Exploring CAN & SPI Protocols with the Arduino®

2019

CANBUS COMMUNICATIONS

Ron Kessler

Anteater Racing

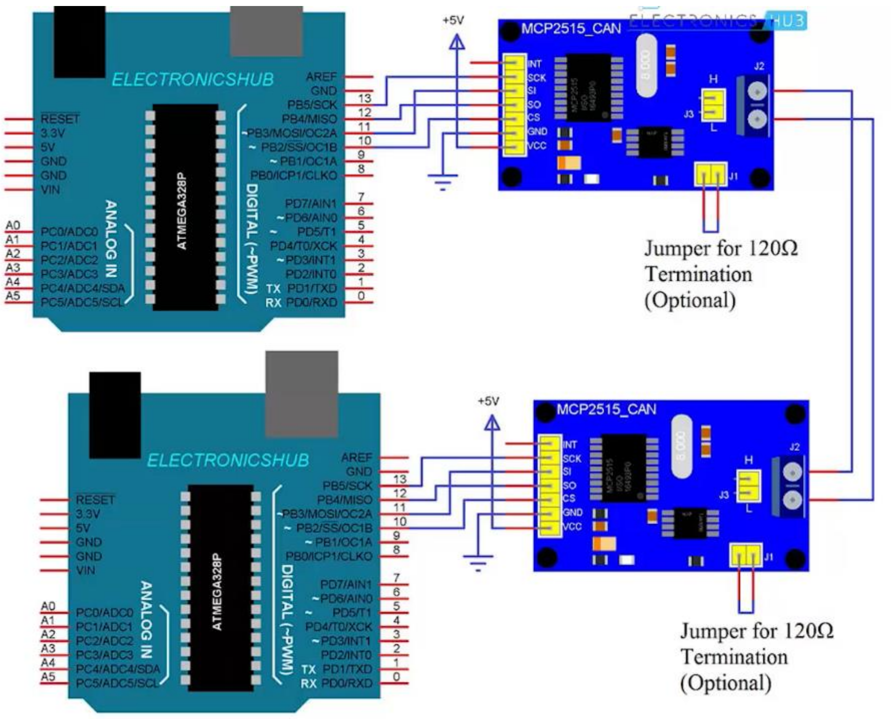
2/22/2019

Working with my CAN-BUS Breadboard Demo: Raw Wave forms

This demo uses a Arduino NANO as a transmitter and an Arduino UNO as a receiver. They communicate using a CANBUS (controller area network). In order to do that, two small modules are used to create the actual CAN bus and they also translate data to and from this bus to the Arduinos. The Arduinos do not have CAN capability built in.

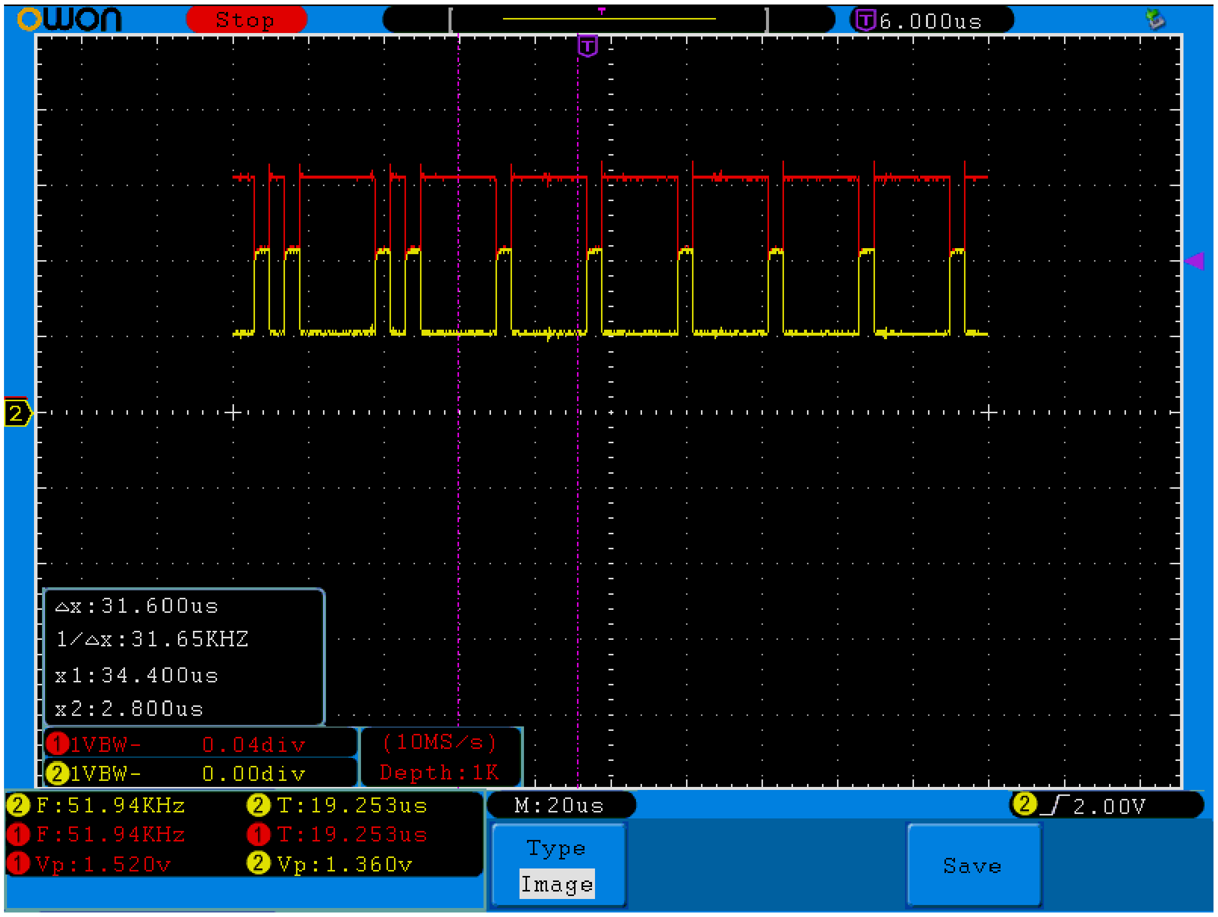
The typical circuit uses MCP2515_CAN modules. The baud rate for the actual CAN bus is programmed into my transmitter sketch on the NANO. This demo is running at 500,000 bits per second.

In order to visualize how this all works, I used my 100Mhz oscilloscope to capture the CAN communication channel. This network uses two wires designated CAN-H and CAN-L (High and Low respectively). They are a mirror image of each other, and the pair are used to accurately and reliably communicate between nodes on the network.



Notice on this first image, that the CAN-High (top) is pulled down and CAN-Low is pulled up when in recessive (not transmitting state) by the terminating resistors.

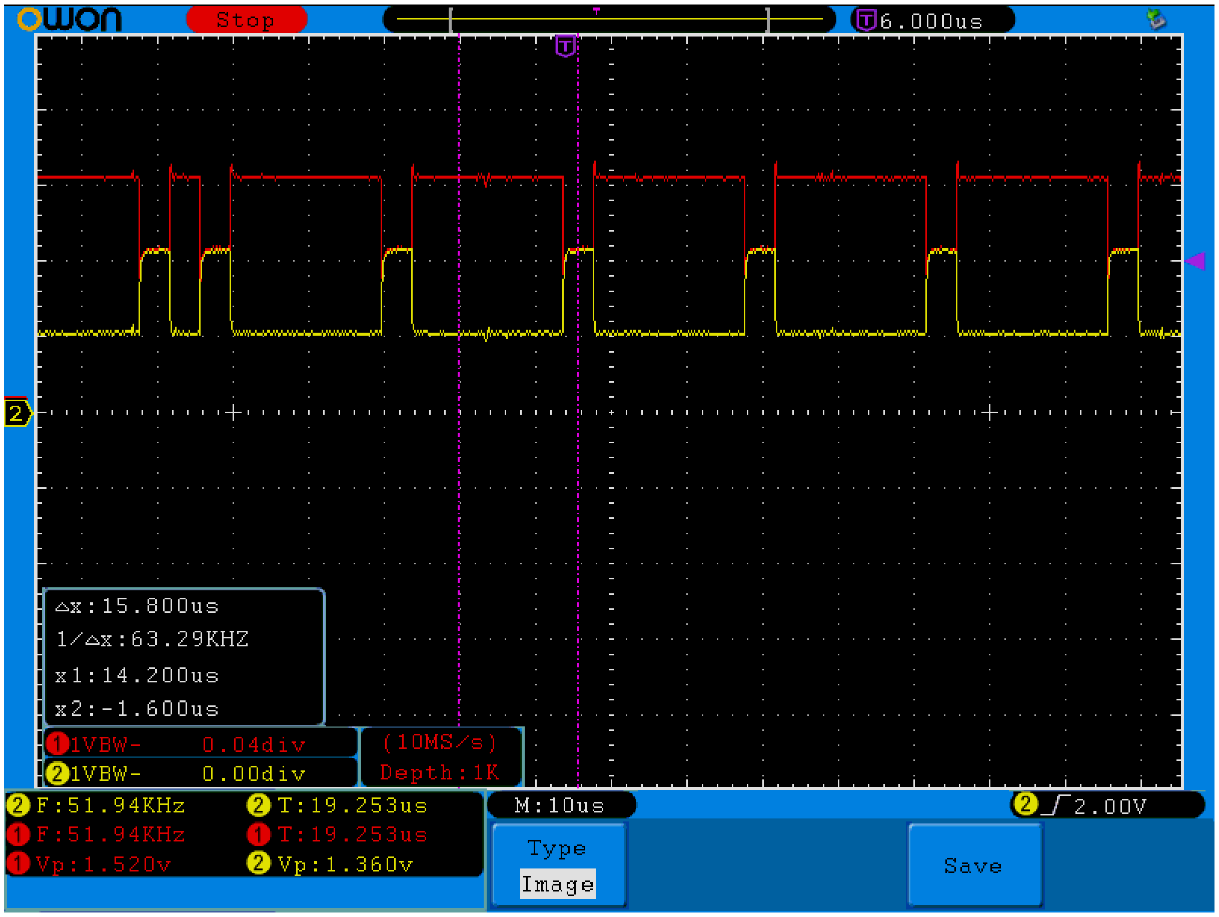
In the dominant state the top red trace is driven up to 3V and the bottom yellow line is driven down to 1V. On average the two signals are around 1.95 volts apart. When the signals are at their maximum delta, that bit is a logical 0 and when they are at the same voltage, the bit is a logical 1. That seems



```
△x:31.600us
1/△x:31.65KHZ
x1:34.400us
x2:2.800us
1 1VBW-      0.04div   (10MS/s)
2 1VBW-      0.00div   Depth:1K
2 F:51.94KHz    2 T:19.253us
1 F:51.94KHz    1 T:19.253us
1 Vp:1.520v     2 Vp:1.360v
```

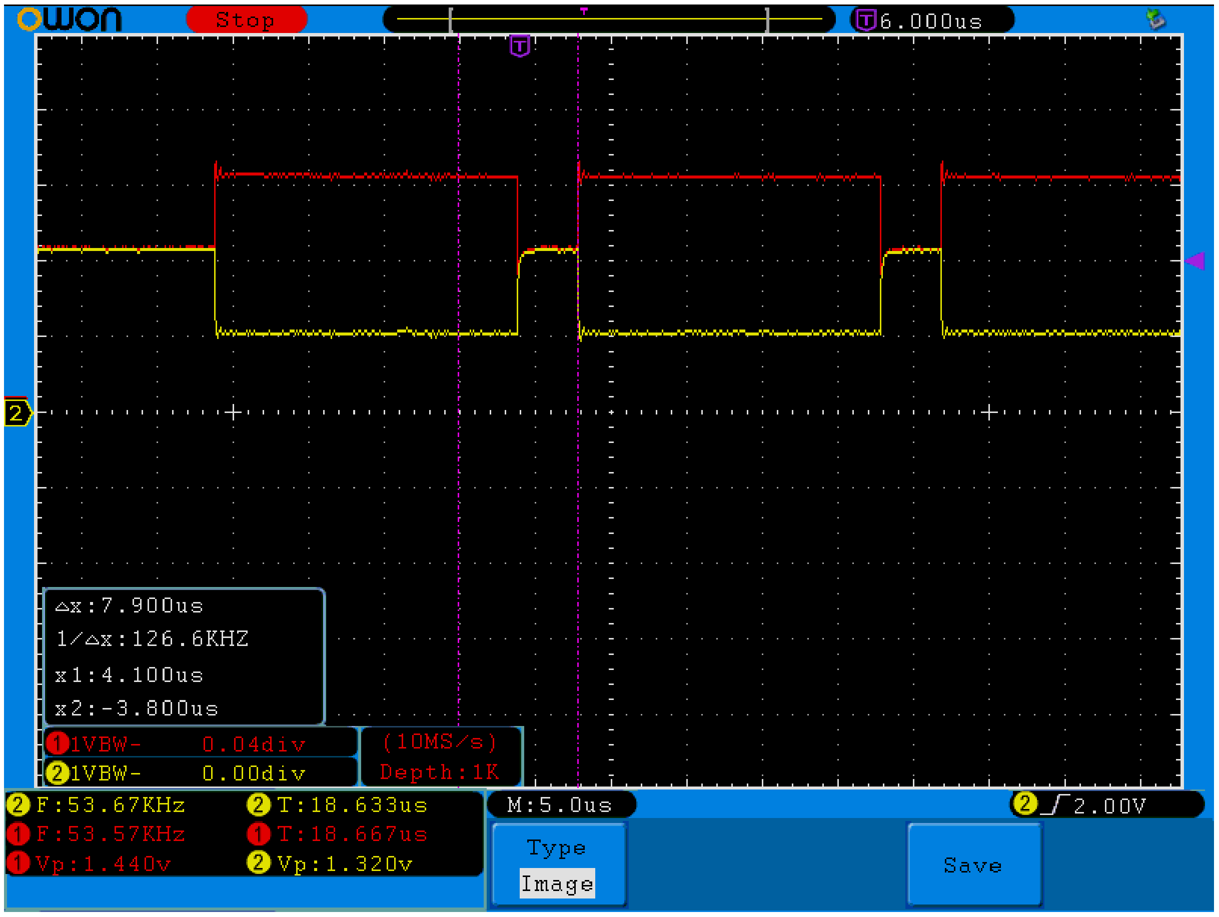
backwards doesn't it. These signals were collected with TWO terminating resistors on the Receiver & Transmitter nodes. The resistors are part of the CAN modules I am using. That measures 60 ohms across the H & L which is what you measure on an actual car.

Here the trace is wider for closer inspection.



```
△x:15.800us
1/△x:63.29KHZ
x1:14.200us
x2:-1.600us
1 1VBW-   0.04div   (10MS/s)
2 1VBW-   0.00div   Depth:1K
2 F:51.94KHz   2 T:19.253us   M:10us   2 2.00V
1 F:51.94KHz   1 T:19.253us
1 Vp:1.520v    2 Vp:1.360v
```

Finally, a closer view yet.



```
△x:7.900us
1/△x:126.6KHZ
x1:4.100us
x2:-3.800us
1 1VBW-      0.04div    (10MS/s)
2 1VBW-      0.00div    Depth:1K
2 F:53.67KHz     2 T:18.633us     M:5.0us     2 2.00V
1 F:53.57KHz     1 T:18.667us
1 Vp:1.440v      2 Vp:1.320v
```

Interpreting SPI Data with the Logic Analyzer

Since the NANO and UNO cannot decode/encode CAN, my CAN modules translate the data into SPI (Serial Peripheral Interface) signals that they can handle.

My Canbus breadboard demo is used to obtain the following results and images. I send 5 bytes from the NANO to the receiver in a loop over CAN. The message is "Hello" (what else). The UNO receives translated CANBUS data via SPI and I measure the packets at the SPI lines coming into the UNO. I am using the Saleae Logic 8 analyzer and Saleae Logic software version 1.2.18 .

Here is a code snippet showing the code from the NANO sent over CAN to my receiving UNO.

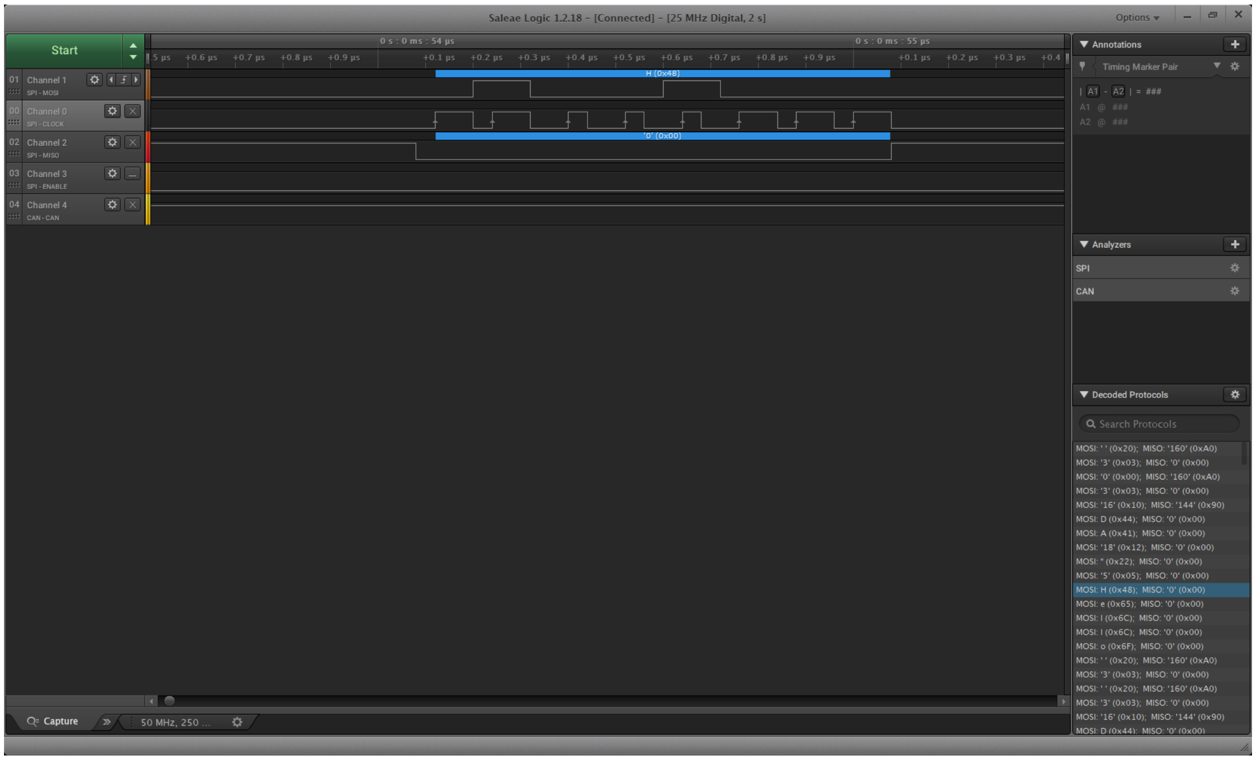
```
//************S T A R T   O F   C A N B U S   T E S T********************
unsigned char  myData[] = {0x48,0x65,0x6C,0x6C,0x6F};      // H,e,l,l,o
//************************************************************************
void setup()
{
        Serial.begin(115200);           //for serial monitor
        //--- set the baudrate of the Bus
        while (CAN_OK != myCAN.begin(CAN_500KBPS))
        {
                Serial.println("CAN BUS initialization failed.....");
                delay(1000);
        }
        Serial.println("CAN BUS Initialized....");
}
void loop()
{
        for(int x = 0;x<5;x++)
          Serial.write(myData[x]);
          Serial.println("");

        //---Make it so! Send the packet of 5 bytes
        myCAN.sendMsgBuf(0x32, 0, 5, myData);
}
```

I first created a character array to hold the data. One letter (one byte) is stored in each of 5 elements in the array. Then I check to make sure the network is up and send each of my pieces of data in order over and over. In this case I chose the #32 to identify my tachometer data. Low ID's have higher priority on CAN networks so if I was transmitting ABS or Crankshaft speed I would choose and number like 0 or 1 for those packets to make sure they were delivered at the expense of say the status of the cabin temperature which is certainly not so critical.

In the next section we will examine the communication packets in detail and see how they appear in the logic analyzer. To obtain the readings, it is only necessary to connect the analyzer to the CAN-L line and the SPI lines. I used a CAN and a SPI analyzer for this demo.

Examining Communication Signals

Here are the first two SPI patterns for my message. This is 'H' (ASCII 48) in 'Hello'. When the clock goes HIGH, data is read. So, look at the eight arrows on the clock pulse. Match up the arrows with the data signal (top trace). When the clock/arrow goes high, read the value of the data bit. Let's walk through it:



```
MOSI: ' ' (0x20);  MISO: '160' (0xA0)
MOSI: '3' (0x03);  MISO: '0' (0x00)
MOSI: '0' (0x00);  MISO: '160' (0xA0)
MOSI: '3' (0x03);  MISO: '0' (0x00)
MOSI: '16' (0x10);  MISO: '144' (0x90)
MOSI: D (0x44);  MISO: '0' (0x00)
MOSI: A (0x41);  MISO: '0' (0x00)
MOSI: '18' (0x12);  MISO: '0' (0x00)
MOSI: " (0x22);  MISO: '0' (0x00)
MOSI: '5' (0x05);  MISO: '0' (0x00)
MOSI: H (0x48);  MISO: '0' (0x00)
MOSI: e (0x65);  MISO: '0' (0x00)
MOSI: l (0x6C);  MISO: '0' (0x00)
MOSI: l (0x6C);  MISO: '0' (0x00)
MOSI: o (0x6F);  MISO: '0' (0x00)
MOSI: ' ' (0x20);  MISO: '160' (0xA0)
MOSI: '3' (0x03);  MISO: '0' (0x00)
MOSI: ' ' (0x20);  MISO: '160' (0xA0)
MOSI: '3' (0x03);  MISO: '0' (0x00)
MOSI: '16' (0x10);  MISO: '144' (0x90)
MOSI: D (0x44);  MISO: '0' (0x00)
```

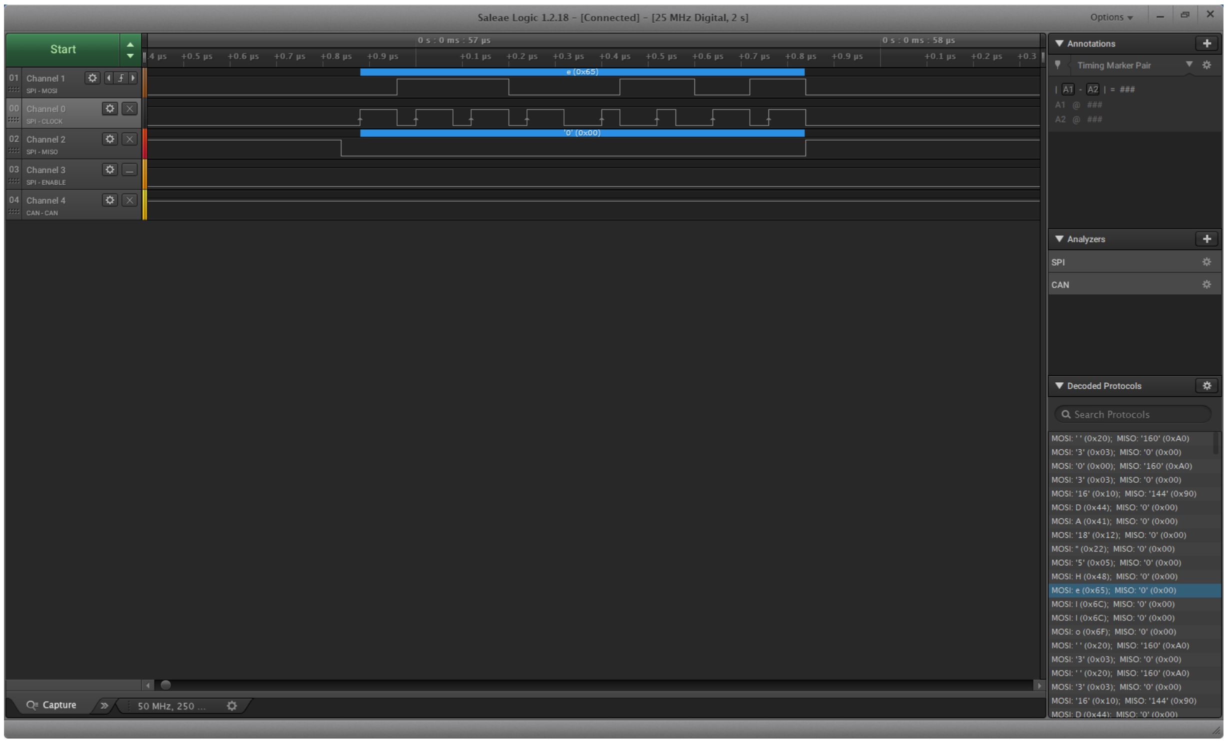
The most significant bit (MSB) is transmitted first (Big Endian).

First nibble = 0100 by examining the arrows (clock going from low to high)
Arrow 1 data is low = 0
Arrow 2 data is high = 1
Arrow 3 data is low = 0
Arrow 4 data is low = 0

Second nibble = 1000
Arrow 5 data is high = 1
Arrow 6 data is low = 0
Arrow 7 data is low = 0
Arrow 8 data is low = 0

So our entire byte = 0100 1000 = 0x48 Hex or 72 decimal or 'H' in ASCII. If you look at the bottom right of the screenshot (decoded protocols) you will see this packet highlighted. Notice the next packet is 'e' followed by 'l', 'l', 'o', space. The analyzer lets us peer into the contents of each piece of information transmitted/received. Notice how critical timing is in digital communication!

Here is the pattern for ‘e’ which is the next letter in the message.



```
MOSI: ' ' (0x20);  MISO: '160' (0xA0)
MOSI: '3' (0x03);  MISO: '0' (0x00)
MOSI: '0' (0x00);  MISO: '160' (0xA0)
MOSI: '3' (0x03);  MISO: '0' (0x00)
MOSI: '16' (0x10);  MISO: '144' (0x90)
MOSI: D (0x44);  MISO: '0' (0x00)
MOSI: A (0x41);  MISO: '0' (0x00)
MOSI: '18' (0x12);  MISO: '0' (0x00)
MOSI: " (0x22);  MISO: '0' (0x00)
MOSI: '5' (0x05);  MISO: '0' (0x00)
MOSI: H (0x48);  MISO: '0' (0x00)
MOSI: e (0x65);  MISO: '0' (0x00)
MOSI: l (0x6C);  MISO: '0' (0x00)
MOSI: l (0x6C);  MISO: '0' (0x00)
MOSI: o (0x6F);  MISO: '0' (0x00)
MOSI: ' ' (0x20);  MISO: '160' (0xA0)
MOSI: '3' (0x03);  MISO: '0' (0x00)
MOSI: ' ' (0x20);  MISO: '160' (0xA0)
MOSI: '3' (0x03);  MISO: '0' (0x00)
MOSI: '16' (0x10);  MISO: '144' (0x90)
MOSI: D (0x44);  MISO: '0' (0x00)
```

Again, match up arrows on the clock and note the data signal (top trace).

First nibble = 0110

Arrow 1 data is low = 0

Arrow 2 data is high = 1

Arrow 3 data is high = 1

Arrow 4 data is low = 0

Second nibble = 0101

Arrow 5 data is low = 0

Arrow 6 data is high = 1

Arrow 7 data is high = 1

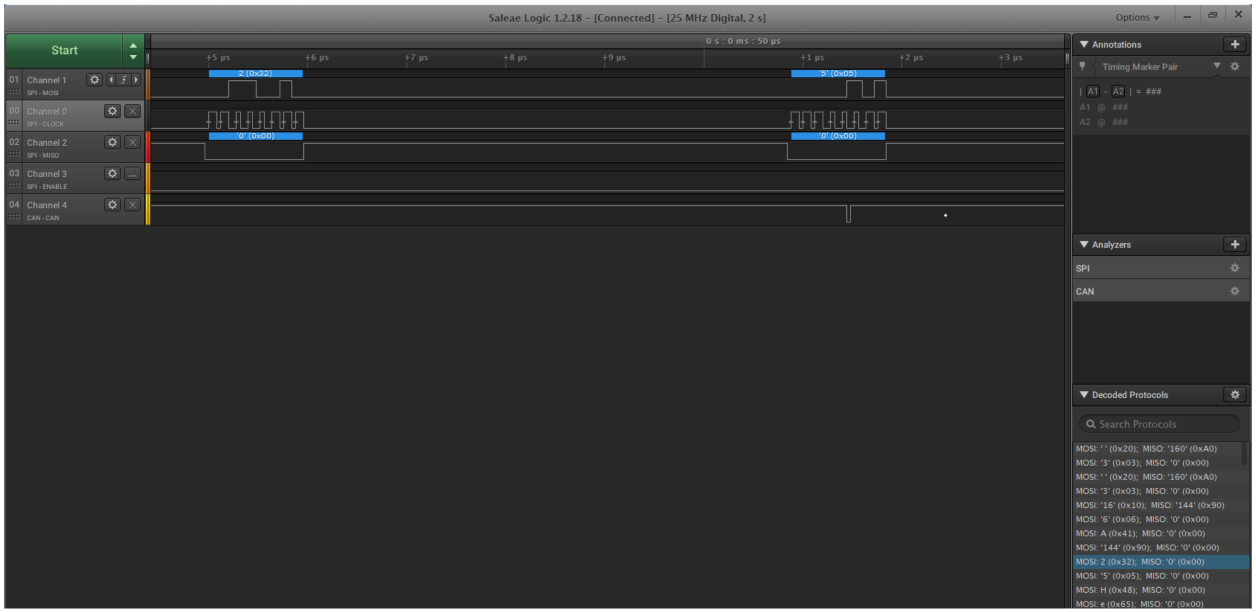
Arrow 8 data is low = 0

So our entire byte = 0110 0110 = 0x65 Hex or 101 decimal or ‘e’ in ASCII

<u>Reading Packet ID and number of Bytes Sent</u>

Now for the last part. Recall that my Arduino sketch used 0x32 as the ID for my sensor. I also told it I was sending five bytes. So, let's look at where that information is. Notice the highlighted packet in the decoded packet window. It shows a value of 0x32. Notice the next packet is 0x05.

The captured image confirms my data is being received in the same sequence I sent it. The packet after the 0x05 is my 'H' (0x48) packet where the message starts. My Arduino code that sends the data is ***myCAN.sendMsgBuf(0x32, 0, 5, myData);*** and we can see that the data is received in exactly the correct order. The ID(32), the #of packets(5), and the actual 5 pieces of data were received without errors.



```
MOSI: ' ' (0x20);  MISO: '160' (0xA0)
MOSI: '3' (0x03);  MISO: '0' (0x00)
MOSI: ' ' (0x20);  MISO: '160' (0xA0)
MOSI: '3' (0x03);  MISO: '0' (0x00)
MOSI: '16' (0x10);  MISO: '144' (0x90)
MOSI: '6' (0x06);  MISO: '0' (0x00)
MOSI: A (0x41);  MISO: '0' (0x00)
MOSI: '144' (0x90);  MISO: '0' (0x00)
MOSI: 2 (0x32);  MISO: '0' (0x00)
MOSI: '5' (0x05);  MISO: '0' (0x00)
MOSI: H (0x48);  MISO: '0' (0x00)
MOSI: e (0x65);  MISO: '0' (0x00)
```

I hope this discussion helps you better visualize the complex communications we all have become accustomed to. I also hope you understand why I get so excited when it actually works!