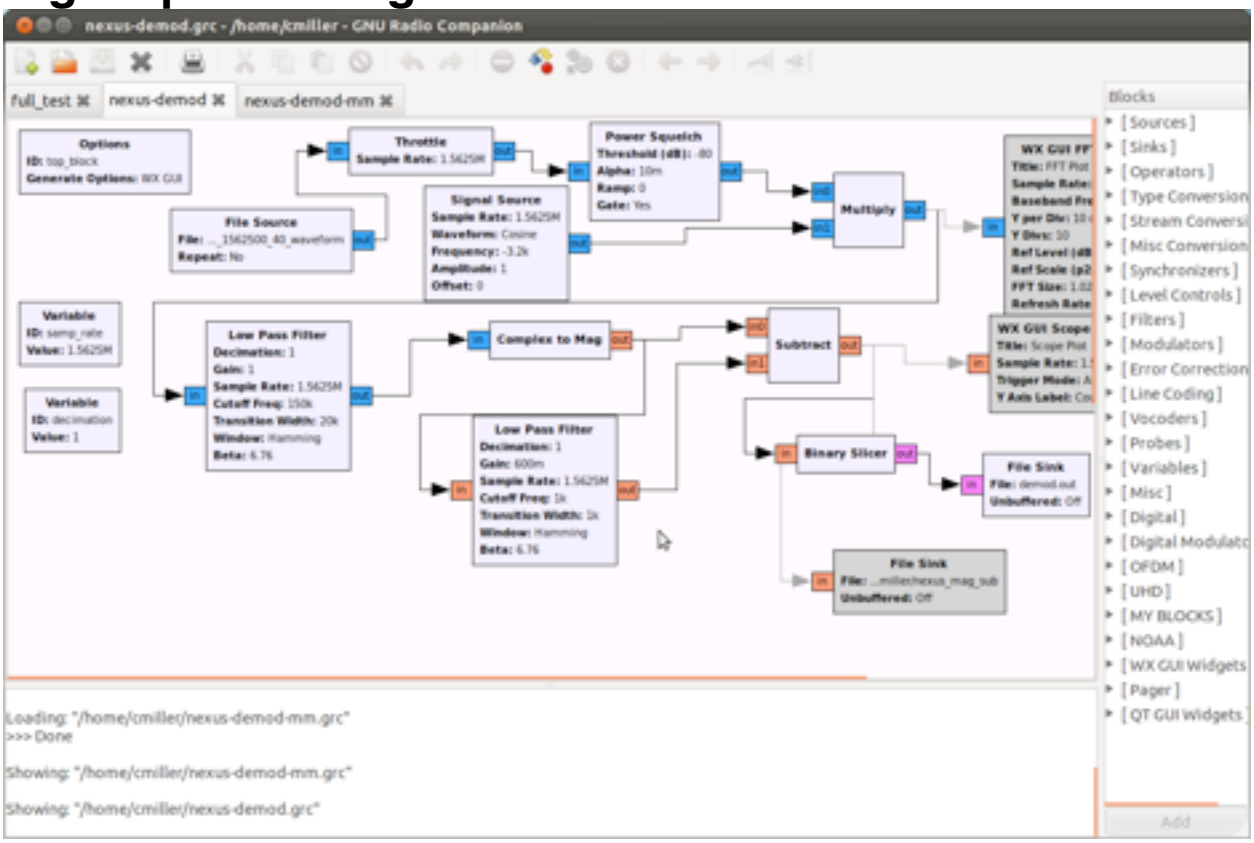
NFC Update 2

Thanks in large part to help from Michael Ossmann, we can sort of read some actual data from the captured signal.

Signal processing



The first thing that we do is a Power Squelsh. I think this eliminates any signal below a certain threshold. Next we reduce the frequency of the signal by 3200Hz. (This value can be obtained by looking at the carrier frequency on a much zoomed in baudline graph of the raw data - use process->transform size). I think this is the amount that the frequency used by the Nexus S differs from the standard 13.56MhZ.

Then we send the signal through a low pass filter of 150k. Then we take the magnitude of the complex waveform. This output should be 0 or above. Then we want to make it where it crosses 0 when it is low, to do this we want to subtract something like the average of the new waveform. For this, we subtract away a low pass filter of the signal at 1k with a gain of 600m. Finally, we send this new output to a binary slicer to get a series of (oversampled) 0's and 1's.

More on this in a bit.

Demodulation by hand

One of the keys is looking at the right time scale. Now that I've seen the data, we see that the Nexus S is using the slowest of the possible data rates, 106kbps which means it must use 100% ASK. 106kbps means that a single bit of data should be transmitted in about 1/106k = .0000094s = 0.01ms. So bytes of data should take on the order of 0.1ms.

You can see around .25ms in Figure 1. That means there could be up to 25 bits of data in this picture. The data from Figure 1 can be digitized to the following if you use a binary slicer (1 if signal is above 0, 0 if signal is below 0) with perfect clock.

```
. . . 1 1 1 1 1 0 1 0 1 1 0 1 0 1 1 0 1 1 0 1 1 0 1 1 1 1 . . .
```

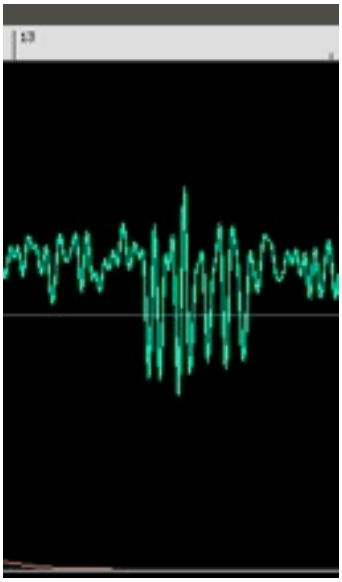


Figure 1: "26"

The decoding follows a Manchester decoding scheme:
`01` -> start
`011` -> end
`10` -> 1
`11` -> 0
(after start or 0) `01` -> 0

Therefore this digitized sequence contains the data bits:

```
. . . 1 1 1 1 1 0 1 0 1 1 0 1 0 1 1 0 1 1 0 1 1 0 1 1 1 1 . . .
                s   0   1   1   0   0   1   0   e
```

so the data is 0110010 and since lsb is first, this becomes the binary 0100110 = 0x26 which is SENS_REQ according to the spec.

Other examples include those like Figure 2, which decodes to 0x93 0x20.

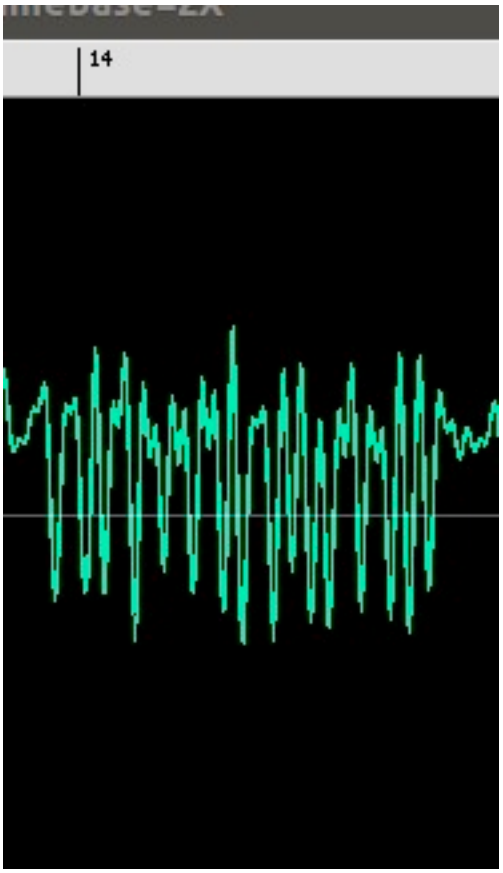


Figure 2, “93,20”

and even longer ones like Figure 3.

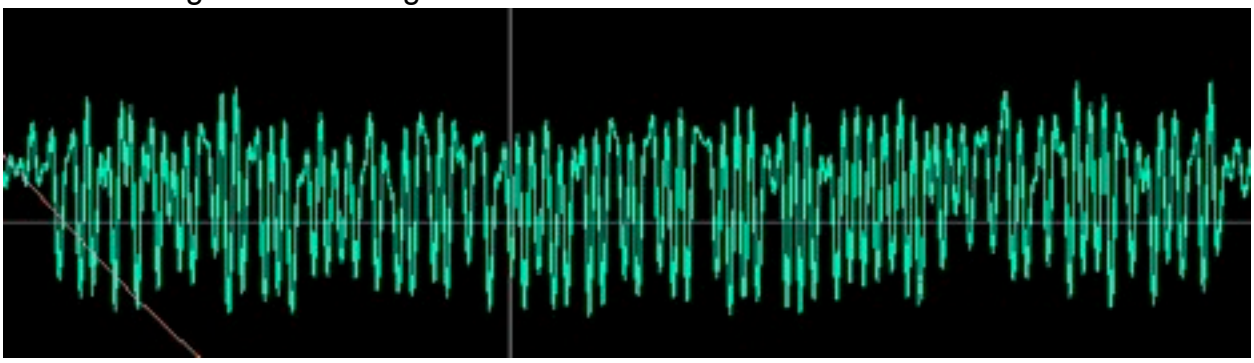


Figure 3, a longer frame correctly decoded to “93 70 88 04 e3 ef 80 99 73”.

To deal with frames longer than 7 bits, the data is packed as follows:

<start> <8 data bits> <1 parity bit> <8 data bits> <1 parity bit> .... <8 data bits> <1 parity bit> <end>

Here is another one worked out

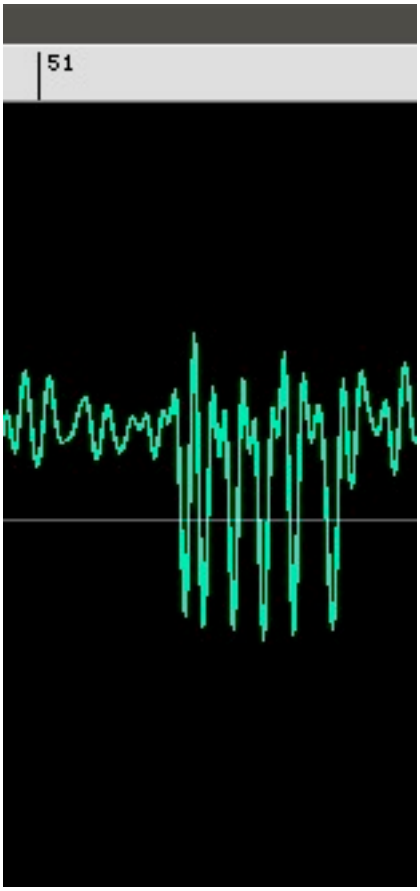


Figure 4: “52”

which looks like:

```
1 0 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 1 1
  s   0   1   0   0   1   0   1   e
```

This is then 1010010 = 0x52 which is ALL_REQ.

However, some don’t work quite right, see Figure 5.

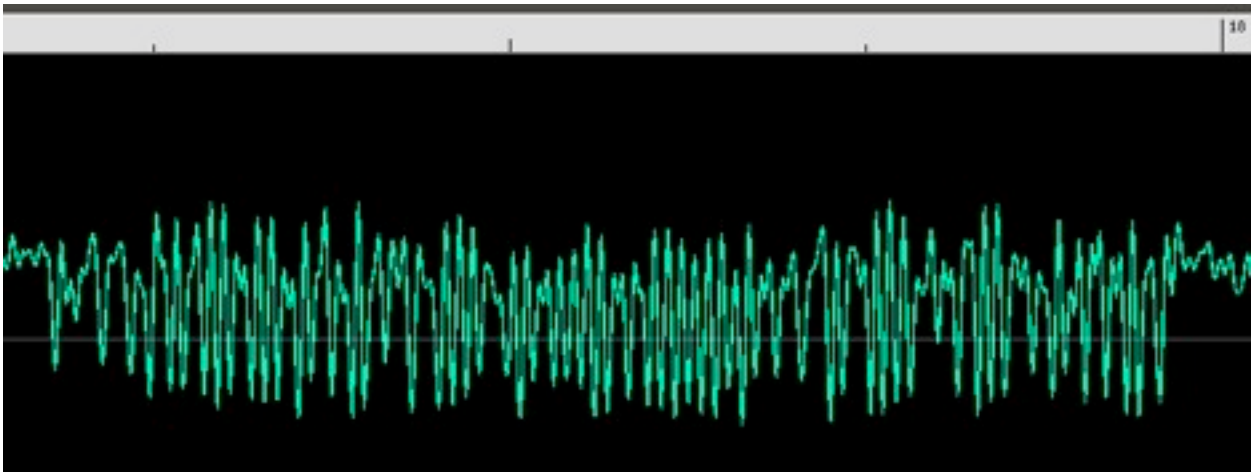


Figure 5: The signal cannot be properly demodulated, presumably due to noise.

Right off the bat, there is a long period of 1's. This would probably digitize as

```
1 0 1 1 1 1 ...
```

which would be an error since after the start "01" there must be either "10" or "01", there cannot be "11".

Demodulating automatically

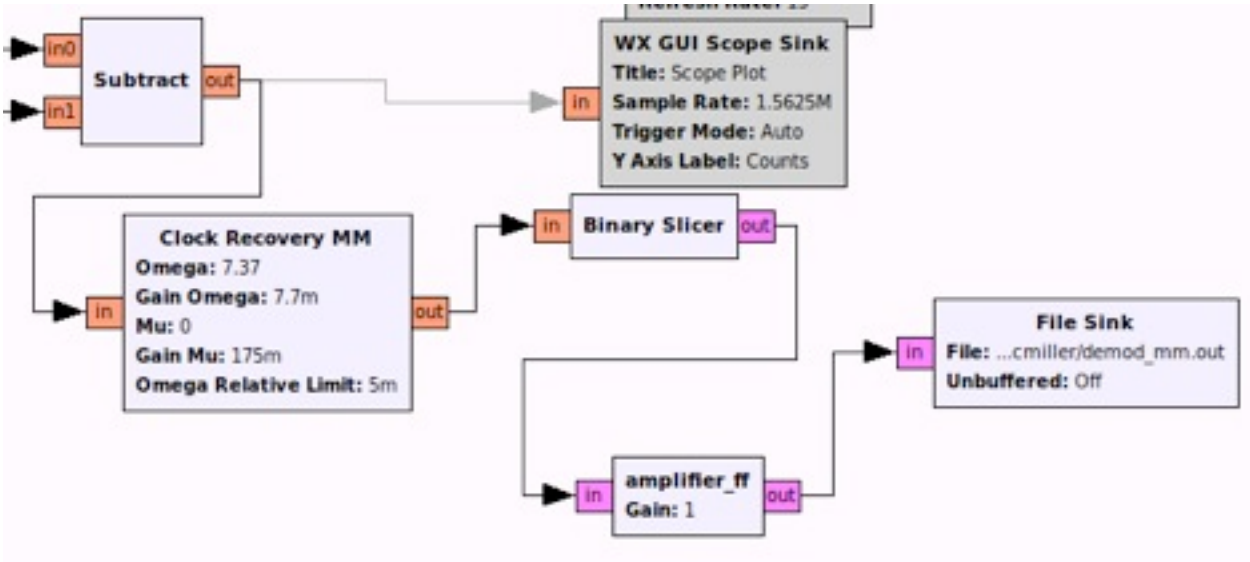
I have a script that will take the output of the binary slicer described in the data processing section and produce decoded data called my-nexus-decode.py. It works pretty well can can output the data streams of all the examples in this doc and more. There are a couple of problems with it. First, it works on oversampled data which is what I have right now. The data I am getting right now is about 7-8 samples per actual "half-bit" of data. (I say half bit because of the way the decoding works, 2 digitized "half bits" become one actual bit of data). My program can deal with this and it actually makes things a little easier because it eliminates some noise since if you don't have 7-8 0's in a row, it is not the beginning of data. This also allows you to deal with clock synchronization since some sequences of bits will be 6, some 7 and some 8 bits long. The other problem (partly because of the oversampling) is the script is really slow. It takes 20 seconds to process 300ms of data. Not exactly real time!

The 7-8 samples per half-bit of data makes sense if you consider that the phone is supposed to be transferring data at 106kbps but that means the actual 0's and 1's (half-bits) need to be coming at a rate of 212kbps. Since we are capturing at 1562500 samples/second that means that we get 1562500/212000 = 7.37 samples per half-bit of data.

Ideally we'd like to get this to 1 sample per half-bit of data so we could eliminate the costly oversampling.

Clock recovery

For clock recovery, I used the Mueller and Miller (M&M) based clock recovery block. This attempts to take the signal and desample it down to 1 sample per bit based on the length of data spikes it sees. It works reasonably well. It is not as accurate as the script I wrote which above which deals with oversampling, but it is orders of magnitude faster. For example, of the first 4 symbols, it decodes 3 correctly and the one it got wrong it only missed 2 half-bits out of 42.



Additionally, I wrote a Gnu Radio Companion block to do the Manchester decoding and print it to the screen which is called amplifier_ff (since I was too lazy to rename the sample code I based it on). The output looks something like:

```
26
93 70 88 04 e3 ef 80 99 73
95 20
71
51 77 10 40 f6 b9
50 00 57 cd
52
```

It is pretty fast, but still not real-time. It takes about 4 seconds to analyze a 2 second recorded sample. I've tried it on three different samples and it works equally well on all three. It doesn't seem to work on a "live" signal, which is problematic.

Moving forward:

Is non-live signal caused by too high sample rate? Will it work with lower sample rate?

Optional:
Clean up signal by adjusting low pass filter?

Optimize the placement of “0”, i.e. the gain value of .6 in subtracted low pass filter. (Not sure this really matters that much)