NES)*. Our system streams video from the NES over the internet to a remote transceiver, which in turn receives input from physical controllers and sends them over the internet to the NES. Due to various hardware issues, including issues with 4-bit ethernet and HDMI input, the video that is sent to the remote FPGA is from a camera pointed at the screen on the NES side. Although as yet untested, the design outlined in this report should enable sending packets sending packets over the internet upon addition of an ARP module by connecting to routers via ethernet.**

Keywords—retro gaming, internet, networking, Field Programmable Gate Arrays

I. General overview of system

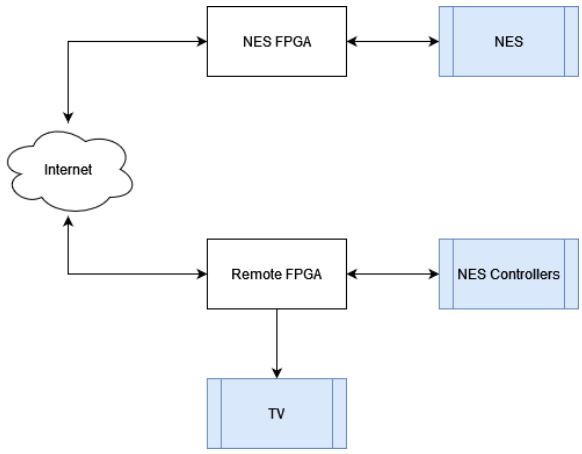

Fig 1. High-level overview of system loop.

The ultimate goal of our project is to revolutionize the classic Nintendo Entertainment System (NES) experience with modern day technology. Previously, the NES could only be played in one place with up to two players. We seek to free gamers of this physical constraint of needing to be in the same room as the NES by providing a network card that would allow users to play remotely without taking away from the playing experience.

Our system consists of two FPGAs: one connected directly to an NES, hereafter referred to as the NES FPGA (N-FPGA), and one that connects to a screen and takes in controller input which we will refer to as the remote FPGA (R-FPGA).

As Fig. 1 demonstrates, the main architecture of our system has the N-FPGA take video from the NES (which we ultimately did using a camera pointed at a TV screen that the NES was hooked up to) and send this to the R-FPGA. The R-FPGA then decodes the video data it receives and outputs it over VGA to the TV. In the reverse direction, the R-FPGA takes input from the NES controllers and sends that to the N-FPGA to decode and output into the NES.

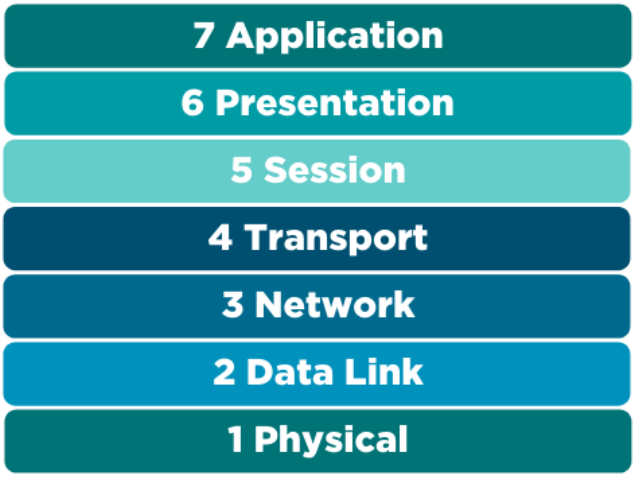

Fig 2. General OSI model consisting of 7 layers

In order to achieve these goals, we developed a system following the OSI model shown in Fig. 2, originally defined in the late 1970s, which provides a general architecture of networked communication (variants of which are still used today). Of this model, we define and implement the bottom 4 layers, while the top 3 layers (consisting of the session, presentation, and application layers) were all encapsulated into a single application layer for sending data.

As described in subsequent sections, the physical and data link layers are implemented using ethernet, the network layer primarily consists of the Internet Protocol (IP), and the transport layer uses the User Datagram Protocol.

Though the system never got fully off the ground, the remainder of this paper will describe the implemented architecture that, if given more time, could be integrated together to allow people from anywhere in the MIT community to play the NES remotely.

II. Network Stack

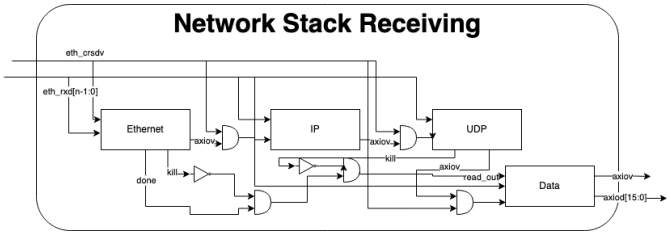

Fig 3. Network receiving block diagram logic

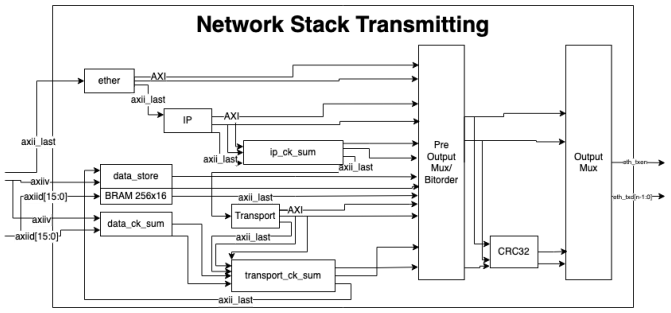

Fig 4. Network transmission block diagram logic

A. Ethernet Receiving and Transmission

Ethernet is one of the primary modes of transferring data in the modern world. Within the OSI model, ethernet actually makes up both the physical layer and the data link layer. The physical layer is the actual ethernet wires you might have plugged into your computer and is the very bits being sent on the wire, while the data link layer is more about the processing of those bits from one end of the cable to another into the actual ethernet jacks.

Our original plan was to use a parameterized ethernet receiving and transmission module to enable 4-bit ethernet on the video FPGA. However, after implementing this parameterized module, we ran into an intractable bug with 4-bit transmission (see *Appendix* for details). We were unable to resolve the error, so we instead will only use the 2-bit ethernet protocol.

We implemented 2-bit ethernet transmission to the spec defined by IEEE 802.3-2008 [1]. Using this standard we can successfully transmit arbitrary data over ethernet to any Media Access Control (MAC) address. MAC addresses are an important part of networking as they provide a unique (either globally or locally) address for every piece of hardware such that communication is able to be differentiated to different devices.

Like with almost any technology, there is a possibility for errors when sending over ethernet which is why the specification includes a 32-bit sequence, known as the Frame Check Sequence (FCS), that allows for the detection of errors within an ethernet frame, which for us is a full packet or a full message. The FCS is calculated using the CRC32-BZIP2 algorithm, which is efficient to do in hardware and less prone to failure than other methods (like the checksums in the network and transport layers described later).

This checksum process is highlighted in Fig. 3 which details the process that a received network packet goes through and in Fig. 4 which details the process for sending a network packet. In Fig. 3, the FCS computation is encapsulated within the Ethernet module. As data is received from the Ethernet wire (the physical layer in the OSI model) it is passed directly into the FCS such that all of the important data within a frame is checked for validity. Once the entire frame is received, the ethernet module reports a *done* signal and a *kill* signal, where *kill* is high if the FCS received does not match the one we calculate for the received data.

When transmitting, a very similar though equally complex process happens. As seen in Fig. 4, as different parts of the Ethernet frame are sent out onto the wire, they are also fed into the CRC32 algorithm such that it can send an accurate FCS at the end of the frame.

This was tested by first verifying correct reception of transmitted packets by another FPGA running our ethernet receiving module. During testing, we used arbitrary data and sent it to the broadcast destination address (FF:FF:FF:FF:FF:FF). Once this was verified to be received correctly, we connected our FPGA to a computer and sniffed the packets using Wireshark, verifying that Wireshark received the packets correctly as well.

B. Internet Protocol v4 (IPv4)

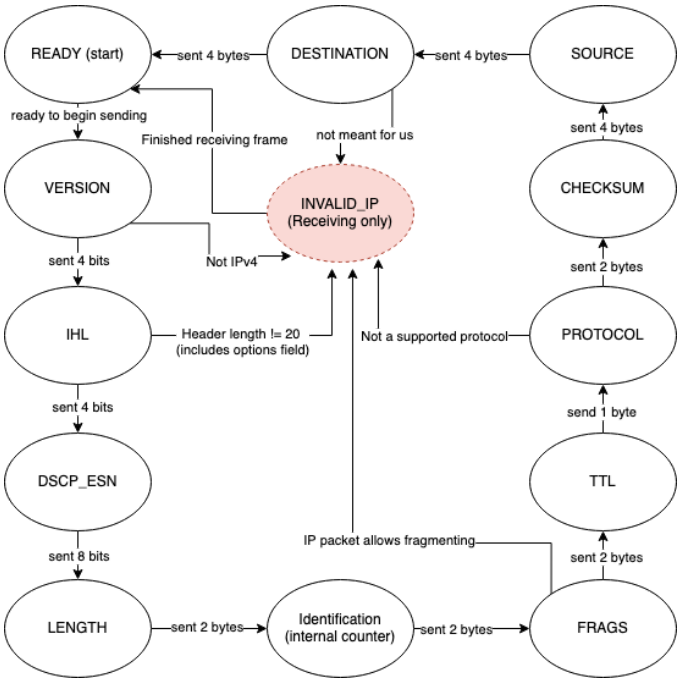

Fig 5. State machine describing IPv4 transmission and reception

The network layer is extremely important for communication. In the lower layers, ethernet provides a means for transmitting data and MAC addresses offer a way to disclose where data should go, but this does not allow for data to traverse between different networks. Without a network

layer, all the transmission of data is just a set of really long extension cords between devices.

All of the communication between the FPGAs within our project use the 4th version of the Internet Protocol (IPv4) as the network layer. IPv4 is the most prominent network layer protocol because of its simplicity yet usefulness. Similar to how ethernet used MAC addresses to uniquely identify devices, IPv4 assigns unique[1] addresses for internetworking, the ability to communicate between different networks. A simpler implementation[2] of IPv4 has been implemented according to RFC 791 [2].

Though our design is meant to work within the MITNet, this network is actually a very large network of networks such that ethernet communication from Building 38 where the N-FPGA is located needs to be able to internetwork with ethernet within other MIT buildings.

Fig. 5 describes the flow of data that is transmitted or received when using IPv4. For simplicity we will not go through what all the fields do, but rather make note of some of the important states within the flow.

An important design choice that we made is making sure that our messages aren't fragmented by keeping their length below 576 bytes[3]. The IPv4 header is always a fixed length of 20 bytes since we don't use the options field. Our messages are then encapsulated in a UDP header which is also a fixed length of 8 bytes. This leaves us the ability to send 558 bytes of data without it being fragmented. The controller data transmitted by R-FPGA is only 8 bits and we only send one line of each frame (240x320 pixels, each 12 bits) at a time which is 480 bytes of data, leaving us plenty of room in each packet.

When receiving, the header checksum is calculated as the IP header streams in and is used to cancel further processing if it is incorrect. This checksum is quite different from the FCS used by ethernet frames. The IPv4 checksum is the one's complement sum of all 16-bit words[4] in the header. This means that when summing all of the words together, if the sum is greater than 0xFFFF, the carry is added back into the sum which causes a rollover (ie 0xFFFE + 0x2 = 0x1). This sum is then complemented and added to the packet header such that when the receiver sums up all of the 16-bit words in the received header, the result should be 0xFFFF.

When transmitting, the header checksum is calculated as the other portions of the header are sent with the exception of the source and destination IPs. This can be seen in Fig. 4 as the output of the IP module is fed into an IP checksum. Since both the source IP address and the destination IP address must be known before we begin transmitting, we are able to precalculate their checksums and add them into the whole header checksum. This was a design decision that we made such that the transmission of packets could be a streamed process, requiring no delay once the data is ready to send.

C. User Datagram Protocol (UDP)

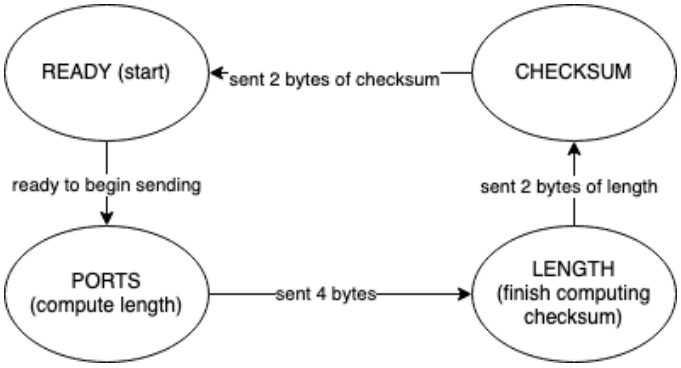

Fig 6. State machine describing the receiving and transmission of UDP headers

The transport layer in the OSI model is not always used since it is not always necessary, unlike the network layer. The transport layer is important for when devices want to send varying amounts of data, while also guaranteeing certain quality-of-service functions such as reliability or low-latency.

For the transport layer, we chose to use the User Datagram Protocol (UDP) for delivering our data in accordance with RFC 768 [3]. The UDP header that is added before the actual transmission of data is fairly small (8 bytes) and only contains 4 fields: the source port, destination port, packet length, and checksum (see Fig. 6).

The ports are designed to allow servers to multiplex UDP packets that they are receiving, but this is inconsequential for us as we are only communicating between 2 FPGAs. The packet length field is meant to be the length of the UDP header plus the length of all of the data being sent. Similarly, the checksum is computed on the UDP header and all of the data using the same checksum calculation that the IPv4 header uses.

As noted, the UDP length and checksum take into account the data that is being transmitted which wasn't a concern for the IPv4 header checksum. This data can have varying length, theoretically, and will affect the checksum meaning that a streaming approach to calculating these values would not work.

For this reason, we chose a different approach to knowing the data length and checksum that can be seen in Fig. 4. We cannot start transmitting until all of the data is ready to be sent, however this does not mean we can't begin calculating the length and checksum of the data as we queue it up to be sent. Since the checksum calculation uses one's complement addition, rather than the FCS's CRC32-BZIP2 algorithm, the checksum of the data can be precalculated and then just added to the UDP header checksum (since the checksums themselves are just 16-bit words).

An alternative consideration for the transport layer would have been to use another common protocol such as the Transmission Control Protocol (TCP). TCP at first was a very alluring choice because if implemented properly, it guarantees

[1] Not actually unique

[2] Simpler is not technically true

[3] This is the total number of bytes that routers are required to be able to send within a single packet, however many can support more data than this.

[4] Words are just chunks of data

reliable transport making sure that every packet arrives (so dropped packets aren't lost) and that even if packets arrive out of order, they will be properly ordered. However, TCP comes with two large drawbacks: larger complexity and high latency. Both are caused by the logic that guarantees reliability.

On the other hand, UDP is designed to be quick and simple because it does not give any reliability guarantees (i.e. if a packet gets dropped the receiver will never even know it was sent). Additionally, packets sent can arrive in any order, and the UDP logic leaves it to the application layer to handle reordering. Since the bottleneck of our design is already the throughput and latency of the network we are using, having a transport protocol with low latency is more important. Additionally, reliability is not a huge concern with video data as there will not be large changes between most frames, so old lines of pixels can be used until the same line of the next frame is received which would be an imperceptible wait time to the human eye.

III. Hardware Interfaces

A. Video Interface

Our original plan was to connect to the NES via HDMI input by connecting an RCA to HDMI adapter between the NES and our FPGA. However, the adapter we received outputs only in 1920x1080 resolution, which our HDMI receiver code was unable to handle correctly. After carefully working through the code and considering our options, we made the unfortunate decision to discard the HDMI interface part of our project. Instead, we have decided to connect our FPGA to a camera, and send that data over the wire instead.

Our video interface consists of two parts: a linebuffer module, and a framebuffer module.

The linebuffer lives on the N-FPGA, and buffers in one line at a time from the camera, then sends this line to the network stack to be sent over the wire. This module works with two 320x16 single-port BRAMs. One BRAM is filled with new pixels directly from the camera, one 16-bit pixel at a time, until the end of the line is reached, then the module immediately sends data out from that BRAM as it reads in data to the other BRAM. With each line, the BRAMs are flipped so that one is constantly being written to while the other is read from. This structure implicitly handles clock domain crossing, as the data is able to be written to the BRAM on the 16.67 MHz clock of the camera, while being read out at the 50 MHz clock of ethernet.

On the R-FPGA, each line is received from the network stack, one pixel at a time. Each pixel is sent to a pixel decoder module, which converts the 16-bit color into 12-bit color to be ready for VGA output, which uses 12-bit color. These pixels are then sent directly into the framebuffer module, which calculates the correct framebuffer BRAM address to write to, based on an internal counter of how many pixels it has received up to that point. The framebuffer BRAM is a (240*320)x12 dual-port BRAM that contains each pixel in the frame. These pixels are then read out to the VGA pins based on timing from a VGA module. This structure also handles clock domain crossing, as the module can write to the BRAM with the 50 MHz ethernet clock, and read from the BRAM on the 65 MHz VGA clock.

B. Controller Interface

We connected an original NES controller to the FPGA using the PMOD ports. We followed the specification laid out in Figure 7. The end of an NES controller cord has 7 pin-outs, however of these, only 5 are connected to the controller itself. The other two do not lead anywhere. Of the five connected wires, one is power, one is ground, one sends the latch signal, one sends the pulse signal, and one sends the data signal. See Figure 4a.

In a typical NES controller setup, every 60Hz, the NES sends a latch signal to the controller. This lasts 12µs, then goes low again. 6µs later, the NES will send the first of 8 pulses that last 12µs total, 6µs high and 6µs low. The pulse line is high at rest, and valid low. Each pulse corresponds to a button, in the order of {A, B, Select, Start, Up, Down, Left, Right}. When pulse is asserted low, the data line becomes high if that button is being pressed, and low if that button is not pressed. See Figure 4b.

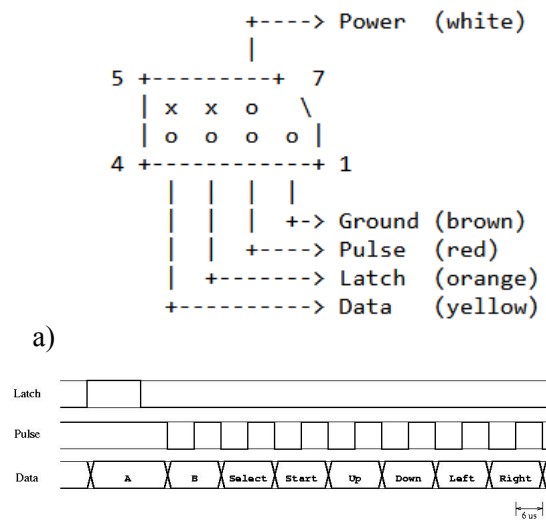

Fig 7. Diagrams of controller information that is sent to the NES[4]. a) A wiring diagram for an NES controller b) The data sheet representing how the NES reads the data from the controller

We were able to mimic this setup on our own FPGAs. The R-FPGA connects to an NES controller via the PMOD ports, and mimics the latch and pulse signals of an NES, and reads the data line in from the controller. The N-FPGA connects to the NES, reads the latch and pulse signals from the NES and mimics the data line back to the console.

Each time the R-FPGA's controller module detects that a button has either been pressed or released, it sends a packet to the network stack with the new button state. This way, we do not clog the wire by sending any controller data over the wire that we do not need to. However, to mitigate a corruption issue we faced with the ethernet packets, where arbitrary bytes of ethernet data were corrupted when they were sent over the wire, we send 20 copies of the packet each time the button

state is changed. The N-FPGA receives these packets, and performs a simple corruption check by checking if two copies of the button state data within a packet are equal. If they are, the FPGA instantly updates its internal buffer of which buttons are being pressed. This buffer will remain constant until a new packet is received, telling it that a new button was pressed or released.

IV. Evaluations

A. Resource Utilization

Our design currently uses 1290 Lookup Tables (LUTs) which is only 2% of the total LUTs. This means that we could have done more combinational logic rather than doing some of our math over multiple clock cycles. Overall, we have room for increased logic to handle VGA & HDMI and other inputs that didn't quite make it.

B. Memory Usage

Currently we do not have the build logs to determine exact memory usage, but based on our design documents, we believe that the following will be true of the final product:

- Both the NES and the remote FPGA have have two network buffer BRAMs (one for receiving and one for sending) that are 16 bits wide and 320 entries deep in order to fully accommodate lines coming in (lines are 320, 12-bit pixels long)[5] resulting in 8 Kib
- The NES FPGA has two line buffers that are 320 entries long and 16 bits wide each, for a total of 10 Kib.
- The R-FPGA has a frame buffer that is 240x320 entries and 12 bits per entry equating to 900 Kib

This results in a total of 728 Kib used in total. This of course might need to be adjusted for HDMI as well as cache's for ARP and other protocols.

C. Packets Dropped

While actual dropped packets are very rare, considering the FPGAs are connected directly over ethernet, the N-FPGA fails to read every packet correctly. Over a test of 50 packets, the FPGA received 50 of them, but correctly read only 41 of them, for a success rate of 82%. This success rate is low for our standards, and makes playing games difficult, as a dropped packet means a button is not registered correctly, a big problem when playing fast-paced video games.

D. Limiting Factors

Our ethernet module implements a 100 Mib ethernet standard, and runs on a clock cycle of 50 MHz. As we currently are only sending button presses over ethernet, and we only read button presses at a rate of 60 Hz as required by the NES controller spec, we will send at a maximum 20*60=1200 packets per second (accounting for sending 20 packets per button press as per our corruption mitigation protocol). This is well under the limit posed by ethernet, so the network is not our limiting factor in our design. With the introduction of routers into our design, and the unreliability that comes with that, this may become the limiting factor, as latency for internet connections can approach the milliseconds. As is, the slowest part of our design is the NES controller itself.

E. High Score in Super Mario Bros.

Miles has currently not achieved the high score in Super Mario Bros from Simmons[6] and currently sits around 130,000[7] points away from that top score.

V. References

[1] IEEE 802.3-2008: IEEE Standard for Information technology--Telecommunications and information exchange between systems--Local and metropolitan area networks--Specific requirements Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications

[2] Internet Program Protocol Specification. RFC 791, September 1981.

[3] User Datagram Protocol. RFC 768, November 1982.

[4] Tresi Arvizo, https://tresi.github.io/nes

VI. Code

https://github.com/wilke0818/6111_final_project

[5] Notice that there is a mismatch between the width of the data and the width of the pixels. This is to more easily accommodate data sending for network protocols. We will handle the differing pixel size accordingly.

[6] He has not achieved it in lab either

[7] This number has been fact checked with the lab white board

APPENDIX

I. Spooky Action: 4-bit Ethernet

While implementing 4-bit ethernet transmission, we ran into a strange bug. After extensive testing, our 4-bit ethernet is able to send the correct header, data, and checksum bits (as calculated using a 4-bit version of the CRC32-BZIP2 module generated online that was recommended by the course staff) to the wire, using a 25MHz clock as specified by the spec [1]. However, this data is not being received correctly by the device on the other side. After even more testing, we realized that the data being received on the other side of the wire was mostly correct, except for the last 4 bits received before the valid bit goes low. These 4 bits (i.e. the last 4 bits of the packet, i.e. the last 4 bits of the FCS) are invariably replaced with the bits `0101`. We determined this is not due to an incorrect checksum calculation, as we probed the bits being sent to the physical layer of ethernet and those bits are correct. We also know it is not due to message content, as we tried to send various different messages to no avail. We know as well that the bug does not lie on the receiving end, as we probed exactly what is being received from the physical layer, and the corruption is present in those bits as well. And just to be sure, we tried three different ethernet cords, two different video boards, and two different receiving boards.

To summarize, it is only the last 4 bits before the valid bit goes low that are being corrupted, and this happens at some point after the bits are sent to the physical layer of the Video Board, and before the bits are received from the physical layer of the receiving board.

II. Spooky Action: Off-by-one Receiving

An interesting yet absolutely terrible bug we encountered was that when receiving controller input on N-FPGA, we found that its carrier signal was high for one cycle too long. This doesn't seem like it might be as big of an issue as it seems, but in fact, this causes the interpretation of where the ethernet FCS is to be off which in turn back-propagates to the data being misread.

After a lot of debugging, we found no apparent issues with sending or receiving which leads us to believe that there is either a hardware issue or that the ethernet jack specification is different than what we believe it is.