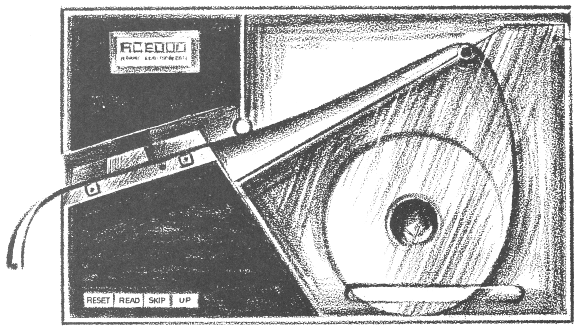
17:05 Up close and personal with Ethernet.

by Andrew D. Zonenberg,
because real hackers don't need PHYs or NICs!

If you're reading this, you've almost certainly used Ethernet on a PC by means of the BSD sockets API. You've probably poked around a bit in Wireshark and looked at the TCP/IP headers on your packets. But what happens after the kernel pushes a completed Ethernet frame out to the network card?

A PC network card typically contains three main components. These were separate chips in older designs, but many modern cards integrate them all into one IC. The bus controller speaks PCIe, PCI, ISA, or some other protocol to the host system, as well as generating interrupts and handling DMA. The MAC (Media Access Controller) is primarily responsible for adding the Ethernet framing to the outbound packet. The MAC then streams the outbound packet over a "reconciliation sublayer" interface to the PHY (physical layer), which converts the packet into electrical or optical impulses to travel over the cabling. This same process runs in the opposite direction for incoming packets.

In an embedded microcontroller or SoC platform, the bus controller and MAC are typically integrated on the same die as the CPU, however the PHY is typically a separate chip. FPGA-based systems normally implement a MAC on the FPGA and connect to an external PHY as well; the bus controller may be omitted if the FPGA design sends data directly to the MAC. Although the bus controller and its firmware would be an interesting target, this article focuses on the lowest levels of the stack.



MII and Ethernet framing

The reconciliation sublayer is the lowest (fully digital) level of the Ethernet protocol stack that is typically exposed on accessible PCB pins. For 10/100 Ethernet, the base protocol is known as MII (Media Independent Interface). It consists of seven digital signals each for the TX and RX buses: a clock (2.5 MHz for 10Base-T, 25 MHz for 100Base-TX), a data valid flag, an error flag, and a 4-bit parallel bus containing one nibble of packet data. Other commonly used variants of the protocol include RMII (reduced-pin MII, a double-data-rate version, which uses less pins), GMII (gigabit MII, that increases the data width to 8 bits and the clock to 125 MHz), and RGMII (a DDR version of GMII using less pins). In all of these interfaces, the LSB of the data byte/nibble is sent on the wire first.

An Ethernet frame at the reconciliation sublayer consists of a preamble (seven bytes of `0x55`), a start frame delimiter (SFD, one byte of `0xD5`), the 6-byte destination and source MAC addresses, a 2-byte EtherType value indicating the upper layer protocol (for example `0x0800` for IPv4 or `0x86DD` for IPv6), the packet data, and a 32-bit CRC-32 of the packet body (not counting preamble or SFD). The byte values for the preamble and SFD have a special significance that will be discussed in the following section.

10Base-T Physical Layer

The simplest form of Ethernet still in common use is known as 10Base-T (10 Mbps, baseband signaling, twisted pair media). It runs over a cable containing two twisted pairs with 100 ohm differential impedance. Modern deployments typically use Category 5 cabling, which contains four twisted pairs. The orange and green pairs are used for data (one pair in each direction), while the blue and brown pairs are unused.

When the line is idle, there is no voltage difference between the positive (white with stripe) and negative (solid colored) wires in the twisted pair. To send a 1 or 0 bit, the PHY drives 2.5V across the pair; the direction of the difference indicates the bit value. This technique allows the receiver to reject noise coupled into the signal from external electromagnetic fields: since the two wires are very close together the induced voltages will be almost the same, and the difference is largely unchanged.

Unfortunately, we cannot simply serialize the data from the MII bus out onto the differential pair; that would be too easy! Several problems can arise when connecting computers (potentially several hundred feet apart) with copper cables. First, it's impossible to make an oscillator that runs at exactly 20 MHz, so the oscillators providing the clocks to the transmit and receive NIC are unlikely to be exactly in sync. Second, the computers may not have the same electrical ground. A few volts offset in ground between the two computers can lead to high current flow through the Ethernet cable, potentially destroying both NICs.

In order to fix these problems, an additional line coding layer is used: Manchester coding. This is a simple 1:2 expansion that replaces a 0 bit with 01 and a 1 bit with 10, increasing the raw data rate from 10 Mbps (100 ns per bit) to 20 Mbps (50 ns per bit). This results in a guaranteed 1–0 or 0–1 edge for every data bit, plus sometimes an additional edge between bits.

Since every bit has a toggle in the middle of it, any 100 ns period without one must be the space between bits. This allows the receiver to synchronize to the bit stream; and then the edge in the middle of each bit can be decoded as data and the receiver can continually adjust its synchronization on each edge to correct for any slight mismatches between the actual and expected data rate. This property of Manchester code is known as self clocking.

Another useful property of the Manchester code is that, since the signal toggles at a minimum rate of 10 MHz, we can AC couple it through a transformer or (less commonly) capacitors. This prevents any problems with ground loops or DC offsets between the endpoints, as only changes in differential voltage pass through the cables.

We now see the purpose of the `55 55 ... D5` preamble: the `0x55`'s provide a steady stream of meaningless but known data that allows the receiver to synchronize to the bit clock, then the `0xD5` has a single bit flipped at a known position. This allows the receiver to find the boundary between the preamble and the packet body.

That's it! This is all it takes to encode and decode a 10Base-T packet. Figure 6 shows what this waveform actually looks like on an oscilloscope.

One last bit to be aware of is that, in between packets, a link integrity pulse (LIT) is sent every 16 milliseconds of idle time. This is simply a +2.5V pulse about 100 ns long, to tell the remote end, "I'm still here." The presence or absence of LITs or data traffic is how the NIC decides whether to declare the link up.

By this point, dear reader, you're probably thinking that this doesn't sound too hard to bit-bang — and you'd be right! This has in fact been done, most notably by Charles Lohr on an ATTiny microcontroller.[13] All you need is a pair of 2.5V GPIO pins to drive the output, and a single input pin.

100Base-TX Physical Layer

The obvious next question is, what about the next step up, 100Base-TX Ethernet? A bit of Googling failed to turn up anyone who had bit-banged it. How hard can it really be? Let's take a look at this protocol in depth!

First, the two ends of the link need to decide what speed they're operating at. This uses a clever extension of the 10Base-T LIT signaling: every 16 ms, rather than sending a single LIT, the PHY sends 17 pulses – identical to the 10Base-T LIT, but renamed fast link pulse (FLP) in the new standard – at 125 μs spacing. Each pair of pulses may optionally have an additional pulse halfway between them. The presence or absence of this additional pulse carries a total of 16 bits of data.

Since FLPs look just like 10Base-T LITs, an older PHY which does not understand Ethernet auto-negotiation will see this stream of pulses as a valid 10Base-T link and begin to send packets. A modern PHY will recognize this and switch to 10Base-T mode. If both ends support autonegotiation, they will exchange feature descriptors and switch to the fastest mutually-supported operating mode.

Figure 7 shows an example auto-negotiation frame. The left 5 data bits indicate this is an 802.3 base auto-negotiation frame (containing the feature bitmask); the two 1 data bits indicate support for 100Base-TX at both half and full duplex.

Supposing that both ends have agreed to operate at 100Base-TX, what happens next? Let's look at the journey a packet takes, one step at a time from the sender's MII bus to the receiver's.

[13] `git clone https://github.com/cnlohr/ethertiny || unzip pocorgtfo17.pdf ethertiny.zip`

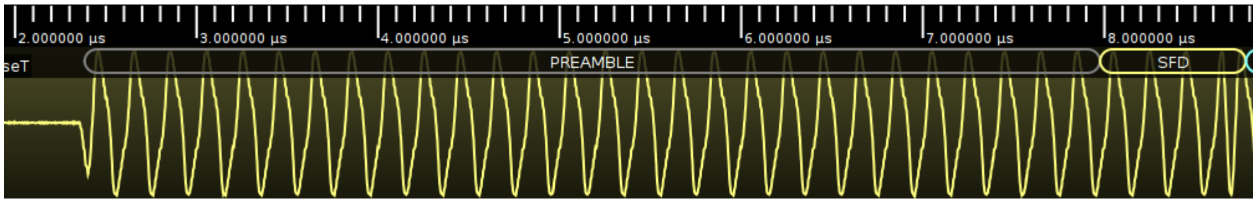


Figure 6. 10Base-T Waveform

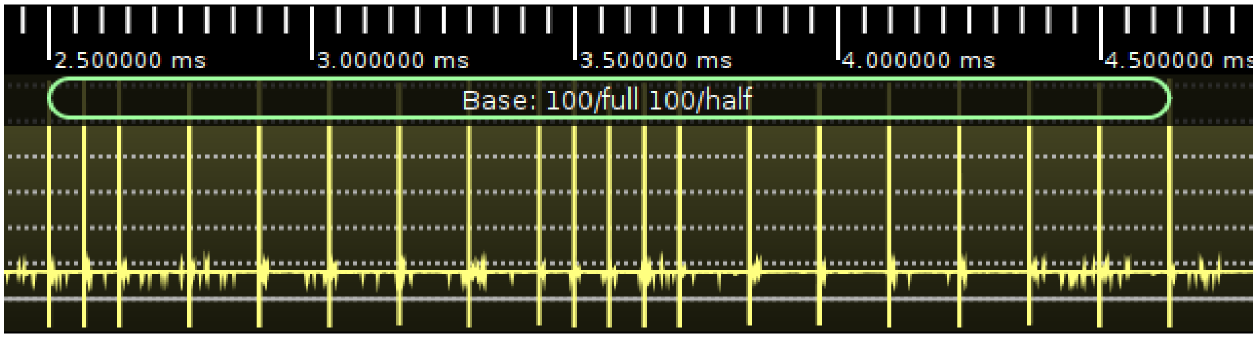


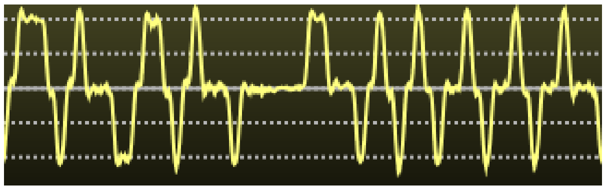
Figure 7. Autonegotiation Frame

First, the 4-bit nibble is expanded into 5 bits by a table lookup. This 4B/5B code adds transitions to the signal just like Manchester coding, to facilitate clock synchronization at the receiver. Additionally, some additional codes (not corresponding to data nibbles) are used to embed control information into the data stream. These are denoted by letters in the standard.

The first two nibbles of the preamble are then replaced with control characters `J` and `K`. The remaining nibbles in the preamble, SFD, packet, and CRC are expanded to their 5-bit equivalents. Control characters `T` and `R` are appended to the end of the packet. Finally, unlike 10Base-T, the link does not go quiet between packets; instead, the control character `I` (idle) is continuously transmitted.

The encoded parallel data stream is serialized to a single bit at 125 Mbps, and scrambled by XORing it with a stream of pseudorandom bits from a linear feedback shift register, using the polynomial $x^11 + x^9 + 1$. If the data were not scrambled, patterns in the data (especially the idle control character) would result in periodic signals being driven onto the wire, potentially causing strong electromagnetic interference in nearby equipment. By scrambling the signal these patterns are broken up, and the radiated noise emits weakly across a wide range of frequencies rather than strongly in one.

Finally, the scrambled data is transmitted using a rather unusual modulation known as MLT-3. This is a pseudo-sine waveform which cycles from 0V to +1V, back to 0V, down to −1V, and then back to 0 again. To send a 1 bit the waveform is advanced to the next cycle; to send a 0 bit it remains in the current state for 8 nanoseconds. The following is an example of MLT-3 coded data transmitted by one of my Cisco switches, after traveling through several meters of cable.



MLT-3 is used because it is far more spectrally efficient than the Manchester code used in 10Base-T. Since it takes four 1 bits to trigger a full cycle of the waveform, the maximum frequency is 1/4 of the 125 Mbps line rate, or 31.25 MHz. This is only about 1.5 times higher than the 20 MHz bandwidth required to transmit 10Base-T, and allows 100Base-TX to be transmitted over most cabling capable of carrying 10Base-T.

The obvious question is, can we bit-bang it? Certainly! Since I didn't have a fast enough MCU, I built a test board (Figure 8) around an old Spartan-6 FPGA left over from an abandoned project years ago.

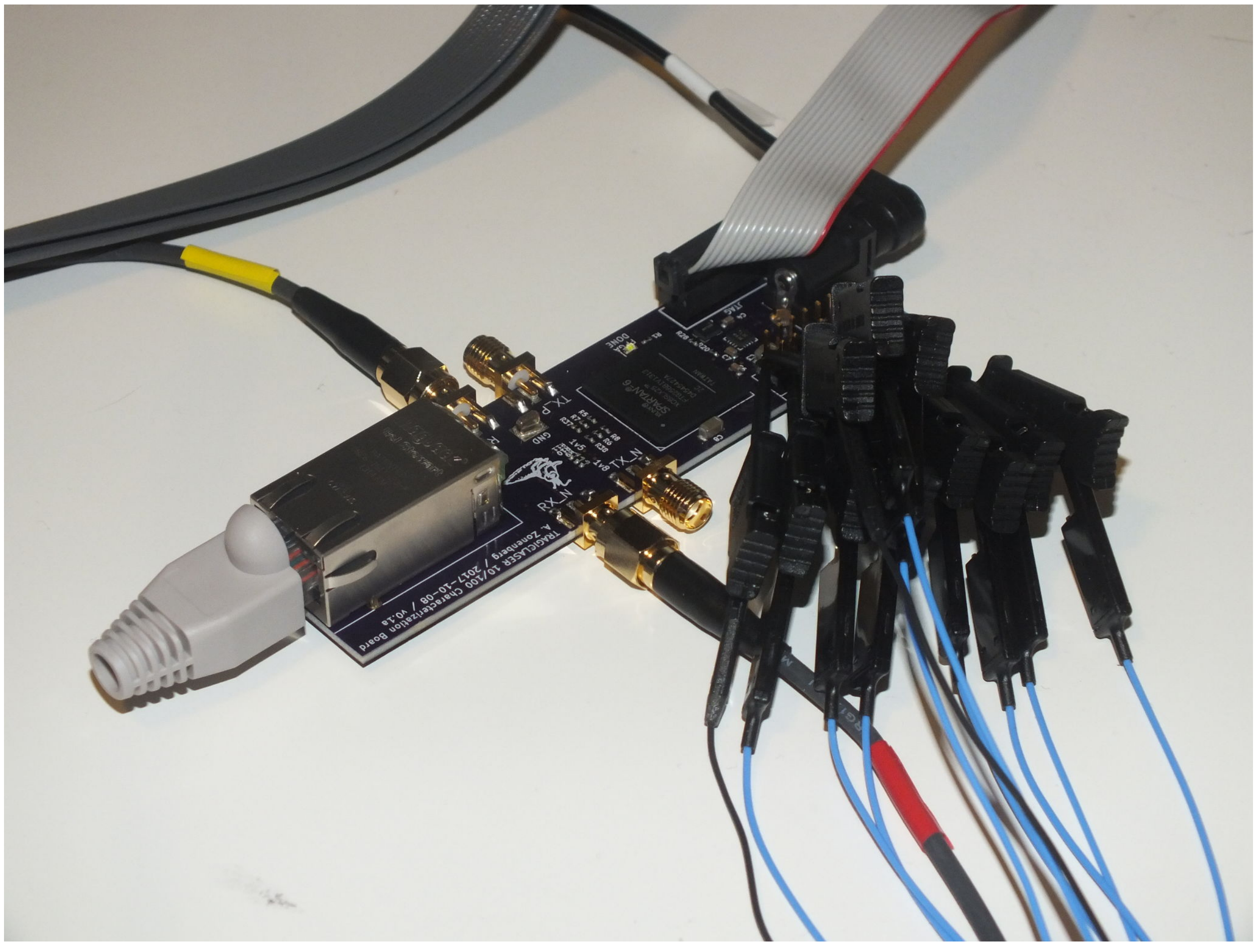


Figure 8. Spartan-6 Test Board

Bit-Banging 100Base-TX

A block diagram of the PHY, randomly code-named TRAGICLASER by @NSANameGen[14], is shown in Figure 9.

The transmit-side 4B/5B coding, serializing, and LFSR scrambler are straightforward digital logic at moderate to slow clock rates in the FPGA, so we won't discuss their implementation in detail.

Generating the signal requires creating three differential voltages: 0, +1, and −1. Since most FPGA I/O buffers cannot operate at 1.0V, or output negative voltages, a bit of clever circuitry is required.

We use a pair of 1K ohm resistors to bias the center tap of the output transformer to half of the 3.3V supply voltage (1.65V). The two ends of the transformer coil are connected to FPGA I/O pins. Since each I/O pin can pull high or low, we have a form of the classic H-bridge motor driver circuit. By setting one pin high and the other low, we can drive current through the line in either direction. By tri-stating both pins and letting the terminating resistor dissipate any charge built up in the cable capacitance, we can create a differential 0 state.

Since we want to drive +/− 1V rather than 3.3V, we need to add a resistor in series with the FPGA pins to reduce the drive current such that the receiver sees 1V across the 100 ohm terminator. Experimentally, good results were obtained with 100 ohm resistors in series with a Spartan-6 FPGA pin configured as LVCMOS33, fast slew, 24 mA drive. For other FPGAs with different drive characteristics, the resistor value may need to be slightly adjusted. This circuit is shown in Figure 10.

This produced a halfway decent MLT-3 waveform, and one that would probably be understood by a typical PHY, but the rise and fall times as the signal approached the 0V state were slightly slower than the 5 ns maximum permitted by the 802.3 standard (see Figure 11).

The solution to this is a clever technique from the analog world known as pre-emphasis. This is a fancy way of saying that you figure out what distortions your signal will experience in transit, then apply the reverse transformation before sending it. In our case, we have good values when the signal is stable but during the transitions to zero there's not enough drive current. To compensate, we simply need to give the signal a kick in the right direction.

Luckily for us, 10Base-T requires a pretty hefty dose of drive current. In order to ensure we could drive the line hard enough, two more FPGA pins were connected in parallel to each side of the TX-side transformer through 16-ohm resistors. By paralleling these two pins, the available current is significantly increased.

After a bit of tinkering, I discovered that by configuring one of the 10Base-T drive pins as LVCMOS33, slow slew, 2 mA drive, and turning it on for 2 nanoseconds during the transition from the +/−1 state to the 0 state, I could provide just enough of a shove that the signal reached the zero mark quickly while not overshooting significantly. Since the PHY itself runs at only 125 MHz, the Spartan-6 OSERDES2 block was used to produce a pulse lasting 1/4 of a PHY clock cycle. Figure 12 shows the resulting waveforms.[15]

At this point sending the auto-negotiation waveforms is trivial: The other FPGA pin connected to the 16 ohm resistor is turned on for 100 ns, then off. With a Spartan-6 I had good results with LVCMOS33, fast slew, 24 mA drive for these pins. If additional drive strength is required the pre-emphasis drivers can be enabled in parallel, but I didn't find this to be necessary in my testing.

These same pins could easily be used for 10Base-T output as well (to enable a dual-mode 10/100 PHY) but I didn't bother to implement this. People have already demonstrated successful bitbanging of 10Base-T, and it's not much of a POC if the concept is already proven.

That's it, we're done! We can now send 100Base-TX signals using six FPGA pins and six resistors!

Decoding 100Base-TX

Now that we can generate the signals, we have to decode the incoming data from the other side. How can we do this?

Most modern FPGAs are able to accept differential digital inputs, such as LVDS, using the I/O buffers built into the FPGA. These differential input buffers are essentially comparators, and can be abused into accepting analog signals within the operating range of the FPGA.

By connecting an input signal to the positive input of several LVDS input buffers, and driving the negative inputs with an external resistor ladder,

[14] `https://twitter.com/NSANameGen/status/910628839566594050`

[15] This wavefrom was captured with a 115 ohm drive resistor instead of 100, causing the output voltage to be closer to 0.9V than the intended 1.0V. After correcting the resistor value, the amplitude was close to perfect.

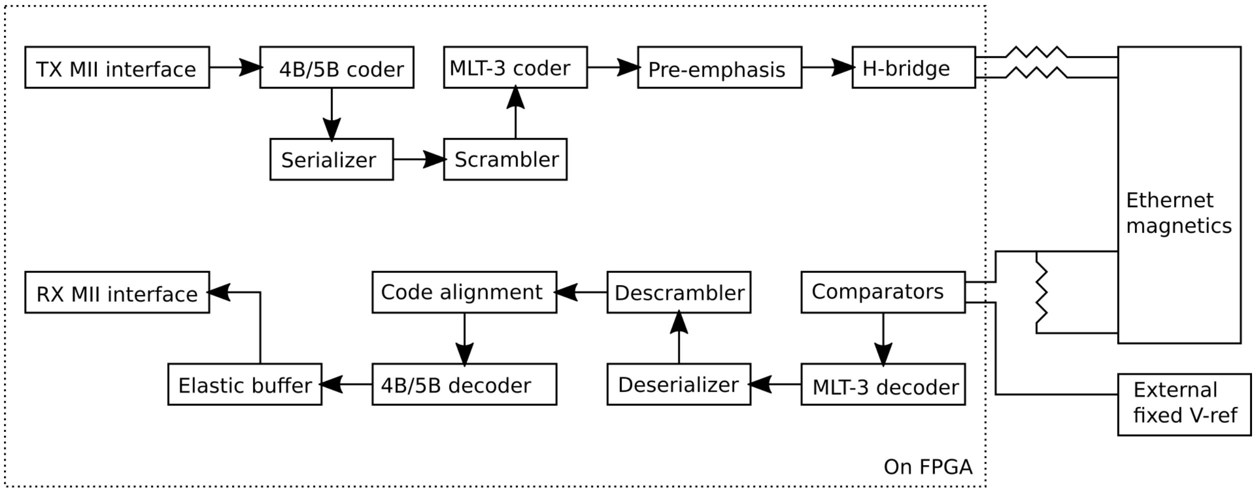


Figure 9. TRAGICLASER Block Diagram

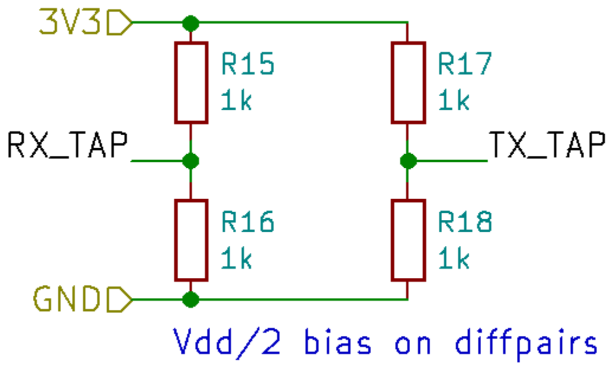


Figure 10. H-Bridge Schematic

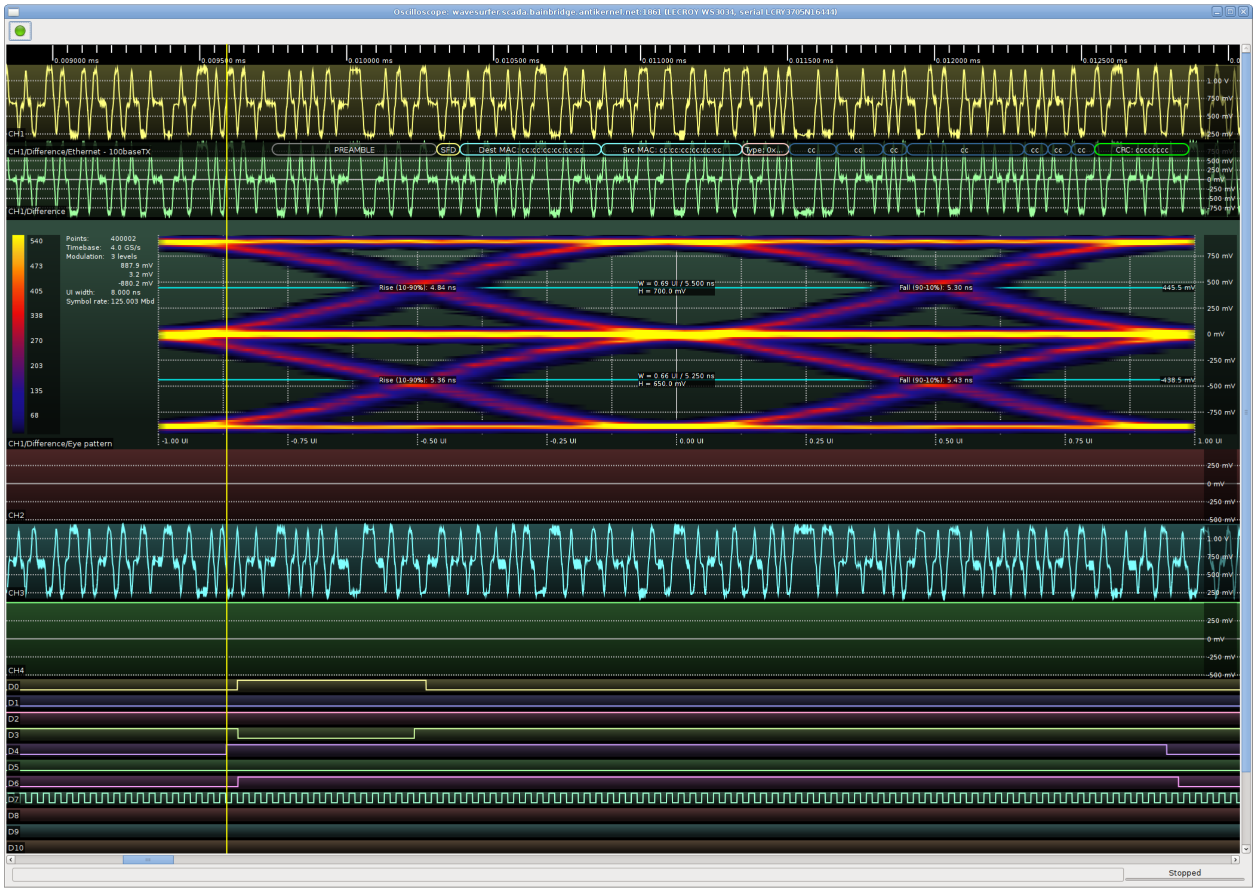


Figure 11. Halfway-Decent Waveform

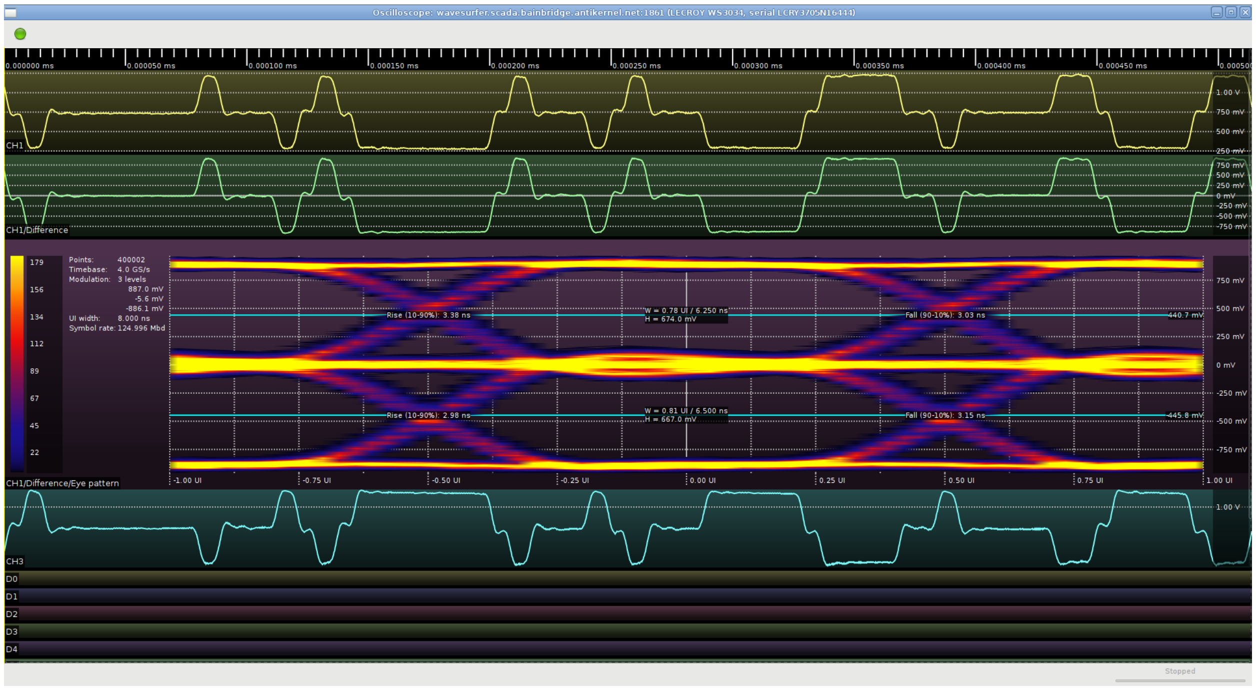


Figure 12. Waveform using Premphasis

we can create a low-resolution flash ADC! Since we only need to distinguish between three voltage levels (there's no need to distinguish the +1 and +2.5, or −1 and −2.5, states as they're never used at the same time) we can use two comparators to create an ADC with approximately 1.5 bit resolution.

There's just one problem: this is a single-ended ADC with an input range from ground to Vdd, and our incoming signal is differential with positive and negative range. Luckily, we can work around this by tying the center tap of the transformer to 1.65V via equal valued resistors to 3.3V and ground, thus biasing the signal into the 0–3.3V range. See Figure 13.

After we connect the required 100 ohm terminating resistor across the transformer coil, the voltages at the positive and negative sides of the coil should be equally above and below 1.65V. We can now connect our ADC to the positive side of the coil only, ignoring the negative leg entirely aside from the termination.

The ADC is sampled at 500 Msps using the Spartan-6 ISERDES. Since the nominal data rate is 125 Mbps, we have four ADC samples per unit interval (UI). We now need to recover the MLT-3 encoded data from the oversampled data stream.

The MLT-3 decoder runs at 125 MHz and processes 4 ADC samples per cycle. Every time the data changes the decoder outputs a 1 bit. Every time the data remains steady for one UI, plus an additional sample before and after, the decoder outputs a 0 bit. (The threshold of six ADC samples was determined experimentally to give the best bit error rate.) The decoder nominally outputs one data bit per clock however due to jitter and skew between the TX and RX clocks, it occasionally outputs zero or two bits.

The decoded data stream is then deserialized into 5-bit blocks to make downstream processing easier. Every 32 blocks, the last 11 bits from the MLT-3 decoder are complemented and loaded into the LFSR state. Since the 4B/5B idle code is `0x1F` (five consecutive 1 bits), the complement of the scrambled data between packets is equal to the scrambler PRNG output. An LFSR leaks 1 bit of internal state per output bit, so given N consecutive output bits from a N-bit LFSR, we can recover the entire state. The interval of 32 blocks (160 bits) was chosen to be relatively prime to the 11-bit LFSR state size.

After the LFSR is updated, the receiver begins XOR-ing the scrambler output with the incoming data stream and checks for nine consecutive idle characters (45 bits). If present, we correctly guessed

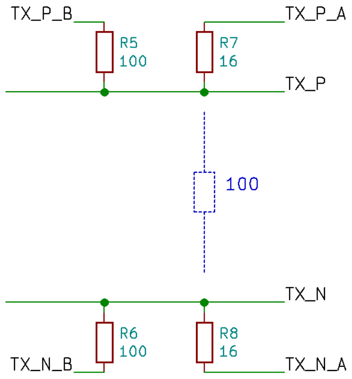


Figure 13. Biasing Schematic

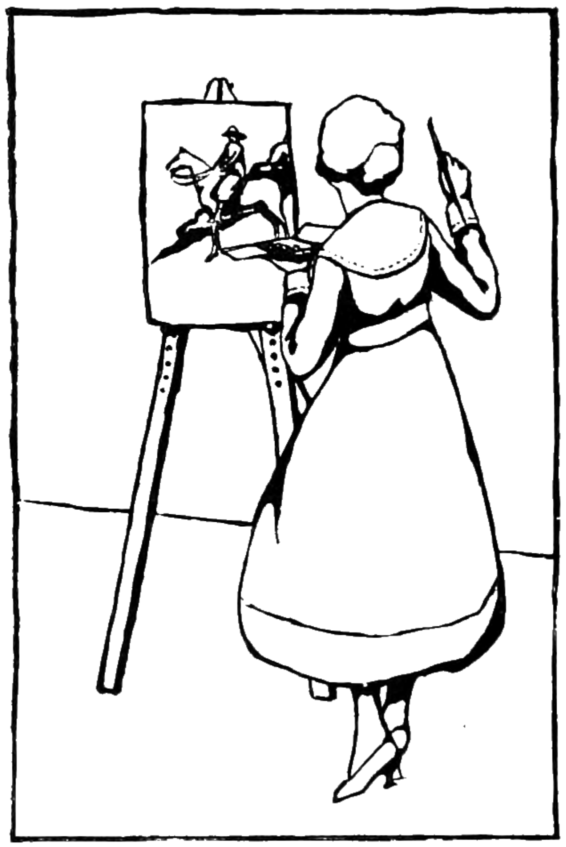
the location of an inter-packet gap and are locked to the scrambler, with probability $1 - (2^{-45})$ of a false lock due to the data stream coincidentally matching the LFSR output. If not present, we guessed wrong and re-try every 32 data blocks until a lock is achieved. Since 100Base-TX specifies a minimum 96-bit inter-frame gap, and we require $45 + 11 = 56$ idle bits to lock, we should eventually guess right and lock to the scrambler.

Once the scrambler is locked, we can XOR the scrambler output (5 bits at a time) with the incoming 5-bit data stream. This gives us cleartext 4B/5B data, however we may not be aligned to code-word boundaries. The idle pattern doesn't contain any bit transitions so there's no clues to alignment there. Once a data frame starts, however, we're going to see a `J+K` control character pair (`11000 10001`). The known position of the zero bits allows us to shift the data by a few bits as needed to sync to the 4B/5B code groups.

Decoding the 4B/5B is a simple table lookup that outputs 4-bit data words. When the `J+K` or `T+R` control codes are seen, a status flag is set to indicate the start or end of a packet.

If an invalid 5-bit code is seen, an error counter is incremented. Sixteen code errors in a 256-codeword window, or four consecutive packet times without any inter-frame gap, indicate that we may have lost sync with the incoming data or that the cable may have been unplugged. In this case, we reset the entire PHY circuit and attempt to re-negotiate a link.

The final 4-bit data stream may not be running at exactly the same speed as the 25 MHz MII clock, due to differences between TX and RX clock domains. In order to rate match, the 4-bit data coming off the 4B/5B decoder (excluding idle characters) is fed into an 32-nibble FIFO. When the FIFO reaches a fill of 16 nibbles (8 bytes), the PHY begins to stream the inbound packet out to the MII bus. We can thus correct for small clock rate mismatches, up to the point that the FIFO underflows or overflows during one packet time.



Test Results

In my testing, the TRAGICLASER PHY was able to link up with both my laptop and my Cisco switch with no issues through an approximately 2-meter patch cable. No testing with longer cables was performed because I didn't have anything longer on hand, however since the signal appears to pass the 802.3 eye mask I expect that the transmitter would be able to drive the full 100m cable specified in the standard with no difficulties. The receiver would likely start to fail with longer cables since I'm not doing equalization or adaptive thresholding, however I can't begin to guess how much you could get actually away with. If anybody decides to try, I'd love to hear your results!

My test bitstream doesn't include a full 10/100 MAC, so verification of incoming data from the LAN was conducted with a logic analyzer on the RX-side MII bus. (Figure 14.)

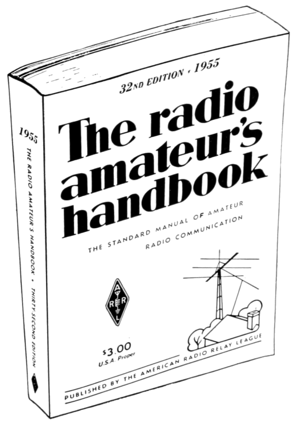
The transmit-side test sends a single hard-coded UDP broadcast packet in a loop. I was able to pick it up with Wireshark (Figure 15) and decode it. My switch did not report any RX-side CRC errors during a 5-minute test period sending at full line rate.

In my test with default optimization settings, the PHY had a total area of 174 slices, 767 LUT6s, and 8 LUTRAMs as well as four OSERDES2 and two ISERDES2 blocks. This is approximately 1/4 of the smallest Spartan-6 FPGA (XC6SLX4) so it should be able to comfortably fit into almost any FPGA design. Additionally, twelve external resistors and an RJ-45 jack with integrated isolation transformer were required.

Further component reductions could be achieved if a 1.5 or 1.8V supply rail were available on the board, which could be used (along with two external resistors) to inject the DC bias into the coupling transformer taps at a savings of two resistors. An enterprising engineer may be tempted to use the internal 100 ohm differential terminating resistors on the FPGA to eliminate yet another passive at the cost of two more FPGA pins, however I chose not to go this route because I was concerned that dissipating 10 mW in the input buffer might overheat the FPGA.

Overall, I was quite surprised at how well the PHY worked. Although I certainly hoped to get it to the point that it would be able to link up with another PHY and send packets, I did not expect the TX waveform to be as clean as it was. Although the RX likely does not meet the full 802.3 sensitivity requirements, it is certainly good enough for short-range applications. The component cost and PCB space used by the external passives compare favorably with an external 10/100 PHY if standards compliance or long range are not required.

Source code is available in my Antikernel project.[16]


32nd Edition · 1955
The radio amateur's handbook
The standard manual of amateur radio communication
$3.00 USA Proper
Published by the American Radio Relay League

1955 EDITION

Big...

Revised...

Complete...

Internationally recognized, universally consulted. A complete and comprehensive treatment of radio and electronics from simple to advanced radio theory and technique. A valuable asset, a constant reference source for the bookshelf of every amateur, engineer, experimenter and technician. Theory, construction, application—all are covered in this widely accepted Handbook—plus a complete catalog section featuring leading manufacturers and suppliers of electronic equipment, components and tubes, providing an excellent buying guide for purchasing agents as well as individual users of parts and equipment.

$3 USA proper **$3.50 US Possessions and Canada** **$4 Elsewhere**

Buckram Bound Edition $5 Everywhere

The American Radio Relay League, Inc.
West Hartford 7, Conn.

[16] `git clone https://github.com/azonenberg/antikernel || unzip pocorgtfo17.zip antikernel.zip`

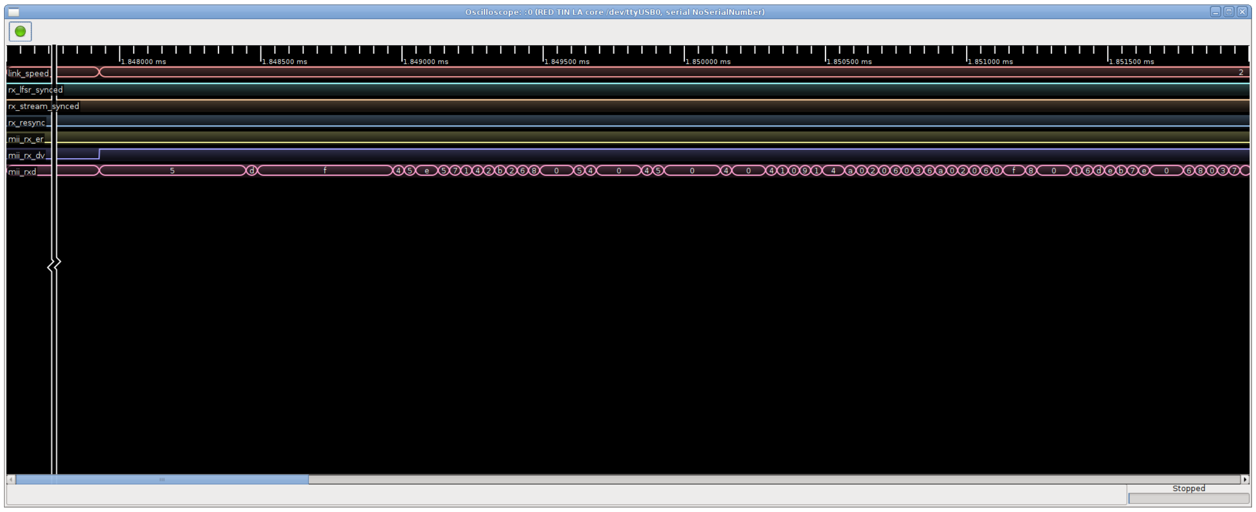
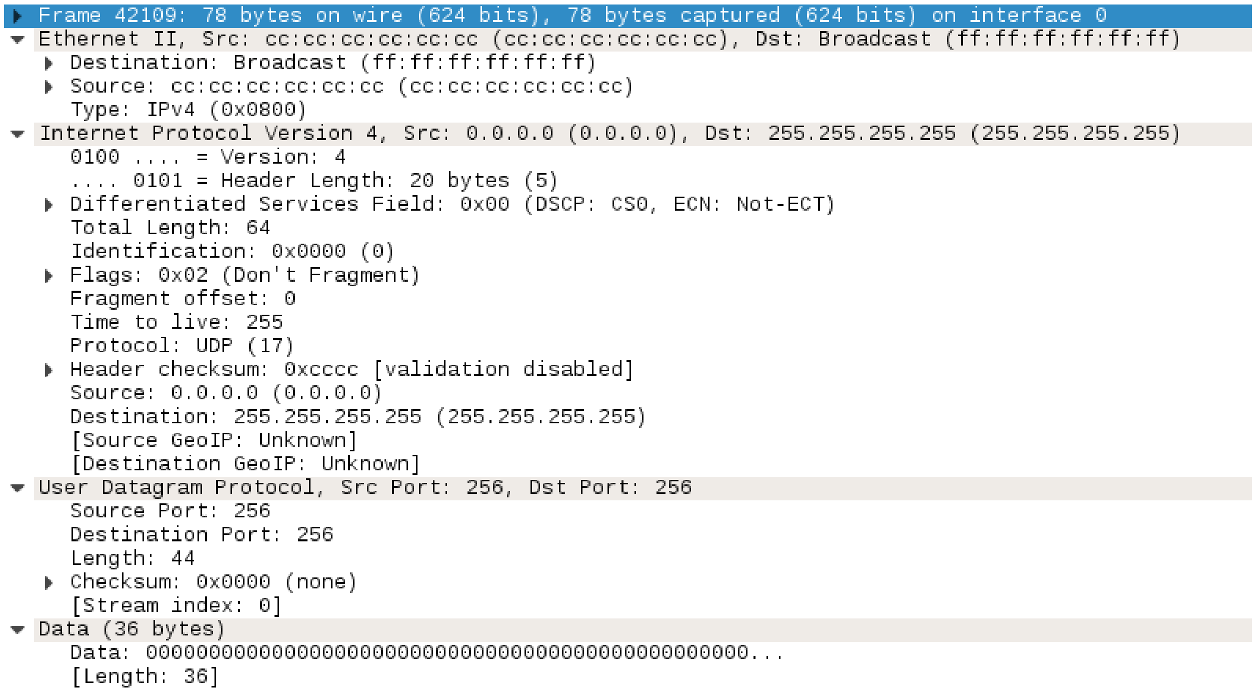


Figure 14. Receiver Verification



```
▶ Frame 42109: 78 bytes on wire (624 bits), 78 bytes captured (624 bits) on interface 0
▼ Ethernet II, Src: cc:cc:cc:cc:cc:cc (cc:cc:cc:cc:cc:cc), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
  ▶ Destination: Broadcast (ff:ff:ff:ff:ff:ff)
  ▶ Source: cc:cc:cc:cc:cc:cc (cc:cc:cc:cc:cc:cc)
    Type: IPv4 (0x0800)
▼ Internet Protocol Version 4, Src: 0.0.0.0 (0.0.0.0), Dst: 255.255.255.255 (255.255.255.255)
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▶ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 64
    Identification: 0x0000 (0)
  ▶ Flags: 0x02 (Don't Fragment)
    Fragment offset: 0
    Time to live: 255
    Protocol: UDP (17)
  ▶ Header checksum: 0xcccc [validation disabled]
    Source: 0.0.0.0 (0.0.0.0)
    Destination: 255.255.255.255 (255.255.255.255)
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
▼ User Datagram Protocol, Src Port: 256, Dst Port: 256
    Source Port: 256
    Destination Port: 256
    Length: 44
  ▶ Checksum: 0x0000 (none)
    [Stream index: 0]
▼ Data (36 bytes)
    Data: 000000000000000000000000000000000000000000000000...
    [Length: 36]
```

Figure 15. Wireshark

17:06 The DIP Flip Whixr Trick: An Integrated Circuit That Functions in Either Orientation

by Joe “Kingpin” Grand

Hardware trickery comes in many shapes and sizes: implanting add-on hardware into a finished product, exfiltrating data through optical, thermal, or electromagnetic means, injecting malicious code into firmware, BIOS, or microcode, or embedding Trojans into physical silicon. Hackers, governments, and academics have been playing in this wide open field for quite some time and there’s no sign of things slowing down.

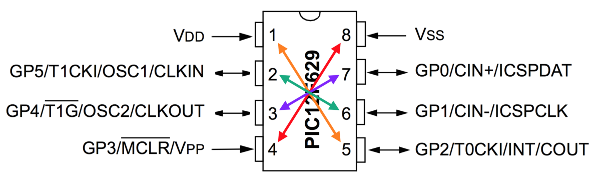
This PoC, inspired by my friend Whixr of `#tymkrs`, demonstrates the feasibility of an IC behaving differently depending on which way it’s connected into the system. Common convention states that ICs must be inserted in their specified orientation, assisted by the notch or key on the device identifying pin 1, in order to function properly.

So, let’s defy this convention!

– – – –   – – –   – – – –

Most standard chips, like digital logic devices and microcontrollers, place the power and ground connections at corners diagonal from each other. If one were to physically rotate the IC by 180 degrees, power from the board would connect to the ground pin of the chip or vice versa. This would typically result in damage to the chip, releasing the magic smoke that it needs to function. The key to this PoC was finding an IC with a more favorable pin configuration.

While searching through microcontroller data sheets, I came across the Microchip PIC12F629. This particular 8-pin device has power and GPIO (General Purpose I/O) pins in locations that would allow the chip to be rotated with minimal risk. Of course, this PoC could be applied to any chip with a suitable pin configuration.



In the pinout drawing, which shows the chip from above in its normal orientation, arrows denote the alternate functionality of that particular pin when the chip is rotated around. Since power (VDD) is normally connected to pin 1 and ground (VSS) is normally connected to pin 8, if the chip is rotated, GP2 (pin 5) and GP3 (pin 4) would connect to power and ground instead. By setting both GP2 and GP3 to inputs in firmware and connecting them to power and ground, respectively, on the board, the PIC will be properly powered regardless of orientation.

– – – –   – – –   – – – –

I thought it would be fun to change the data that the PIC sends to a host PC depending on its orientation.

On power-up of the PIC, GP1 is used to detect the orientation of the device and set the mode accordingly. If GP1 is high (caused by the pull-up resistor to VCC), the PIC will execute the normal code. If GP1 is low (caused by the pull-down resistor to VSS), the PIC will know that it has been rotated and will execute the alternate code. This orientation detection could also be done using GP5, but with inverted polarity.

The PIC’s UART (asynchronous serial) output is bit-banged in firmware, so I’m able to reconfigure the GPIO pins used for TX and RX (GP0 and GP4) on-the-fly. The TX and RX pins connect directly to an Adafruit FTDI Friend, which is a standard FTDI FT232R-based USB-to-serial adapter. The FTDI Friend also provides 5V (VDD) to the PoC.

In normal operation, the device will look for a key press on GP4 from the FTDI Friend’s TX pin and then repeatedly transmit the character ’A’ at 9600 baud via GP0 to the FTDI Friend’s RX pin. When the device is rotated 180 degrees, the device will look for a key press on GP0 and repeatedly transmit the character ’B’ on GP4. As a key press detector, instead of reading a full character from the host, the device just looks for a high-to-low transition on the PIC’s currently configured RX pin. Since that pin idles high, the start bit of any data sent from the FTDI Friend will be logic low.