From Oscilloscope to Wireshark: A UDP Story

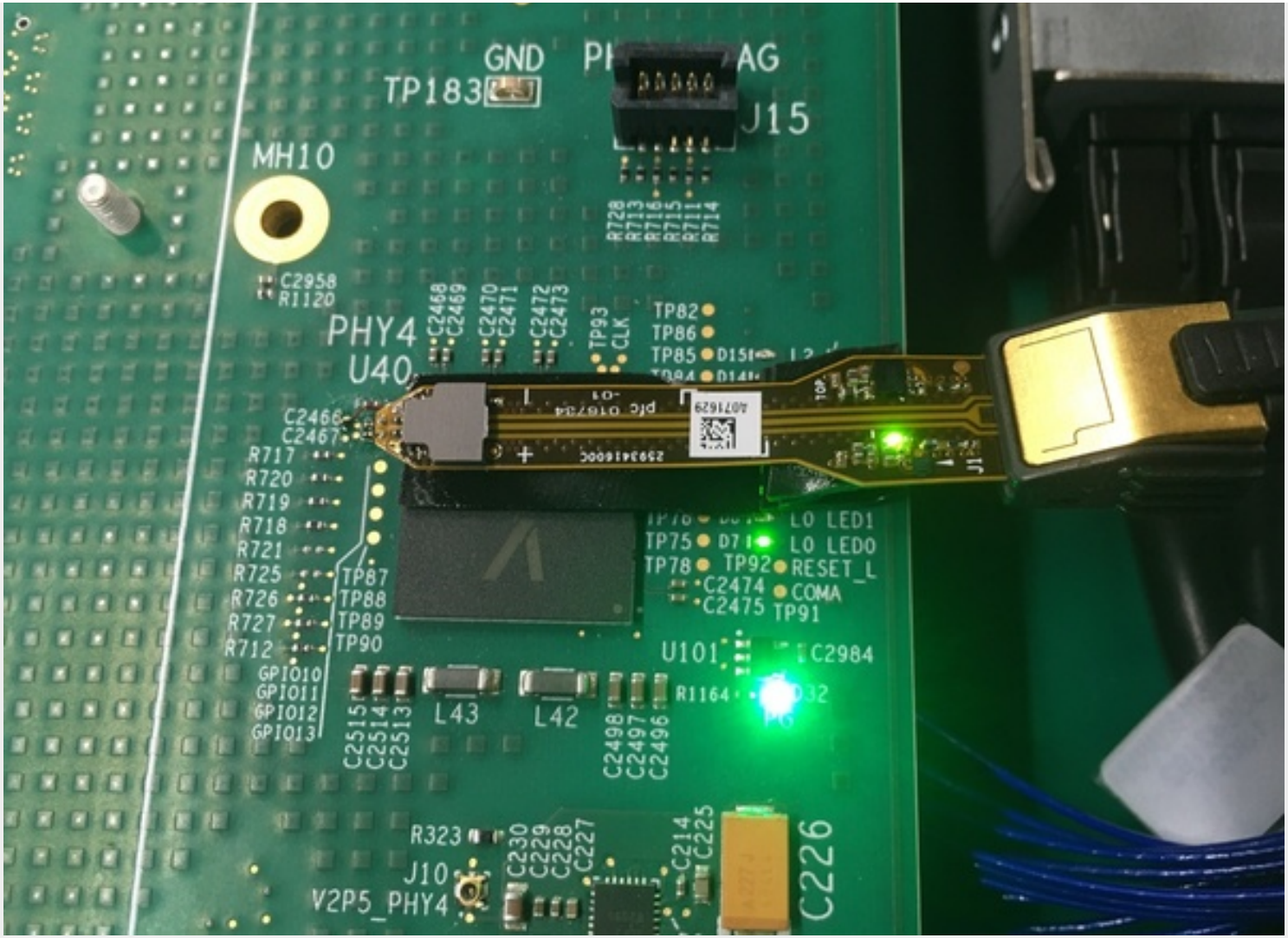
UDP is a transport-level protocol for sending messages through an IP network.

It sits at level 4 in the OSI model:

7	Application
6	Presentation
5	Session
4	**Transport**
3	Network
2	Data link
1	Physical

Like many of you, I've got hardware on my desk that's sending UDP packets, and the time has come to take a closer look at them.

Most "low-level" networking tutorials will bottom out somewhere at "use `tcpdump` to see raw packets". We'll be starting a bit lower in the stack; specifically, *here*:

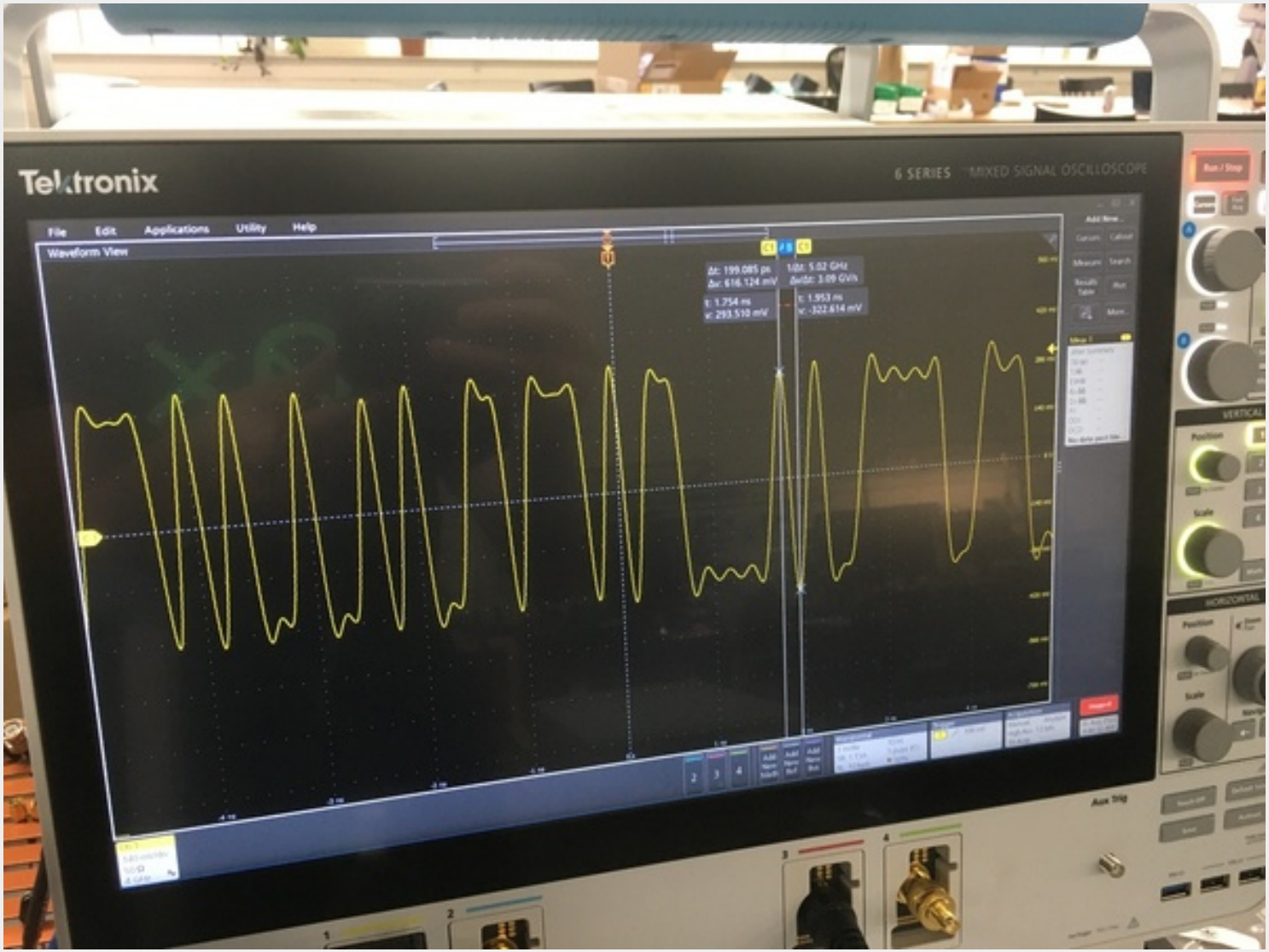


(click on images for the original, full-size files)

This is a high-speed active differential probe soldered to an Oxide Computer Company rack switch. We're going **all the way down** to the metal.

(Huge thanks to Eric for the careful soldering that made this possible!)

Looking at the signals on an oscilloscope, we see data zooming down the wires:



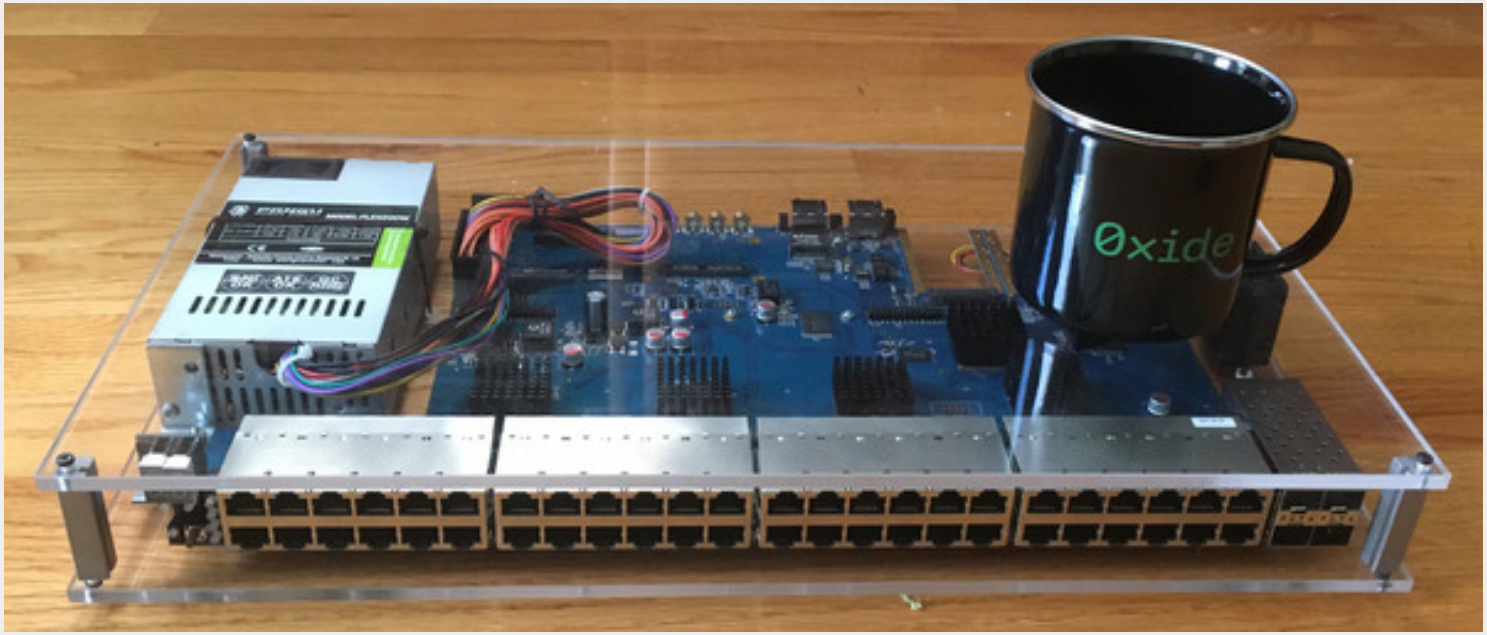
The rest of this post will take us from these raw voltage waveforms all the way to decoded UDP packets. Hold on tight, we're going from L1 all the way to L4.

First, a bit of context

I work at Oxide Computer Company, writing embedded software for the rack-scale computer.

Over the past few months, I've been focused on the **management network**, which is a low-speed network between each server's Service Processor. The service processor is roughly equivalent to a baseboard management controller; it allows for lights-out management of the rack.

The heart of the management network is the VSC7448, a 52-port, 80G ethernet switch chip. This is the *slower* switch, and it's still a beast; here's the dev kit:



(mug for scale)

This decoding work was part of tracking down a nasty bug which caused a subset of links to only work *some* of the time. The root cause turned out to be a misconfiguration of the switch IC, but the hunt was an interesting dive into the physical layer of modern networking.

Loading the waveforms

That's enough context, back to work!

The oscilloscope doesn't have a built-in QSGMII analyzer (and we'll want to do fairly sophisticated processing of the data), so I wanted to export waveform data to my computer.

How much data should I capture? Analog waveforms can easily add up to multiple gigabytes, so I'd like to capture a small amount while still catching a packet or two.

I knew that a device on the network was emitting about 30K UDP packets per second, or one packet every 33 µs. I configured the oscilloscope to collect 100M samples at 1 TSPS (tera-sample per second, 10^{12}), which multiplies out to 100 µs of data; this means we should catch 1-3 UDP packets.

After hunting down a USB key, I ended up with a 191M `.wfm` file to process.

Fortunately, the `.wfm` file format is documented by Tektronix.

In about 400 lines of code, I implemented a simple parser using `nom` (which has great support for this kind of binary format). The parser is overkill: it decodes the entire file based on the specification. In practice, we only care about two

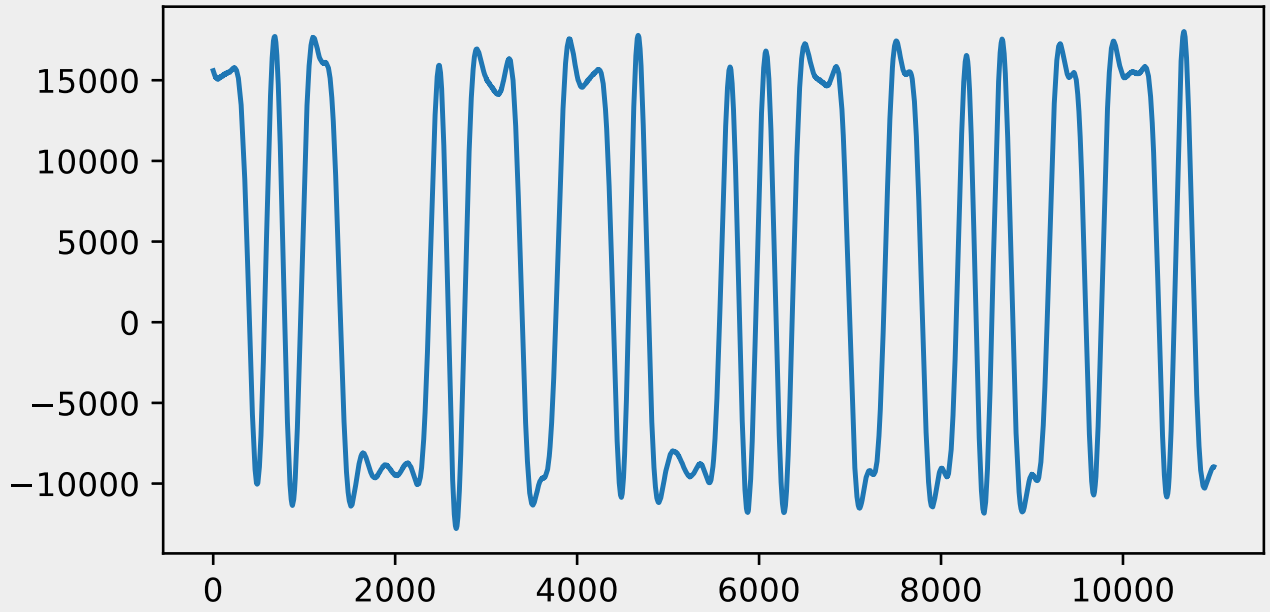
things:

- The sample waveform (which is an array of `i16`)
- The sample rate (in seconds per sample)

It's also possible to write a decoder in three lines of Python, once you know where the data stream starts in the file:

```
import numpy as np
data = open('udp-spam.wfm', 'rb').read()
pts = np.frombuffer(data[904:-1], dtype=np.int16)
```

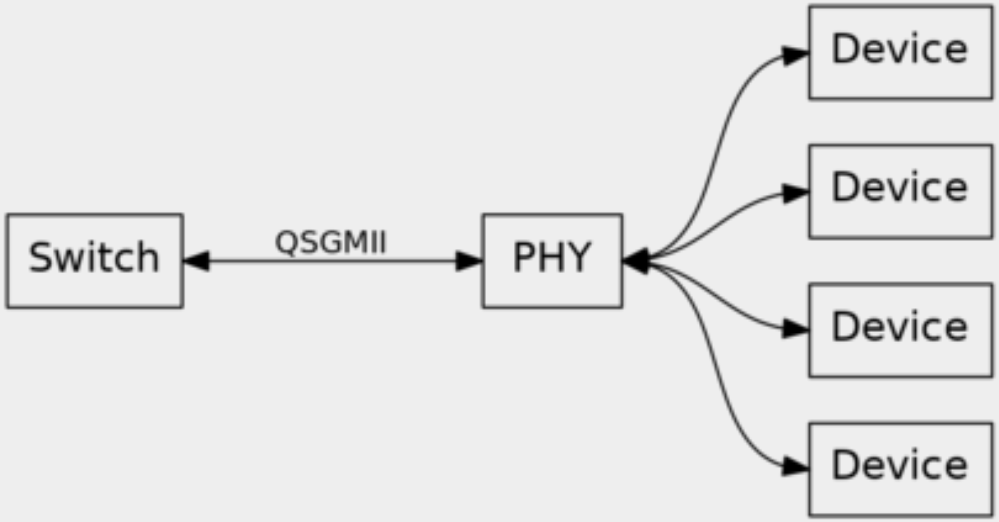
Plotting a chunk of this data, it looks like what I was seeing on the oscilloscope:



Now, we have to figure out what it means.

QSGMII for dummies

Thinking back to the probes soldered to our PCB, we are probing the link between the main VSC7448 switch and a VSC8504 PHY. The PHY is acting as a port expander: it's taking four ports from the switch (combined into a single Tx/Rx channel), and splitting them into four separate channels.



(This is only necessary because the VSC7448 does not have enough pins to drive all 52 of its ports directly)

The link between the switch and the PHY is using **QSGMII**, which stands for quad serial gigabit media-independent interface. QSGMII is a protocol for communication between a Media Access Control (MAC) block and an ethernet PHY.

Specifically, QSGMII is a way to pack four SGMII channels into a single Tx/Rx

pair (hence the "quad"), running 4× as fast (5 GBPS instead of 1.25 GBPS)

The SGMII and QSGMII standards are both freely available, and are both quite readable (especially compared to IEEE 802.3-2015).

Let's start with the encoding.

8b/10b decoding

Both SGMII and QSGMII use 8b/10b encoding, which is a way to pack a stream of (8-bit) bytes into 10-bit "code-groups" (sometimes called "symbols") with various desirable properties:

- On average, there are the same number of 0s and 1s in the stream
- There are enough bit transitions to recover the clock

The code-groups are all smushed together, so when you look at the raw data, it's not obvious where code-groups begin or end.

To recover the code-group framing, we need to look for **comma characters**, which are characters of the form `1100000` or `0011111`. These are (almost) the only characters which have five 0s or 1s in a row.

First, let's convert the `i16` values to binary:

```
let pts: Vec<bool> = t.pts.iter().map(|p| *p < 0).collect();
```

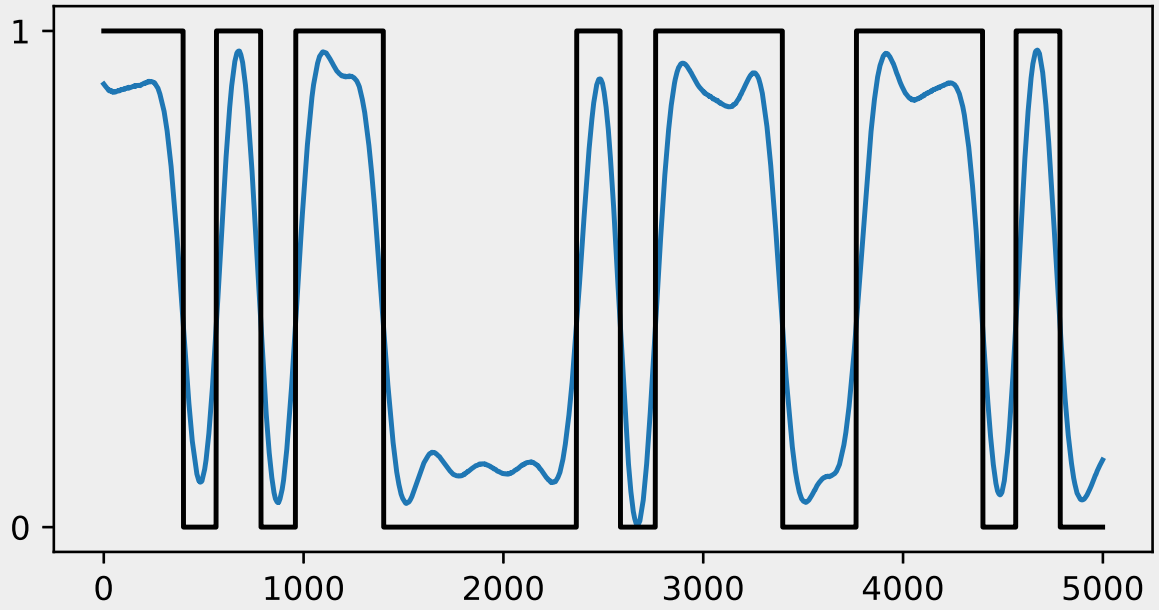
Next, we'll start by picking out places where the signal changes from 0 to 1 or vice versa:

```
let crossings = pts.iter()
```

```
    .zip(&pts[1..])
    .enumerate()
    .filter(|(_, (a, b))| a != b)
    .map(|(i, _)| i)
    .collect()
```

We know our sample rate (1 TPSP) and the nominal QSGMII bit rate (5 GHz); this means that a single-bit pulse (e.g. `010`) should be a 200-sample pulse. In turn, we expect a comma character to be roughly 1000 samples long (200 × 5).

Here's a chunk of data (normalized to 0-1); can you spot the comma character?

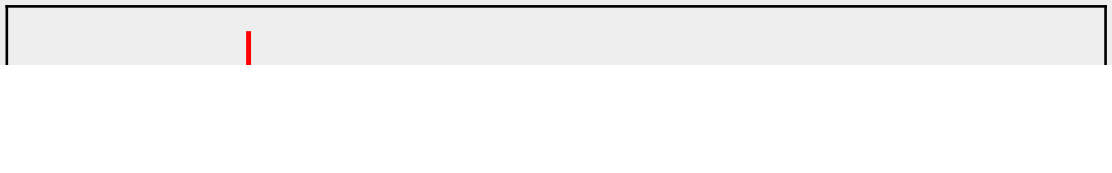


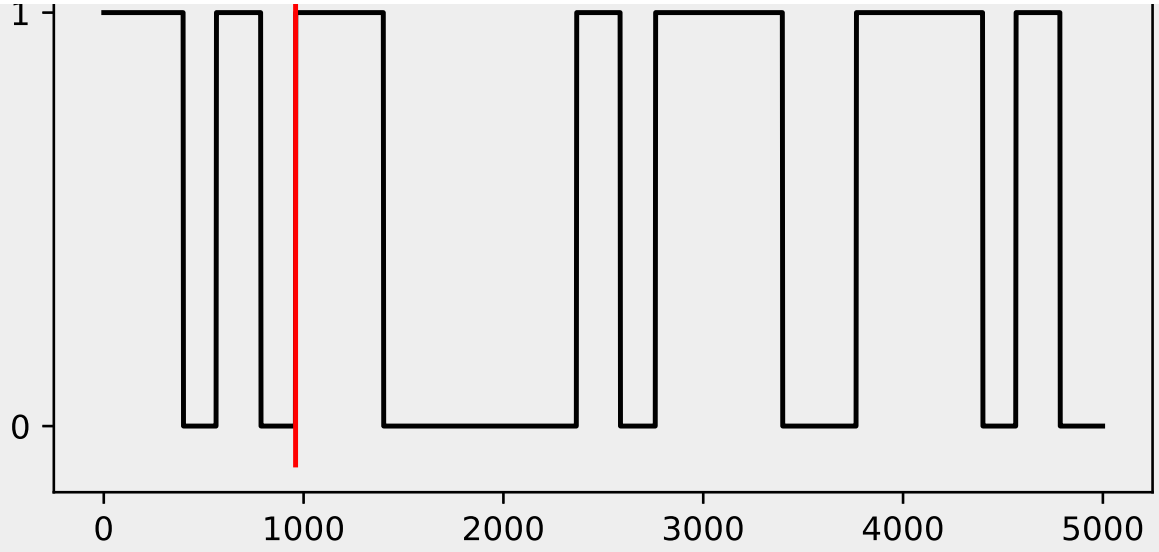
We can detect comma characters by picking out places where the signal doesn't change for > 900 samples, i.e. 4.5 bit lengths:

```
// We detect a potential comma if there are >= 4.5 bit lengths between
// transitions, since a comma has 5 identical bits in a row.
let comma_length = samples_per_clock * 9 / 2;
let commas = crossings
    .iter()
    .zip(&crossings[1..])
    .filter(|&(a, b)| b - a >= comma_length)
    .map(|(a, _b)| *a - samples_per_clock * 2)
    .filter(|a| {
        // A comma is either 0011111 or 1100000.  We detected the crossing
        // at the beginning of the 11111 or 00000 run, so we backed up by
        // two clock periods (in the `map` above).  Now, we advance by by
        // 1/2 clock period to land in the middle of the bit.
        let i = a + samples_per_clock / 2;
        let mut value = 0;
        for j in 0..7 {
            value = (value << 1) | (pts[i + j * samples_per_clock] as u16);
        }
        value == 0b1100000 || value == 0b0011111
    })
    .collect();
```

(Notice that we check for `0011111`/`1100000`, not just `11111`/`00000`; certain other data patterns can produce a run of 5 identical bits, but lack the leading `00`/`11`)

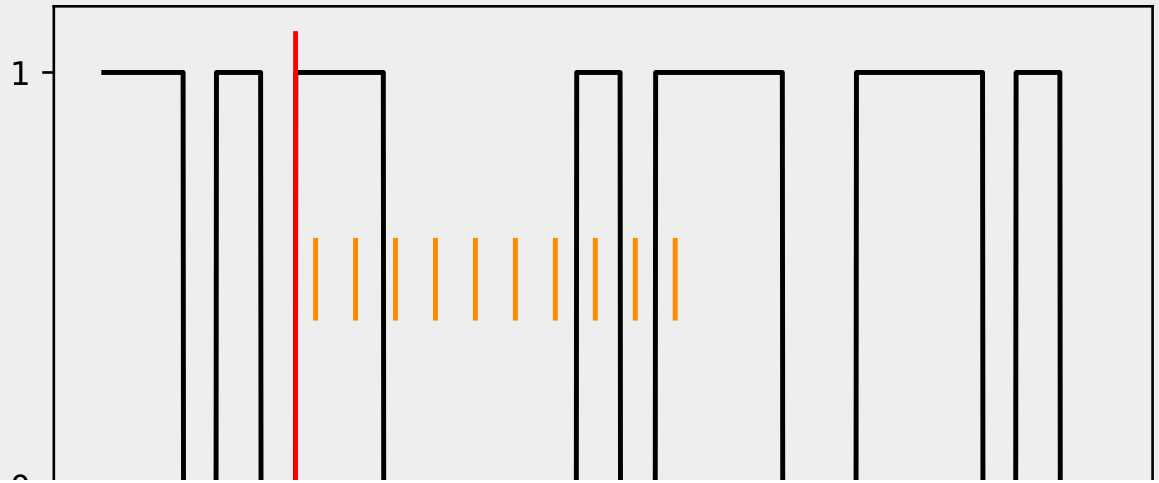
The comma character is right here in our data:

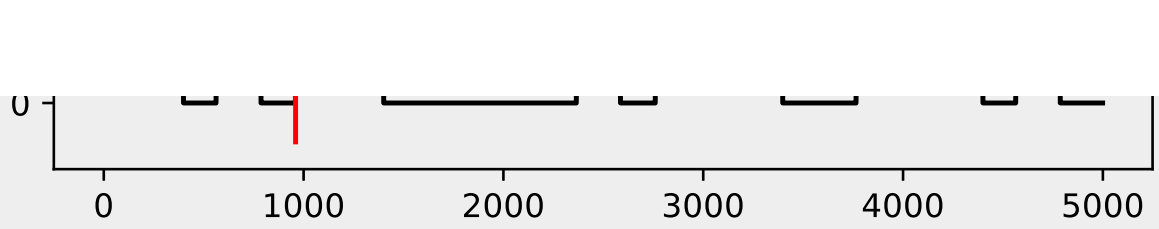




(I'm going to stop showing the analog trace now, to keep the graphs readable)

Once synchronized with the comma character, we know that bits are spaced at the bit sample rate, so we can read the 10-bit code-group. In this case, it's `1100000101`:





However, it's not *quite* that easy!

The oscilloscope and switch may not have *exactly* the same clock rate. If we go a long time between comma characters, we may end up sampling at the wrong position in the waveform!

It turns out that we need to synchronize in two places:

- Comma characters tell us when a new code-group starts
- Bit transitions help us keep the clock in sync

Here's what that looks like:

```
// The comma iterator points to the bit transition at the beginning of the
// comma codegroup, i.e. 0011111 or 1100000
let mut iter_comma = commas.iter().cloned().peekable();

// Index at which to sample the data.  We start at a half-cycle offset from
// the bit transition at the beginning of the comma character.
let mut i = iter_comma.next().unwrap() + samples_per_clock / 2;

// The zero-crossing iterator points to bit transitions, so we can stay in
// sync even if it's been a long time since the last comma character.
let mut iter_cross =
    crossings.iter().cloned().filter(move |c| *c > i).peekable();
```

```
let mut bit = 0;
let mut value: u16 = 0;
let mut cgs = vec![];
while i < pts.len() {
    // Resynchronize based on bit transitions
    if let Some(&j) = iter_cross.peek() {
        if i > j {
            i = j + samples_per_clock / 2;
            iter_cross.next();
        }
    }
    // Resynchronize and reset on comma characters
    if let Some(&j) = iter_comma.peek() {
        if i > j {
            i = j + samples_per_clock / 2;
            if bit != 0 {
                warn!("Off-sync comma character at {j} ({})", cgs.len());
            }
            bit = 0;
            value = 0;
            iter_comma.next();
        }
    }

    value = (value << 1) | (pts[i] as u16);
    bit += 1;
    i += samples_per_clock;
```

```
    if bit == 10 {
        cgs.push(value);
        bit = 0;
        value = 0;
    }
}
```

This gives us a `Vec<u16>`, with each item storing a 10-bit code-group. To convert to actual code-groups, we just need a **huge** look-up table:

```
#[derive(Copy, Clone, Debug, Eq, PartialEq)]
pub enum Codegroup {
    D0_0,
    D1_0,
    D2_0,
    D3_0,
    D4_0,
    // etc...
    K23_7,
    K27_7,
    K29_7,
    K30_7,
}

impl TryFrom<u16> for Codegroup {
    type Error = ();
    fn try_from(value: u16) -> Result<Self, Self::Error> {
        use Codegroup::*;
```

```
        let out = match value {
            0b100111_0100 | 0b011000_1011 => D0_0,
            0b011101_0100 | 0b100010_1011 => D1_0,
            0b101101_0100 | 0b010010_1011 => D2_0,
            // etc...
            0b110110_1000 | 0b001001_0111 => K27_7,
            0b101110_1000 | 0b010001_0111 => K29_7,
            0b011110_1000 | 0b100001_0111 => K30_7,
            _ => return Err(()),
        };
        Ok(out)
    }
}
```

At this point, we've gone from raw voltage readings to a stream of codegroups. Here's a few dozen samples from the stream:

```
K28.5
D24.2
K23.7
D24.2
D16.2
D24.2
K28.1
D24.2
K28.5
D24.2
D16.2
```

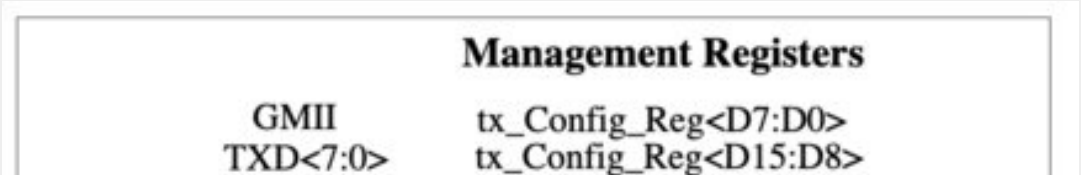
```
D24.2
D16.2
K29.7
K28.1
K29.7
K28.5
K23.7
D16.2
K23.7
D16.2
K28.5
K28.1
K28.5
```

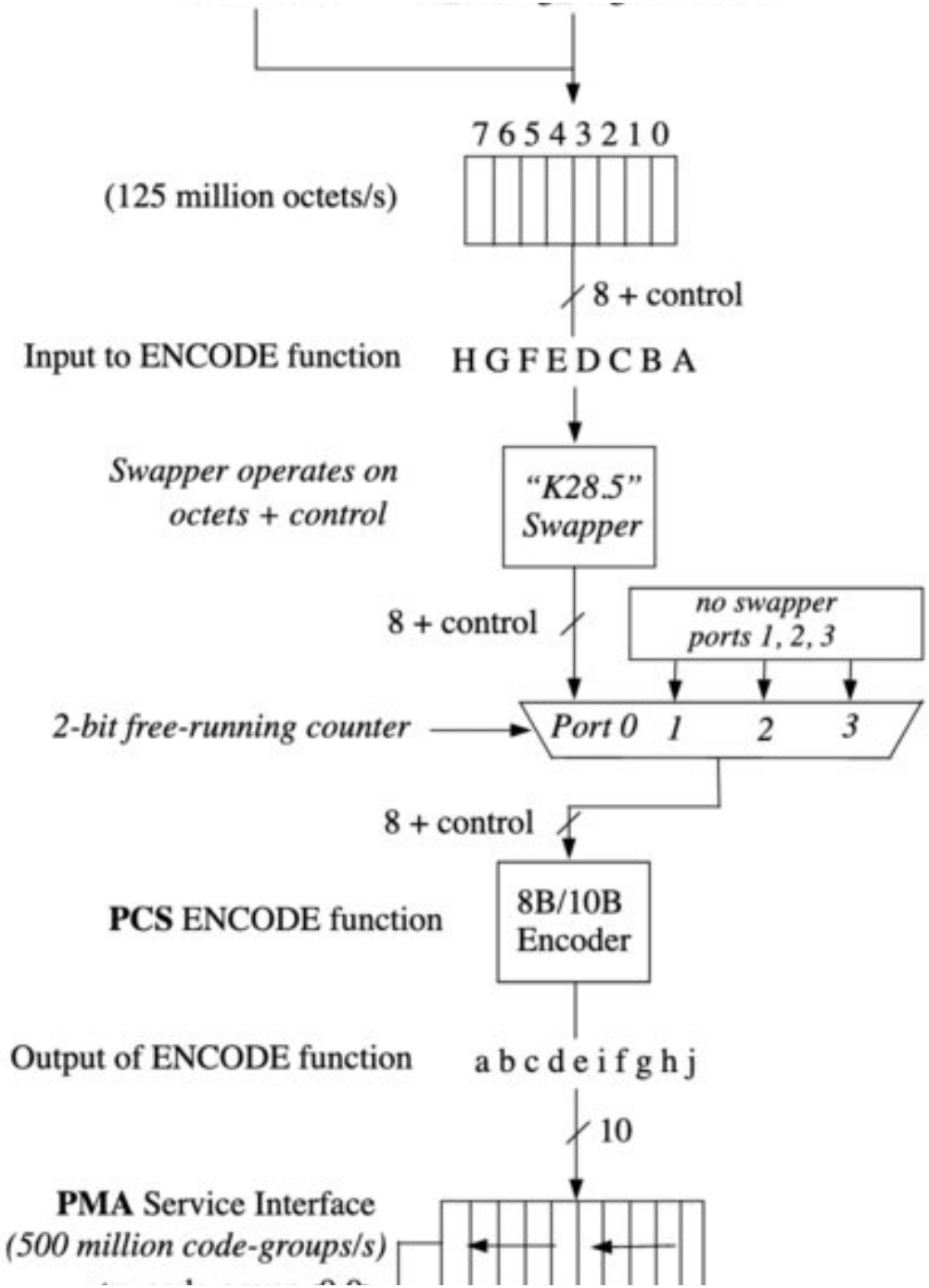
If you've done any low-level networking, some of this may be starting to look familiar!

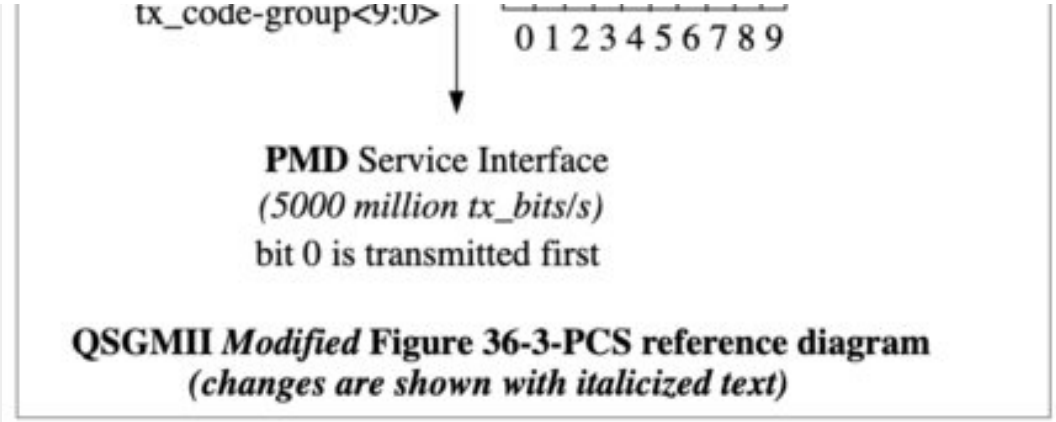
Separating the streams

Remember that QSGMII combines four separate SGMII streams into a single physical link. How does this work, and how do we distinguish them?

Let's go to the QSGMII standard:




7 6 5 4 3 2 1 0
(125 million octets/s)
8 + control
Input to ENCODE function
H G F E D C B A
Swapper operates on
octets + control
"K28.5"
Swapper
8 + control
no swapper
ports 1, 2, 3
2-bit free-running counter
Port 0 1 2 3
8 + control
PCS ENCODE function
8B/10B
Encoder
Output of ENCODE function
a b c d e i f g h j
10
PMA Service Interface
(500 million code-groups/s)



QSGMII *Modified* Figure 36-3-PCS reference diagram
(changes are shown with italicized text)

Code-groups from the individual ports are interleaved in order by the demultiplexer in the center of the diagram, so we'd expect to see

```
port0, port1, port2, port3, port0, port1, port2, port3, ...
```

To uniquely identify Port 0, QSGMII uses the "K28.5 swapper", shown in the diagram right before the demultiplexer: `K28.5` is replaced with `K28.1` for Port 0.

This works because `K28.5` appears frequently in SGMII traffic: it's part of the idle pattern and also used for start / end of packet delimiters.

By detecting this code-group, we can split the data into four streams:

```
// Split the codegroups into separate streams for each channel
let mut channel: Option<usize> = None;
let mut streams: [Vec<Codegroup>; 4] = Default::default();
for (i, mut cg) in cgs.iter().cloned().enumerate() {
    // Detect the K28.1 sync character
    if cg == Codegroup::K28_1 {
```

```
        if channel.map(|c| c != 0).unwrap_or(false) {
            warn!("Out of sync K28.1 at codegroup {}", i);
        }
        channel = Some(0);
        cg = Codegroup::K28_5; // Unswap K28.1 -> K28.5
    }
    // Only record codegroups after we've found Port 0
    if let Some(c) = channel.as_mut() {
        streams[*c].push(cg);
        *c = (*c + 1) % 4;
    }
}
```

Starting at the first `K28.1`, we now have four per-port streams:

```
 Port 0 | Port 1 | Port 2 | Port 3
--------|--------|--------|-------
 K28.1  | D24.2  | K28.5  | D24.2
 D16.2  | D24.2  | D16.2  | K29.7
 K28.1  | K29.7  | K28.5  | K23.7
 D16.2  | K23.7  | D16.2  | K28.5
 K28.1  | K28.5  | K28.5  | D16.2
 D16.2  | D16.2  | D16.2  | K28.5
 K28.1  | K28.5  | K28.5  | D16.2
 D16.2  | D16.2  | D16.2  | K28.5
 K28.1  | K28.5  | K28.5  | D16.2
 D16.2  | D16.2  | D16.2  | K28.5
```

From code-groups to packets

We're getting closer! The next step is turning this stream of code-groups into packets. To understand how this works, we'll consult IEEE 802.3-2015, the IEEE Standard for Ethernet:

Table 36–3—Defined ordered sets

Code	Ordered Set	Number of Code-Groups	Encoding
/C/	**Configuration**		Alternating /C1/ and /C2/
/C1/	Configuration 1	4	/K28.5/D21.5/Config_Reg[a]
/C2/	Configuration 2	4	/K28.5/D2.2/Config_Reg[a]
/I/	**IDLE**		Correcting /I1/, Preserving /I2/
/I1/	IDLE 1	2	/K28.5/D5.6/
/I2/	IDLE 2	2	/K28.5/D16.2/
	Encapsulation		
/R/	Carrier_Extend	1	/K23.7/
/S/	Start_of_Packet	1	/K27.7/
/T/	End_of_Packet	1	/K29.7/
/V/	Error_Propagation	1	/K30.7/
/LI/	**LPI**		Correcting /LI1/, Preserving /LI2/

/LI/	LPI		Correcting /LI1/, Preserving /LI2/
/LI1/	LPI 1	2	/K28.5/D6.5/
/LI2/	LPI 2	2	/K28.5/D26.4/

[a]Two data code-groups representing the Config_Reg value.

This table defines a mapping from code-groups to "ordered sets". We can create a few more types, then write a decoder:

```
#[derive(Copy, Clone, Debug, Eq, PartialEq)]
enum OrderedSet {
    Configuration(Codegroup, Codegroup),
    Idle,
    CarrierExtend,
    StartOfPacket,
    EndOfPacket,
    ErrorPropagation,
    LinkPartnerIdle,
}

enum Pcs {
    OrderedSet(OrderedSet),
    Data(u8),
}

/// Decodes a set of codegroups into PCS-level control and data
fn decode(cgs: &[Codegroup]) -> Vec<Pcs> {
    let mut cg_iter = cgs.iter().cloned();
```

```
    let mut pcs = vec![];
    while let Some(cg) = cg_iter.next() {
        // Table 36-3
        let s = match cg {
            Codegroup::K28_5 => match read_k28_5(&mut cg_iter) {
                Some(c) => Pcs::OrderedSet(c),
                None => break,
            },
            Codegroup::K23_7 => Pcs::OrderedSet(OrderedSet::CarrierExtend),
            Codegroup::K27_7 => Pcs::OrderedSet(OrderedSet::StartOfPacket),
            Codegroup::K29_7 => Pcs::OrderedSet(OrderedSet::EndOfPacket),
            Codegroup::K30_7 => Pcs::OrderedSet(OrderedSet::ErrorPropagation),
            d => {
                if !d.is_data() {
                    warn!("Unexpected special codegroup: {:?}", d);
                    continue;
                }
                Pcs::Data(u8::from(d))
            }
        };
        pcs.push(s);
    }
    pcs
}

/// Reads the data that follows a K28.5 codegroup, forming an ordered set
///
/// Returns `None` if the stream terminates.
```

```
fn read_k28_5(
    mut iter: impl Iterator<Item = Codegroup>
) -> Option<OrderedSet> {
    let out = match iter.next()? {
        Codegroup::D21_5 | Codegroup::D2_2 => {
            let d1 = iter.next()?;
            let d2 = iter.next()?;
            if !d1.is_data() {
                warn!("Unexpected special codegroup: {:?}", d1);
            }
            if !d2.is_data() {
                warn!("Unexpected special codegroup: {:?}", d2);
            }
            OrderedSet::Configuration(d1, d2)
        }

        Codegroup::D5_6 | Codegroup::D16_2 => OrderedSet::Idle,
        Codegroup::D6_5 | Codegroup::D26_4 => OrderedSet::LinkPartnerIdle,
        c => {
            // "A received ordered set that consists of two
            //  code-groups, the first of which is /K28.5/ and the
            //  second of which is a data code-group other than
            //  /D21.5/ or /D2.2/ (or /D6.5/ or /D26.4/ to support
            //  EEE capability), is treated as an /I/ ordered set."
            //  [36.2.4.12]
            warn!("Unexpected codegroup after K28.5: {:?}", c);
            OrderedSet::Idle
        }
```

```
    };
    Some(out)
}
```

Printing the decoded results for port 0, we're **almost there**:

```
/I/ x2024
/S/
55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 5
/T/
/R/
/I/ x1939
/S/
55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 5
/T/
/R/
/I/ x1436
```

There's a repeating pattern here:

- Repeated idle characters (`/I/`)
- Start of packet (`/S/`)
- Packet data (beginning with `55 55 55...`)
- End of packet (`/T/`)
- Carrier extend (`/R/`)

The packet data is a bit curious, though. `0x55` is part of the ethernet frame

preamble, but it should only appear 7 times, not... 69.

It turns out there's a simple explanation: this link is running at 100M mode, but (Q)SGMII only runs at gigabit speeds. Rate adaption is a simple matter of repeating each byte 10x, with the first `/S/` replacing one instance of `D0`:

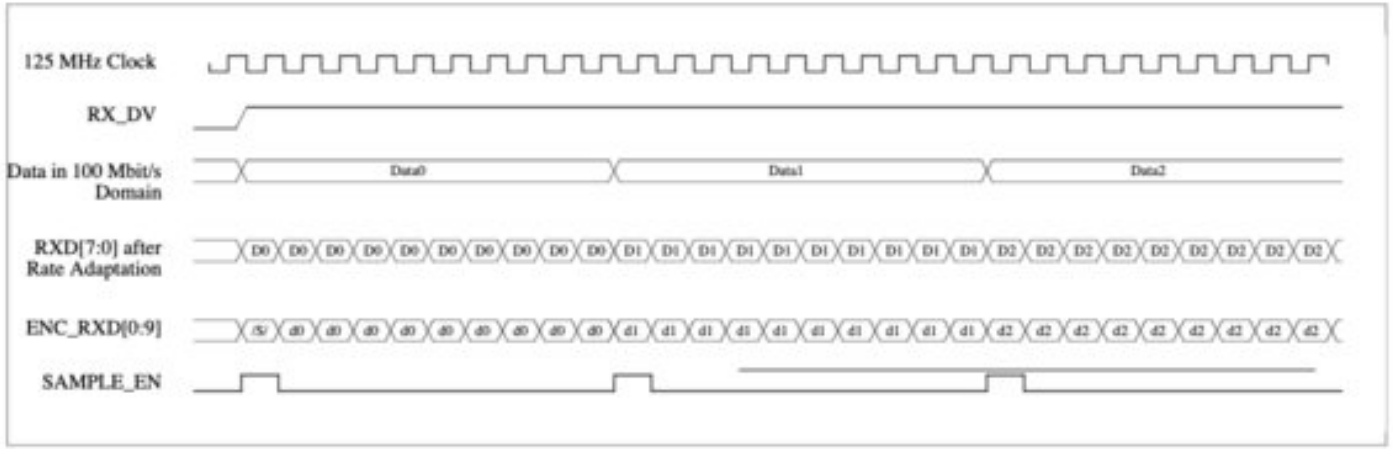


Figure 7 Data Sampling in 100 Mbit/s mode

Taking every 10th byte from the packet data, we see something very promising:

```
/I/ x2024
/S/
55 55 55 55 55 55 55 d5 33 33 00 00 00 01 0e 1d f3 5c 9d 24 86 dd 60 00 00 00 00
/T/
/R/
/I/ x1939
/S/
55 55 55 55 55 55 55 d5 33 33 00 00 00 01 0e 1d f3 5c 9d 24 86 dd 60 00 00 00 00
```

```
/T/
/R/
/I/ x1436
```

Delimited by start and end markers, we see data beginning with

```
55 55 55 55 55 55 55 d5
```

which is the ethernet frame preamble!

Continuing to decode the ethernet frame with our eyes:

- The destination MAC address is `33:33:00:00:00:01`, an IPv6 "all-nodes" multicast address
- The source MAC address is `0e:1d:f3:5c:9d:24`
- The ethertype is `0x86DD`, which indicates IPv6

We've finally made it to the ethernet frame level!

Storing and analyzing packets

Decoding ethernet frames with our eyes gets old fast.

Luckily, there are lots of good tools for working with frame data. Using the `pcap` library, we can write out a `.pcap` file to be analyzed with Wireshark.

Here's our full analyzer, going from `.wfm` to four `.pcap` files:

```
$ cargo run --release -- udp-spam.wfm
[2022-08-08T13:22:35Z INFO  qsgmii] Opening udp-spam.wfm
[2022-08-08T13:22:36Z INFO  qsgmii] Loaded 100000000 samples
[2022-08-08T13:22:36Z INFO  qsgmii] Found 290319 crossings
[2022-08-08T13:22:36Z INFO  qsgmii] Using 200 samples per QSGMII clock
[2022-08-08T13:22:36Z INFO  qsgmii] Found 21596 commas
[2022-08-08T13:22:36Z INFO  qsgmii] Found 49776 code-groups
[2022-08-08T13:22:36Z WARN  qsgmii] Skipping short packet []
[2022-08-08T13:22:36Z WARN  qsgmii] Skipping short packet []
[2022-08-08T13:22:36Z INFO  qsgmii] Wrote 2 packets from port 0 to out.pcap.0
[2022-08-08T13:22:36Z WARN  qsgmii] Skipping short packet []
[2022-08-08T13:22:36Z WARN  qsgmii] Skipping short packet []
[2022-08-08T13:22:36Z INFO  qsgmii] Wrote 2 packets from port 1 to out.pcap.1
[2022-08-08T13:22:36Z WARN  qsgmii] Skipping short packet []
[2022-08-08T13:22:36Z WARN  qsgmii] Skipping short packet []
[2022-08-08T13:22:36Z INFO  qsgmii] Wrote 2 packets from port 2 to out.pcap.2
[2022-08-08T13:22:36Z WARN  qsgmii] Skipping short packet []
[2022-08-08T13:22:36Z WARN  qsgmii] Skipping short packet []
[2022-08-08T13:22:36Z INFO  qsgmii] Wrote 2 packets from port 3 to out.pcap.3
```

The whole pipeline – from loading the `.wfm` to writing the `.pcap` file – runs in about 410 milliseconds on my computer. Considering I put no effort into optimization, this isn't too bad!

Using `tshark`, we can confirm that these are UDP packets:

```
$ tshark -r out.pcap.0
    1   0.000000 fe80::c1d:f3ff:fe5c:9d24 → ff02::1      UDP 82 997 → 8 Len=8
```

```
    2   0.000000 fe80::c1d:f3ff:fe5c:9d24 → ff02::1      UDP 82 997 → 8 Len=8
```

At this point, we've entered the realm of conventional packet-handling tools. For example, using `tshark -V`, we can get a (very) verbose explanation of the packets:

```
$ tshark -V -r out.pcap.0
Frame 1: 82 bytes on wire (656 bits), 82 bytes captured (656 bits)
    Encapsulation type: Ethernet (1)
    Arrival Time: Aug  8, 2022 09:22:36.372717000 EDT
    [Time shift for this packet: 0.000000000 seconds]
    Epoch Time: 1659964956.372717000 seconds
    [Time delta from previous captured frame: 0.000000000 seconds]
    [Time delta from previous displayed frame: 0.000000000 seconds]
    [Time since reference or first frame: 0.000000000 seconds]
    Frame Number: 1
    Frame Length: 82 bytes (656 bits)
    Capture Length: 82 bytes (656 bits)
    [Frame is marked: False]
    [Frame is ignored: False]
    [Protocols in frame: eth:ethertype:ipv6:udp:data]
Ethernet II, Src: 0e:1d:f3:5c:9d:24 (0e:1d:f3:5c:9d:24), Dst: IPv6mcast_01 (33:33:00:00:00:01)
    Destination: IPv6mcast_01 (33:33:00:00:00:01)
        Address: IPv6mcast_01 (33:33:00:00:00:01)
        .... ..1. .... .... .... .... = LG bit: Locally administered address (this is NOT the factory default)
        .... ...1 .... .... .... .... = IG bit: Group address (multicast/broadcast)
    Source: 0e:1d:f3:5c:9d:24 (0e:1d:f3:5c:9d:24)
        Address: 0e:1d:f3:5c:9d:24 (0e:1d:f3:5c:9d:24)
```

```
        .... ..1. .... .... .... .... = LG bit: Locally administered address (this is NOT the factory default)
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
    Type: IPv6 (0x86dd)
    Trailer: 1e3002580c01009b
    Frame check sequence: 0x00000c01 [unverified]
    [FCS Status: Unverified]
Internet Protocol Version 6, Src: fe80::c1d:f3ff:fe5c:9d24, Dst: ff02::1
    0110 .... = Version: 6
    .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
        .... 0000 00.. .... .... .... .... .... = Differentiated Services Codepoint: Default (0)
        .... .... ..00 .... .... .... .... .... = Explicit Congestion Notification: Not ECN-Capable Transport (0)
    .... 0000 0000 0000 0000 0000 = Flow Label: 0x00000
    Payload Length: 16
    Next Header: UDP (17)
    Hop Limit: 64
    Source Address: fe80::c1d:f3ff:fe5c:9d24
    Destination Address: ff02::1
User Datagram Protocol, Src Port: 997, Dst Port: 8
    Source Port: 997
    Destination Port: 8
    Length: 16
    Checksum: 0x52aa [unverified]
    [Checksum Status: Unverified]
    [Stream index: 0]
    [Timestamps]
        [Time since first frame: 0.000000000 seconds]
        [Time since previous frame: 0.000000000 seconds]
    UDP payload (8 bytes)
```

```
Data (8 bytes)

0000  01 02 03 04 05 06 07 08                           ........
    Data: 0102030405060708
    [Length: 8]
```

These are the UDP packets I was looking for!

Misson accomplished.

Thanks to Eric, Arjen, RFK, and everyone else at Oxide that helped with this investigation!

As further reading, "Gigabit Ethernet and Fibre Channel Technology" describes the physical and data link layers in great detail.

For a less home-brew workflow, check out Andrew Zonenberg's excellent work on `glscopeclient`, which can now do this kind of QSGMII decoding.