Blind Reverse Engineering a Wireless Protocol
Or
Adventures in Amateur Signals Intelligence

Introduction

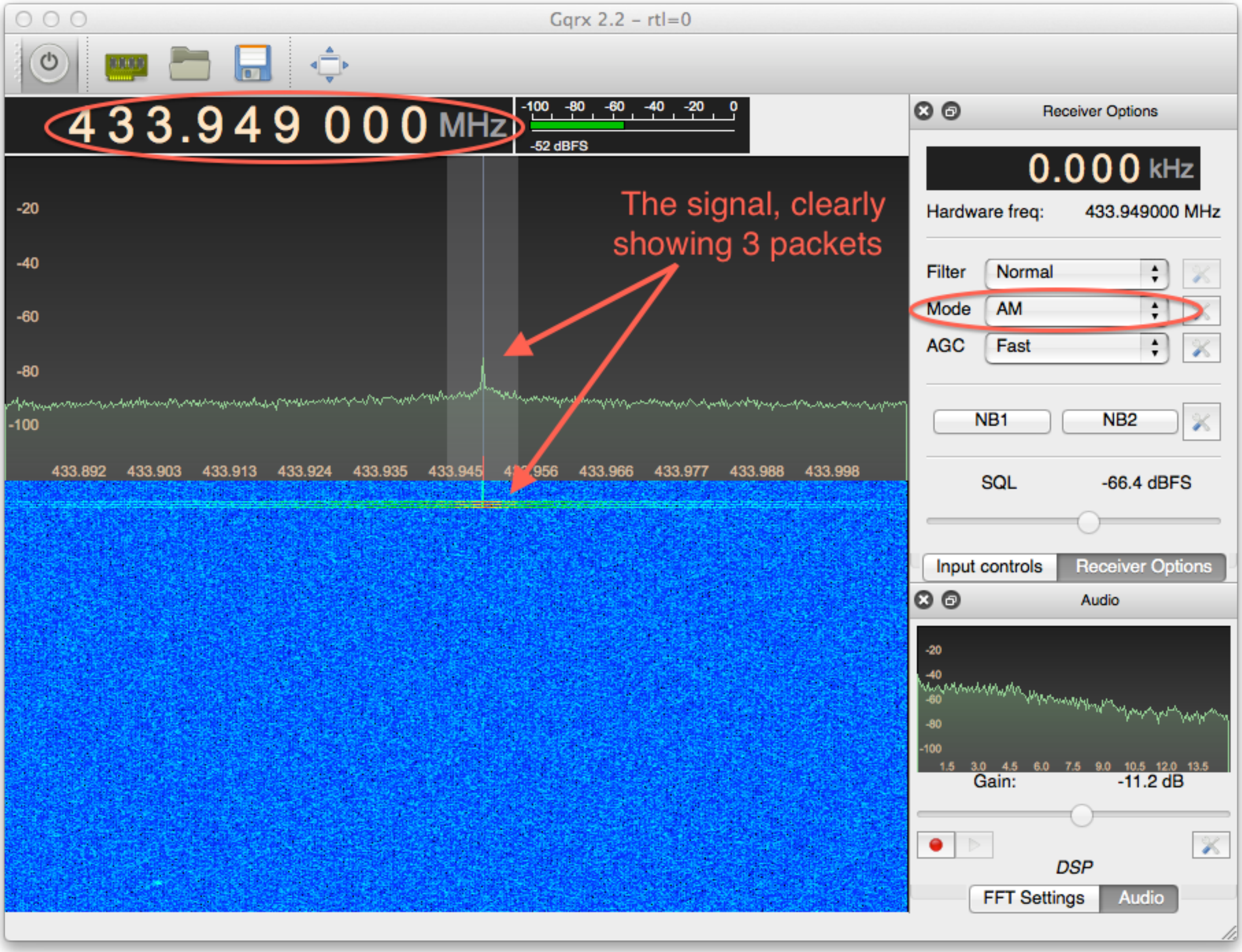
This is a document describing the steps taken in an unusual case of black-box reverse engineering. In it I describe the detection, recording, and decoding of an entirely unknown digital radio signal. However, I also include my many errors, side-tracks, and total screw-ups. One thing I learned on this project is that when you are reverse engineering, you have to be wrong a thousand times before you're finally right. The project began in the field of software defined radio, and ended in the field of protocol reverse engineering with the intention to integrate the received data into a decoding application. This was a very interesting exercise (to me at least) and I discovered that not much has been written on the subject for non-specialists. I hope this will serve to inform and assist others with similar obscure technical investigations.

It all started while I was reading up on interesting projects that could be done with my new software defined radio. I ran across www.windytan.com and was inspired by several hacks performed by Oona Räisänen of Helsinki, Finland. Her work in decoding DARC transmissions from Helsinki metropolitan busses and a sideband data stream from a news helicopter were very impressive. The idea of decoding digital transmissions captured my imagination, so I started trolling around the RF spectrum looking for anything of interest.

Because there's an awful lot of spectrum to investigate, I needed to narrow down the field a bit. I read up on the FCC spectrum allocation and decided that any of the ISM (Industrial, Scientific, Medical) bands would be good places to search, so I kept to those mostly. I eventually found a transmission that repeated every 43 seconds at 433.949 Mhz. The signal was strong and consistent, the perfect target for analysis. Decoding this signal became my weird little personal CTF challenge.

Capturing the Signal

Capturing the signal was almost trivially simple once its frequency was known. I already knew the frequency I wanted to capture, and the default bandwidth setting in my SDR software seemed fine. I listened to the signal using AM and FM and decided that AM was the correct modulation method. When I listened to it using FM demodulation it sounded like loud static. When I listened to it with AM demodulation it sounded like a fast stuttering pattern. In other words, it sounded digital. Recording the audio of these transmissions was as easy as pressing record. GQRX (the SDR software I use) is kind enough to pipe sound output to WAV files for your convenience.



The signal, clearly showing 3 packets

Hardware freq: 433.949000 MHz

Filter: Normal

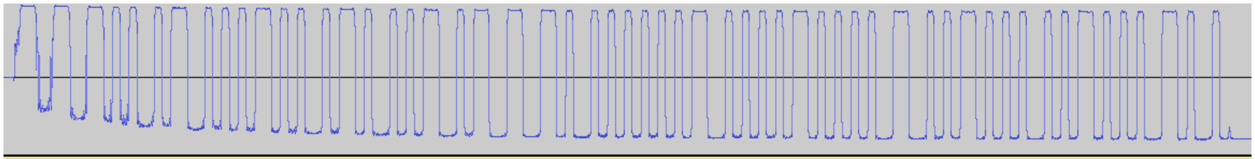
Mode: AM

AGC: Fast

SQL -66.4 dBFS

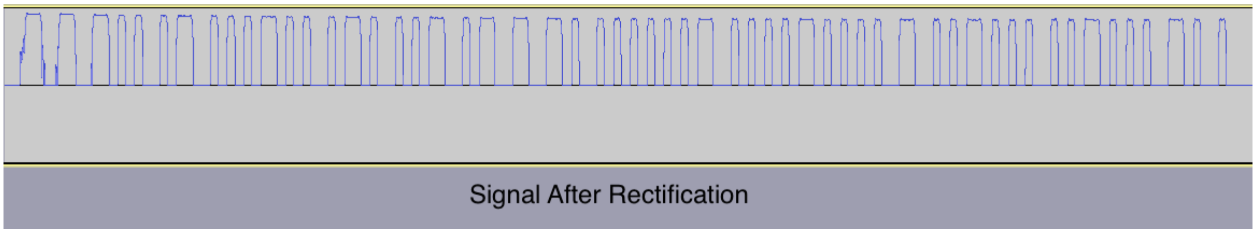
Gain: -11.2 dB

The next step was to take a look at the recordings. Simply opening the files using Audacity and selecting “Waveform (dB)” from the track’s drop-down menu displayed the obviously digital data. Luckily for me, this meant that the signal was using one of the simplest digital modulation methods (i.e. how a digital signal gets converted to an analog one). This simple method is called Amplitude Shift Keying and it’s no more complicated than a digital high is transmitted with one power level and a digital low is transmitted at a slightly lower power level. There are many other digital modulation methods that could have been used which would have required me to work out a more complicated demodulation solution. However, it was still “analog” in that it had noise on every rising and falling edge and it was full wave, meaning that it rises above “zero” and falls below “zero”.



Signal Before Processing

This is obnoxious and makes reading the rises and falls of the signal very hard visually. I found a third-party Audacity plugin called Rectivert that allowed me to solve the full wave problem by simply deleting everything below “zero”. Then I used the standard Audacity “hard limiter” effect to cut off everything above a decibel level of my choosing. After that I had a clear, easy to follow digital signal.



Signal After Rectification

Decoding

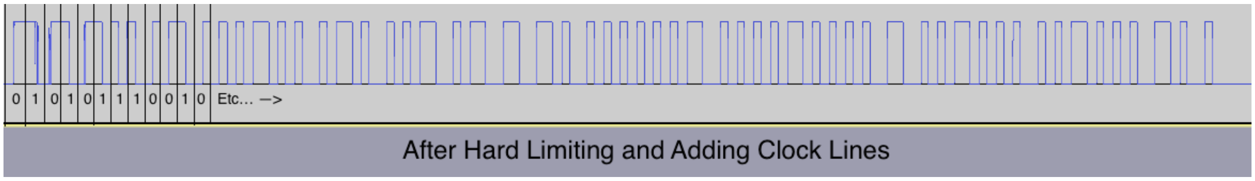
Now that the easy stuff was all done, I got to start in on the actual grunt work. First, I needed to know how the digital signal converted to 0’s and 1’s. This is less obvious than it sounds. If you thought that a high signal always equaled 1 and a low signal always equaled 0 in all cases, then you are probably a victim of oversimplified tutorials, just like I was. It is significantly more complicated than this because there are all sorts of difficulties in getting the receiving machine to correctly interpret what the sending machine meant. One such difficulty is that the signal being sent needs to encode within its shape the speed of data transmission. A gross example of this is if computer A sends two bytes, “1100 0101” at a rate of 1 bit per millisecond and computer B is reading the data at a rate of two bits per millisecond then the result would be “1111 0000 0011 0011”. There are several other factors that come into play here as well, but the point is that engineers have devised several different methods of converting digital 0’s and 1’s into various combinations of high and low signals with specific rules about when and how the combinations are done. Each of these various methods is called a line code.

I made my first time-consuming, but ultimately unhelpful side journey at this point. I didn’t know anything about line codes except that they exist and I blindly assumed that there would be some program that would input a variable signal from an audio file (or something) and decode various line codes. If not that, I at least thought I could find a program that would tell me what line code I was looking at. After much fruitless research I decided that GNURadio-Companion might be my solution. GRC is a very powerful tool that has “blocks” of radio components. This is the “software” part of software defined radio. The blocks are extremely low level and you can put them together to build, in software, just about anything you can buy in hardware. The hours I put into learning GRC taught me a heck of a lot about radio theory, but didn’t help me with my immediate problem. At the time of this writing GRC doesn’t appear to have blocks that deal with line codes. I briefly toyed with the idea of learning Python and writing my own blocks, but then I returned to reality.

After a bit more study I realized that most line codes can be distinguished from each other visually. With this new familiarity I was able to determine that the line code in use was called “Manchester”. Or so I thought. In actual fact, it is not Manchester at all, but that didn’t stop me from “decoding” dozens of messages and pondering over them. Oddly enough, the line coding that was actually in use produced data that was similar enough that I could still derive some useful data while using the wrong line code. Most of the data that followed was correct, but occasional strange results kept me suspicious of the it. It took quite a while before the real method became clear.

Manchester line coding is fairly common, and is interesting in that the 0’s and 1’s are encoded in the *change* of state of the signal rather than the state of the signal. The change from high to low is one possible digit and the change from low to high is the other. Unfortunately, there’s two different interpretations of the manchester line code so, depending on the opinions of the people who designed the equipment, all the 0’s might be 1’s or vice versa. I just picked one standard and ran with it, keeping in mind that I might have picked the wrong one. I decided to interpret all transitions from low to high as a 0, and all transitions from high to low as a 1.

Converting each captured message into binary was done visually. There simply wasn’t a program around that I could find to do it for me. This was time consuming, but I eventually got pretty fast at it. I simply took a screen shot of each recording, zoomed into the signal, imported that into a image editor, drew vertical lines to break up transitions, and typed a 0 or a 1 respectively into a spreadsheet. After all of the preceding work, I finally had binary. And no idea whatsoever what it meant.



After Hard Limiting and Adding Clock Lines

Amateur Signals Intelligence

The general idea behind “black-box” testing is that you do not examine the device or program itself, just its inputs and outputs. However, implicit in the idea of black-box testing is the knowledge that you actually have access to the black box! You can turn it on and off, alter its inputs, change its location or other environmental factors, all while measuring its outputs and correlating the changes you are introducing to the changes you find in your measurements. Search the internet and you’ll find dozens of people who have reverse engineered the signals from a key fob, or a wireless sensor, or (amusingly) their bathroom scale. The difference is, they had the object and could manipulate it. I did not know what device was doing the transmitting, did not have access to it, and could not cause any changes to its inputs or environment. Essentially, I was engaged in signals intelligence, a practice not commonly found outside of the military and clandestine services communities.

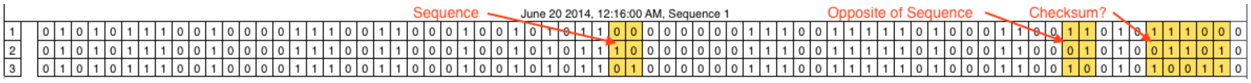
Metadata

With this in mind, all that I thought that I could do was collect data and compare it to other data hoping to find enough correlations to determine the purpose and meaning of part or all of the binary data. Early on in the process I realized that I had more useful information than just the binary. I had data *about* the data. Most specifically, the time, down to the second of each transmission. This became useful enough that I ended up throwing out all of my collected data from before I started keeping track of the time and date of each transmission. That was a lot of wasted work, but trying to correlate data with differing levels of value and accuracy was asking for trouble. I also noted every other clue I had about the transmission even if they were obvious. Examples include 1) Frequency, 2) Bandwidth, 3) The signal was very strong and thus the transmitter was probably very close, 4) The transmissions occurred exactly every 43 seconds, 5) Each transmission consisted of 3 separate, but very similar packets, 6) Each packets was exactly the same length, with similar content, 7) There appeared to be only one device doing any communicating, so either it was simply broadcasting, or any device responding was farther away from me and I wasn't receiving that signal.

Correlation

The methods I used to correlate the data evolved over time, but followed the basic pattern of throwing everything into a spreadsheet organized by transmission and packet (remember, 3 packets per transmission) and highlighting those columns with bits that changed and then looking for patterns in the changes. When looking for changes, I could look for changes over time (changes that spanned various different messages) or for changes within a single message (changes that occurred only within the three packets of one message). It was also important to keep in mind that a correlation did not necessarily mean a causal relationship. Here is a sample message highlighting the three fields that did not remain consistent from one packet to the next.

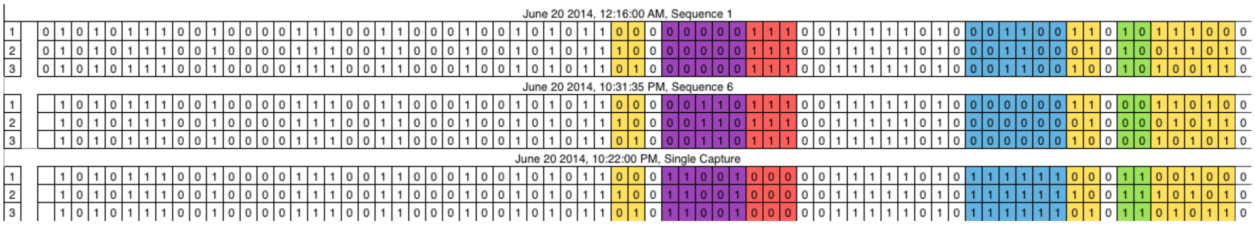
The first pair on the left look to me like a counter and the second pair is clearly the compliment (reverse) of the first pair. There could be a relationship there. The third set of 5 bits doesn't have any immediately obvious relationship to anything else at the moment.


Sequence
Opposite of Sequence
Checksum?

June 20 2014, 12:16:00 AM, Sequence 1

1	0	1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	0	0	0	0	0	0	0	0	1	1	1	0	0	1	1	1	1	1	0	1	0	0	0	1	1	0	0	1	1	0	1	0	1	1	1	0	0	0
2	0	1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	1	0	0	0	0	0	0	0	1	1	1	0	0	1	1	1	1	1	0	1	0	0	0	1	1	0	0	0	1	0	1	0	0	1	1	0	1	0
3	0	1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	0	1	0	0	0	0	0	0	1	1	1	0	0	1	1	1	1	1	0	1	0	0	0	1	1	0	0	1	0	0	1	0	1	0	0	1	1	0

By adding more data I revised and improved my understanding of the protocol and it's variables. Here's a sample of three packets after adding a significant number of additional messages so as to find more variables. I'm skipping messages that were essentially the same so as to make the differences more obvious. I colored those bits similarly that I believed had a clear relationship.



June 20 2014, 12:16:00 AM, Sequence 1

1	0	1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	0	0	0	0	0	0	0	0	1	1	1	0	0	1	1	1	1	1	0	1	0	0	0	1	1	0	0	1	1	0	1	0	1	1	1	0	0	0
2	0	1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	1	0	0	0	0	0	0	0	1	1	1	0	0	1	1	1	1	1	0	1	0	0	0	1	1	0	0	0	1	0	1	0	0	1	1	0	1	0
3	0	1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	0	1	0	0	0	0	0	0	1	1	1	0	0	1	1	1	1	1	0	1	0	0	0	1	1	0	0	1	0	0	1	0	1	0	0	1	1	0

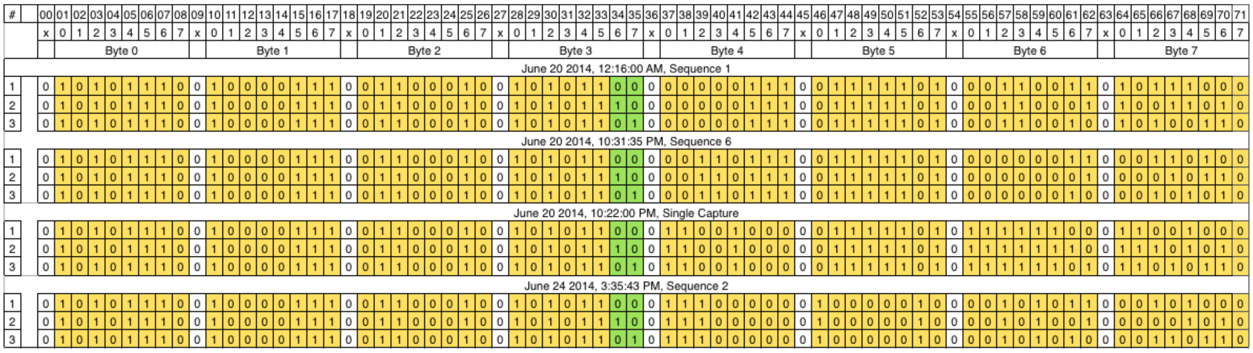
June 20 2014, 10:31:35 PM, Sequence 6

1		1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	0	0	0	0	0	1	1	0	1	1	1	0	0	1	1	1	1	1	0	1	0	0	0	0	0	0	0	1	1	0	0	0	1	1	0	1	0	0
2		1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	1	0	0	0	0	1	1	0	1	1	1	0	0	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0
3		1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	0	1	0	0	0	1	1	0	1	1	1	0	0	1	1	1	1	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	1	0

June 20 2014, 10:22:00 PM, Single Capture

1		1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	0	0	0	1	1	0	0	1	0	0	0	0	0	1	1	1	1	1	0	1	0	1	1	1	1	1	1	0	0	0	1	1	0	0	1	0	0	0
2		1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	1	0	0	1	1	0	0	1	0	0	0	0	0	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	0	0	1	1	1	0	1	0	1	0
3		1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	0	1	0	1	1	0	0	1	0	0	0	0	0	1	1	1	1	1	0	1	0	1	1	1	1	1	1	0	1	0	1	1	0	1	0	1	1	0

By continuing to add more packets I eventually detected the pattern of bytes. The basic pattern of a single packet was a start bit (0) followed by a single byte for a 9-bit word with 8 words per packet, making a total of 72 bits per packet. Note: This is false, but I didn't figure that out until I switched over to the correct line coding technique.



#		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71
		x	0	1	2	3	4	5	6	7	x	0	1	2	3	4	5	6	7	x	0	1	2	3	4	5	6	7	x	0	1	2	3	4	5	6	7	x	0	1	2	3	4	5	6	7	x	0	1	2	3	4	5	6	7	x	0	1	2	3	4	5	6	7	x	0	1	2	3	4	5	6	7
			Byte 0									Byte 1									Byte 2									Byte 3									Byte 4									Byte 5									Byte 6									Byte 7							
June 20 2014, 12:16:00 AM, Sequence 1																																																																									
1		0	1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	0	0	0	0	0	0	0	0	1	1	1	0	0	1	1	1	1	1	0	1	0	0	0	1	1	0	0	1	1	0	1	0	1	1	1	0	0	0
2		0	1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	1	0	0	0	0	0	0	0	1	1	1	0	0	1	1	1	1	1	0	1	0	0	0	1	1	0	0	0	1	0	1	0	0	1	1	0	1	0
3		0	1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	0	1	0	0	0	0	0	0	1	1	1	0	0	1	1	1	1	1	0	1	0	0	0	1	1	0	0	1	0	0	1	0	1	0	0	1	1	0
June 20 2014, 10:31:35 PM, Sequence 6																																																																									
1		0	1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	0	0	0	0	0	1	1	0	1	1	1	0	0	1	1	1	1	1	0	1	0	0	0	0	0	0	0	1	1	0	0	0	1	1	0	1	0	0
2		0	1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	1	0	0	0	0	1	1	0	1	1	1	0	0	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0
3		0	1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	0	1	0	0	0	1	1	0	1	1	1	0	0	1	1	1	1	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	1	0
June 20 2014, 10:22:00 PM, Single Capture																																																																									
1		0	1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	0	0	0	1	1	0	0	1	0	0	0	0	0	1	1	1	1	1	0	1	0	1	1	1	1	1	1	0	0	0	1	1	0	0	1	0	0	0
2		0	1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	1	0	0	1	1	0	0	1	0	0	0	0	0	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	0	0	1	1	1	0	1	0	1	0
3		0	1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	0	1	0	1	1	0	0	1	0	0	0	0	0	1	1	1	1	1	0	1	0	1	1	1	1	1	1	0	1	0	1	1	0	1	0	1	1	0
June 24 2014, 3:35:43 PM, Sequence 2																																																																									
1		0	1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	0	1	1	0	0	0	1	0	1	0	0	0
2		0	1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	1	0	0	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0	1	0	1	0
3		0	1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	0	1	0	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	0	1	0	0	0	0	1	1	0	1	1	0

One very important datum is that the counter is incrementing in reverse order - 00, 10, 01. That tells me the data is being transmitted Least-Significant-Bit first, i.e. binary 1000 = decimal 1. This is opposite of the way that binary is usually represented in textbooks, tutorials, etc. I am used to seeing the lowest significant bit on the right side of the number, not the left. 0001 = decimal 1, 0010 = decimal 2. Neither method is right or wrong, it's just a matter of convenience and familiarity.

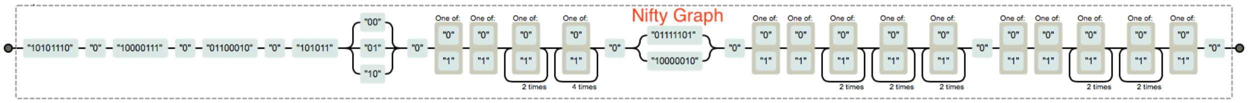
Checksum

After all that I decided to look for checksums and parity data. I already believed that byte 6 was a parity or checksum byte because of two properties it had. First, if any one variable changed between two packets, this byte *always* changed and it always changed in the same way. Second, if two or more variables changed between two packets, this byte always changed, but not in the same way in both packets. I began testing it by taking the simplest parity method I knew of (XOR) and just trying it out on the bytes preceding that byte in reverse order. Example: The hex equivalents of a packet from byte 0 to byte 6 were 75 E1 46 35 E0 BE CC. I first calculated E0 xor BE and found that the answer was not CC. Then I calculated 35 xor E0 xor BE, and so on. I found the match with E1 xor 46 xor 35 xor E0 xor BE = CC. I tried this on several more bytes and it worked consistently (If I had continued to check even more I would have found occasional errors due to the incorrect line coding technique I was using). I tried this in reverse order because it's logical that any data that doesn't change (such as a manufacturer ID, device ID, or protocol ID) would probably be near the beginning of the packet and that is the kind of data that is least likely to be included in a checksum.

Failure of Analysis

Now that I had a clear delineation of bytes, I could start analyzing them individually and in combination. The kinds of things I was looking for were the kinds of things you would find in almost any networking or serial data transfer protocol such as; 1) Source Address, 2) Destination Address, 3) Protocol Type, 4) Sequence (already identified), 5) Date, 6) Time, 7) Message Length, 8) Payload, 9) Parity or Checksum bytes, 10) Start bits (already identified) 11) Stop bits (confirmed as not present), 12) Device ID, 13) Device type.

At this point my research went totally off the rails. Rather than just getting down to the hard work of trial and error with a hex editor, I thought I could find a magic application that would analyze the data for me and point me towards relationships between different packets and bytes. This turned out to be false. I did find one program, Netzob, that is actually designed to do exactly what I needed. Unfortunately, it doesn't analyze any data group smaller than 4 bits. I already knew that I had at least 1 variable that was only 2 bits long and had no idea whether there would be more. Rather than risk getting false results and not knowing it, I chose not to pursue the use of Netzob for this project. Nevertheless, it looks like an awesome tool. I also tried regular expressions and finite state machines but all I got was a nifty graph (courtesy of www.regexper.com).



In a fit of desperation, I turned to data mining tools and algorithms, but stepped back from the horror of that unspeakable knowledge before my mind was shattered. That way madness lies.

The Document

As I progressed through my analysis, I repeatedly scoured the internet with a variety of search engines but kept coming up with nothing that identified the messages I recorded. However, after finding the 8-bit/1-bit pattern, I tried looking up "wireless protocol bytes separated by 0's" and… Eureka! I finally discovered a document describing exactly what I was seeing. At http://members.upc.nl/m.beukelaar/Crestaprotocol.pdf is a document that perfectly matched up with my data. It also informed me that I was listening to an environmental sensor of some type manufactured by a Dutch company called Cresta. They make personal and home products such as clock radios that combine with a variety of weather sensors to display the humidity, temperature, wind speed and direction, etc. Using this document I was able to determine that I was listening to a temperature-only sensor. I was even able to start reading the temperature from my captured packets.

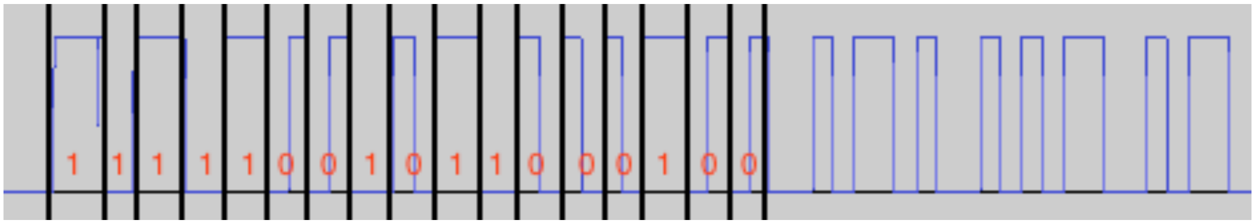
All Is Not Well…

While this seemed like a total success, there were problems that still nagged at me. First, the PDF describes a "decryption" technique that is necessary to perform before you can get useable data, but the decryption consists of nothing more than a few bitwise operations.

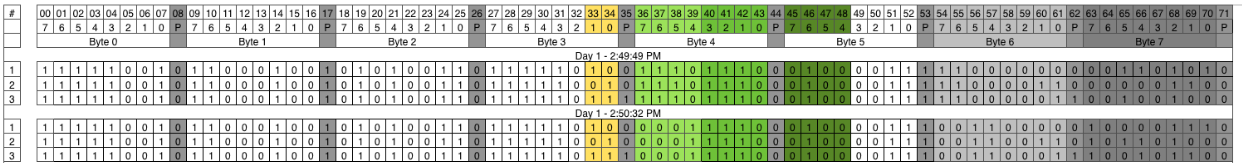
That's not real encryption! And who the heck encrypts the weather anyways? It didn't make sense. Why would a company bother to invent a terribly bad encryption technique in order to obscure data that isn't private? The next oddity was that the last byte is described in the PDF as an "exotic" checksum. Huh? What company would bother to invent their own checksum technique? Are standard checksums not good enough? Frankly, I'm totally impressed that the author of the document actually figured out the combination of bitwise operations that would convert the incorrectly decoded bit-stream into (mostly) useable data. It's amazing. Finally, I knew there must be something wrong because the temperatures I was capturing would do odd things from time to time. For example; in a sequential group of packet captures, the temperature went from the 70's fahrenheit to almost 110 fahrenheit in about 2 minutes, then dropped back into the 80's. That's clearly not possible on Earth.

Enlightenment

After some time banging my head into the desk, I got back to work. To make a really long story short, I finally found a forum post indicating that the line coding technique used was "a type of bi-phase mark encoding, differential manchester". By looking those two techniques up I found that they are not, in fact, related at all, but I decided to check them out regardless. I took a new series of recordings and decoded several messages using both line codes and compared them. When decoding using Differential Manchester things just clearly didn't make sense, but when I decoded the messages with Bi-Phase Mark Encoding, it all fell into place.



In Bi-Phase Mark Encoding, 0's are represented by a change of state occurring during a clock cycle. 1's are represented by the state of the signal remaining the same throughout the clock cycle.



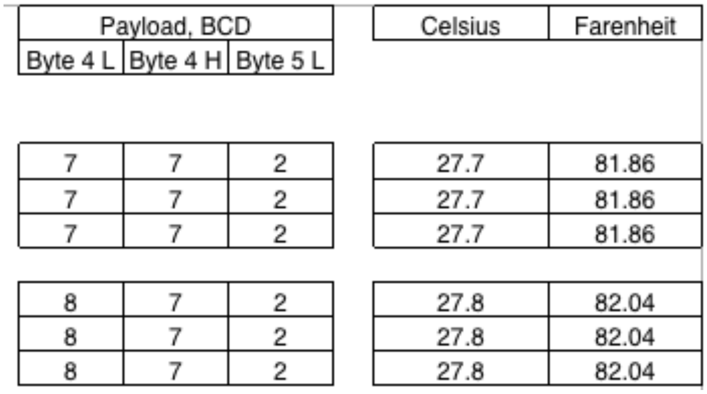
#	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17
	7	6	5	4	3	2	1	0	P	7	6	5	4	3	2	1	0	P
	Byte 0									Byte 1								
1	1	1	1	1	1	0	0	1	0	1	1	0	0	0	1	0	0	1
2	1	1	1	1	1	0	0	1	0	1	1	0	0	0	1	0	0	1
3	1	1	1	1	1	0	0	1	0	1	1	0	0	0	1	0	0	1
1	1	1	1	1	1	0	0	1	0	1	1	0	0	0	1	0	0	1
2	1	1	1	1	1	0	0	1	0	1	1	0	0	0	1	0	0	1
3	1	1	1	1	1	0	0	1	0	1	1	0	0	0	1	0	0	1

#	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
	7	6	5	4	3	2	1	0	P	7	6	5	4	3	2	1	0	P
	Byte 2									Byte 3								
																Day 1 - 2:49:49 PM		
1	0	1	0	1	0	0	1	1	0	1	1	1	1	1	0	1	0	0
2	0	1	0	1	0	0	1	1	0	1	1	1	1	1	0	0	1	0
3	0	1	0	1	0	0	1	1	0	1	1	1	1	1	0	1	1	1
																Day 1 - 2:50:32 PM		
1	0	1	0	1	0	0	1	1	0	1	1	1	1	1	0	1	0	0
2	0	1	0	1	0	0	1	1	0	1	1	1	1	1	0	0	1	0
3	0	1	0	1	0	0	1	1	0	1	1	1	1	1	0	1	1	1

#	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53
	7	6	5	4	3	2	1	0	P	7	6	5	4	3	2	1	0	P
	Byte 4									Byte 5								
1	1	1	1	0	1	1	1	0	0	0	1	0	0	0	0	1	1	1
2	1	1	1	0	1	1	1	0	0	0	1	0	0	0	0	1	1	1
3	1	1	1	0	1	1	1	0	0	0	1	0	0	0	0	1	1	1
1	0	0	0	1	1	1	1	0	0	0	1	0	0	0	0	1	1	1
2	0	0	0	1	1	1	1	0	0	0	1	0	0	0	0	1	1	1
3	0	0	0	1	1	1	1	0	0	0	1	0	0	0	0	1	1	1

#	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71
	7	6	5	4	3	2	1	0	P	7	6	5	4	3	2	1	0	P
	Byte 6									Byte 7								
1	1	1	0	0	0	0	0	0	0	0	0	1	1	0	1	0	1	0
2	1	1	0	0	0	0	1	1	0	0	0	0	0	0	1	1	0	0
3	1	1	0	0	0	0	0	1	1	0	0	1	0	0	1	0	0	0
1	0	0	1	1	0	0	0	0	0	1	0	0	0	0	1	1	1	0
2	0	0	1	1	0	0	1	1	0	1	0	1	1	0	1	0	0	0
3	0	0	1	1	0	0	0	1	1	1	0	0	1	0	1	1	0	0

There are indeed 8 bytes, each separated by 1 bit. That one bit is an even parity bit which ensures that every full 9-bit word has an even number of 1's in it. If there are an even number of ones in the first 8 bits, then the parity is a 0. If there are an odd number of 1's in the first 8 bits, then the parity is a 1. The xor checksum in byte 6 worked constantly, and, most importantly, the temperature data now made sense. By referring back to the PDF document I found earlier, and accounting for the fact that it would be slightly off due the the line-coding error, I determined which sections of the packet contained the payload. The three 4-bit words containing temperature data are highlighted in shades of green. You may also notice that the counter is slightly different now since it starts at 1 instead of at 0. It's a bit unusual to see something so close the hardware count like a human instead of like a machine, but whatever. It still counts and the data still works.



Payload, BCD			Celsius	Farenheit
Byte 4 L	Byte 4 H	Byte 5 L		
7	7	2	27.7	81.86
7	7	2	27.7	81.86
7	7	2	27.7	81.86
8	7	2	27.8	82.04
8	7	2	27.8	82.04
8	7	2	27.8	82.04

The 4-bit temperature words are in Binary Coded Decimal and reading them in reverse order (just to make it easy) reveals the current temperature in Celsius. A quick conversion gives me the consistently accurate temperature in Fahrenheit (mis-spelled above for your amusement).

Conclusion

So there it is, my foray into the twin dark arts of signals intelligence and blind reverse engineering. I hope this can serve as an example/primer for anyone else trying to figure out what the flood of digital signals around us actually mean.To be completely honest, this whole experiment would probably have been a pulsating mass of fail if the target of my investigation had been anything more complicated than a thermometer. Luckily, I got the thermometer. What will you get?

-Rory O'Hare