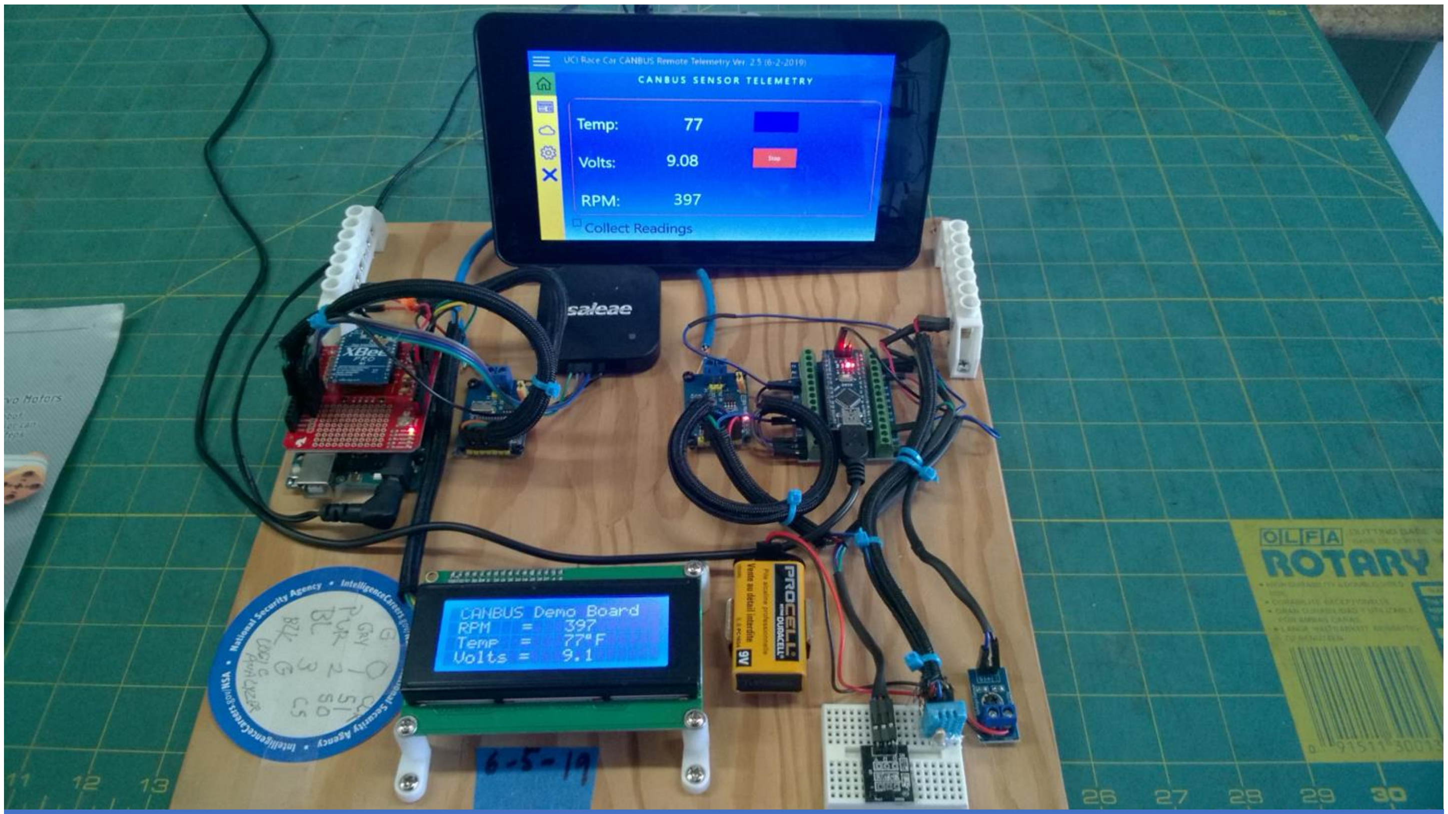


Working with CANBUS Serial Communication Networks Part 2

Bit shifting: How to stream telemetry data & decode packets with the Saleae Logic 8 analyzer

Ronald P. Kessler, Ph.D., MCSE 6/7/19 UCI Race car Engineering

Controller Area Networks (CAN)

Introduction

The CANBUS Demo Board

This paper describes how my CANBUS Demo board is used to transmit sensor data via two Arduino microcontrollers via the CANBUS protocol. You can see the board layout on the cover page. Currently, it is configured to send three pieces of data. I transmit RPM (Hall sensor), temperature (DHT-11), and voltage sensor (Anmbest_MD132). You can see the sensors in the lower right side of the image. A drill is used to activate the Hall sensor. The drill has a magnet attached and when it spins the Hall sensor converts the signal and transmits directly into the Arduino Nano on the right side of the board. This data is then sent to an MCP 2515 CANBUS module. Since the Arduino cannot process CANBUS directly, it uses the SPI (Serial Peripheral Interface) protocol to send data to the first CAN module. The MCP 2515 then transmits my data to a second CANBUS module via CAN protocol at 500 megabits/sec.

I used CAT-5 cable as my CANBUS (Blue cable). The second CAN module then converts the sensor data to SPI so the Arduino UNO can use it. The UNO processes the data and updates the LCD. Then the readings are sent via the onboard XBEE radio to a PC or Raspberry Pi GUI application to complete the wireless communication.

To keep things simple, I will only address the process of transmitting RPM data from the Hall sensor to the NANO and over to the UNO. The logic analyzer is attached to the UNO via the SPI pins. The wiring and capture settings are available at the end of the report.

Formatting Data for Streaming

There are several things you must consider and understand when using serial communication channels with microcontrollers. As you know, one byte can only hold 256 values (0-255 or 2^8) and this severely limits my RPM demo board. I need to be able to transmit RPM in a higher range to handle automotive applications. Therefore, I will need to transmit two bytes of data. A 16-bit unsigned int can hold 65,536 values (2^{16}) and will work perfectly.

In this scenario, let's assume we have an RPM reading of 650. Since 650 cannot be expressed in one byte, I will break the 650 value into two bytes and then transmit one after the other. Arduino© and other serial protocols expect the most significant bit (MSB) to be sent first followed by the least significant bit (LSB). The direction the bytes are packaged up and transmitted is referred to as Endianness. When sending packets via Ethernet, Wi-Fi, Bluetooth, Cell phone, or XBEE radios, the transmitting and receiving devices must know what order the data is being sent.

For example, it doesn't matter whether I send the values 4,67,24,33 starting with the 4 first, or the other way around. I could send the values starting with 33. The point is they must agree, or you will be transmitting gibberish. When we send data with the MSB first, we say we are using BIG Endian. If the system requires the LSB first, you would be using LITTLE Endian.

RPM Encoding/Transmitting

Now let's take a closer look at how I packaged up my sensor data. Specifically, I want to describe how to package raw data into two bytes to accommodate RPM values greater than 255.

1. Assume we have a reading of 650 rpm and want to transmit that to an Arduino/CAN network. Since the largest number we can send in one byte is 255, we must split the number 650 into two bytes. In short, we take 16 bits and break them up into two 8 bit packets.

2. In binary, 650 = 0000 0010 1000 1010

3. So, first we grab the MSB by right shifting 650 by 8 bits. This is like deleting the right 8 bits. All we have left are the 0000 0010 bits. These bits = 2 in decimal.

4. Try it yourself. In your calculator, enter 650. Click the SHR button (Shift Right), then click 8, then press ENTER.
   The result is 2. 0000 0010 1000 1010 >> 8 = 0000 0010 which is 2 in decimal. In coding, ">> 8" means shift right 8 bits. The yellow bits shift right 8 places and push the green bits into the bit bucket, where they are discarded. We are left with only 8 bits which now equal 2 in decimal.

5. Now create the low byte by grabbing it from the original value of 650. Clear your calculator, and type in 650. Then click the AND button, type in 255, press ENTER. The result = 138. Here is how it looks in binary.

6.
```
0010 1000 1010  (650)   original value
     1111 1111  (255)   AND 255 to 650
     1000 1010  (138)   Result = 138
```
   When we use the AND bitwise operator, we essentially copy/grab every bit in our original number that corresponds to a bit in the byte 255. This byte that contains 255 is called a 'Mask'. It is used to select bits we want and ignore bits we don't want. So, we will transmit 2 (0010) as the high byte followed by 138 (1000 1010) as the low byte.
7. Now let's see how Arduino code packages our RPM data into two bytes:

```
myRPM = 650;

//---now split 16 bit int into 2 bytes
uint8_t lowbyte = myRPM & 0xFF;
uint8_t highbyte = myRPM >> 8;

//---assign to my data array for transmitting
myData[0] = highbyte;                    //high byte
myData[1] = lowbyte;                     //low byte
```

   I would use a loop to transmit the array data here.

8. Masks are used in a lot of scenarios. They are used in computer networking. For example, IP address hold two pieces of information. The first tells the computer/phone what the address is of the network it is connected to or wants to connect to. The second piece of data tells our devices the ID number they must use on the network. This type of mask is called a *subnet mask*. Any device that wants to connect to a network (Wi-Fi, Ethernet, Bluetooth, etc.) must have a unique ID number and know the network ID it is connected to. Let's take a closer look.

   Assume my IP address = 192.168.1.2

   Traditional IP addresses are configured as 32 bits arranged in four bytes called octets which are separated by periods. A subnet mask of 255.255.255.0 indicates the first three octets are the network ID and the last octet (0) is used for device Id's. By ANDing the mask value to the IP address, the computer grabs the first three octets and ignores the 4th one. We did the same thing but with only two numbers. These first three octets are the Network ID. The 4th one is used as the unique ID of our device. Since it is only 8 bits, computers on a network can only be numbered from 0-255. That should sound familiar. However, you cannot use 1 or 255 as a device ID. That's a long story!

   Here is how it works for the 1st octet:

   192 = 1100 0000
   255 = <u>1111 1111</u>   Subnet mask for first octet
   AND= 1100 0000 which equals our original value of 192. Another of way of saying this is, "for every bit in this octet where the corresponding mask bit = '1', grab it and make it part of the network ID". The 0 part of the mask means do NOT make these last 8 bits part of the Network ID.

   This is how the O/S determines which network our devices are on and that is used to route our requests to the correct network. So, you can see the AND bitwise operations can be used to extract a precise number of bits from a stream of bits. Let's get back to our CANBUS discussion.

RPM receiving End

Now that we know how to package up our bits into two bytes, let's see how to decode them at the receiving end of our Com link which is the Arduino UNO in this case.

1. To reassemble our two 8-bit bytes back into a 16-bit integer, we will LEFT shift the MSB then OR that result with the value of the LSB. Recall we RIGHT shifted the MSB when we transmitted. So now we left shift to start the recombination of the two bytes.
2. Look at this code snippet from the Arduino UNO sketch I used for my receiver end of the CANBUS:

```
unsigned char buf[2];                   //an array to hold my values
unsigned int receivedRPM = 0;           //must be 16 bits because we broke up the RPM into 2
                                        bytes in the sender app
myCAN.readMsgBuf(&len, buf);                    //read it
receivedRPM = buf[0] << 8 | buf[1];     // the '|' pipe is an OR bitwise logic operation
```

3. Let's break it down. We read the input stream in my buf[2] array. Since the MSB is transmitted first (Big Endian) buf[0] holds that part.

4. Get the MSB by left shifting it 8 bits. We transmitted '2' in the transmitting part. After left shifting we get:
   receivedHighbit = buf[0] << 8 which equals 512.
   In binary it looks like this: 0000 0010 << 8 = 0010 0000 0000 = 512

   Recall that we transmitted 138 as the LSB. Since it is less than 255, we do not have to do any processing with it. We can simply use OR logic to 'glue' the two bytes back into one 16 bit int. The receivedLowbit = buf[1];

   MSB   0010 0000 0000 = 512
   LSB   <u>0000 1000 1010</u> = 138
   OR   0010 1000 1010 = 650 which is our original RPM value! Remember, OR logic says, 'if either bit is True (1) OR if they are both True then the result is True'. It is just a cool way to grab all the bits from both bytes and re-assemble them.

5. In code, the operation looks like this: receivedRPM = buf[0] << 8 | buf[1];

Examining Data Packets with the Logic Analyzer

Now lets examine how packets appear when the RPM is transmitted in two bytes as I described in the previous section. In this instance the CAN-ID of my packets is 0x20 (32 decimal). The RPM being received is 313. So, to examine the actual packets, we must again look at how the RPM data was packaged.

To get the MSB: Use the calculator and do 313 >> 8 = 1.

Again, if you view the binary values it makes more sense.

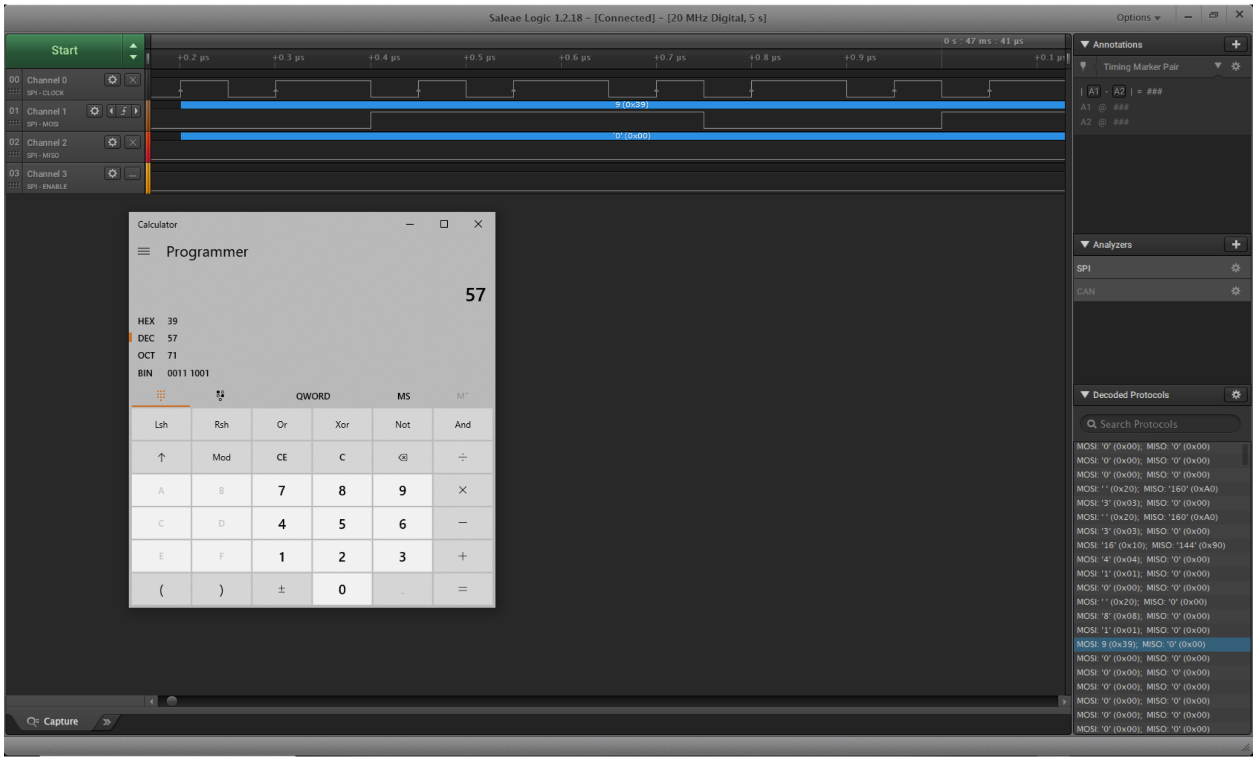
```
            0000 0001      0011 1001 = 313
```

If we right shift these bits 8 positions, we move the left byte over into the position of where the 00111001 bits are and they are discarded. This leaves us with 0000 0001 which = 1. Your calculator will confirm this. So the first byte sent over the CAN is the value 1.

Next, I need to get the value of the LSB which is the 0100 1011 part. To do that we grab those bits as they are by doing an AND bitwise operation as I showed you before.

On the calculator, type in 313 again. Click the AND button. Type 255 then enter. The result is 57. Notice in the calculator 57 decimal = 39 in hex (0x39). I have left the calculator on screen to show you that in the image below.

This screen shot from my Saleae Logic 8 analyzer.The highlighted packet on the lower right of the image shows the decoded packet is 0x39 (57 decimal) and the packet above is 0x01 or 1 in decimal. Those two packets contain our RPM value of 313! The MSB is in the 0x01 packet and the LSB is in the 0x39 packet. They both have different values because we moved the bits into different positions in order to break up a large number into two separate bytes. The MSB was transmitted first. That is why it shows up above



```
57
HEX 39
DEC 57
OCT 71
BIN 0011 1001
```

```
MOSI: '0' (0x00);  MISO: '0' (0x00)
MOSI: '0' (0x00);  MISO: '0' (0x00)
MOSI: '0' (0x00);  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: '4' (0x04);  MISO: '0' (0x00)
MOSI: '1' (0x01);  MISO: '0' (0x00)
MOSI: '0' (0x00);  MISO: '0' (0x00)
MOSI: ' ' (0x20);  MISO: '0' (0x00)
MOSI: '8' (0x08);  MISO: '0' (0x00)
MOSI: '1' (0x01);  MISO: '0' (0x00)
MOSI: 9 (0x39);  MISO: '0' (0x00)
MOSI: '0' (0x00);  MISO: '0' (0x00)
MOSI: '0' (0x00);  MISO: '0' (0x00)
MOSI: '0' (0x00);  MISO: '0' (0x00)
MOSI: '0' (0x00);  MISO: '0' (0x00)
MOSI: '0' (0x00);  MISO: '0' (0x00)
MOSI: '0' (0x00);  MISO: '0' (0x00)
```

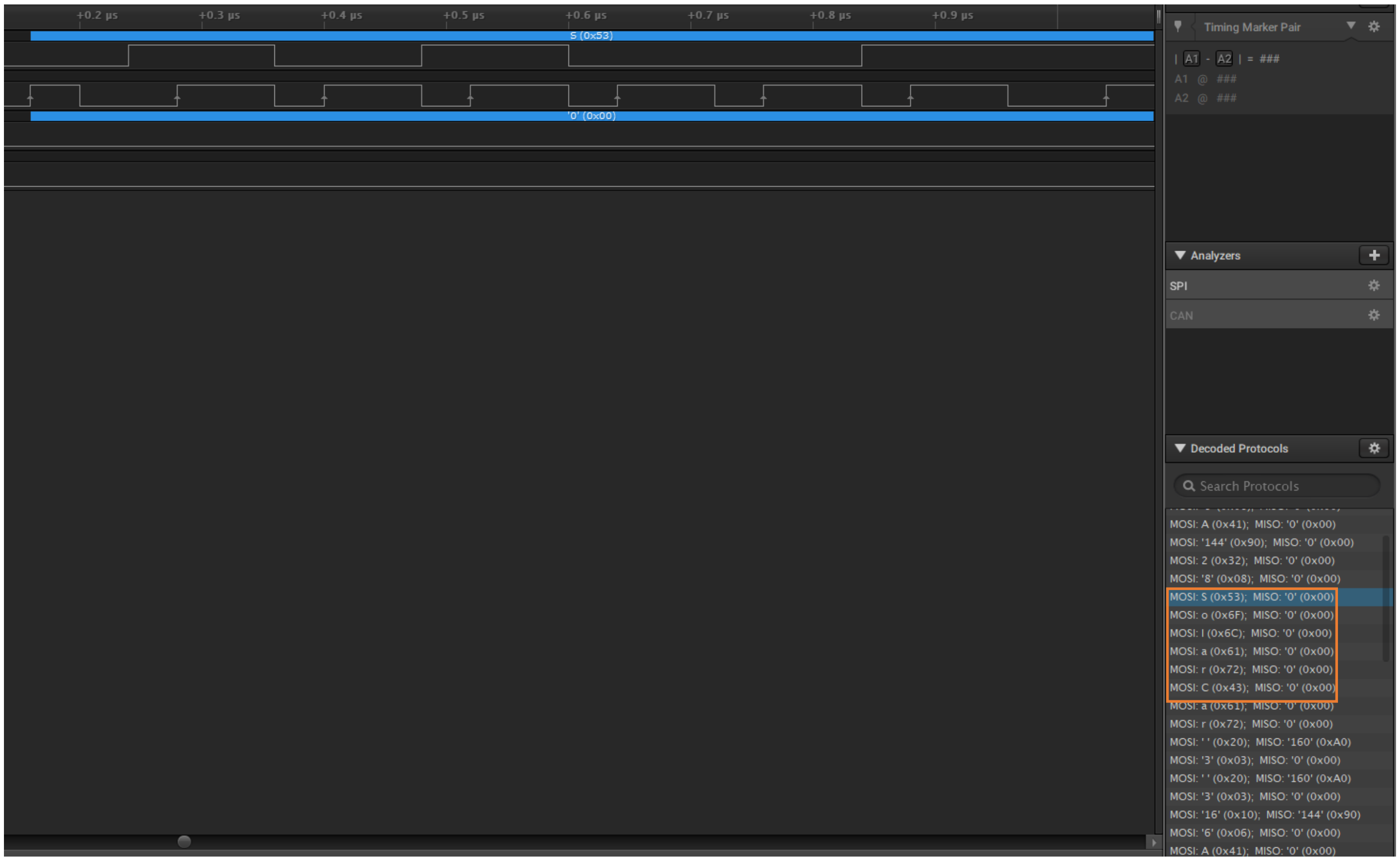
the LSB (0x39) packect. You can see the remaining data packets contain 0 because I did not transmit any more data. Although I could have made the remaining bits hold temperature or battery voltage if I needed to.

The first few times you do this kind of programming it will be easy to get lost. Try to remember that all we want to do is break up a 16 bit number into two parts. When we move bits around into different positions, they have a different value. And that is what can get you off track. That is fine. Because all we have to do is shift them back into a 16 bit number and we have our original value back.

So let's review how we *do* get our original RPM value of 313 to display on our screen. Recall we sent the values 1 and 57 in our two bytes. Here is how we recombine them.

1. On your calculator type in 1 then click the LSH button (Left shift), type 8, and hit enter. This gives 256. Leave the result alone on the calculator.
2. Now, click the OR button, type in 57, then press enter. This results is 313 which is our original data.
3. Again, in code you can use receivedRPM = buf[0] << 8 | buf[1];

For fun, I showed my solar race car class what a message looks like via CANBUS. If you expand this page you will notice the highlighted packet begins with a value of ‘S’ followed by ‘o’, ‘l’,’a’,’r’,’C’,’a’,’r’. Instead of sending RPM data, I sent a text message by packaging each letter as a byte in my array using the hex value of each letter that was sent over the wire.



```
MOSI: A (0x41);  MISO: '0' (0x00)
MOSI: '144' (0x90);  MISO: '0' (0x00)
MOSI: 2 (0x32);  MISO: '0' (0x00)
MOSI: '8' (0x08);  MISO: '0' (0x00)
MOSI: S (0x53);  MISO: '0' (0x00)
MOSI: o (0x6F);  MISO: '0' (0x00)
MOSI: l (0x6C);  MISO: '0' (0x00)
MOSI: a (0x61);  MISO: '0' (0x00)
MOSI: r (0x72);  MISO: '0' (0x00)
MOSI: C (0x43);  MISO: '0' (0x00)
MOSI: a (0x61);  MISO: '0' (0x00)
MOSI: r (0x72);  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: '6' (0x06);  MISO: '0' (0x00)
MOSI: A (0x41);  MISO: '0' (0x00)
```

Conclusion

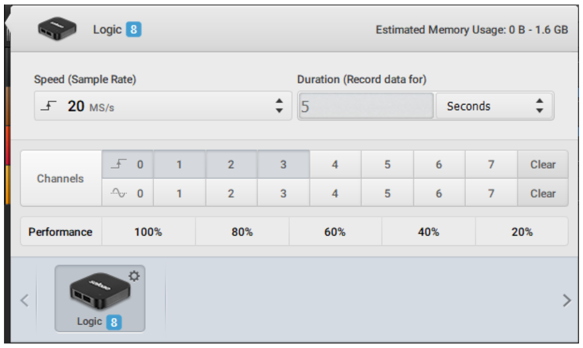
By looking at the actual packets being transmitted, you can see very clearly how our data is streamed on serial communication lines. Just remember, you must understand how each network protocol is designed so you can package up your data in the correct format. And please don't forget about Endianess. Not all protocols use Big Endian like Arduino does. Also, there are other ways to do this bit manipulation. In fact, the Arduino library has a highbyte() and lowbyte() functions that do all this for you. It is fine to use them but it is really important to understand how this all works first.

Packet Capturing and Analyzer Setup Details

For completeness, I have included some information that may assist you in gathering CANBUS packets like I did. I used a *Saleae Logic 8* analyzer and chose the SPI decoder protocol. The sample rates are shown in this image.

I collected RPM data on my demo board that I described in the beginning. When I stopped the drill motor, the LCD displayed 313. I left everything running and then started the capture. The Arduino circuit keeps tranmitting the last value so it was an easy way to verify my data was being received.

The packets were captured from the SPI ports on the Arduino UNO since it was the receiving end of the network as shown on the next page.



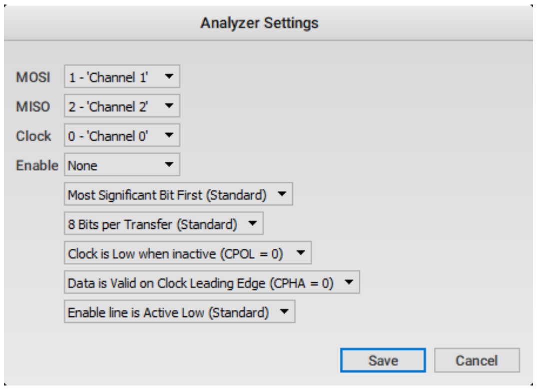
Logic 8 — Estimated Memory Usage: 0 B - 1.6 GB

Speed (Sample Rate): 20 MS/s

Duration (Record data for): 5 Seconds

Channels: 0 1 2 3 4 5 6 7 Clear

Performance: 100% 80% 60% 40% 20%



Analyzer Settings

MOSI: 1 - 'Channel 1'

MISO: 2 - 'Channel 2'

Clock: 0 - 'Channel 0'

Enable: None

Most Significant Bit First (Standard)

8 Bits per Transfer (Standard)

Clock is Low when inactive (CPOL = 0)

Data is Valid on Clock Leading Edge (CPHA = 0)

Enable line is Active Low (Standard)

Here is a closer look at the SPI Arduino connections for the Logic analyzer.

Saleae Pins	SPI	Arduino Pins
Green = 0	| SPI Clock	13
Gray = 1	| SI	11
Purple = 2	| SO	12
Blue = 3	| CS	10
Black = Ground		GND