Over the Air Remote Code Execution on the DEF CON 27 Badge via NFMI

World's first NFMI exploitation, sorta
or
OTARCEDC27NFMIOMGWTFBBQ

Agenda

1. Introduction
2. Intro to the Badge
3. Buffer Overflow and Proof of Concept
4. NFMI Specs
5. Convert Analog Signal into Symbols
6. Convert Symbols to Bytes
7. Reverse Engineer CRC, Craft Packets
8. Remote Crash the Badge
9. Hacking NFMI Firmware
10. OTA RCE Demos

Because Any Meeting without an Agenda is a Trap

Introduction

- Seth Kintigh
- Hardware Security Development Lifecycle Engineer a Dell
- Hobbyist programmer since 1987, learned cipher breaking from my grandma, mom was meteorologist and COBOL programmer, dad is an Electrical Engineer
- WPI graduate
  - BS EE with minors in CS and Physics
  - MS EE with concentration in crypto and infosec
- Started career as an EE, but shifted to network security in 2004
- At home at low levels and layers

Near Field Magnetic Inductance (NFMI)

- Short range wireless **physical layer** that communicates by coupling a tight, low-power, non-propagating magnetic field between devices.
  - The concept is for a transmitter coil in one device to modulate a magnetic field which is measured by means of a receiver coil in another device. [1]
- Not radio!
  - Radio waves (electromagnetic waves) decay at $\mathbf{1/r^2}$
  - Magnetic fields decay at $\mathbf{1/r^6}$
    - Very short range, 2m tops, 10s of cm badge to badge, a 1-2 cm in my experiments.
    - Short range makes it more secure
  - Low absorption by body tissue, unlike radio, good for a “Body Area Network”
  - More efficient than radio over short distances
- Used in some hearing aids and some proximity cards as part of the NFC protocol

1: https://en.wikipedia.org/wiki/Near-field_magnetic_induction_communication

2: https://www.nxp.com/products/wireless/miglo/nfmi-radio-for-wireless-audio-and-data-streaming:NXH2261UK

Almost No Information Available

- Weird levels of secrecy for hearing aid tech (dreams of Apple earbuds?)
  - No data sheet (**very** weird)
  - No protocol info at all
  - No development kits
  - No samples
  - NDA required with minimum orders in the 100,000s of units

Software Defined Radio (SDR)

- Functions that were once performed in specialized hardware can now be done in software, hence Software Defined Radio
  - I used GNURadio to modulate and demodulate signals
- I use HackRF to tune and receive/transmit signals
- Instead of an antenna I use a coil, basically an electromagnet (or half of a transformer), to send and receive signals
- I used Python for everything else
  - Pulling clean packets out of noise, unmasking/unobfuscating packets, convolution and deconvolution of symbols, converting symbols to/from data, computing their bizarre CRC format, writing packets into .wav files for transmit

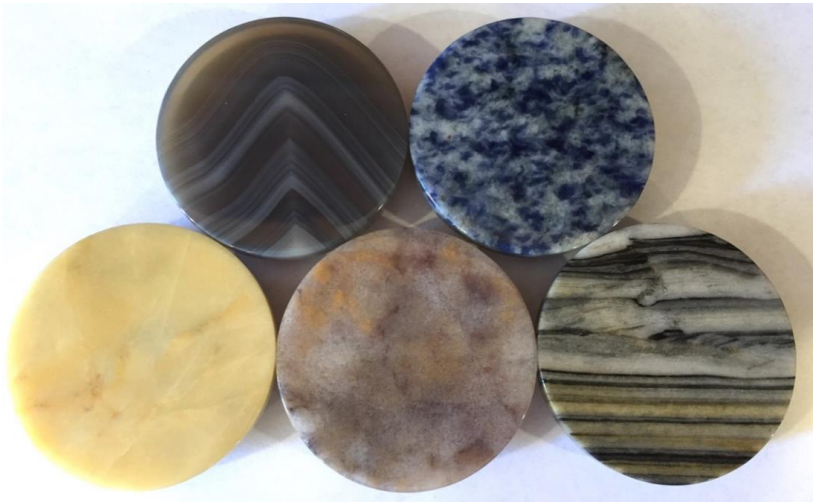
Other Terms You Should Know

- Buffer overflow: write data to a stack variable then keep on writing until you overwrite the return address of a function. Rewrite the return address to some point at your code or somewhere interesting.
- SWD/J-TAG: low level debugging interface for hardware. Like GDB, for hardware – read or write registers, memory and flash, step the clock one cycle at a time, good stuff.
- Convolution code: In telecommunication, a convolutional code is a type of error-correcting code that generates parity symbols via the sliding application of a boolean polynomial function to a data stream.[1]

[1] https://en.wikipedia.org/wiki/Convolutional_code

The DEF CON 27 Badge Game

- The badges communicate with each other via NFMI, also LEDS and beep depending on activity
- They were part of a game:
  - Must communicate with 1 each of “magic” versions of the badge types: Speaker, Village, Contest, Artist, Goon
  - “Prize” is a piezoelectric rick-roll
- Cut from crystalline stone, see great presentation on how they were made[2]

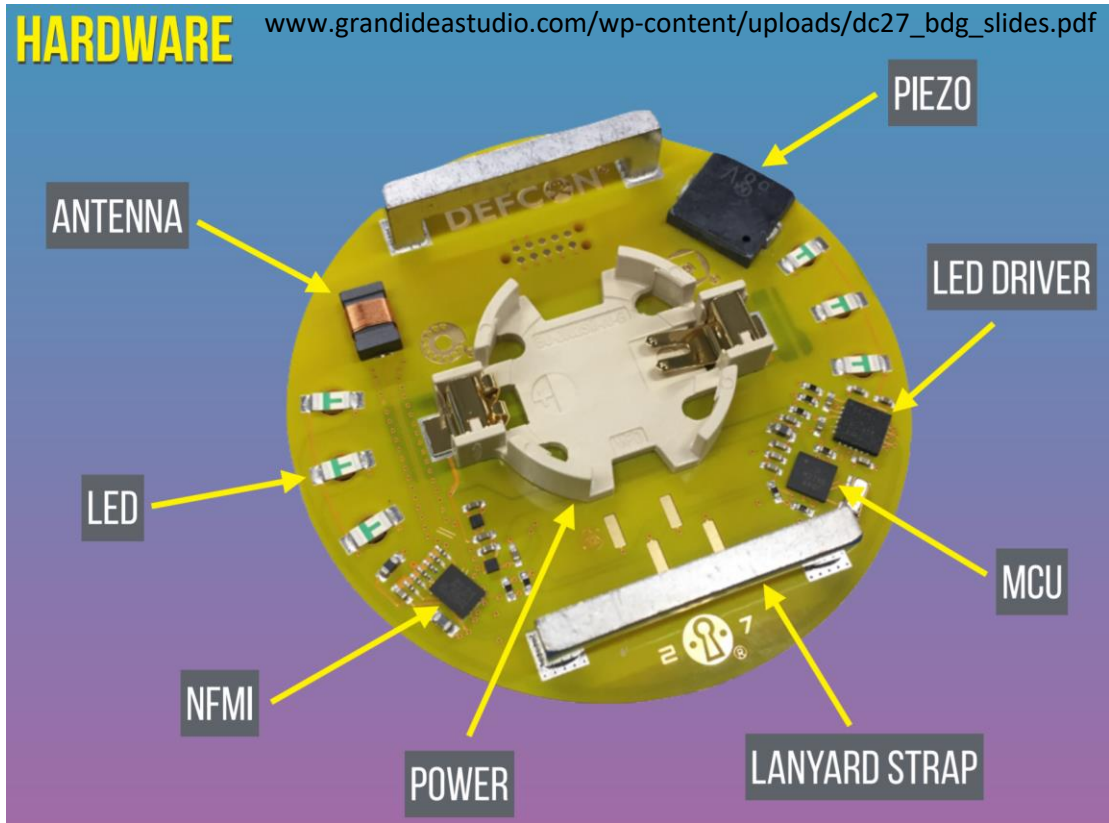


[1] http://www.grandideastudio.com/defcon-27-badge/

[2] https://www.youtube.com/watch?v=gnZQcWIX02A

DEF CON 27 Badge Hardware

- Badge has an MCU, NFMI chip, LEDs, and piezoelectric speaker
- MCU communicates to the NFMI chip via UART
- When the MCU boots up, the MCU loads the badge firmware
- Within that firmware is an NFMI protocol firmware patch
- The MCU sends that firmware patch to the NFMI chip during bootup



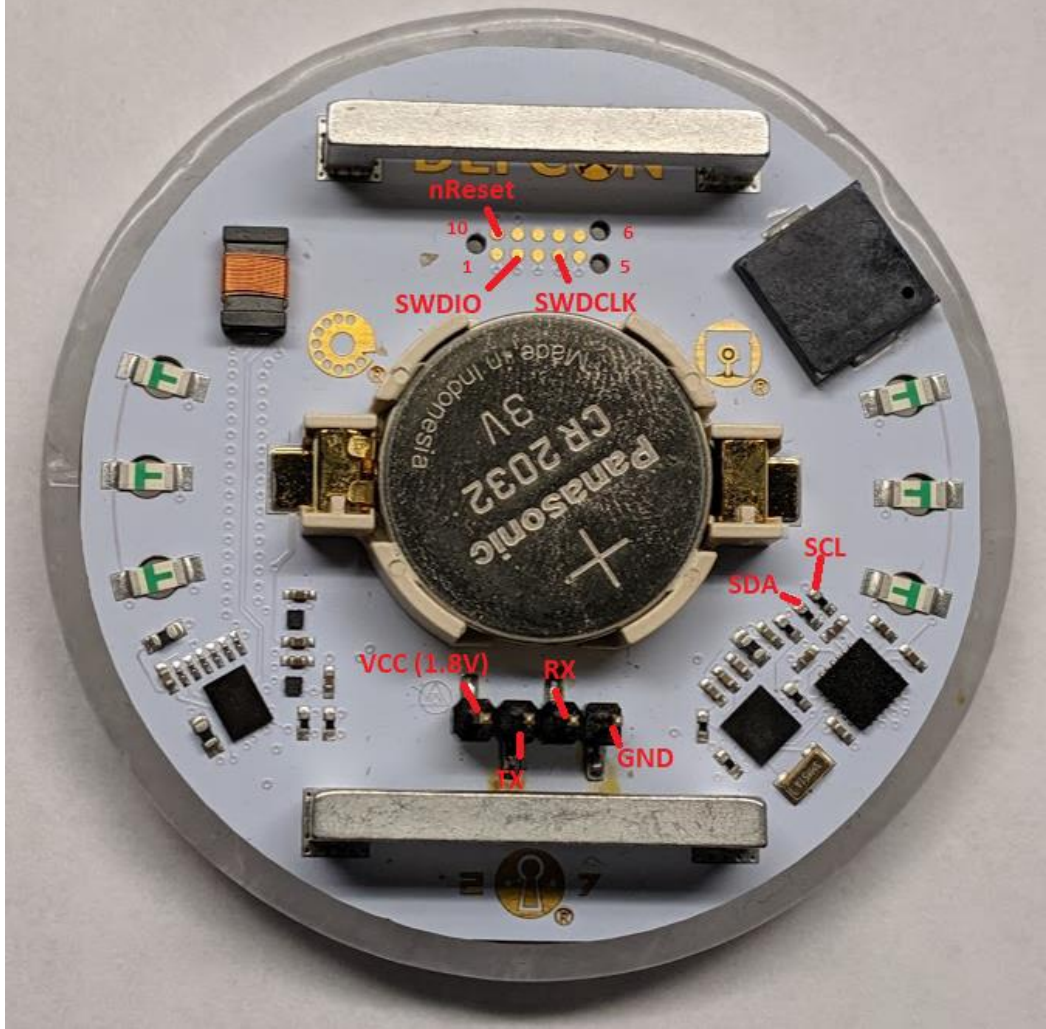
DEF CON 27 Badge Debug Interfaces

- There are pads for serial and SWD communication with the MCU
  - Solder on leads or use pressure connectors

"Depopulating the connectors stops the hackers!"

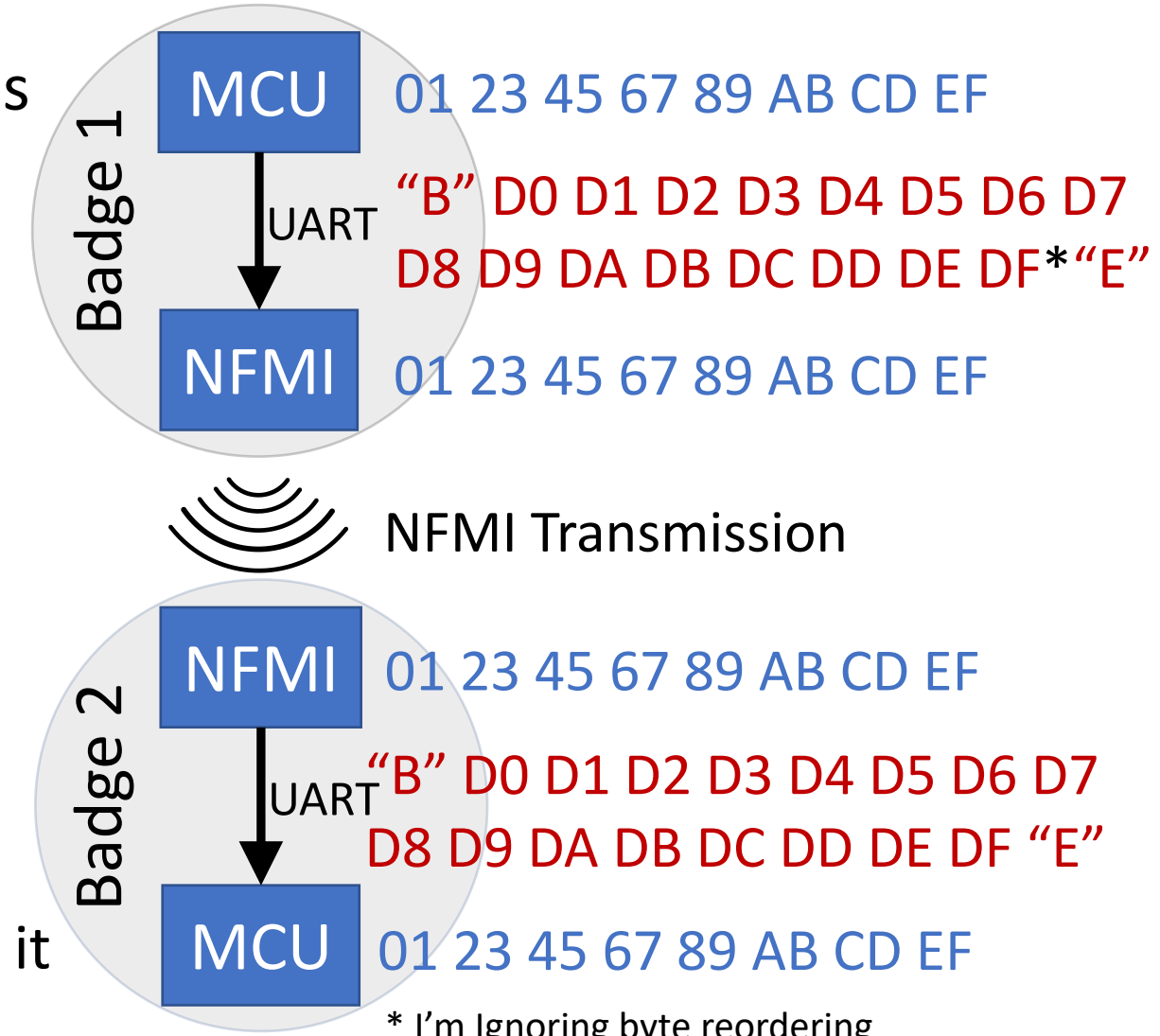
- Serial port shows a terminal interface for the badge MCU
- JTAG/SWD allows rewriting of the MCU firmware, and full debugging control over the MCU, including stepping the clock, reading registers, etc.

www.reddit.com/r/Defcon/comments/cpmpja/defcon_27_badge_hacking_for_beginners/



Padding UART for Fun and Profit

- Badge MCU wants to transmit 8 bytes
- MCU pads that to 18 bytes
- Sends via UART to NFMI chip
- NFMI chip un-pads and transmits



- Receiving NFMI re-pads data
- Sends via UART to MCU
- Badge MCU strips padding, puts data on ring buffer until it's ready to process it

Buffer Overflow

- To find game clues I spent the first few hours reverse engineering the firmware
  - Then the source code was released… But I've never been given the correct answers before
- Found a buffer overflow so obvious I was sure it was a part of the game
  - (Narrator: It wasn't)

```
// extract the data contents from the buffer until we reach the packet footer
i = 0;
while ((ch = nxhRingBuffer[nxhTxIndex]) != 'E')
{
    if (nxhRxIndex == nxhTxIndex)  // we've reached the end of the buffer
        return 1;

    dataBlob[i] = ch;
    i++;
    nxhTxIndex++;
    nxhTxIndex %= LPUART0_RING_BUFFER_SIZE;
}
```

Buffer Overflow Proof of Concept

- I Verified it was exploitable by simulating a large packet
  - I started by writing a buffer overflow exploit in ARM code
  - I used JTAG to write it directly onto the MCU's ring buffer for receiving NFMI data
  - The badge read the data and executed my code injection, proving it was exploitable

```
W1 0x1ffffcd6 0x04  // set nxhTxIndex
w1 20002f00 42  d0 d0 d0 d0  d0 d0 d0 d0  d0 d0 d0 d0  d0 d0 d0 d0 // “B”, 16 bytes of padded nibbles
w1 20002f11 00 00 00 00  11 11 11 11  44 44 44 44  55 55 55 55 // Overwrite other stack vars, R4, R5
w1 20002f21 33 4c 00 00  e5 f1 ff 1f  11 11 11 11  22 22 22 22 // LR→[POP R0-R2,R3,R4,PC], R0→str addr, R1, R2
w1 20002f31 33 33 33 33  44 44 44 44  2d 04 00 00  48 61 63 6b // R3, R4, LR->printf, then my string
w1 20002f41 20 74 68 65  20 70 6c 61  6e 65 74 0d  0a 00 45    // end of string, “E”
w1 0x1ffffcd4 0x51 // set nxhRxIndex
```

  - Demo!
- Now I just needed to do that with a real NFMI transmission

NFMI Specs are Tough to Find

- Some NXP NXH2261UK info in marketing pamphlets, blogs, and FCC filings:
  - Center frequency: 10.579 MHz[2][5], 10.6 MHz[1], 10.56 MHz antenna on badge[3]
    - 12.288 MHz oscillator?[6]
  - Bandwidth: 596 kbps[1][4] and/or 400 kHz[1] or 568.7 kHz[7]
    - Supports streaming via I^2C?
  - D8PSK/8-DPSK modulation[1][7]
  - Up to 2 audio Tx, 2 audio Rx[6]
    - Firmware suggests it has 8 queues, each 16 bytes (group chat?)

1: https://www.futureelectronics.com/resources/get-connected/2017-06/future-electronics-near-field-magnetic-induction

2: https://www.nxp.com/products/wireless/miglo/nfmi-radio-for-wireless-audio-and-data-streaming:NXH2261UK

3: http://www.grandideastudio.com/wp-content/uploads/dc27_bdg_bom.pdf

4: https://www.nxp.com/docs/en/fact-sheet/MIGLOFS.pdf

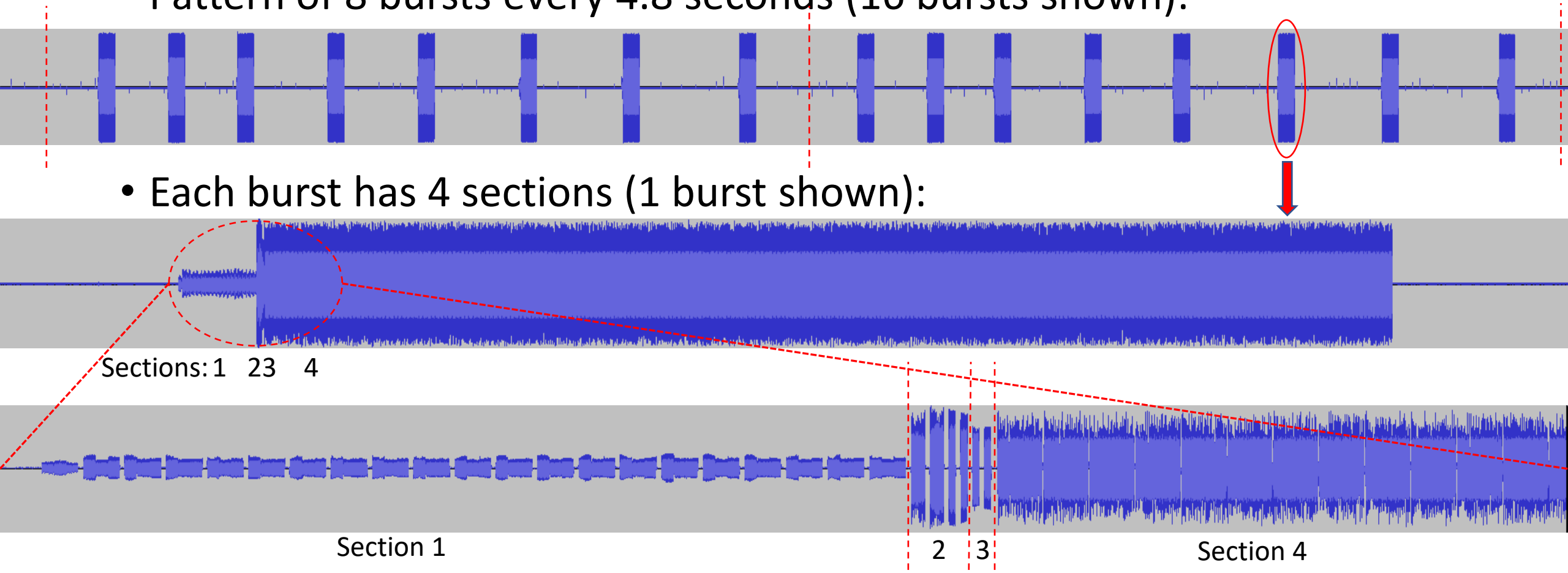
5: https://fccid.io/TTUBEOPLAYE8R/RF-Exposure-Info/RFExp-3568435

6: https://www.52audio.com/wp-content/uploads/2018/06/NXP%E6%81%A9%E6%99%BA%E6%B5%A6%E3%80%8A%E6%81%A9%E6%99%BA%E6%B5%A6%E7%9C%9F%E6%97%A0%E7%BA%BF%E8%80%B3%E6%9C%BA%E5%92%8C%E4%BD%8E%E5%8A%9F%E8%80%97%E6%B8%B8%E6%88%8F%E8%80%B3%E6%9C%BA%E7%9A%84%E8%A7%A3%E5%86%B3%E6%96%B9%E6%A1%88%E3%80%8B.pdf

7: https://apps.fcc.gov/eas/GetApplicationAttachment.html?id=5049516

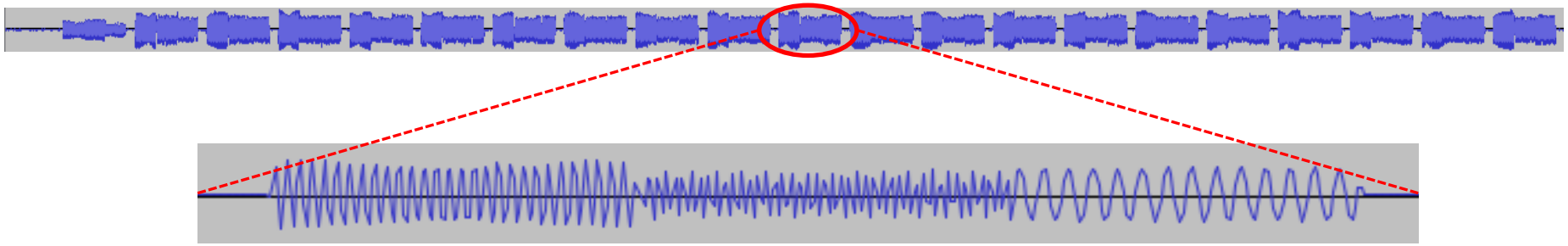
Start by Analyzing the Analog Signal

- Pattern of 8 bursts every 4.8 seconds (16 bursts shown):
- Each burst has 4 sections (1 burst shown):

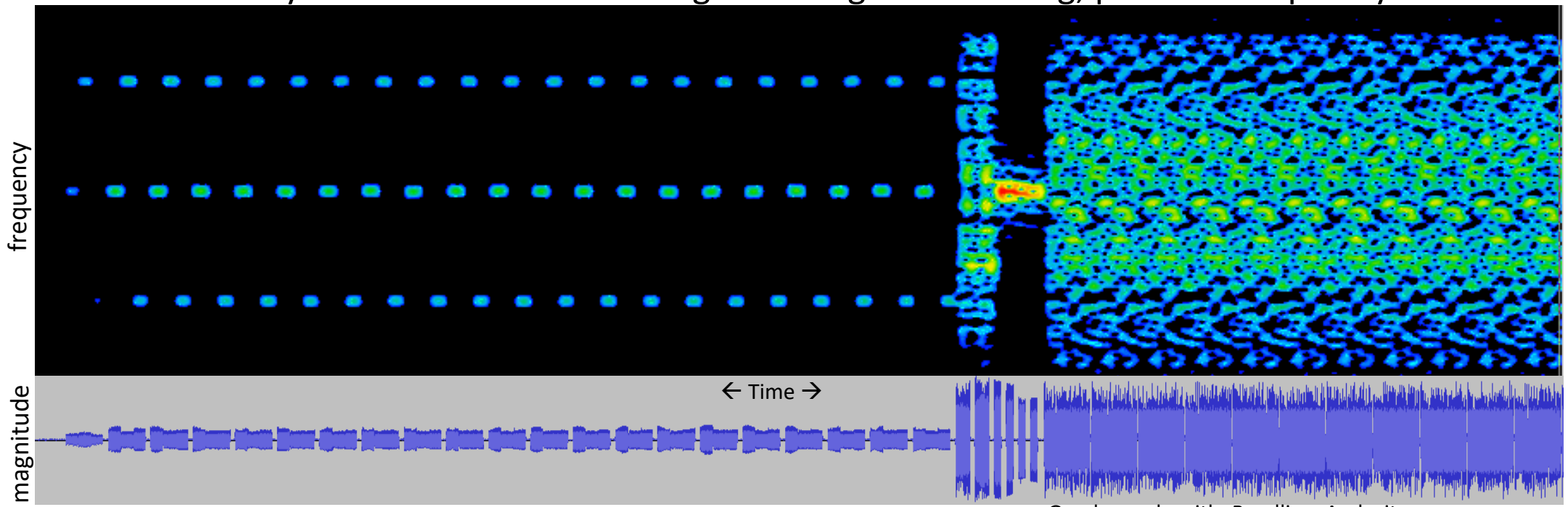


Raw signal capture, no down-conversion

Section 1



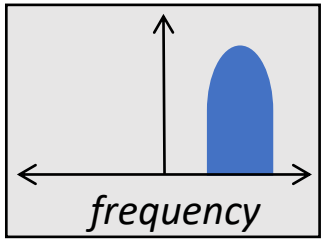
- Section 1 is 21 pulses of 3 frequencies then a pause/null
  - Appears to be timing signal using/mimicking trinary FSK modulation (TFSK?)
  - Center of frequency plot is 10.569MHz, other pulses shifted +/- 150MHz
    - Probably to tell the receiver the signal strength and timing, possible frequency info

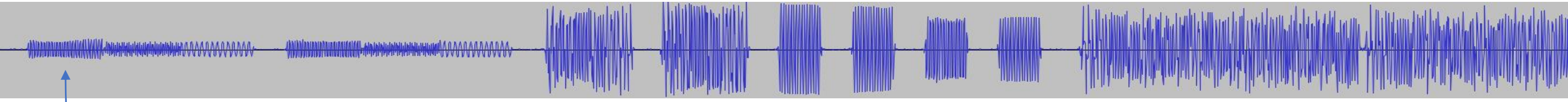


Graphs made with: Baudline, Audacity

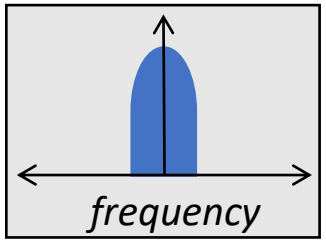
Quick Note on Down-Conversion

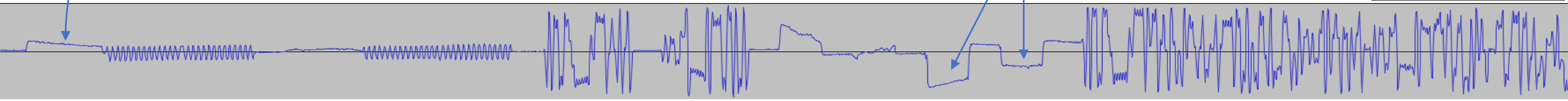
- It's the process of shifting a signal to an Intermediate Frequency
- This is the raw signal, most energy at 10.569MHz +/- 200KHz:





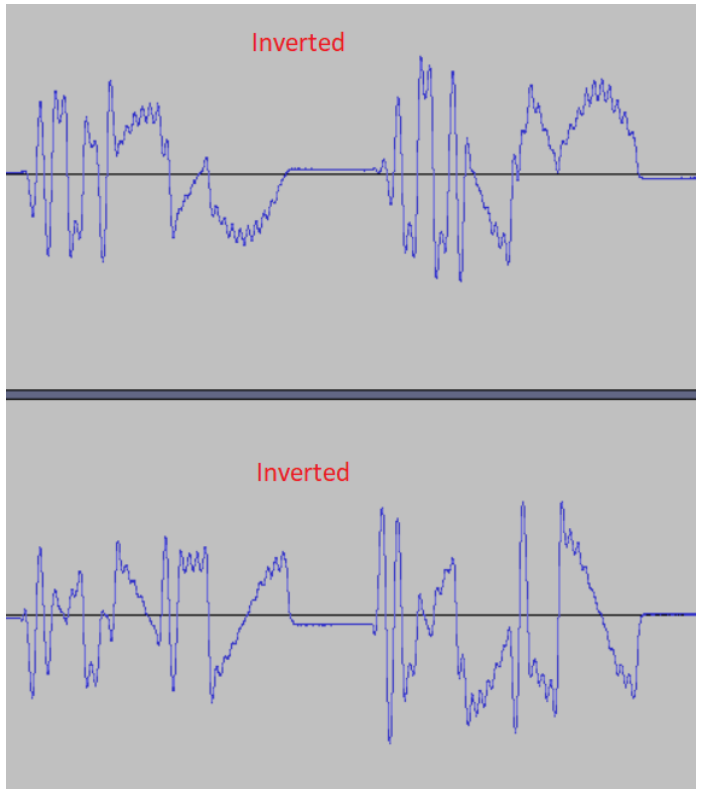
- Below is the signal when the HackRF is tuned to 10.569MHz, and it down-converts the signal by 10.569MHz
  - Most energy is now centered at 0Hz +/-200KHz
  - Signals that were at 10.569MHz are now at *almost* 0Hz (sections 1 and 3)
  - Data is more visually obvious (sections 2 and 4)

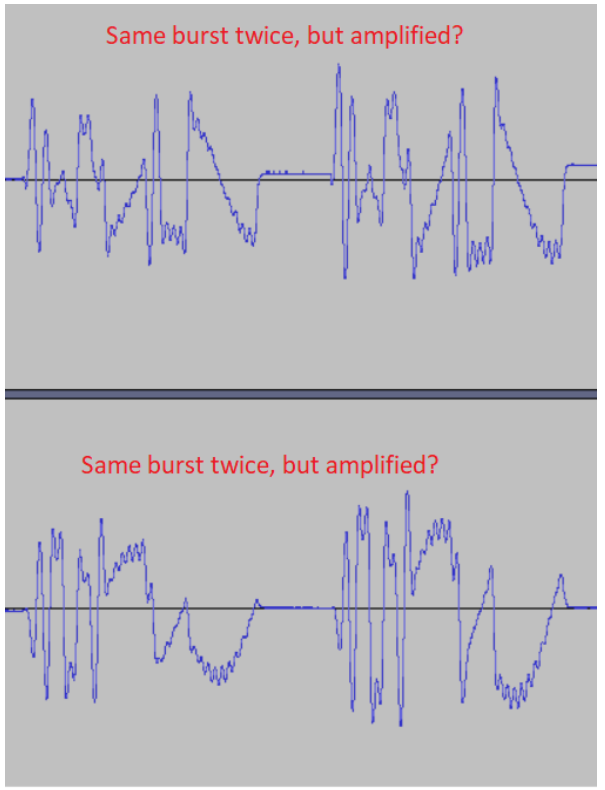


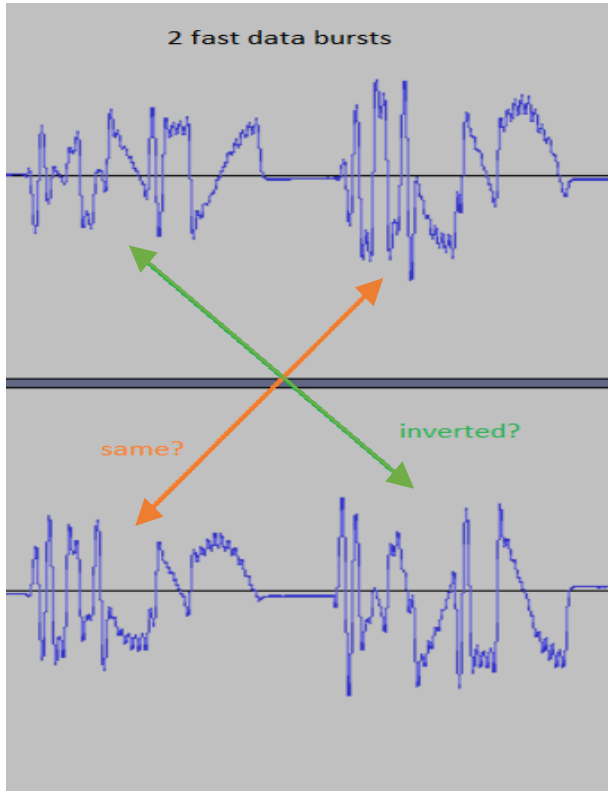


Section 2: Preambles

- Section 2 held one of a few patterns, often repeated twice, or inverted, or with magnitude and phase swapped



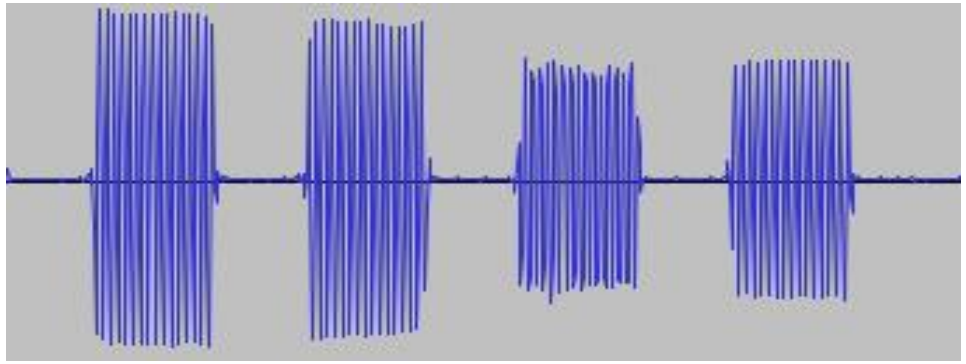




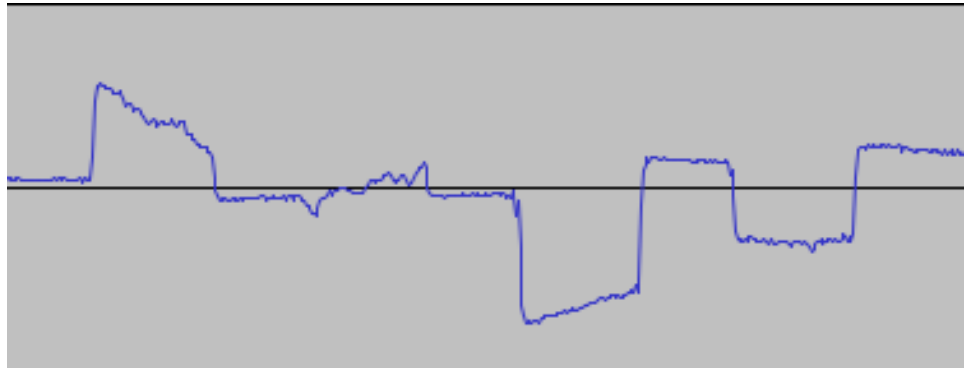
Demodulated by 10.56MHz so center appears at 0Hz

Section 3: More Timing?

- Seems to be more timing(/strength/freq) data, communicated with 4 bursts of the center frequency followed by a pause/null

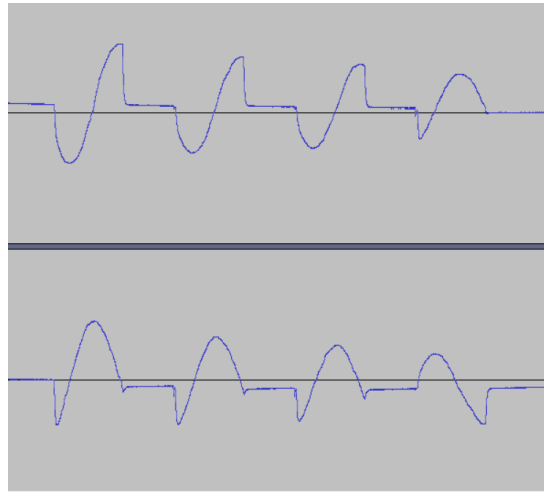


Magnitude with no down-converting



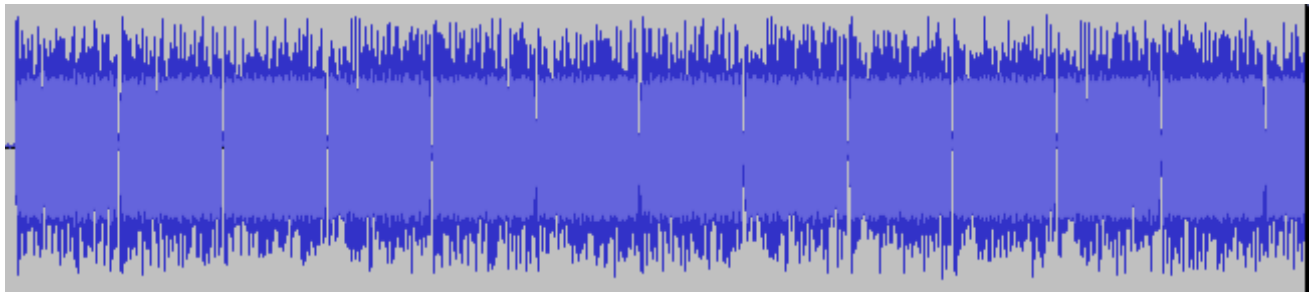
Mag down-converted by 10.569MHz to almost 0Hz

Frequency drift happens

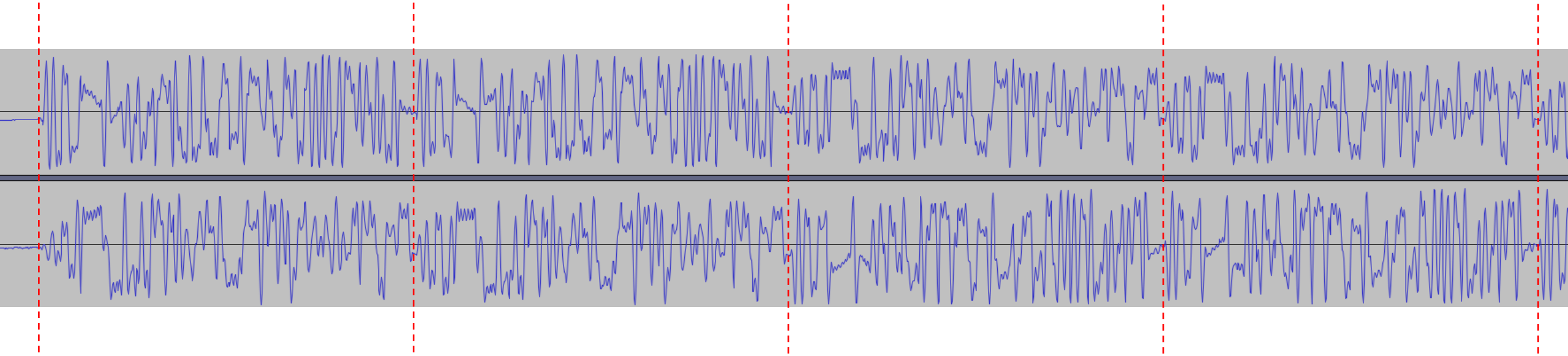


Mag and phase down-converted

Section 4: Data!

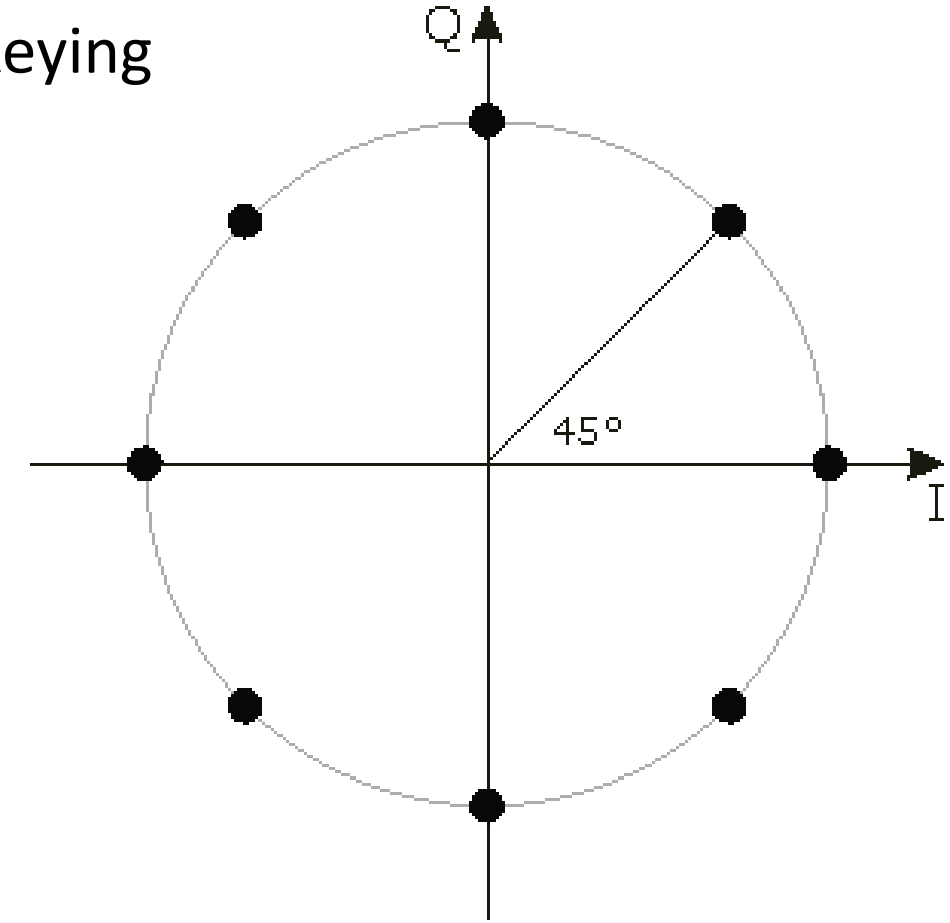


- 271(?!) copies of the data packet
  - Each starts with one of 8 variations of Section 2 preamble
  - Followed by data
  - Then a brief null/pause
- Sometimes exact copies, sometimes inverted, sometimes I and Q swap



D8PSK Modulation

- **D**ifferential, **8**-point constellation, **P**hase **S**hift **K**eying
- PSK modulations transmit data by modulating a carrier frequency using carefully timed cosine “I” and sine “Q’ inputs
- If you plot these signals by magnitude of the “in-phase” (I) and “quadrature” (Q) components the result is a constellation of 8 points
- Each phase is a “**symbol**” 0-7
- Differential means the information is transmitted as the difference between 2 symbols, modulo 8



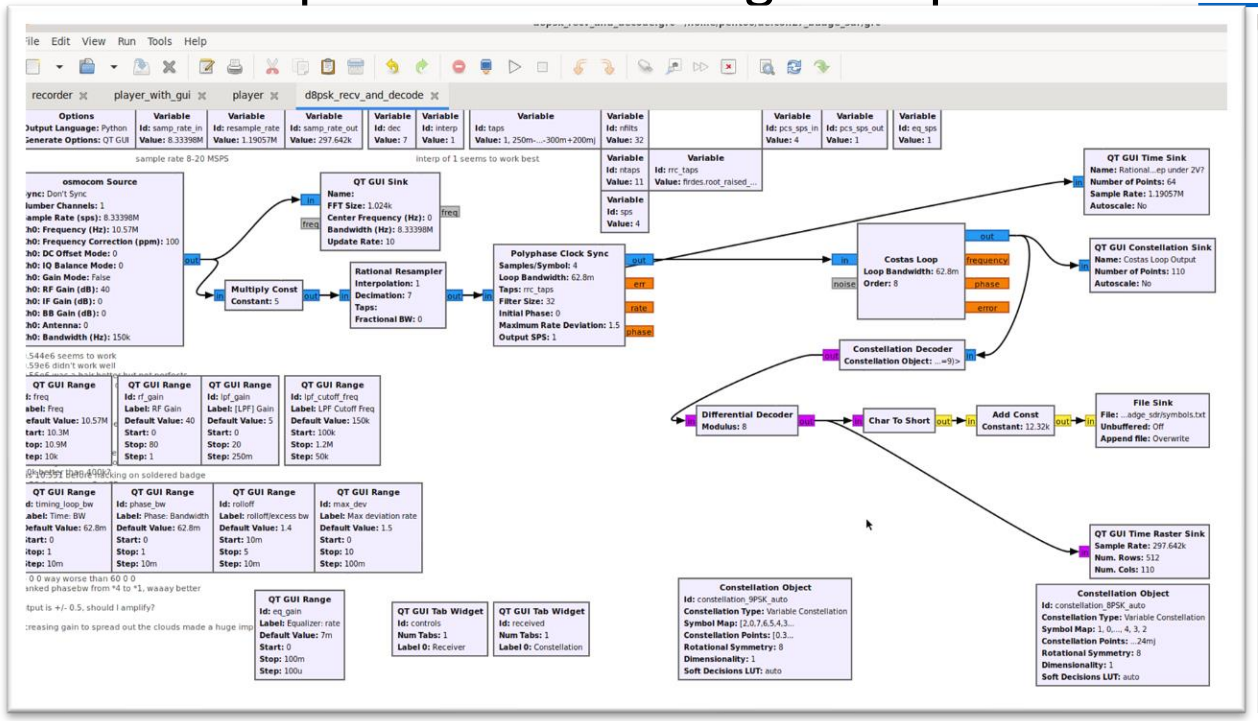
8PSK constellation

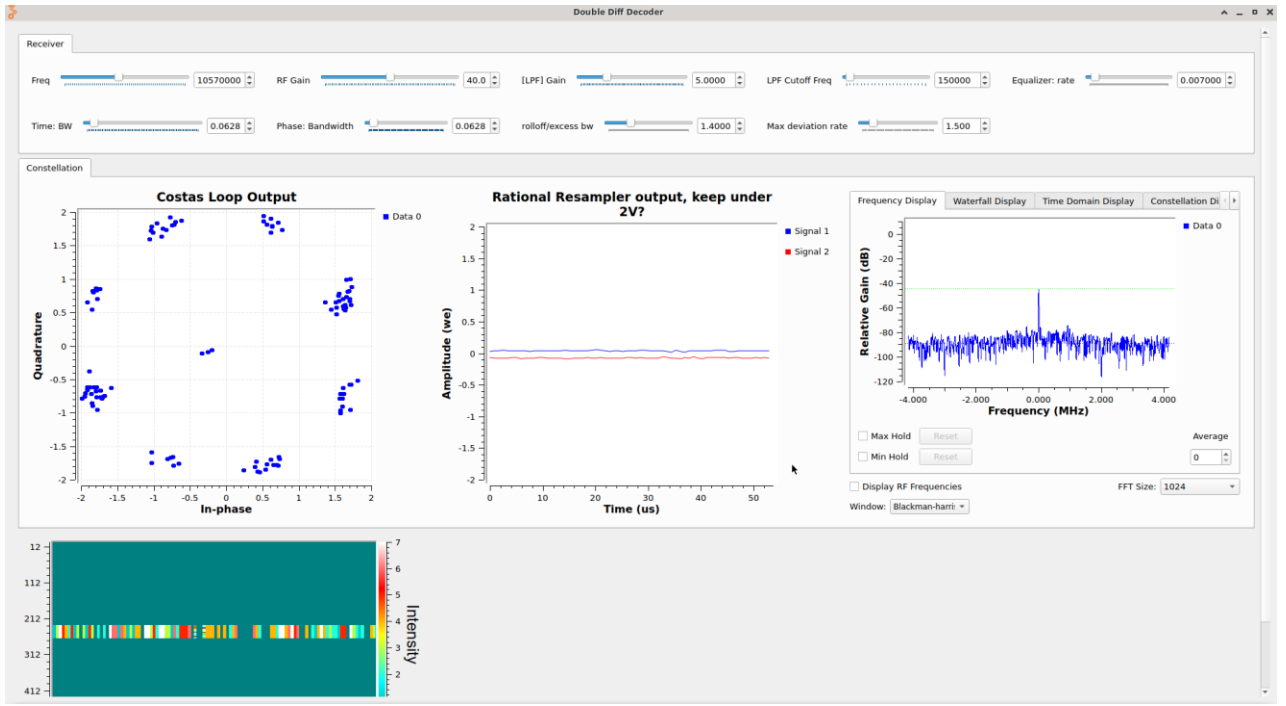
Frequency and Sample Rate

- Center frequency seems to move
  - Different badges have different freqs? Temperature? Sample rates? Anger?
- 1.5515MHz worked for a long time, then 1.4MHz, then 1.569MHz
- Sample rate of 2Msps and 1.192Msps (corresponding to 596kHz bandwidth) didn’t work as well as 1.19055Msps (595,275Hz bandwidth)
  - Latter value gave me 4 samples per symbol, 440 samples per packet

Reverse Engineering the Analog Signal

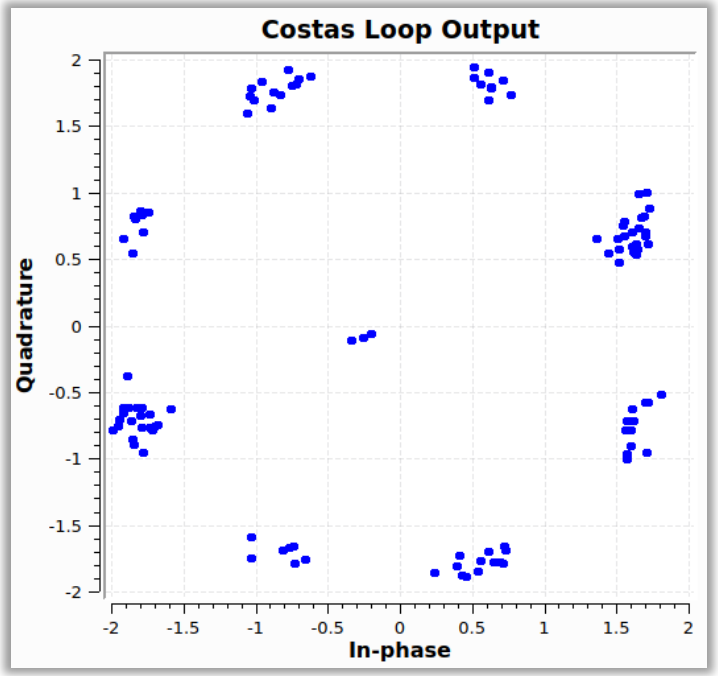
- HackRF to receive the signal
- Used GNURadio to write a D8PSK demodulator to output symbols
  - Nightmare of poor docs, broken examples, months of guessing and checking
  - I published working examples here: https://github.com/skintigh/GNURadio_examples





Dealing with Noise and Nulls

- The 271 copies of section 4 varied a lot, only some of that due to noise
  - Structure seems* to be: 3 null symbols, 1 random symbol, then 106 symbols
- Nulls?!?
  - Appeared to be noise at first
  - Sort of a 9th symbol in D8PSK, the null forms a 9th dot at the center of the constellation (D9PSK?)
  - Related to NXP's CoolFlux BSP audio chip?
    - Uses nulls in a OFDM-DQPSK signal [1]
      - Simple to find timing using a low pass filter[1]



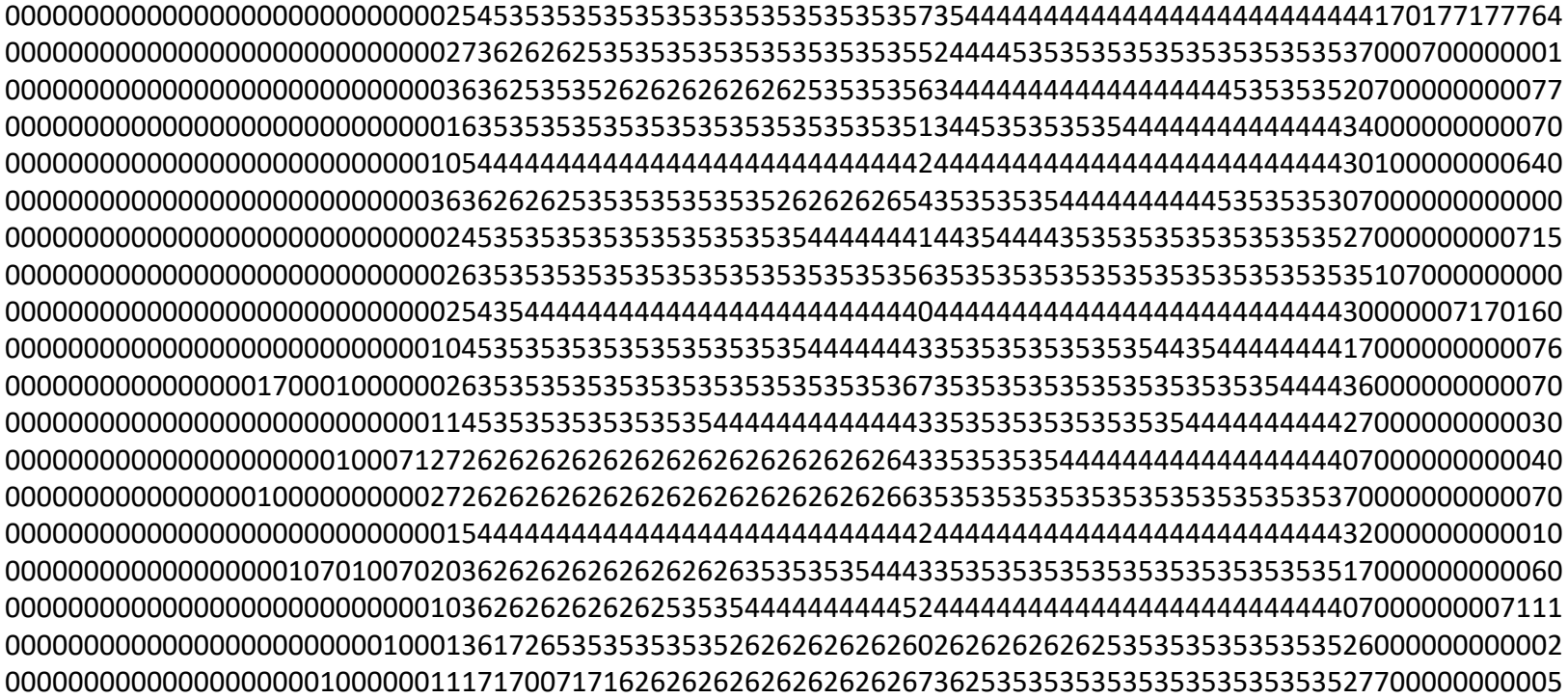
110 Symbols Plotted by Phase and Delay

* Or maybe random symbol R, 2 nulls, 7-R? Signals have 11 tiny samples that look like: .I…….I.

[1] DSP-Based Implementation of a Digital Radio Demodulator on the ultra-low power processor CoolFlux BSP

Section 1-3 Symbols

- Section 1: 21 x 99 symbols
  - Timing?



- Section 2: 2 copies of 44 symbols
  - **Preamble!**
- Section 3: 4 x 56 symbols
  - Timing?

44404040246000006420004000000000210000000005
44404040246000006420004000000000210000700000

40000000000000003000000000000
21000000000000003000000000000
11000000000000003000000000000
02000000000000002100000000000

Section 4 Symbols

- Section 4: 271 copies of 110 symbols
  - Not all identical copies dues to noise
- Wrote a Python tool to count packet variations
  - Outputs 1 copy of the most common packet in this section

00735350305015510000552000442775576644063173443234570742120043137542513031530202076616413144077352775447244044
51714440404024600000642000442774675644063173443233670732120043137542513031530202076616413144077352775447244044
61704440404024600000642000442774675644063173443233670732120043137542513031530202076616413144077352775447244044
50724440404024600000642000442774675644063173443233670732120043137542513031530202076616413144077352775447244044
62004440404024600000642000442774675644063173443233670732120043137542513031530202076616413144077352775447244044
50724440404024600000642000442774675644063173443233670732120043137542513031530202076616413144077352775447244044
61704440404024600000642000442774675644063173443233670732120043137542513031530202076616413144077352775447244044
50014440404024600000642000442774675644063173443233670732120043137542513031530202076616413144077352775447244044
61704440404024600000642000442774675644063173443233670732120043137542513031530202076616413144077352775447244044
50724440404024600000642000442774675644063173443233670732120043137542513031530202076616413144077352775447244044
51004440404024600000642000442774675644063173443233670732120043137542513031530202076616413144077352775447244044
50104440404024600000642000442774675644063173443233670732120043137542513031530202076616413144077352775447244044

Preambles

- 20 fixed symbols, then 12 that form 1 of 3 patterns
- **Section 2**: alternates randomly between 2 sequences <with 12 nulls in-between?>
  - 4440 4040 2460 0000 6420 0040 0000 0000
  - 4440 4040 2460 0000 6420 0042 3133 4224
- **Section 4**: All 271 copies start (after null junk) with:
  - 4440 4040 2460 0000 6420 0044 2774 6756

Structure of Section 4 Data Symbols

Null Preamble	Packet Data	Checksum
5001 444040402460000064200044277467 56	4406317344323367073212004313754251303153020207661641314407735277	5447244044

Header:

- 4 bytes null/primer
- Preamble 8x4 symbols
  - 20 fixed, 12 variable

Packet Data 16x4 symbols:

- 16 symbol counter
  - Changes every burst, seems to increment
- 4 symbols for user data size
  - Learned this by modifying badge firmware
- 44 symbols for 11 bytes of user data
  - 12 symbols/3 bytes unused by badge

Packet Footer

- 10 symbols

Packet Data

Counter	Size	8 Bytes of User Data	Unused Data
427723574223216 7	0732	1200431375425130315302020766164 1	31440773527

Finding the Mask Symbols for Data 5-11

- So now we have a stream of differential symbols, do we have data?
  - Not even close
- Every symbol after the preamble* are masked.
  - Setting the badge to transmit 8 bytes of 0x00 confirmed this
    - This gave me 32 symbols of the mask
  - Modified badge firmware to send 11 bytes of 0x00
    - That gives us the mask for those 11 user bytes (packet data bytes 5 though 15):

    5 4 3 5 7 0 3 6 4 0 7 3 0 4 1 4 4 2 7 4 6 2 0 2 0 7 6 6 1 6 4 1 5 2 3 4 4 7 5 1 1 2 5 5
      - I don't see a pattern.

* and maybe the last 12 symbols of the preamble are masked, too.

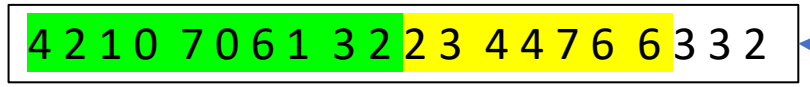
Finding the Mask Symbols for Data 0-4

- The symbols changed with every burst -- alternated between changing 1 symbol and (usually) 7
  - Every 8 packets: 8 symbols changed. Every 32: 9 changed. Every 128: 10 changed
  - Conclusion: it's a counter, 2 bits per symbol, with convolution on half(?!) of the bits
- After some reboots, the first symbol alternated between 0 and 4, on others between 2 and 6
  - Every other location could be any symbol 0-7
  - The first group of 4 symbols had 128 patterns, the next group of 4 had 256 patterns
  - Conclusion: the counter increments by 2, and sometimes starts odd, sometimes even
    - I didn't know which was odd or even, so I guessed 0/4 was even. I guessed… poorly.
- Tail of 6 changed symbols obfuscates other symbols
  - Even if I assume byte3 == 0 and know byte 4, I can only compute the last 2 of 8 symbols

0000 1111 2222 XXXX XX44

Finding Mask Symbols for Data 0-4

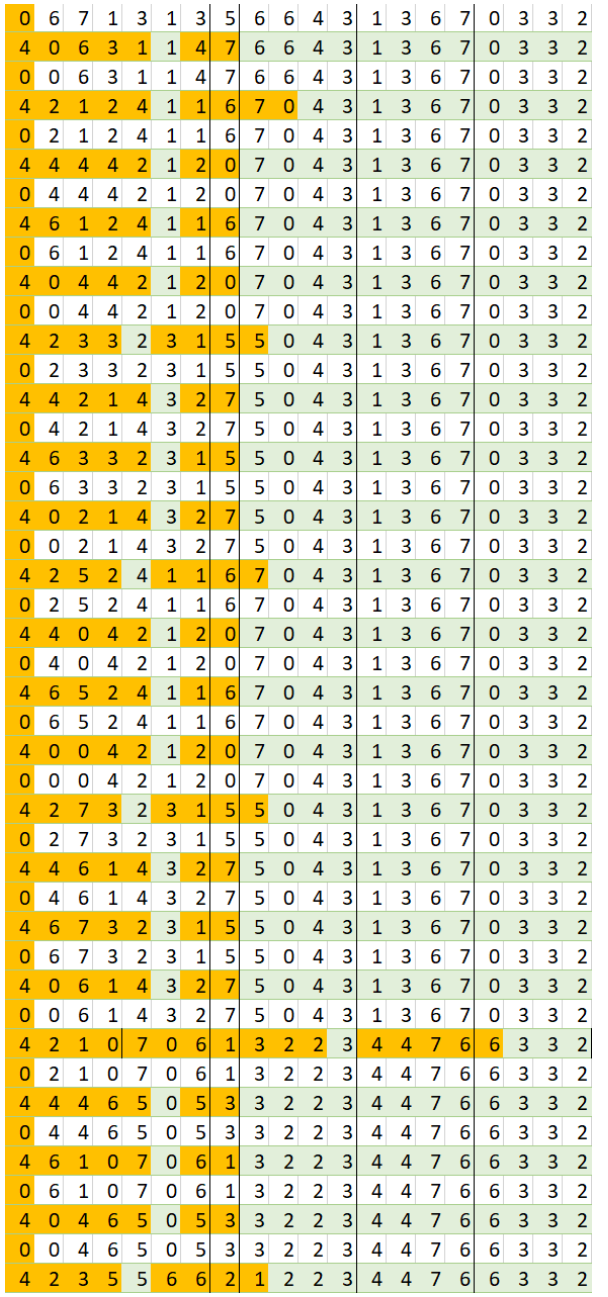
- I Recorded the counter for a week, observed transitions
  - This gave me a mask for the first 10 symbols



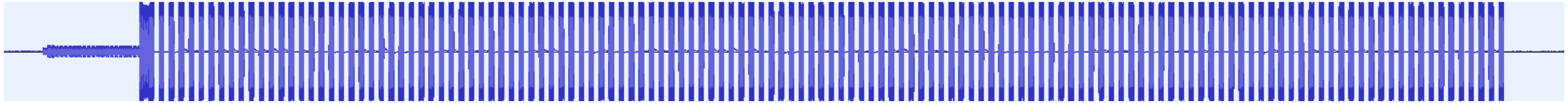
Confirmed mask

Last symbol to increment plus its tail of 6 changes

  - Confirmed the count doesn't start at zero
- Took 19.1 hours to get 9th symbol, 76.5 hours get 10th
  - At this rate it will take 51 days for 12 symbols
  - 36 years for 16
  - 9139 years for 20
- Didn't understand well enough to brute force them
- Needed a smarter method



Finding the Mask Symbols for Data 0-4



- Then I got lucky by getting unlucky – I broke a badge
- Badge became very angry
  - Instead of 271 x 110 symbol packets, it transmitted a pattern of 108-symbols followed by 108 nulls
  - Counter moved over 16 symbols!?!
  - Transmitted around 1.4 MHz?
- Conclusions:
  - Mask is 4210 7061 3242 2275 0332
    - Assuming my 0/1 guess was right…
  - It's better to be lucky than smart

4	2	1	0	7	0	6	1	3	2	4	2	2	2	7	5	4	5	6	6	4	5	7	4	7	2	3	7	3	5	6	1
4	2	1	0	7	0	6	1	3	2	4	2	2	2	7	5	0	3	5	5	4	7	0	1	5	2	3	7	3	5	6	1
4	2	1	0	7	0	6	1	3	2	4	2	2	2	7	5	4	3	5	5	4	7	0	1	5	2	3	7	3	5	6	1
4	2	1	0	7	0	6	1	3	2	4	2	2	2	7	5	0	1	0	7	2	7	7	3	5	2	3	7	3	5	6	1
4	2	1	0	7	0	6	1	3	2	4	2	2	2	7	5	4	1	0	7	2	7	7	3	5	2	3	7	3	5	6	1
4	2	1	0	7	0	6	1	3	2	4	2	2	2	7	5	0	7	5	5	4	7	0	1	5	2	3	7	3	5	6	1
4	2	1	0	7	0	6	1	3	2	4	2	2	2	7	5	4	7	5	5	4	7	0	1	5	2	3	7	3	5	6	1
4	2	1	0	7	0	6	1	3	2	4	2	2	2	7	5	0	5	0	7	2	7	7	3	5	2	3	7	3	5	6	1
4	2	1	0	7	0	6	1	3	2	4	2	2	2	7	5	4	5	0	7	2	7	7	3	5	2	3	7	3	5	6	1
4	2	1	0	7	0	6	1	3	2	4	2	2	2	7	5	0	3	3	2	5	4	3	5	3	2	2	0	2	0	0	1
4	2	1	0	7	0	6	1	3	2	4	2	2	2	7	5	4	3	3	2	5	4	3	5	3	2	2	0	2	0	0	1
4	2	1	0	7	0	6	1	3	2	4	2	2	2	7	5	0	1	2	4	7	4	4	7	3	2	2	0	2	0	0	1
4	2	1	0	7	0	6	1	3	2	4	2	2	2	7	5	4	1	2	4	7	4	4	7	3	2	2	0	2	0	0	1
4	2	1	0	7	0	6	1	3	2	4	2	2	2	7	5	0	7	3	2	5	4	3	5	3	2	2	0	2	0	0	1
4	2	1	0	7	0	6	1	3	2	4	2	2	2	7	5	4	7	3	2	5	4	3	5	3	2	2	0	2	0	0	1
4	2	1	0	7	0	6	1	3	2	4	2	2	2	7	5	0	5	2	4	7	4	4	7	3	2	2	0	2	0	0	1
4	2	1	0	7	0	6	1	3	2	4	2	2	2	7	5	4	5	2	4	7	4	4	7	3	2	2	0	2	0	0	1

Finishing up the Data Mask

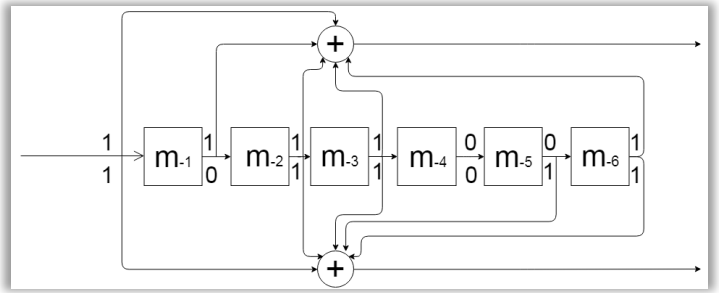
- Discovered counter moves at ludicrous speed when updating Tx packet
  - That's why counter starts at 346,637
- Wrote a script to update the Tx packet repeatedly then capture packets
  - Got mask and proved counter is 4 bytes
    - In case you want to send 2 billion packets over 41 years
- Data mask resulted in some erratic values when decoding sequential counts
  - Wrong guess about what 1 and 0
  - The broken badge locked the first (unused!) byte at 1 just to screw with me?
    - Mask for 1: 4210 7061 3242 2275 0332
    - Mask for 0: 2336 7701 3203 2275 0332

Checksum Mask

- That left only the mask for the Checksum/CRC
  - No way to get that until I know the algorithm and all data values
    - To get the values I had to understand the convolution
- So I guessed and moved on

Counter Indicates Values are Convolved

- As the counter increments, it either changes:
  - 1 symbol for odd bits
  - 6 of the next 7 symbols for even bits
- This means a 1-bit change is being spread out over 7 symbols
  - They are using convolution, possibly like the one used on the Voyager space probe
    - Voyager shift registers[1]:
      - 1111001
      - 1011011



- But only for **half** of the bits…?!?

Counter
427723574223
027723574223
446543654223
046543654223
467723574223
067723574223
406543654223
006543654223
421413701423
021413701423
444233061423
044233061423

[1] https://en.wikipedia.org/wiki/Convolutional_code

Reverse Engineering Convolution Code

- Started by changing 1 bit at a time
- Any 1 odd bit always changed 1 symbol, and always by 4
- Any 1 even bit changed 6 of the next 7 symbols
  - Amount of change depended on mask and distance from the set bit
- Even bit change (show in code and mask-bit logic)
  - Symbol positions **0, 2, 3, 6**: mask in [1,2,5,6]) * 4 + 2;
    - [bit0^bit1, 1, 0]
  - Symbol position **1**: (mask mod 4) * 2 + 3
    - [bit0^bit1, ~bit0, 1]
  - Symbol position **5**: mask in [1,3,5,7]) * 6 + 1
    - [bit0, bit0, 1]

Symbol position after even bit

Mask (or previous)	0	1	2	3	4	5	6
0	2	3	2	2	0	1	2
1	6	5	6	6	0	7	6
2	6	7	6	6	0	1	6
3	2	1	2	2	0	7	2
4	2	3	2	2	0	1	2
5	6	5	6	6	0	7	6
6	6	7	6	6	0	1	6
7	2	1	2	2	0	7	2

Reversing Convolution Code, Multiple Bits

- That math worked for decoding 1 even bit set in any position
- But multiple even bits were a mess
- After a lot of ugly math and dead ends, I realized the math that worked for a mask worked for *any* precious value

Reversing Convolution Code, Even Bits

- We can now convolve (or de-convolve) any number of even bits

Value of each byte:	0x00	0x00	0x01	0x00	0x05	0x00
Even bits of that byte:	0 0 0 0 0 0 0 0		1 0 0 0 0 0 0 0		1 1 0 0 0 0 0 0	
Mask of 0:	2 3 3 6 7 7 0 1		2 3 3 6 7 7 0 1		2 3 3 6 7 7 0 1	
+ Position 0	0 0 0 0 0 0 0 0		6 0 0 0 0 0 0 0		6 2 0 0 0 0 0 0	
sum	2 3 3 6 7 7 0 1		0 3 3 6 7 7 0 1		0 5 3 6 7 7 0 1	
+ Position 1	0 0 0 0 0 0 0 0		0 1 0 0 0 0 0 0		0 5 1 0 0 0 0 0	
sum	2 3 3 6 7 7 0 1		0 4 3 6 7 7 0 1		0 2 4 6 7 7 0 1	
+ Position 2	0 0 0 0 0 0 0 0		0 0 2 0 0 0 0 0		0 0 2 6 0 0 0 0	
sum	2 3 3 6 7 7 0 1		0 4 5 6 7 7 0 1		0 2 6 4 7 7 0 1	
+ Position 3	0 0 0 0 0 0 0 0		0 0 0 6 0 0 0 0		0 0 0 2 2 0 0 0	
sum	2 3 3 6 7 7 0 1		0 4 5 4 7 7 0 1		0 2 6 6 1 7 0 1	
+ Position 4	0 0 0 0 0 0 0 0		0 0 0 0 0 0 0 0		0 0 0 0 0 0 0 0	
sum	2 3 3 6 7 7 0 1		0 4 5 4 7 7 0 1		0 2 6 6 1 7 0 1	
+ Position 5	0 0 0 0 0 0 0 0		0 0 0 0 0 7 0 0		0 0 0 0 0 7 1 0	
sum	2 3 3 6 7 7 0 1		0 4 5 4 7 6 0 1		0 2 6 6 1 6 1 1	
+ Position 6	0 0 0 0 0 0 0 0		0 0 0 0 0 0 2 0		0 0 0 0 0 0 6 6	
sum	2 3 3 6 7 7 0 1		0 4 5 4 7 6 2 1		0 2 6 6 1 6 7 7	
Convolved value:	2*Mask-sum 2 3 3 6 7 7 0 1		2*Mask-sum 4 2 1 0 7 0 6 1		2*Mask-sum 4 4 0 6 5 0 1 3	

All math is modulo 8

Convoluted Convolution

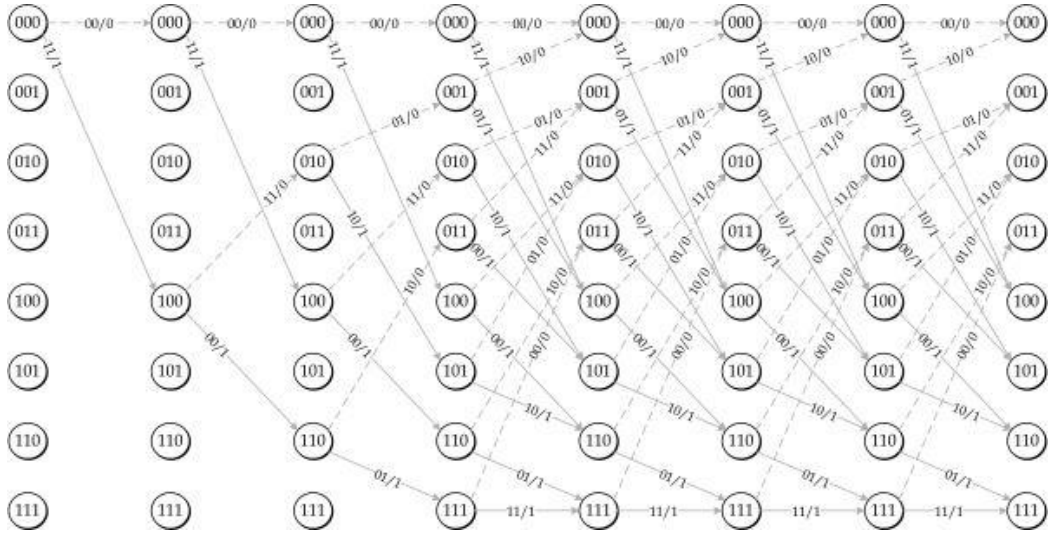
- Now I could decode the counter and it was sequential… mostly
- Sometimes multiple odd bits convolved
  - I looked for patterns, added rules until it worked right
- All the rules don't care what the current bit is, only look at previous bits
  - Up to 2 bytes ago!
  - Some rules are triggered by 0 bits, not just 1s
  - Rules are:
    - xxx011x
    - xxx101x
    - 1xxxx1x
    - 11xxxxx

odd bits	0	1	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0
encoded	0	4	4	0	0	0	0	0	0	0	4	0	4	0	0	0	0	4	0	0	4	4	4	4	4	0	0	0	0	0	0
xxx011x	x	x	x	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0
xxx101x	x	x	x	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1xxxx1x	x	x	x	x	x	x	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	4	0	0	0	0	0	0	0
11xxxxx	x	x	x	x	x	x	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4	0
sum	0	4	4	4	0	0	0	4	0	0	4	0	4	4	0	0	0	4	4	0	4	4	0	0	4	0	4	4	4	4	0

← Time →

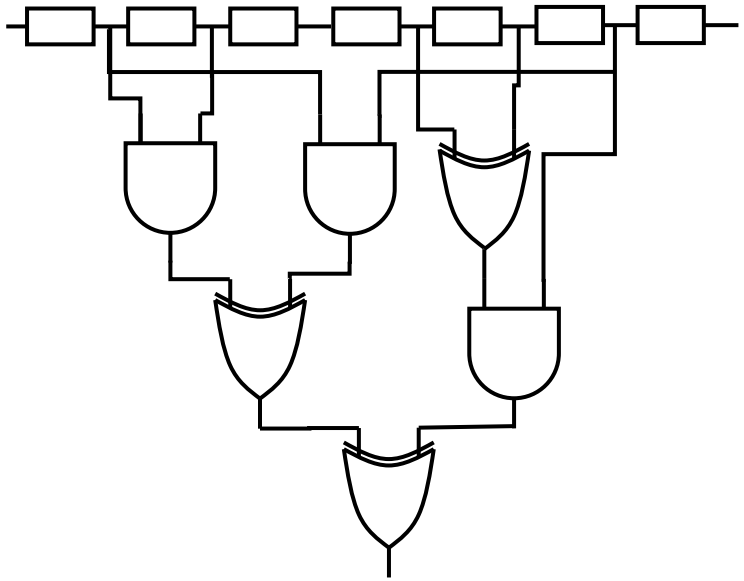
Convolution

- Convolution of even bits might be Trellis Code Modulation TCM*
  - I ended up figuring it out in a spreadsheet and writing some ugly python code, but it works
    - Easier than deciphering GNURadio documentation



Rate = 2/3, m = 7?

- Convolution of odd bits is… odd
  - xxx011x
  - xxx101x
  - 1xxxx1x
  - 11xxxxx



*Can TCM convolve 1 bit into a pattern like 2322012? Is that middle 0 possible?

CRC Reverse Engineering

- Each packet ended with 10 unknown symbols, equivalent of 20 bits
  - 2^{20} possible combinations, yet only 2^{12} patterns were common/real
    - 12 bits stored in 20 bits?!?
    - Changing 1 symbol can alter the next 6, won't that overwrite the nulls and primer?
- All 10 symbols, "20 bits," had to be correct
  - Altering any in a packet replay attack caused it to be rejected by the NFMI chip
- Must reverse engineer this in order to craft custom packets
  - Tried the tool "CRC Reveng" but it didn't seem to work
  - Wrote Python programs to try all possible 12-bit algorithms, didn't work?!?

CRC Reverse Engineering, Continued

- Observed 1-bit difference in counter resulted in predictable checksum change
- Checksum values were built up by XORing values from a table, just like a CRC!
  - I used counter values to get the CRC table values for the low bits
  - Noticed that updating a TX packet would fast-forward the counter by ~300,000 ticks
    - Wrote a program to speed the counter through more bits
  - Wrote another program to “bit walk” every byte I could control
    - Set 1 bit per packet
  - Wrote a program to ingest all this data and compute almost the entire CRC table

CRC Reverse Engineering, Continued

- I eyeballed a pattern in the 10 symbols that revealed a how the 12-bit CRC was stored
  - First 2 symbols store 4 bits of CRC data
  - Next 2 symbols store 3 bits
  - Next 2: 2 – only in odd bits
  - Next 2: 2 – only in odd bits
  - Next 2: 1 – only in odd bits
- Only the first 4 symbols can cause a tail of 6 symbol changes
  - Odd bits only changes 1 symbol
  - CRC is stored in 10 symbols total, ***including*** the 6-symbol tail of changes from convolution!
    - Wicked smaht
- Bits are shuffled around, for reasons
  - This is what thwarted CRC Reveng, and my brute force attempts
  - Later tried CRC Reveng with the bits rearranged, and it worked!

CRC Reverse Engineering, Continued

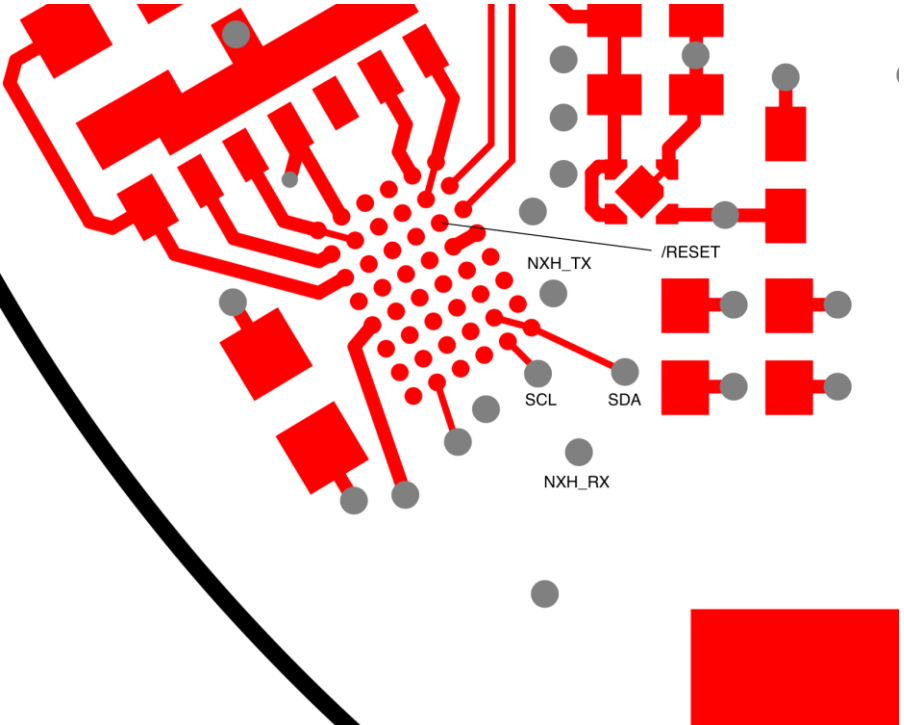
- Now I can compute a CRC for the 16 data bytes, but I need the mask
- Originally I guessed randomly at the mask and based all the math off that
  - I think this worked since CRC is built by XOR, with a non-zero base value for an empty packet, and I was effectively XORing that base again that with my mask symbols
    - 9 of the bits change one symbol by 4, so that works like XOR
    - 3 bits can cause a trail of 6 changed symbols, which made some packets with those CRC values unreliable?
- I *think* the CRC doesn't protect the Preamble
  - Tried making packets with a ton of preamble variations, none worked
- I assumed 16 0s in the data would have a CRC of 0 based the mask off that
  - It worked! Reliably!

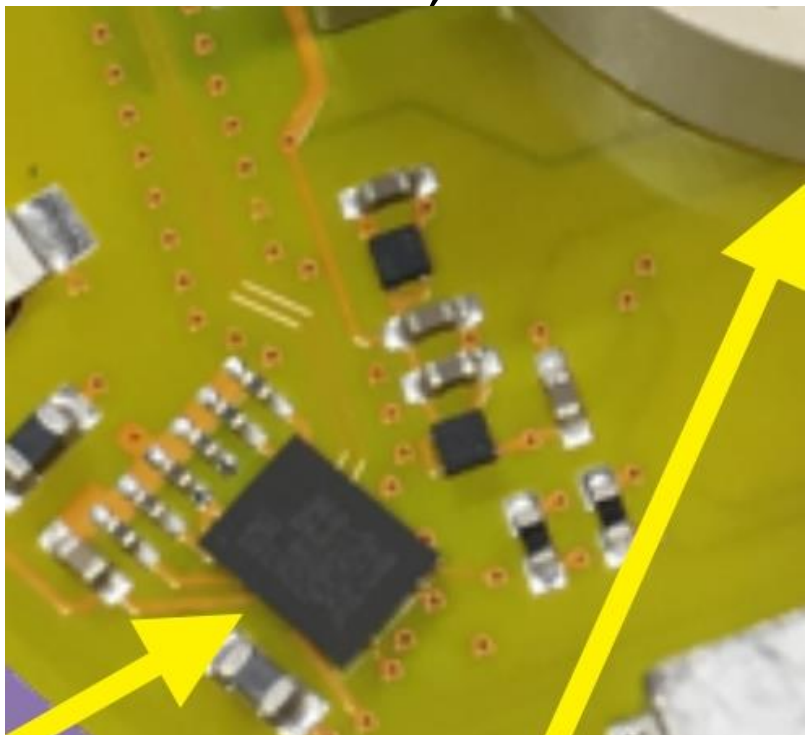
Crafted Packets at Last!

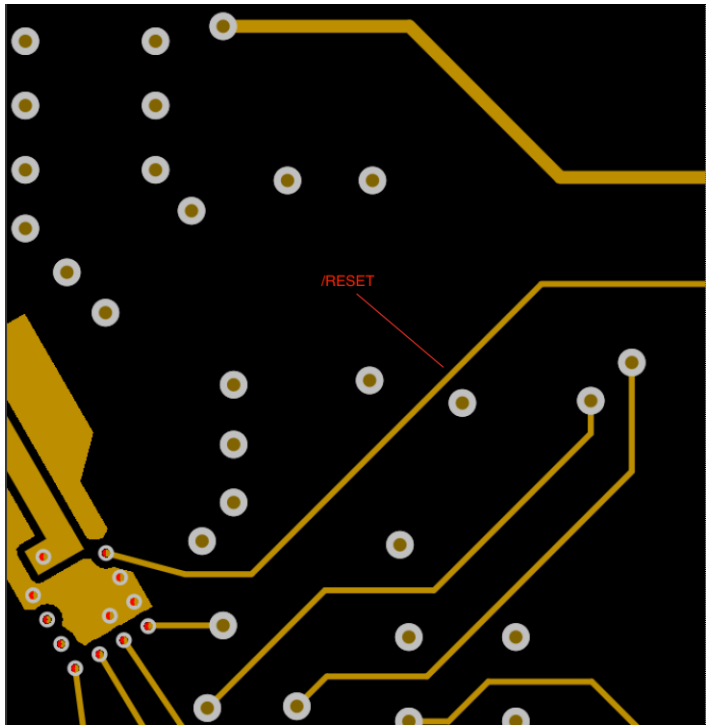
- With the CRC and mask I can now craft my own 16-byte packet!!!
  - Will release my tools on github
- But I need a 36 byte packet to overflow the badge…
- I knew this from the start, hoped I would figure it along the way
  - No field for that in the packet
  - Preamble tinkering was a bust
- Time to reverse engineer the NFMI firmware

Extracting the NFMI firmware

- To use SWD on the NFMI chip, first I had to find the reset line
  - Knew it was on a middle layer
  - Found out which ball it was from
  - Scoured slides to find the line to that ball, then Joe Grand sent me confirmation

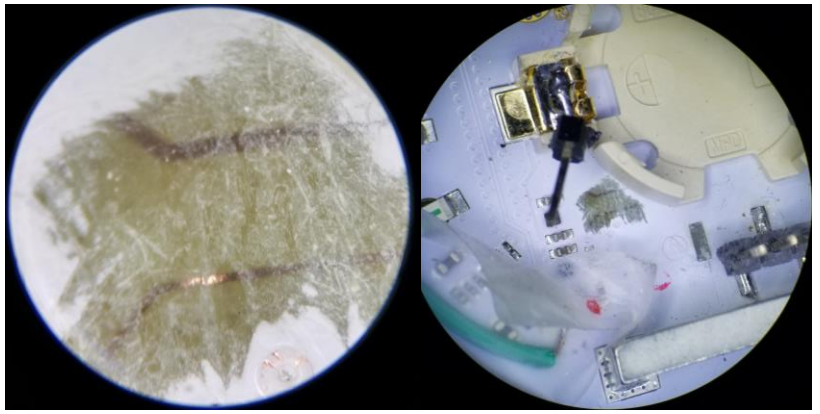






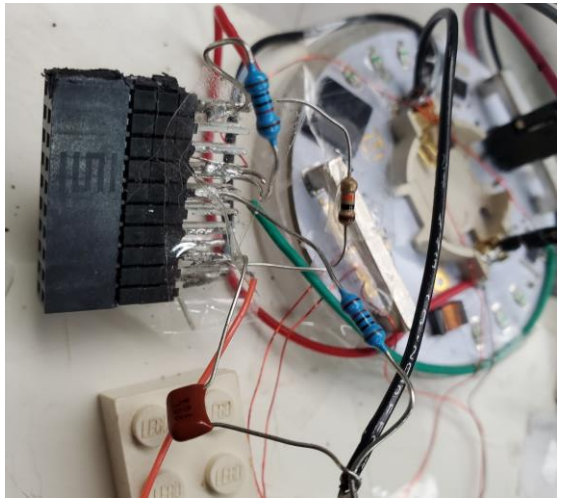
Extracting the NFMI firmware

- Scratched a layer off the badge circuit board to expose a reset line on the middle layer



- Cut it and soldered on a wire
  - The trace is about the size of a hair
  - Used a stereo microscope at Artisan's Asylum

- Connecting SWD to an undocumented chip
  - Added pull-ups, pull downs, resoldered everything
  - Then tried settings for random chips until one worked
- Extracted NFMI memory space 0-0x18000… **without** the NFMI protocol code



Reversing the NFMI Firmware

- I extracted the NFMI protocol code from a section of the badge's firmware, assembled the pieces together and dropped it into Ida Pro
  - Nothing in there indicated a packet length field
  - Coded to drop packets with more than 11 bytes
    - (Later I removed that, but I still don't know how to craft a longer packet, and faking the length resulted in uninitialized data)
- But I had seen (and logged) oversize packets occurring spontaneously and crashing the badge, what was going on?

```
B df df df df df df df df df df df df df df df df dd dd dd dd dd dd E -- length = 22
0xFFFFFFFFFFFFFFFF
-> Unique ID: 0xFFFFFFFF
-> Badge Type: Unknown
-> Magic Token: Yes
-> Game Flags: 11111111

B df df df df df df 42 df df df df df df df df df df df df df df df dd dd dd dd dd dd 42 df df df df df df df df df df df df df df df df dd dd dd dd dd dd E -- length = 52

Welcome to the DEFCON 27 Official Badge
```

Off By One Bug to the Rescue

```
// If ring buffer isn't full, add the data
if (((nxhRxIndex + 1) % LPUART0_RING_BUFFER_SIZE) != nxhTxIndex)
{
    nxhRingBuffer[nxhRxIndex] = data;
    nxhRxIndex++;
    nxhRxIndex %= LPUART0_RING_BUFFER_SIZE;
}
```

- NFMI sends packet to badge via UART,
  - Badge checks for 2 bytes of space before copying 1 byte at a time to the ring buffer
    - One-by-one copying allows partial packets, off-by-one allows odd-sized packets
- If I send the right pattern of packets, I can leave a buffer with 18 free bytes
  - NFMI chip sends ‘B’, 16 padded bytes and ‘E’ to badge MCU via UART
  - Badge writes ‘B’ and 16 bytes of padded data, sees only 1 byte free so it drops the ‘E’

- The badge reads packets from the beginning of the ring buffer, freeing more space

- Write a second packet before the badge empties ring buffer

- The badge firmware finds the first ‘B’ then copies the next 33 bytes before finding ‘E’
  - B16B16E ≈ B33E

Off By One Bug, Continued

- But wait, there's more!
  - Keep hammering with the largest size packet, B22E (11 user bytes):
    - Fill the ring buffer so the last entry is:
      - B22
    - 24 bytes read from the front allows 24 added to the end:
      - B22B22
    - Badge reads faster than we can transmit, so there is lots of space when the next packet arrives:
      - B22B22B22E
    - Badge reads that "B", then copies **68 bytes** to dataBlog buffer before finding the 'E'
      - The dataBlog buffer is only **18-byte** long…
  - Might be able to do even more
    - Fill the buffer with tiny packets to make reads take as long as possible
    - Then write larger packets that get truncated and keep the pain train going

Demo: Crash a Def CON 27 Badge via NFMI

- Now we can crash a stock badge at will!
  - It just takes a while with a 2048-byte buffer
    - 2048 / (a maximum of 24 bytes x 8 bursts / 4.8 seconds) x 2 = a boring demo
  - So I cheated and set the buffer to 72 bytes
- Process:
  - Initialize the buffer
    - Might not know where nxhTxIndex and nxhRxIndex are pointing, so completely fill the buffer by blasting it for a while
      - Ring buffer may have unread packets, indexes might be moved by “RO” or “RC” after reboot.
    - Stop transmitting
    - Have the badge read packets and completely drain the buffer (Often see B20 for a 72-byte buffer after a reboot)
    - Indexes are now equal
  - Attack the buffer
    - Send 1 packet of total length = LPUART0_RING_BUFFER_SIZE % 24
      - Length 8 or B6E for a 2048 byte buffer, nothing for a 72 byte buffer
    - Blast the buffer full, last entry is B22
    - Have the badge start reading, keep blasting, last entry will grow to B22B22B22E
  - Splat

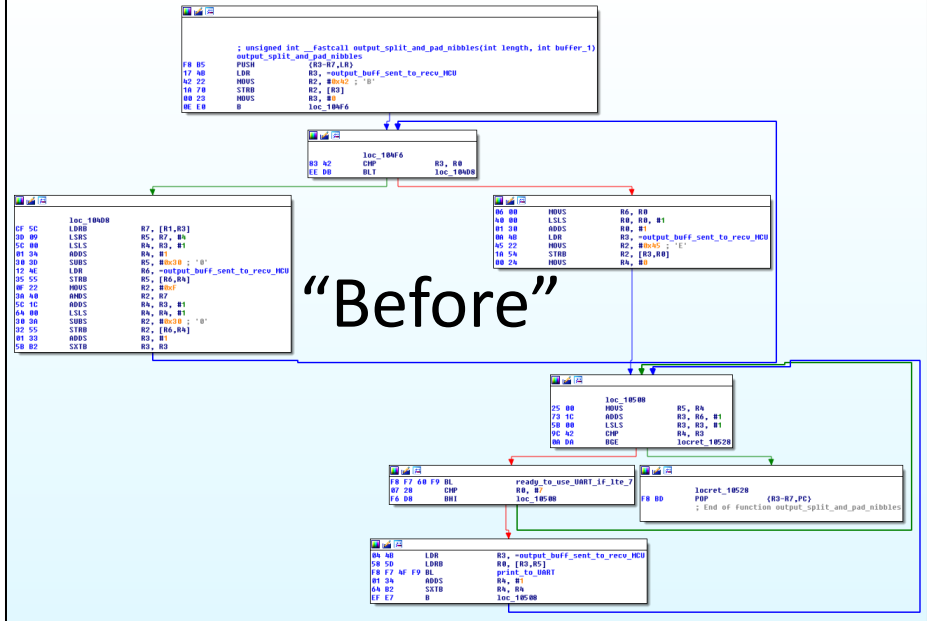
Can We Do Something More Interesting?

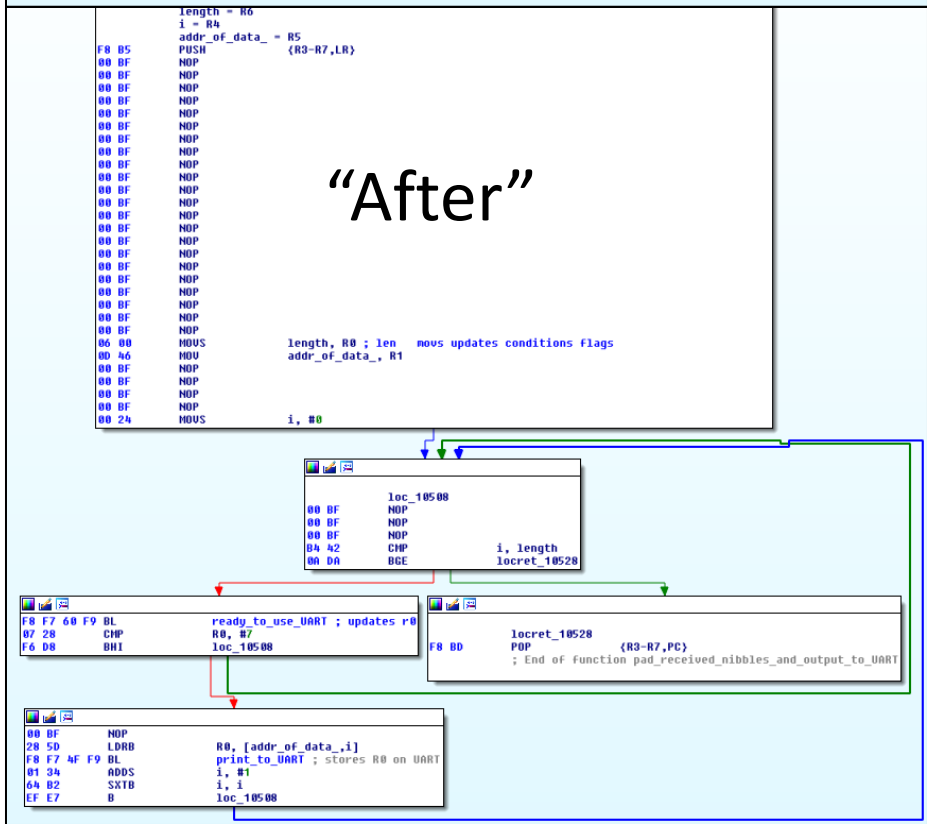
- dataBlob is 18 bytes but takes up 20 bytes of the stack, then there are 3 registers on the stack for 12 more bytes, then the LR register
  - We need to overwrite 32 bytes of junk, then up to 4 bytes of the LR
  - LR can only contain 000024CB* when the ring buffer is full
    - 25D7 after 1 packet read failure, 20xx-23xx when updating game state in non-interactive mode
  - Little endian lets us recycle the top bytes
- **One problem**: all data is padded with Dx
  - Send 36 bytes: LR = DxDxDxDx: Invalid
  - Send 35 bytes: LR = 00DxDxDx: Invalid
  - Send 34 bytes: LR = 0000DxDx: Data and a BXLR, not helpful
  - Send 33 bytes: LR = 000024Dx: 000024DD may return to waiting for a packet
    - Send 33 bytes: LR = 000025Dx: 000025DF might display ascii art
- I can crash a stock badge but not run arbitrary code ☹

* +1 for thumb 👍

Fixing the NFMI Protocol

- The only way around that was to cheat
  - Modify the victim badge's firmware to modify the NFMI firmware at boot up
- Found code that padded output with 0xDn
  - Removed that padding code
  - Removed the stupid 'B' and 'E' crap too
- Original game still works
  - Use 10 user data bytes to send "B", data, "E"





NFMI Proprietary Firmware Format

- Lastly I had to figure out the bizarre NFMI firmware format
  - 3 sections, each consists of
    - Data Segment
      - CAFEBABE (what is this, Java?)
      - Length
      - Base address
      - CRC-32/POSIX of header
      - Data (Length x 2 bytes)
      - Padding 0xFFFF, if Length is odd
      - Checksum/mixed-up-CRC
    - Additional data segments (optional)
    - End
      - 0 (Length)
      - 0 (Base)
      - Checksum/mixed-up-CRC
- Put that into the original firmware, time for some fun

10

- Live Demos of over-the-air remote code execution of arbitrary code
  - Demos that exist so far:
    - I can write an arbitrary string to the debug console, currently “Seth was here!!”
    - I can write a string from the badge memory to the console, like “Goon” or “Speaker”
  - Demos in progress
    - Play the RickRoll music
    - Play arbitrary sounds, perhaps SOS
    - Do something with the LEDs on the badge.
    - Trigger the ASCII art function in the badge
- Demo of POC
- Demo of oversized packet
- Demo of CRC table?

Oddities

- First packet data always contains the following in ASCII: “0403E045”
  - 0x45E00304 stored in NXH memory at 17DFC and 17E20 but not referenced by code
  - Might be a output buffer address computed by the NFMI…?
- Also got error packets with “0D047039”
  - 0x3970040D = ????
- NFMI firmware has the entries:

  00C0: 2281A100, “rev53481M”

  43E2: 2281A100, “rev53481MS”
- 2281A100 ≡ 0x00A18122 = $10584354_{10} \stackrel{?}{=} 10.584354$Mhz?
  - Or is it the rev values backwards? **1** 05 **8435** 4 backwards?

Remaining Mysteries: Preambles

- Preamble bytes 0-4 suggest signal could actually be a DD8PSK (DD9PSK?) because the differences between the first 20 symbols 4440 4040 2460 0000 6420 are:
  - ?000 4444 2222 0000 6666
- Preamble bytes 5-7 – what do they mean?
  - Assuming the mask is 0040 0000 0000 (20 00 00), preamble values are:
    - 0040 0000 0000 = 00 00 00
    - 0042 3133 4224 = C0 FF 55
    - 0044 2774 6756 = 80 87 00 ← the one used in packets
- Unknown if CRC protects any preamble bytes, or just the data
- I made packet with the section 2 preambles and all possible CRCs, none worked
  - Tried a lot of versions of the last 3 preambles bytes,
  - Occasionally one that was close to the original would work, sporadically
    - Probably because noise turned it back into the original preamble

Remaining Mysteries

- Where does the mask come from?
  - Tried all the first 20 or so PRBS, nada
  - Tried changing endianness, reversing the bits, splitting odd and even bits
    - Haven’t tried double-diffing or un-diffing the bits
- Must be an easy way to stream or send longer packets
  - Throughput right now is about 22 bytes per second…
- What the heck is that convolution?