Reverse engineering the MINI Cooper automotive CAN bus message format.

Software study group presentation

February 2011

LOGIC PD™

Why spend time reverse engineering a CAN bus?

> Everyone has heard that modern cars use the CAN bus to send messages between various parts of the vehicle.
> Those messages must be interesting or the manufacturer wouldn’t build an entire bus to transport them right?
> Wouldn’t you like to know what the various bits of your car are saying to each other?

Controller Area Network bus overview. (Copied from Wikipedia)

- CAN is a message based protocol, designed specifically for automotive applications but now also used in other areas such as industrial automation and medical equipment.
- Development of the CAN-bus started originally in 1983 at Robert Bosch GmbH. The protocol was officially released by the SAE in 1986. The first CAN controller chips, produced by Intel and Philips, came on the market in 1987. Bosch published the CAN 2.0 specification in 1991.

So… CAN I get access to the data somehow?

- **ISO 11898-1**: CAN Data Link Layer and Physical Signaling. (Costs $, so I didn't read it.)
- Microchip AN228: A CAN Physical Layer Discussion
  - 2-wire differential signaling. NICE!
- Lots of CAN-to-USB converters out there. I borrowed one from a local FAE and wired it to the MINI
- A single CAN bus connects the Instrument cluster, Engine Control Module, Automatic Transmission Module, Antilock Braking System, Automatic Stability Control, Electro Hydraulic Steering and the Steering Angle Sensor together. (What could possibly go wrong!)

Problem #1 data transmission rate?

- Since the CAN bus is differential, looking at one wire with respect to ground looks approximately like this:

  &@!#^(@&^$#(*&(!*^@%$(!^%@#%!&@^%#(&!^
- Trying to hook two oscilloscope probes up, inverting one signal and adding them while balancing a scope on your lap without accidentally unplugging it from the extension cord, dropping it on the garage floor or shorting out the CAN bus didn't help much.
- How about the "I'll know the data is right when I see it" method: Pick rates until it looks right. This is pretty easy. There are only a few well-known possibilities.

What do you get?

> The protocol adapter does all the hard work. You get nice messages out of it once you pick the right bit rate.
> CAN messages are well-defined:
  + I got a timestamp, Message ID and message bytes.
  + All messages had 8 data bytes.

```
327a 0153 00 51 00 00 00 ff 00 80
327a 01f0 0a 20 0a 00 0a 00 0a 00
327a 01f8 00 00 00 00 fe ff 00 00
327c 0316 01 00 00 00 00 00 00 00
327c 0336 00 00 fe 02 82 15 b0 67
```

The parts of a message:

```
> 327a 0153 00 51 00 00 00 ff 00 80

Timestamp:         327a
Message ID:        0153
8 bytes of data: 00 51 00 00 00 ff 00 80
```

I recorded 1.2 messages while I drove around. There were 13 different massage IDs

Count	Message ID
158865	0x0153
158865	0x01f0
158865	0x01f8
216338	0x0316
216334	0x0329
216335	0x0336
124942	0x0545
5432	0x0613
5432	0x0615
5432	0x0618
6010	0x061a
2209	0x061f
556	0x0630

Things to notice:

- The number of messages received and the message IDs are inversely-related! What? I got more messages with low numbered IDs than I got messages of high numbered IDs.
- Using Google, I found this: "A message consists primarily of an ID **which represents the priority of the message** and up to eight data bytes."
- That would explain it. Lower message IDs are higher priority and occur more often.
- (Don’t look too closely at the table and complain that some of the counts and message IDs don't fall nicely into the description above. Real life is messy.)

What to look at first?

- How about message ID 0x0153 since it's the highest priority message encountered.
- grep " 0153" log1_clean | head

```
327a 0153 00 51 00 00 00 ff 00 80
3281 0153 00 51 00 00 00 ff 00 80
3288 0153 00 51 00 00 00 ff 00 80
328f 0153 00 51 00 00 00 ff 00 80
3296 0153 00 51 00 00 00 ff 00 80
329d 0153 00 51 00 00 00 ff 00 80
32a4 0153 00 51 00 00 00 ff 00 80
```

BORING!!!!!

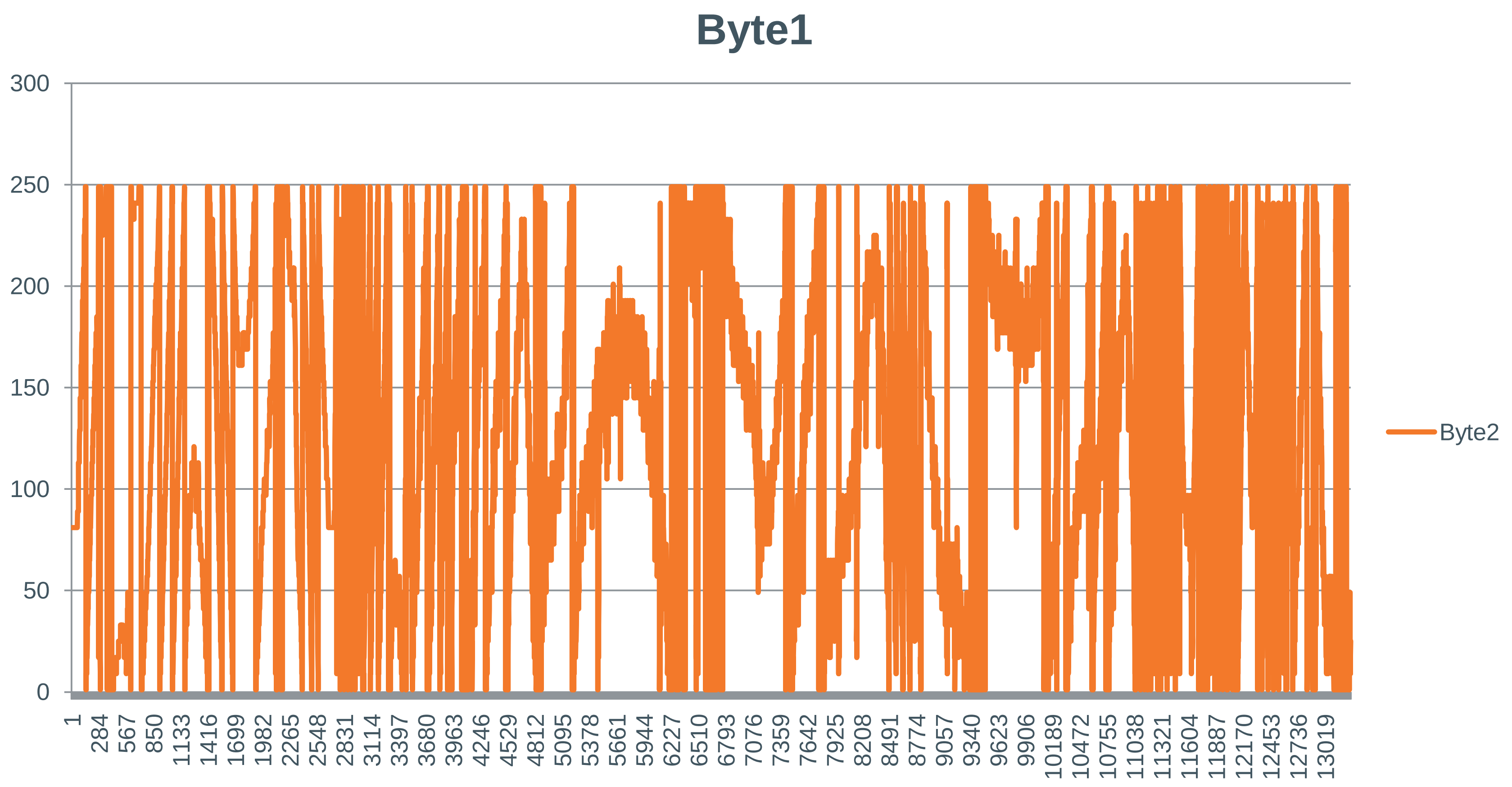
Let's toss the timestamp, sort the data, remove all the duplicates and see if there's something more interesting going on.

> ```
> grep " 0153" log1_clean | cut -d' ' -f2- | sort | uniq | head
> 0153 00 01 01 00 00 ff 00 80
> 0153 00 01 02 00 00 ff 00 80
> 0153 00 01 03 00 00 ff 00 80
> 0153 00 01 04 00 00 ff 00 80
> 0153 00 01 05 00 00 ff 00 80
> 0153 00 01 06 00 00 ff 00 80
> 0153 00 01 07 00 00 ff 00 80
> ```

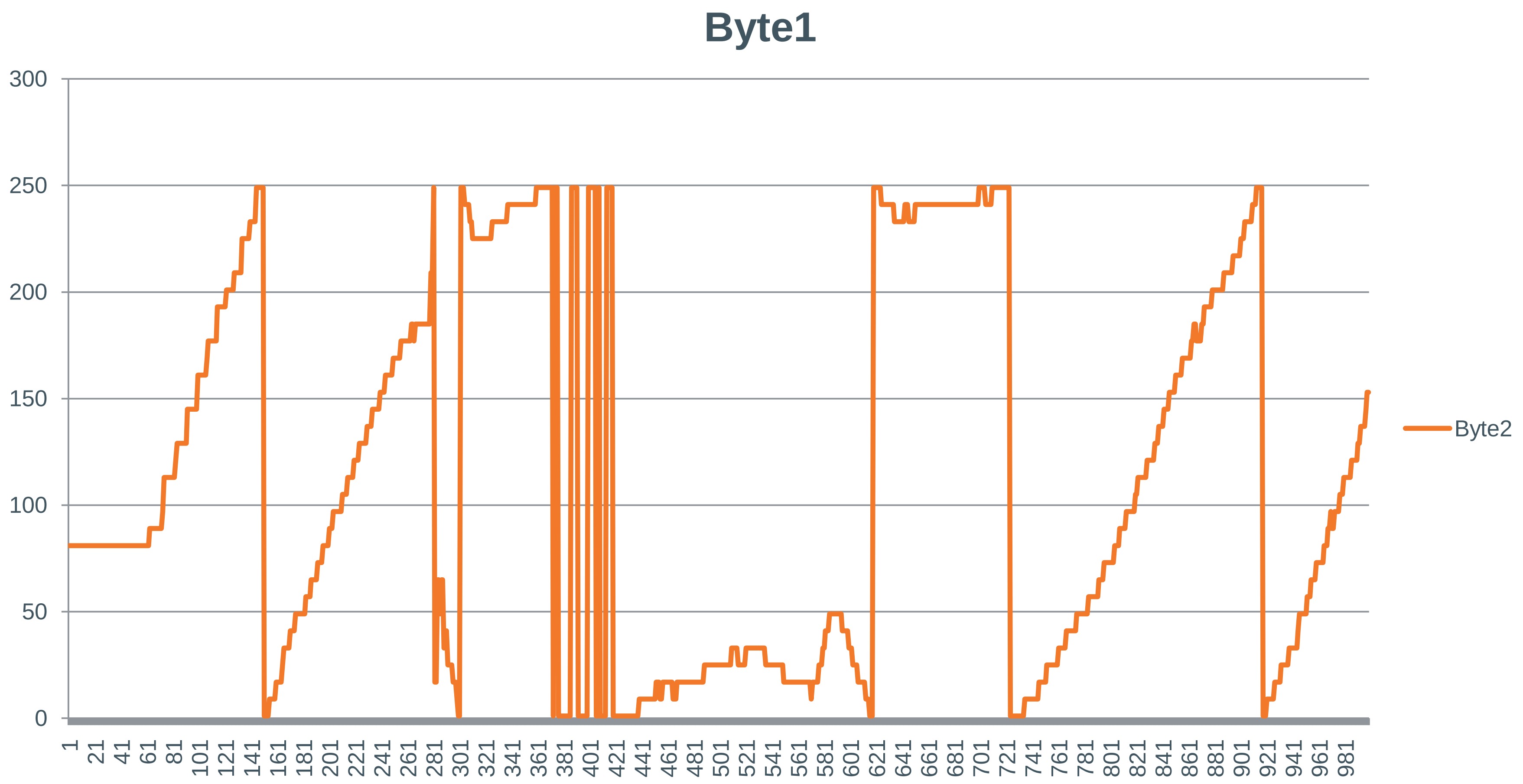
> That looks more interesting.

> There's data in byte #1 at least. Let's look there.

Byte #1 data: Looks like *something's* there.



Just look at the first 1,000 messages. Looks better!



This is a pattern to look out for:

> The plot looks continuous from 0 through about 153 and then starts over at zero and looks continuous again for a while.

> This is the classic pattern for the lower-significance bits of a larger bit field.

> Imagine looking at a ramp from 0 to 100, but cover up the most significant digit. You'll see 0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9

> That's just what we're seeing, so let's look for the upper bits. They are probably in the packet. Byte #2 maybe?

But, not so fast. It would be great to know which bits change in the packet first.

> Some of the data in the packet might be the same for all packets in the log. How can we tell which bits CHANGE at least once in the log?

> Thanks to Cooper and Elmquist, this algorithm works:

1) Record the data for the first packet of a type.
2) Exclusive-or the new packet with the saved packet to get just the bits that are different.
3) OR those bits into an array holding all the bits that have changed.

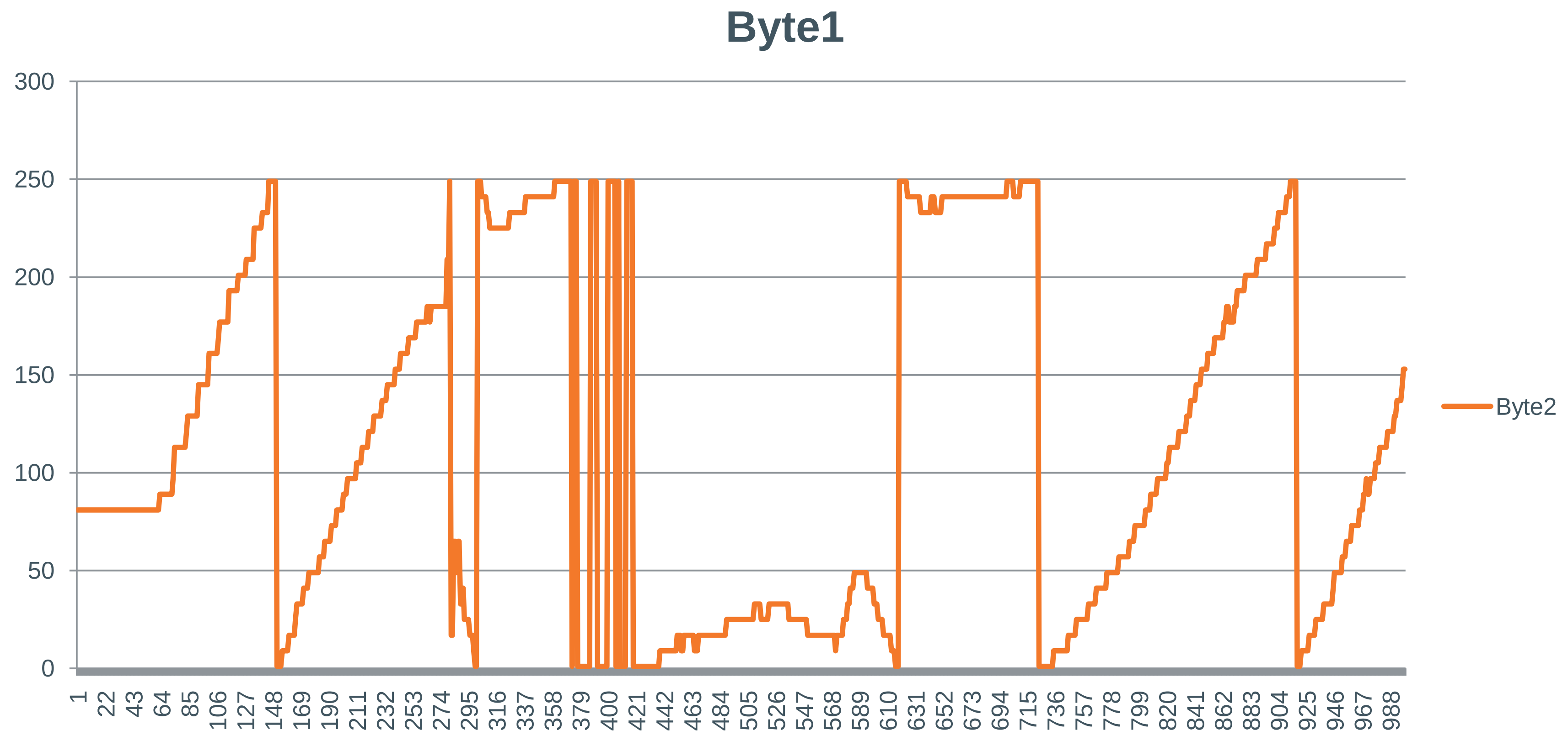
At the end, print out the array.

Which bits changed in which packets:

```
> 153 10 f8 3f 00 00 00 00 80
> 1f0 ff e7 ff c7 ff 27 ff 87 ← This ID looks fun.
> 1f8 7f 00 00 00 00 00 00 00 ← One 7-bit value?
> 316 01 00 ff 7f 00 00 00 00
> 329 f3 7f 00 00 00 ff 00 00
> 336 ff 7f 00 02 ff 1f ff 7f
> 545 12 ff ff 00 00 00 00 00
> 613 00 00 00 00 00 df f7 00
> 615 00 00 00 03 02 00 00 00 ← Only 3 bits change!
> 618 00 00 3f 00 00 00 00 00 ← Just one 6-bit field?
> 61a 07 00 00 ff 00 ff 00 00
> 61f ff ff 0f c0 42 00 00 00
> 630 00 00 00 00 00 00 00 00 ← This ID is real boring.
```

Back to the packet ID 0x0153

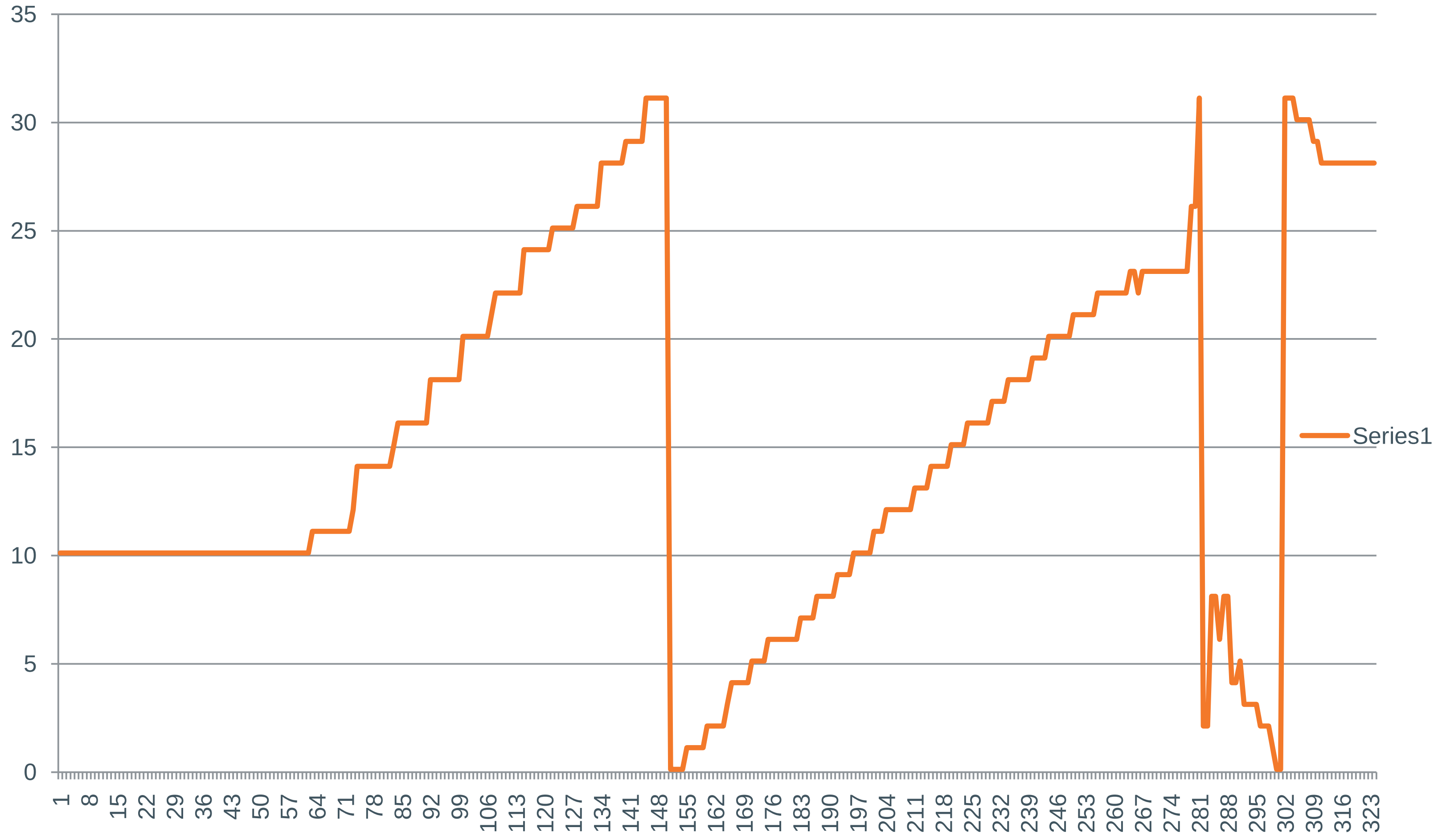
> We plotted byte #1 and got this: (Let's look closer at it.)



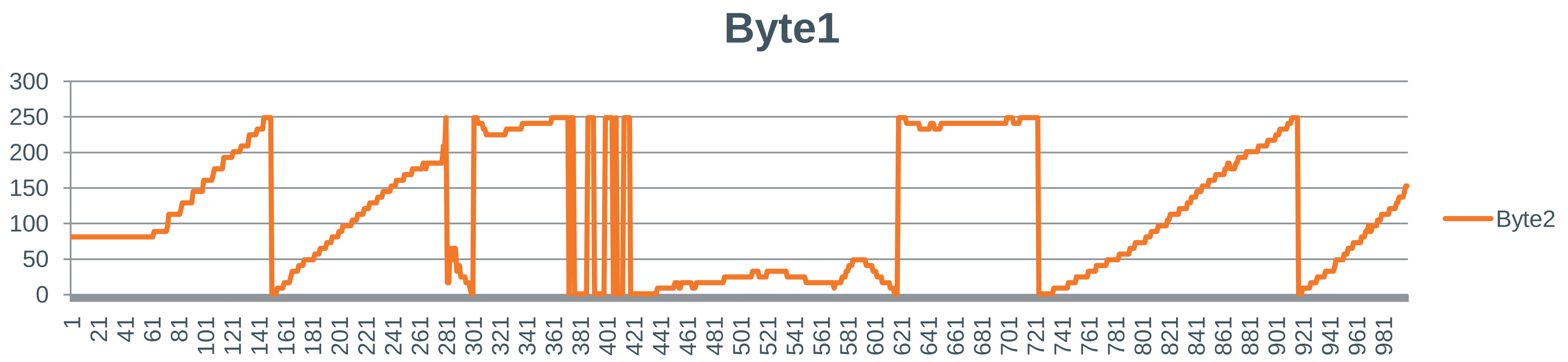
But byte #1 has bits-changed of 0xF8.

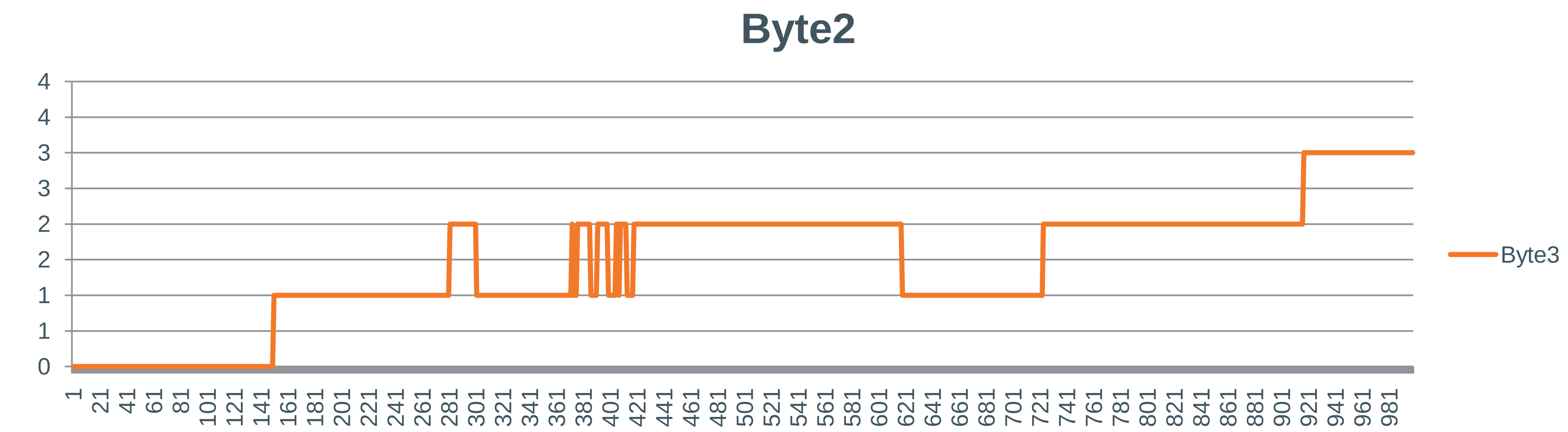
- Why do I see a 0x51 (D'81)in there? The 1 bit never changes since we see a change mask of 0xF8
- It looks like the least-significant bits don't change.
- Looking at the beginning and removing dups we see this:
  - 81
  - 89
  - 97
  - 113
  - 121
- HEY! They go up by 8 each time. (16 if you do the math)
- The least-significant 8 bits probably don't matter. >>3

Now the data goes up and down by 1 (with a bias)



So where are the more significant bits? Next byte?

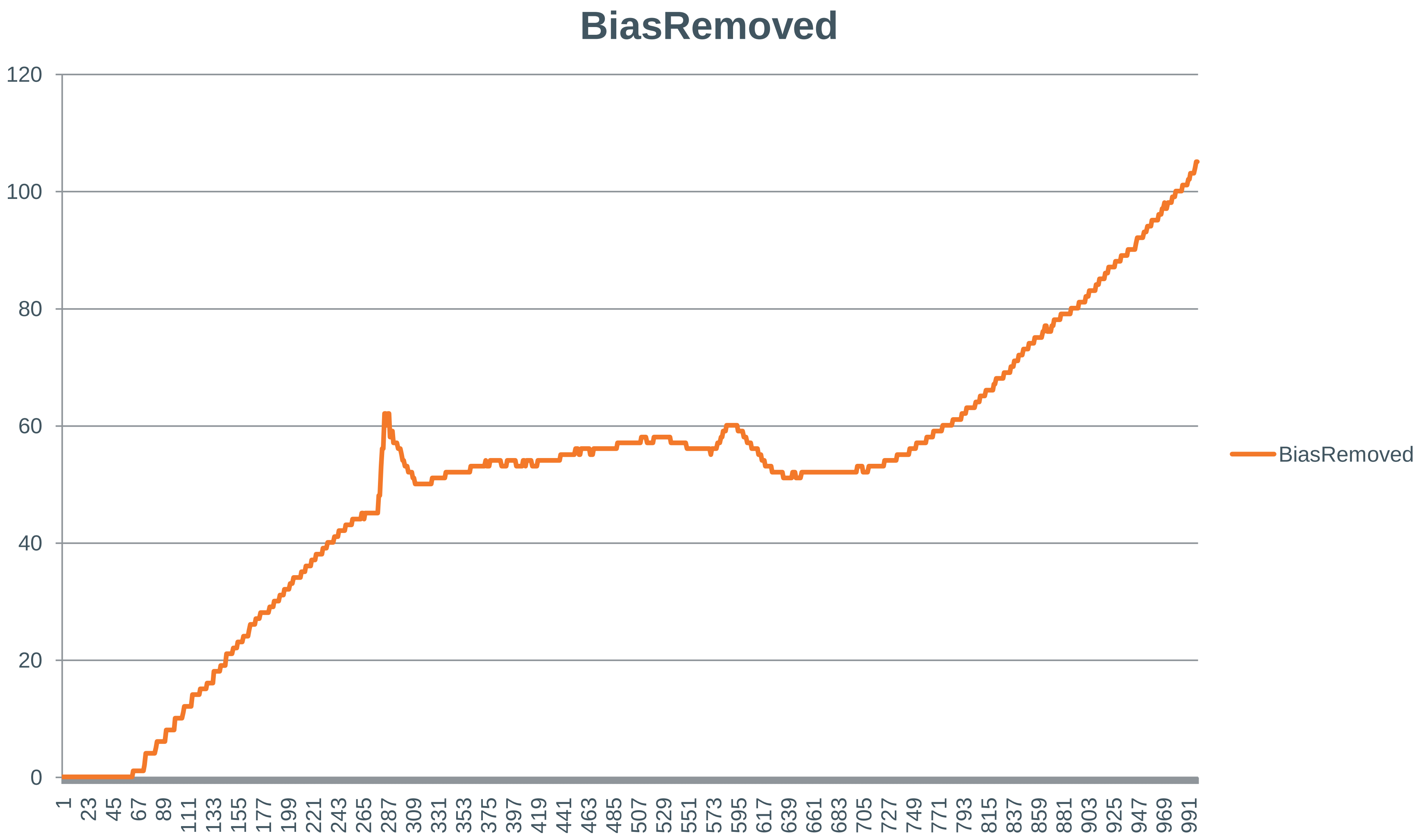




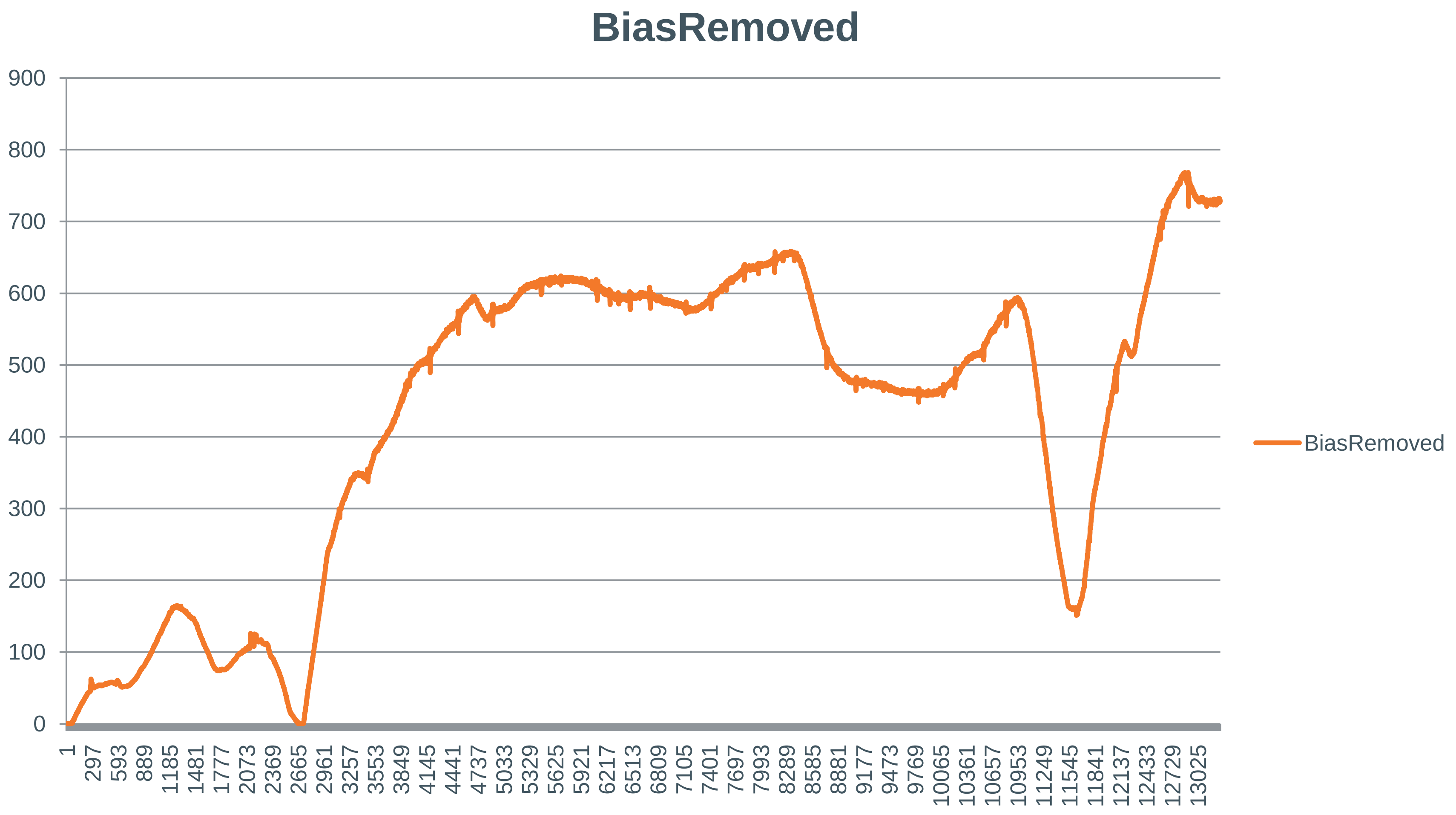
See how byte #2 goes up or down?

- Every time byte #1 has a discontinuity, byte #2 increments or decrements.
- This is EXACTLY what we want to see if a bit field is split into two parts. All we need to do is glue them together.
- Take byte #2 shift it left 8 bits (multiply by 256) and add it to byte #1.
- Then shift the whole mess right 3 bits (divide by 8)
- Then subtract 10.
- Simple right? What do you get then?

Final graph of who-knows? Looks fabulous!



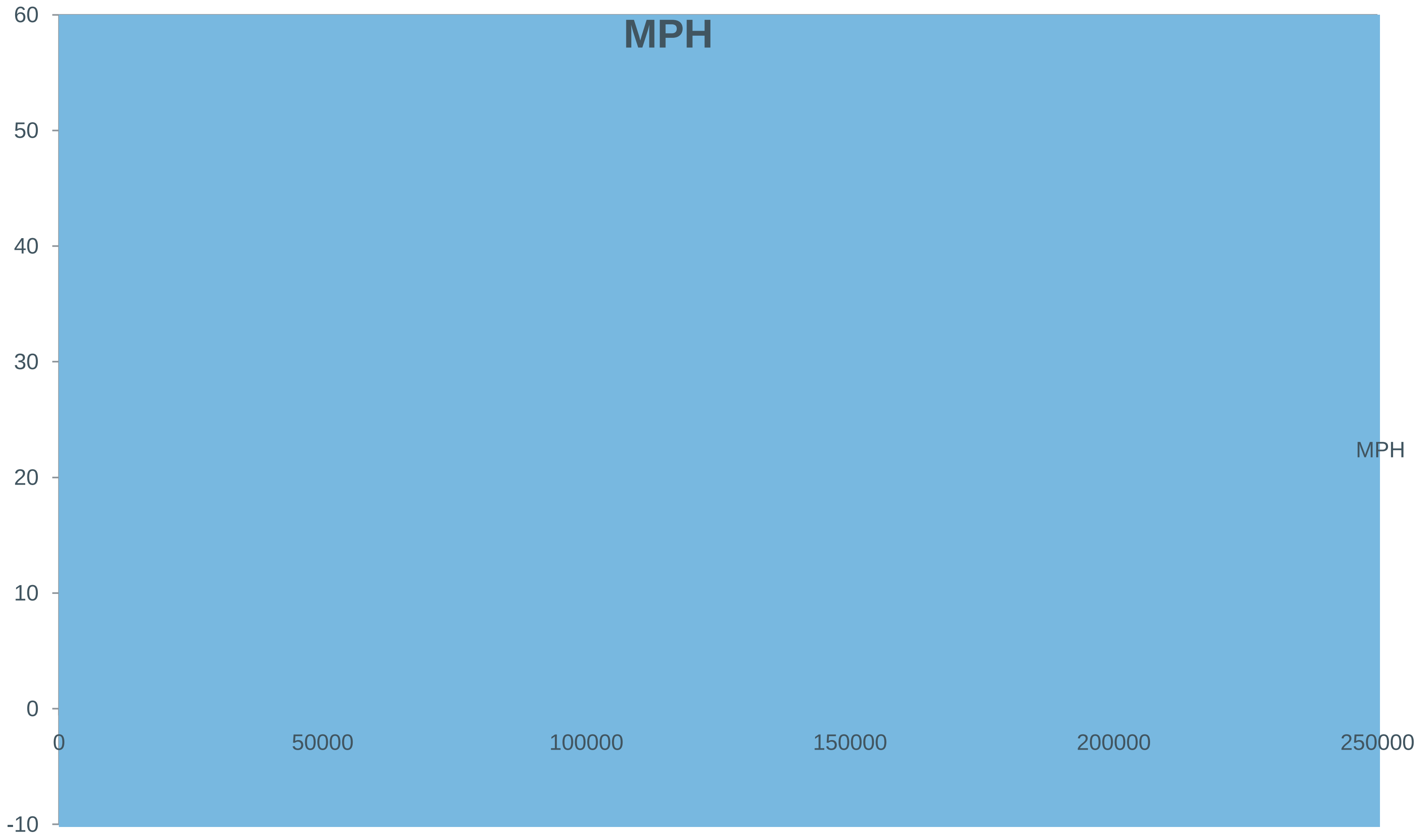
Longer plot of whatever it is.



What is it?

- I wrote down what I did:
- 1) Back down driveway
- 2) Stop
- 3) Drive through neighborhood
- This could be vehicle speed! (Maybe)
- In what units? MPH, KPH, furlongs-per-fortnight?
- MPH? (probably not since I don’t drive 600 MPH.)
- Somewhere along the line I decided I needed to divide by 22 to get MPH. That seems dubious, but I don’t have records of the drive anymore, so OK, let’s go with that.

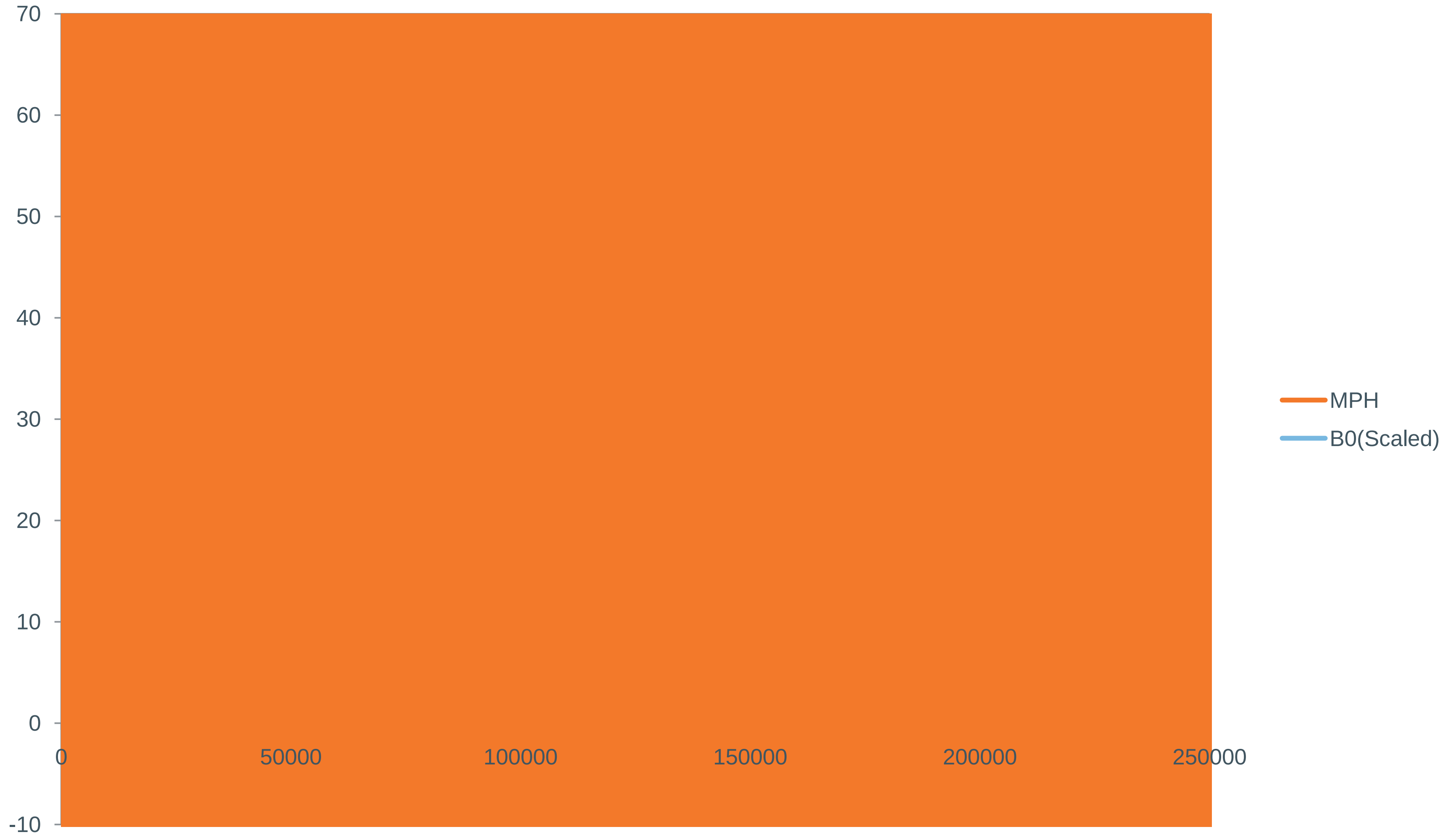
Longer Speed plot. (Captured from older dataset)



OK, we nailed that one. What are the other bits?

- Byte #0 has changing data in the 2^4 bit (0x10)
- Byte #7 has changing data in the 2^7 bit. (0x80)
- (Looking at byte #7 it’s 0x80 until I turn the car off, so it’s not very interesting.)
- Plot byte #0 on top of the speed:

Speed and Byte #0



That’s interesting.

> It looks like Byte #0 bit 2^4 goes to a 1 whenever I’m decelerating.

> I know from looking at OBD data that that happens when the fuel system goes “open-loop” This could be an indication of that condition. It would be easy to correlate it with a live OBD reading to make sure.

OK, now just do the same thing with all the other message types…

- I found these values over the course of about 2 weeks of investigating:

Message type 0x153

	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Bits used	10	F8	3F	00	00	00	00	80
For What?	00 or 10 (Fuel open-loop?)	Vehicle or wheel speed						80 when car is on?

Message type 0x1F0

	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Bits used	FF	E7	FF	C7	FF	27	FF	87
For What?	8 bits + next 3 bits Wheel speed	00, 20, 40, 60 pattern + 3 bits of wheel speed	8 bits + next 3 bits Wheel speed	00, 20, 40, 60 pattern + 3 bits of wheel speed	8 bits + next 3 bits Wheel speed	00, 20, 40, 60 pattern + 3 bits of wheel speed	8 bits + next 3 bits Wheel speed	00, 20, 40, 60 pattern + 3 bits of wheel speed

Message type 0x1F8

	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Bits used	7F	00	00	00	00	00	00	00
For What?	Correlated to Wheel speed							

LOGIC PD™

Message type 0x316

	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Bits used	01	00	FF	7F	00	00	00	00
For What?	Ignition on		RPM * 6 (LSBFirst)					

Message type 0x329

	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Bits used	F3	7F	00	00	00	FF	00	00
For What?	11, 62, 80, C0 pattern	Coolant Temp * 4 in C				Throttle position		

Message type 0x336

	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Bits used	FF	7F	00	02	FF	1F	FF	7F
For What?	Correlated with RPM. Ignition advance?			Ignition on?	?	?	?	?

Message type 0x545

	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Bits used	12	FF	FF	00	00	00	00	00
For What?	12 when not running. Parking Brake?	Fuel consumed?						

Message type 0x613

	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Bits used	00	00	00	00	00	DF	F7	00
For What?						?	?	

Message type 0x615

	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Bits used	00	00	00	03	02	00	00	00
For What?				?	Ignition on?			

Message type 0x618

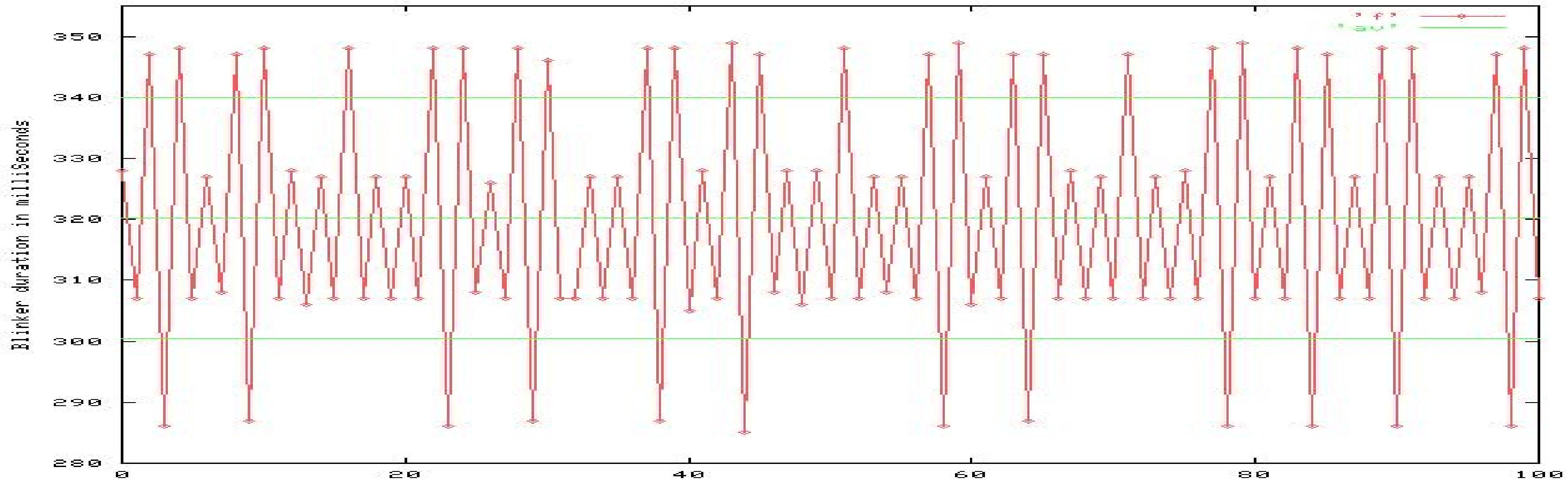
	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Bits used	00	00	3F	00	00	00	00	00
For What?			?					

Message type 0x61A

	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Bits used	07	00	00	FF	00	FF	00	00
For What?	?			Odo?		?		

Message type 0x61F

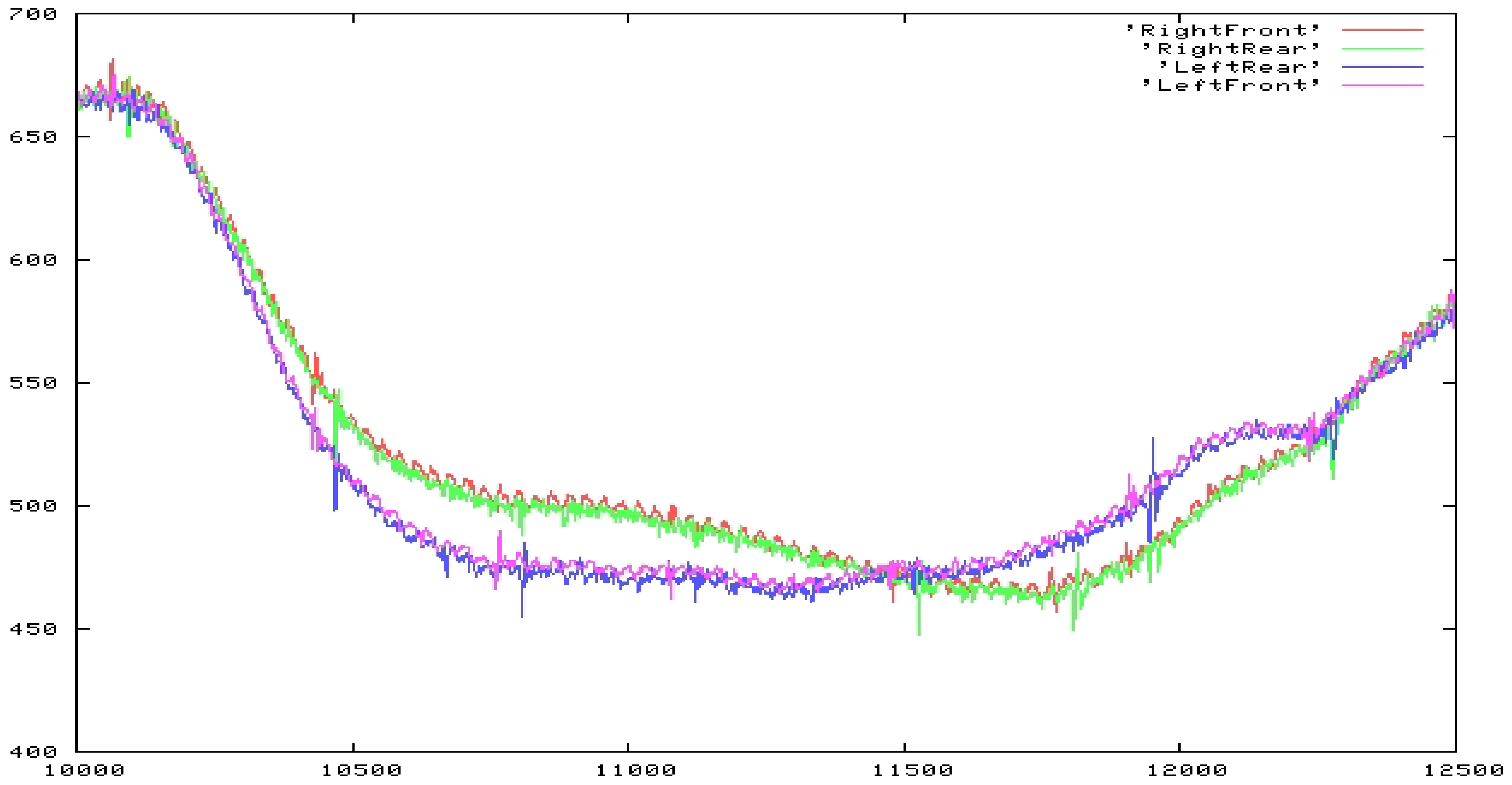
	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Bits used	FF	FF	0F	C0	42	00	00	00
For What?	?	?	?	?	Bit 2 blinker			



Message type 0x630

	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Bits used	00	00	00	00	00	00	00	00
For What?								

I even managed to figure out which wheel sensor was which. How?



Which wheel is which?

- See that there are two lines that are almost identical? I'd say that's probably a front wheel and a rear wheel.
- I turned right, left then straight. Which side is left?
  - As you turn right, the right wheels travel less distance, so they'll go slower.
- OK, we have right vs. left figured out. Front vs. back?
- See the sharp spikes? Could that be sticks or cracks in the road which will make the wheel speed change?
- There are places where a spike happens in one line and then happens later in the other line. That looks like front vs. rear to me.

That’s it.

> This project was great fun.

> I suggest you try it with some random data stream someday.