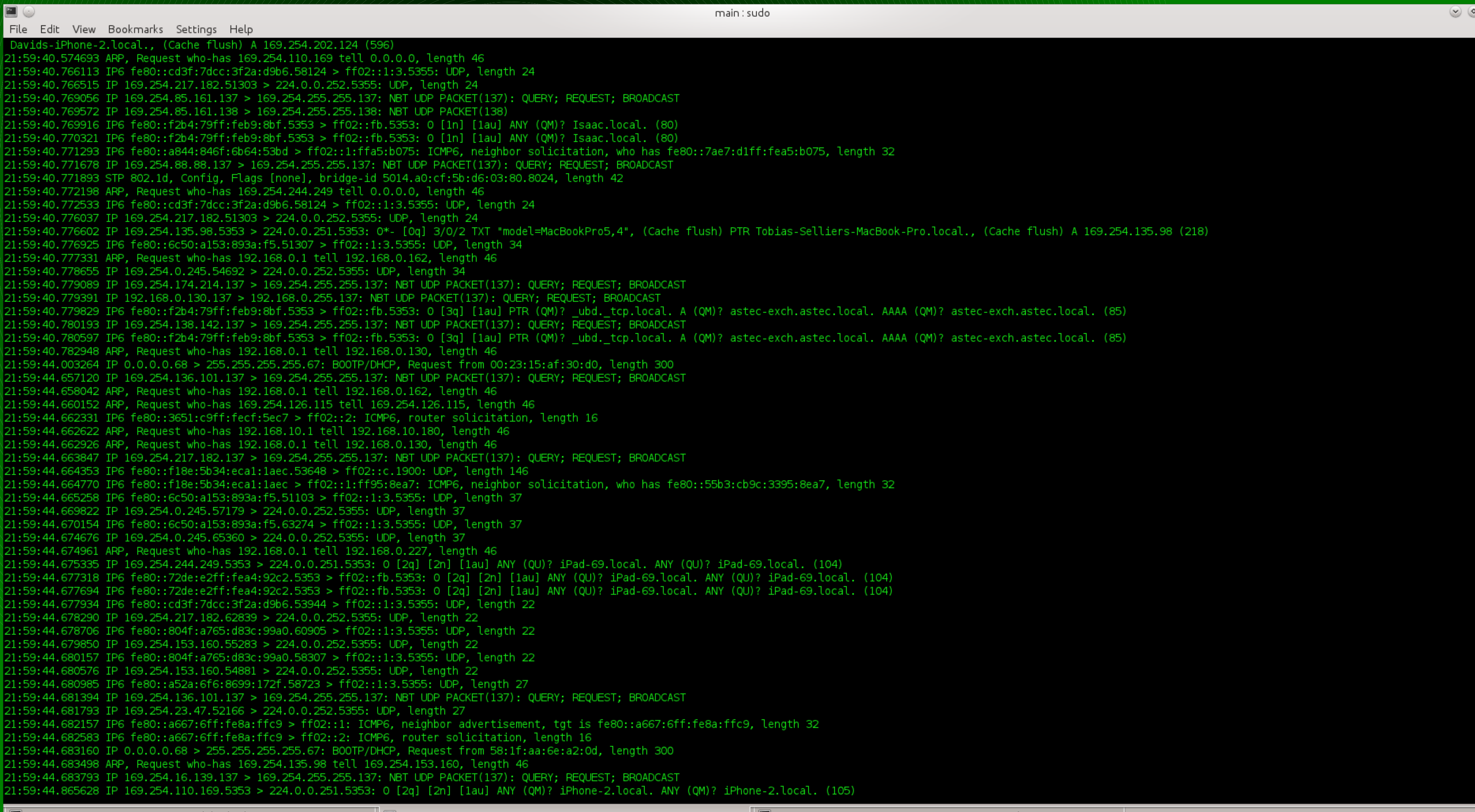
0x0000



“We are not as strong as we think we are”

- Rich Mullins

<GHz or bust!

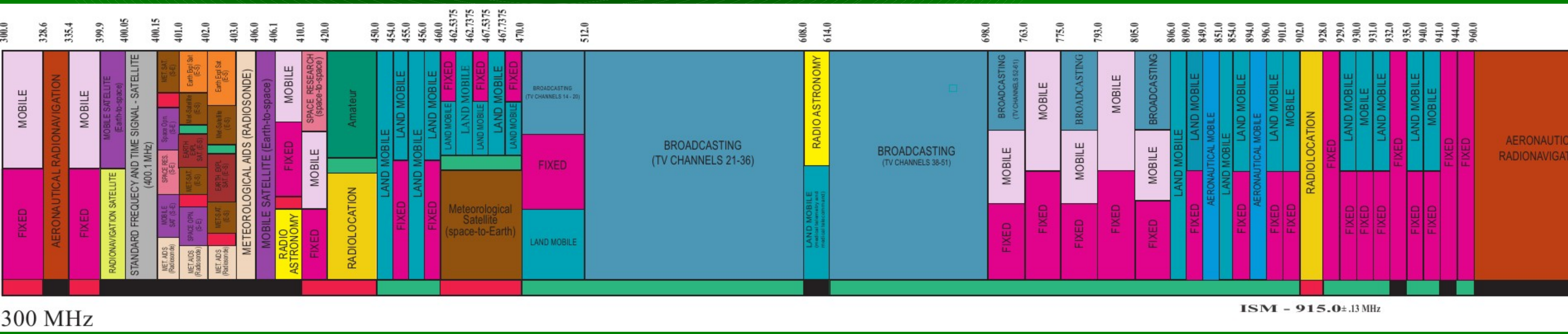
leveraging the power of the chipcon 1111
and RfCat

0x0001 – workshop plan - ejercicios

- lessons to teach:
  - play around with mods/baud/etc...
  - using the dongle to tune in and listen
  - using the dongle to determine, and transmit
  - playing with the dongle... it's just fun!
- toys to play with:
  - Garage door opener
  - Keyless entry fob
  - Power Meter
  - Glucometer
  - IMME 1
  - IMME 2 / dongle

0x0002 – installing the client

- once you have a cc1111 dongle flashed with RfCat...
- install client according the the README
- blackhat release:
  - https://rfcat.googlecode.com/files/rfcat-blackhat2012.tgz
  - https://rfcat.googlecode.com/files/rfcatChronos-bh12.hex
  - https://rfcat.googlecode.com/files/rfcatDons-bh12.hex



0x1000 – intro to <GHz and unlicensed ISM

- ITU-R 5.138, 5.150, and 5.280 rules
- US FCC Rules(title 47) parts 15 and 18
  - Industrial – power grid stuff and more!
  - Science – microwave ovens?
  - Medical – insulin pumps and the like
- Popular ISM bands (vary around the world)`:
  - 300 : 300 – 330 MHz
  - 433 : 433.050 – 434.790 MHz
  - 868 : 863.000 – 870.000 MHz
  - 915 : 902.000 – 928.000 MHz
  - cc1111 does 300-348, 372-460, 779-928... but we've seen more.
- Other ISM includes 2.4 GHz and 5.8 GHz
  - cc2531.... hmmm... maybe another toy?

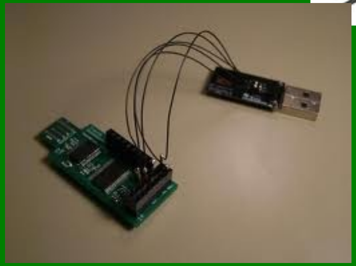
0x1010 – what plays <GHz?

- Industry, Science, Medical bands, US and EU
- Cell phones
- Cordless Phones
- Personal Two-Way Radios
- Car Remotes
- Pink IM-ME Girl Toys!
- TI Chronos Watches
- Medical Devices (particularly 401-402MHz, 402-405MHz, 405-406MHz)
- Power Meters
- custom-made devices
- Old TV Broadcast
- much, much more...

0x1020 – how do we play with it?

- cc1110/cc1111 do 300-348MHz, 391-464MHz, 782-928MHz
  - and more...
- RFCAT uses the CC111x on some common dongles

  - Chronos dongle (sold with every TI Chonos watch)
  - “Don's Dongles”, aka TI CC1111EMK
  - IMME (currently limited to sniffer/detection firmware)
- but there are some catches
  - rf comms configuration?
  - channel hopping sequence?
  - bluetooth and DSSS? (not hap'nin)



0x1030 – why do i care!?

- the inner rf geek in all of us
- your security research may require that you consider comms with a wireless device
- your organization may **have** 900MHz devices that should be protected!

0xE1 – workshop ej 1 – getting started

$ tar zxf rfcat-blackhat2012.tgz

$ cd rfcat-blackhat2012

$ sudo python setup.py install (trust me!)

$ rfcat -r

```
atlas@blah:~/hacking/Hardware/rfcat$ rfcat -r
'RfCat, the greatest thing since Frequency Hopping!'

Don't you wish this were a CLI!?  Sorry.  Maybe soon...
For now, enjoy the raw power of rflib, or write your own device-specific CLI!

currently your environment has an object called "d" for dongle.  this is how
you interact with the rfcat dongle, for :
    >>> d.ping()
    >>> d.setFreq(433000000)
    >>> d.setMdmModulation(MOD_ASK_OOK)
    >>> d.makePktFLEN(250)
    >>> d.RFxmit("HALLO")
    >>> d.RFrecv()
    >>> print d.reprRadioConfig()


In [1]:
```

0xE2 – workshop ej 2 – listen to teacher

```
$ rfcat -r
>>> d.setMdmModulation(MOD_ASK_OOK)
>>> d.setMaxPower()
>>> d.setMdmDRate(9600)
>>> d.makePktVLEN()                    # variable length packet
>>> d.RFlisten()
```

0x2000 - intro to the cc1111 core

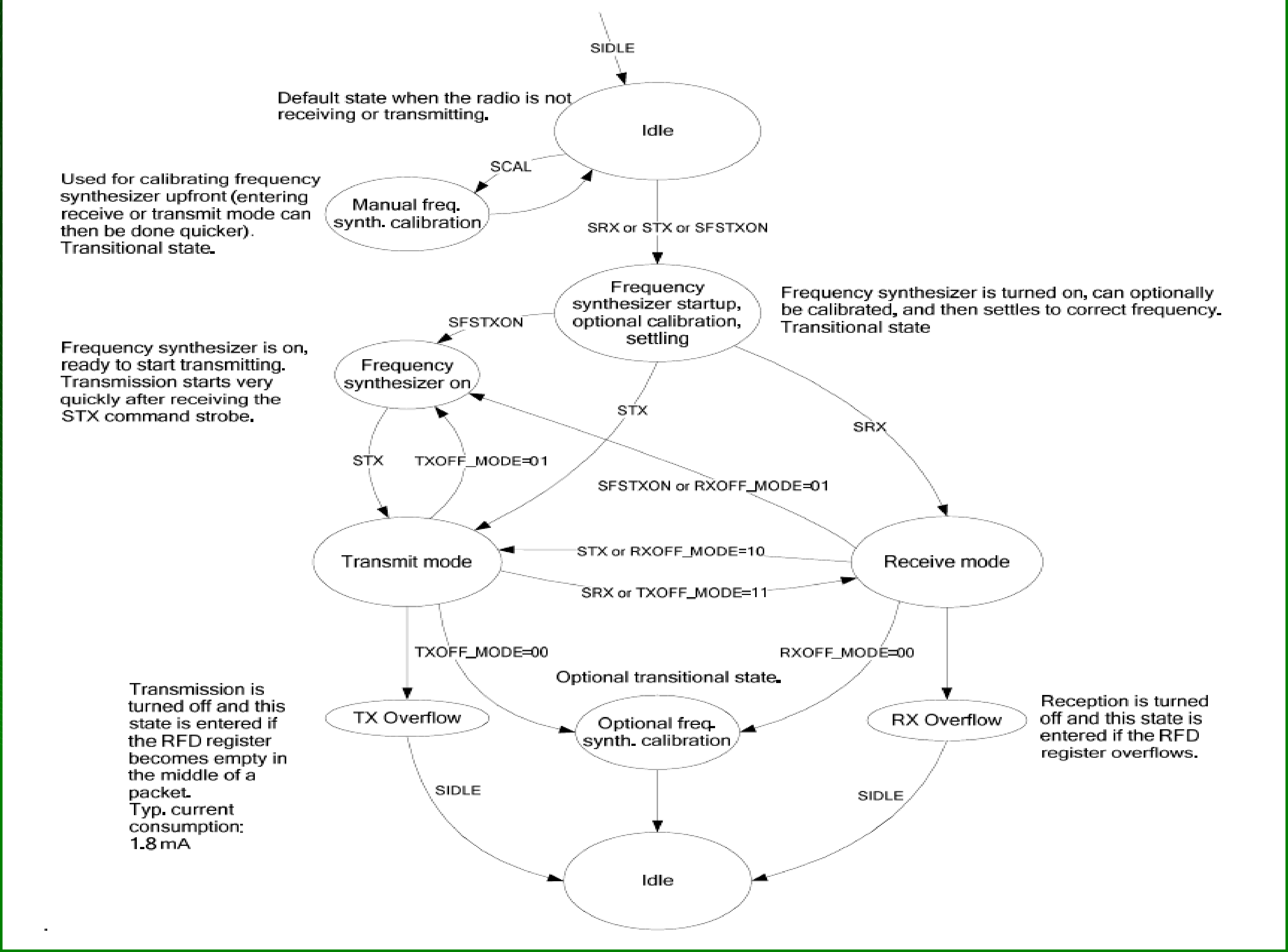
- for the devs in the house...
  - mcu
  - radio state engine
  - radio configuration
  - usb
  - timers
  - dma

0x2010 – cc1111 mcu

- modified 8051 core
  - 8-bit mcu
  - single-tick instructions
  - 256 bytes of iram
  - 4kb of xram
  - XDATA includes all code, iram, xram
  - execution happens anywhere :)
- register access to radio, dma, crypto, usb, timers, adc
- registers are simply memory locations is the XDATA address space

0x2020 – cc1111 radio state engine

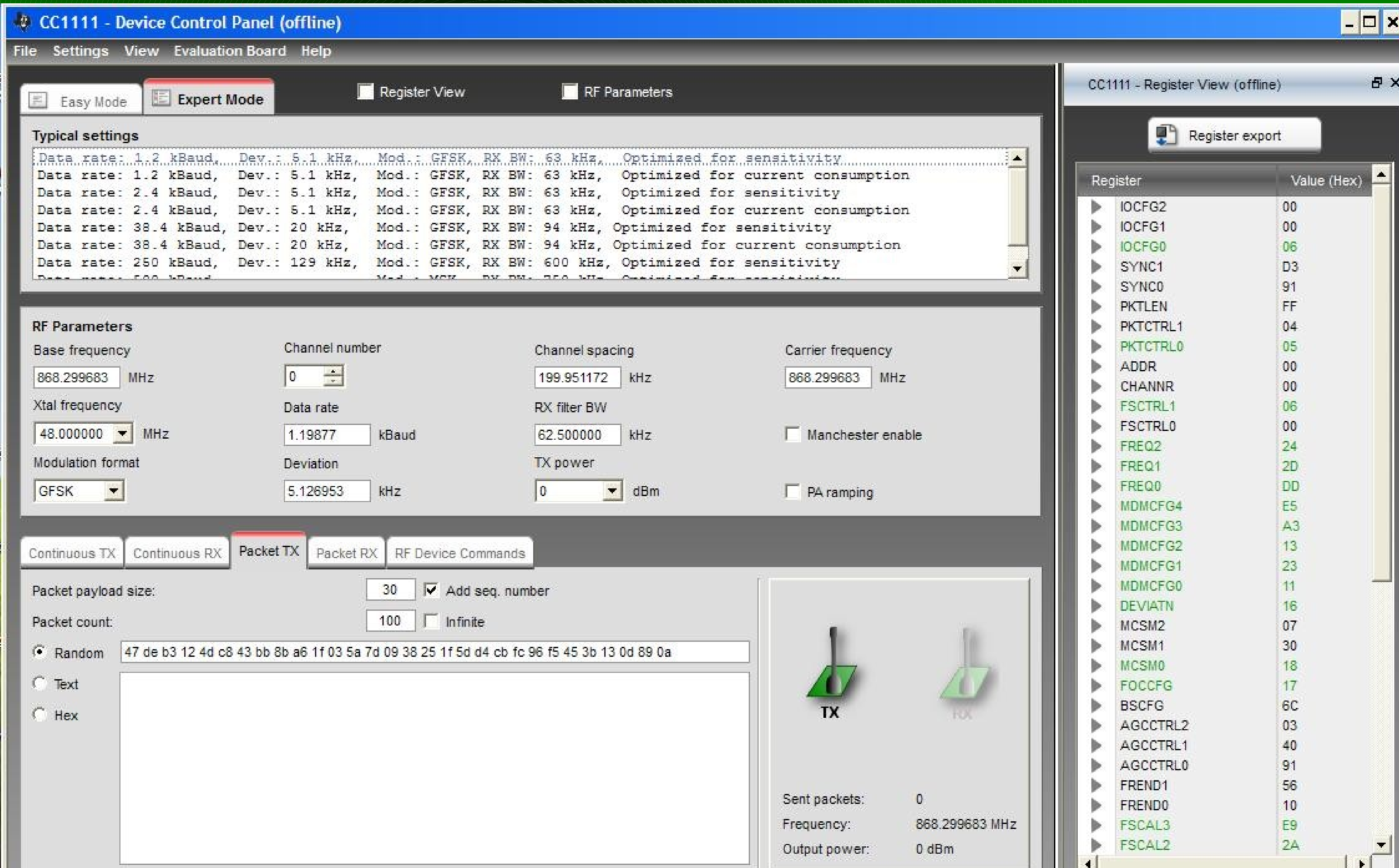
- IDLE
- CAL
- FSTXON
- RX
- TX



0x2030 – cc1111 radio configuration

- configuring the radio is done through updating a set of 1-byte registers in varying bit-size fields
  - MDMCFG4 – MDMCFG0 – modem control
  - PKTCTRL1, PKTCTRL0 – packet control
  - FSCTRL1, FSCTRL0 – frequency synth control
  - FREND1, FREND0 – front end control
  - FREQ2, FREQ1, FREQ0 – base frequency
  - MCSM1, MCSM0 – radio state machine
  - SYNC1, SYNC0 – SYNC word, or the SFD
  - CHANNR, ADDR – channel and address
  - AGCCTRL2, AGCCTRL1, AGCCTRL0 – gain control

0x2040 – Smart RF Studio (ftw)



0x2050 – cc1111 radio notes

- Data Rate, Bandwidth, and Intermediate Frequency and Freq-Deviation depend on each other

- put the radio in IDLE state before configuring
- put the radio in IDLE state before configuring
- put the radio in IDLE state before configuring

- STROBE (SIDLE, STX, SRX, SCAL...)
  - then wait for the MARCSTATE == MARC_STATE_whatever
- CCA impacts entering TX state from RX
  - but not from IDLE state

0x2060 – usb

- usb is a world unto itself, with a massive standard and substandards
  - gg: usb-in-a-nutshell
  - gg: usb complete jan axelson
- cc1111's usb controller is accessed using:
  - registers for config/control of usb
  - registers indicating usb events that occur
  - endpoint-specific FIFO buffers
    - messages go there before sending to host
    - messages arrive there from host
  - usb “descriptors” as necessary by spec
    - host uses these to query the device
- our firmware provides all this and more

0x2100 – RfCat for devs

- cc1111usb.c provides usb descriptors and framework
  - shouldn't need much tinkering
- cc1111rf.c provides the core of the radio firmware
  - shouldn't need much tinkering
- application.c provides the template for new apps
  - copy it and make your amazing toy
- txdata(buffer, length) to send data IN to host
- registerCbEP5OUT() to register a callback function to handle data OUT from host
  - data is in ep5iobuf[]
- transmit(*buf, length) allows you to send on the RF pipeline
- appMainLoop() – modify this for handling RF packets, etc...
- follow the examples, luke!
  - RfCat's “application” source is appFHSSNIC.c

0xE3 – heartfelt communication

- pick a friend (or set of friends)
- agree on who will xmit and who will recv

```
$ rfcat -r                 # common (xmit/recv)
>>> d.setMdmModulation(MOD_GFSK)
>>> d.makePktFLEN(20)
>>> d.setFreq( 915200000 )
--- recver ---
>>> d.RFlisten()
--- xmitter ---
>>> d.RFxmit(“hello my name is <name>”)
```

- what happened?

0xE3.5 – your closer friends...

- two problems: length and sync word!
- both xmitter and recver (not necessarily at once):
  - increase the packet length

```
>>> d.makePktFLEN(35)
```

  - now agree upon a 16-bit sync word (0 – 0xffff)

```
>>> d.setMdmSyncWord(<syncword>)
```

  - now try xmit/recv again
  - now reverse roles
  - now learn the power of the dark side!

```
>>> d.discover()
>>> d.discover(IdentSyncWord=True)
>>> help(d)      # ahhhhhhh.....
```

0x3000 – inquiring minds want to know!

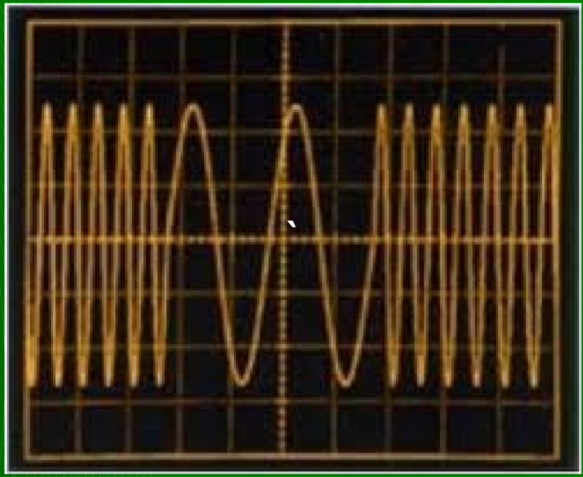
- frequencies
- modulation (2FSK/GFSK, MSK, ASK/OOK, other)
- intermediate frequency (IF)
- baud rate
- channel width/spacing/hopping?
- bandwidth filter
- sync words / bit-sync
- variable length/fixed length packets
- crc
- data whitening?
- any encoding (manchester, fec, enc, etc...)

0x3010 – interesting frequencies

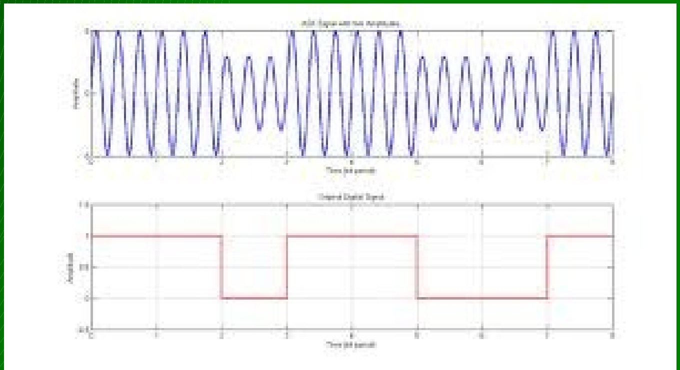
- 315MHz – car fobs, garage door openers
- 433MHz – medical devices, car fobs
- 868MHz – EU loves this range
- 915MHz – NA stuff of all sorts (power meters, insulin pumps, industrial plant equipment, industrial backhaul)

- (the rest are not covered by RfCat... yet)
- 2.4GHz – 802.11/wifi, 802.15.4/zigbee/6lowpan, bluetooth
- 5.8GHz – cordless phones
- FREQ2, FREQ1, FREQ0

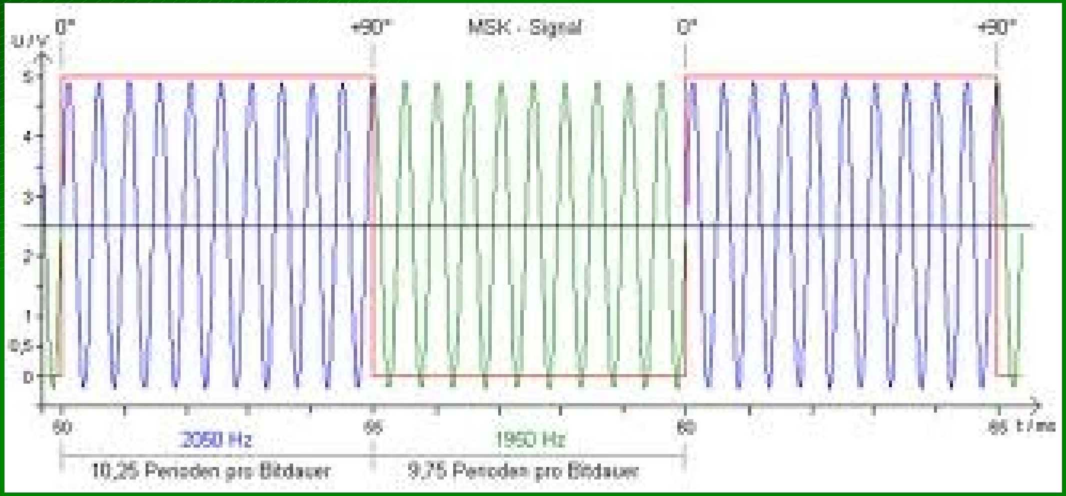
0x3020 – modulations



- 2FSK/GFSK – Frequency Shift Key
  - (digital FM)
  - power meters, DECT/CT2



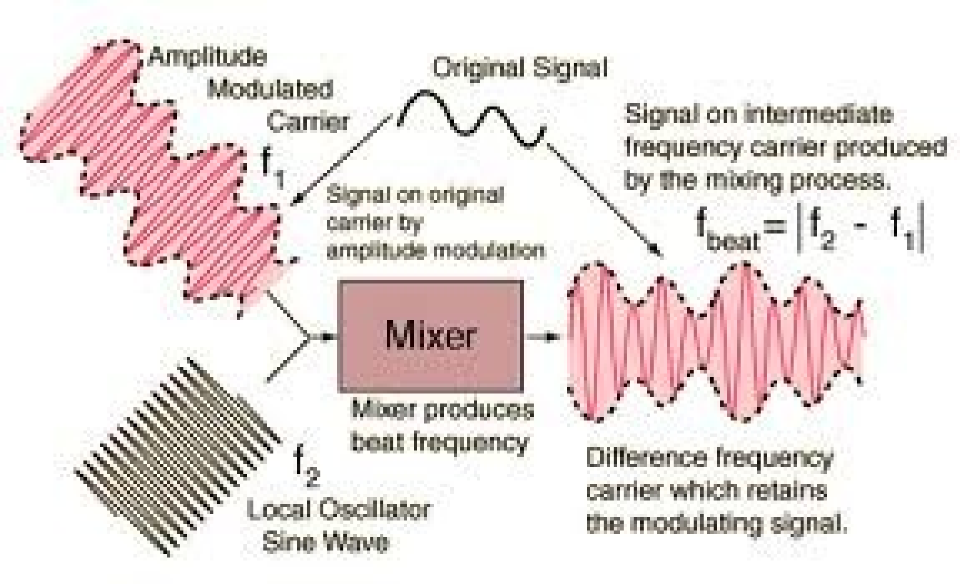
- ASK/OOK – Amplitude Shift Key
  - (digital AM)
  - morse-code, car-remotes, etc...
- MSK – Minimal Shift Key (a type of quadrature shift modulation like QPSK)
  - GSM



- MDMCFG2, DEVIATN

0x3030 – intermediate frequency

- mix the RF and LO frequencies to create an IF (heterodyne)
  - improves signal selectivity
  - tune different frequencies to an IF that can be manipulated easily
  - cheaper/simpler components
- cc1111 supports a wide range of 31 different IF options:
  - 23437 hz apart, from 0 – 726.5 khz
- Smart RF Studio recommends:
  - 140 khz up to 38.4 kbaud
  - 187.5 khz at 38.4 kbaud
  - 281 khz at 250 kbaud
  - 351.5khz at 500 kbaud
- FSCTRL1



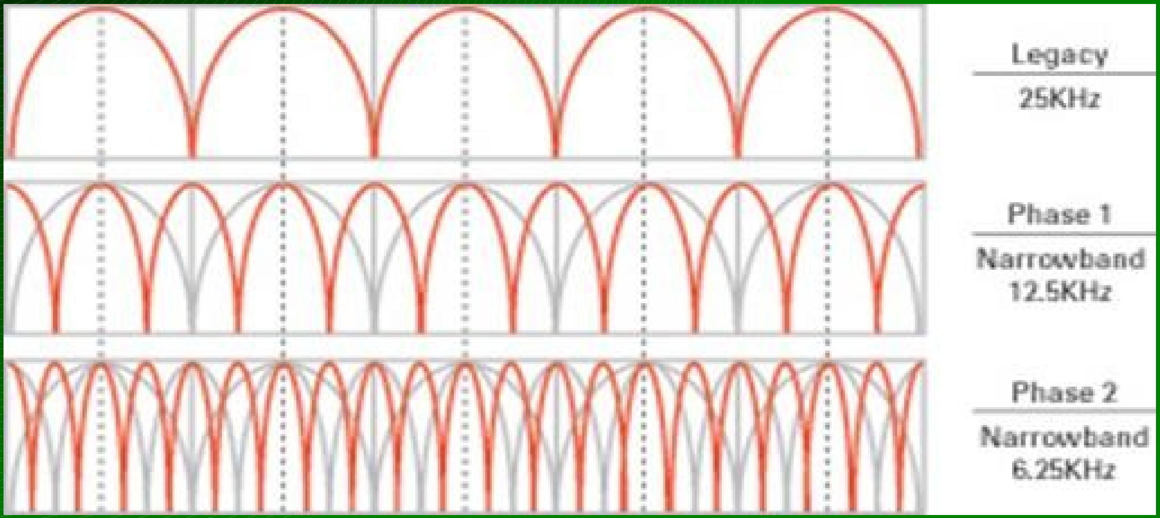
0x3040 – data rate (baud)

- much like your modems or old
- the frequency of bits
  - some can overlap and get garbage!
    - garbage can be good...
- baud has significant impact on IF, Deviation and Channel BW
- seeing much use of 2400, 19200, 38400, 250000
  - and other oddballs like 4760
- MDMCFG3 / 4

0x3050 – channel width / spacing

- simplifying frequency hopping / channelized systems
- real freq = base freq + (CHANNR * width)

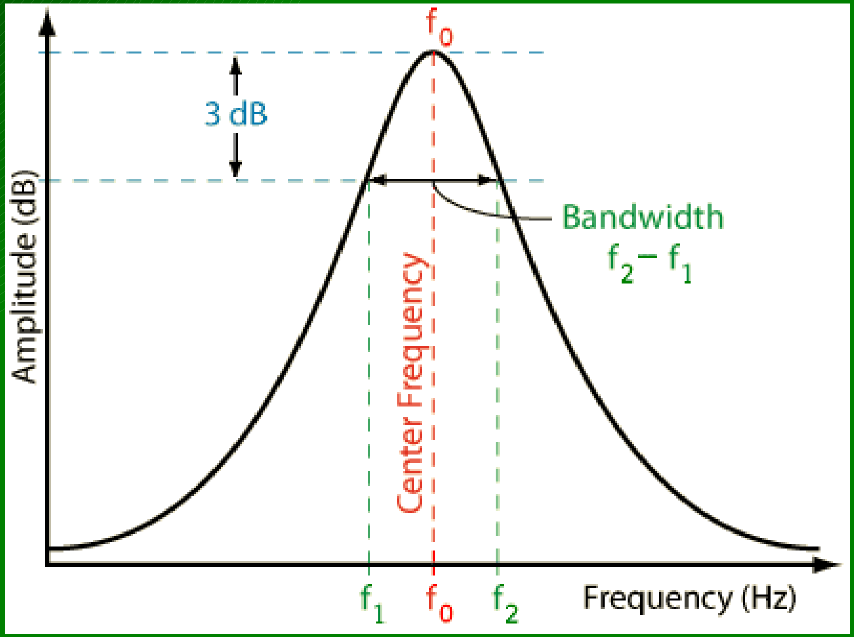
- MDMCFG0 / 1



0x3060 – bandwidth filter

- programmable receive filter
- provides for flexible channel sizing/spacing

- total signal bw = signal bandwidth + (2*variance)
- total signal bw **wants** to be less than 80% bw filter!
- MDMCFG4



0x3070 – preamble / sync words

- identify when real messages are being received!
- starts out with a preamble (1 0 1 0 1 0 1 0...)
- then a sync word (programmable bytes)
  - marking the end of the preamble
  - aka 'SFD' – start of frame delimiter
- configurable to:
  - nothing (just dump received crap)
  - carrier detect (if the RSSI value indicates a message)
  - 15 or 16 bits of the SYNC WORD identified
  - 30 out of 32 bits of double-SYNC WORD
- SYNC1, SYNC0, MDMCFG2

0x3080 – variable / fixed-length packets

- packets can be fixed length or variable length
- variable length assumes first byte is the length byte
- both modes use the PKTLEN register:
  - Fixed: the length
  - Variable: MAX length
- soon, with contributions from Major Malfunction:
  - Infinite mode (very long FIXED length, eg. 1024)

- PKTCTRL0, PKTLEN

0x3090 – CRC – duh, but not

- crc16 check on both TX and RX
- uses the internal CRC (part of the RNG) seeded by 0xffff
- DATA_ERROR flag triggers when CRC is enabled and fails
- some systems do this in firmware instead

- PKTCTRL0

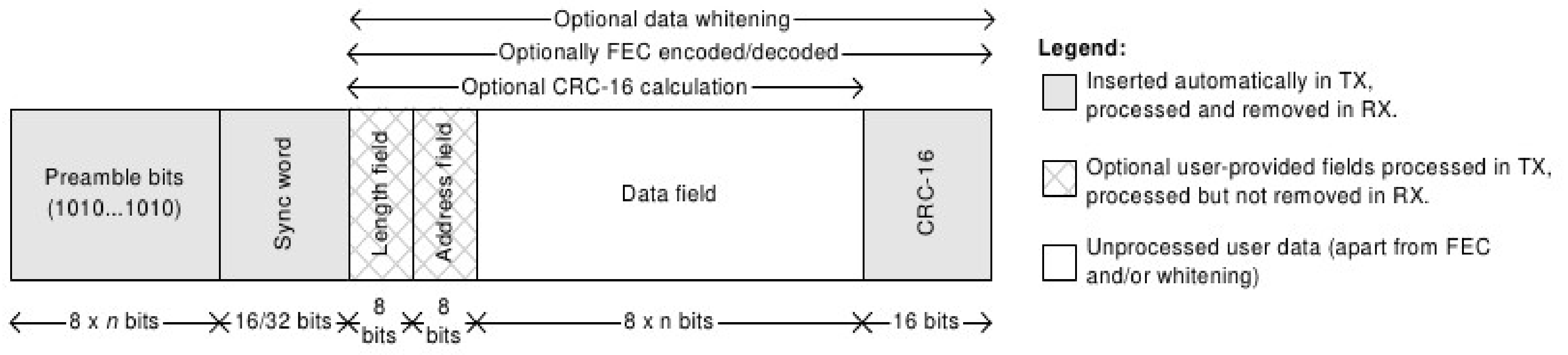
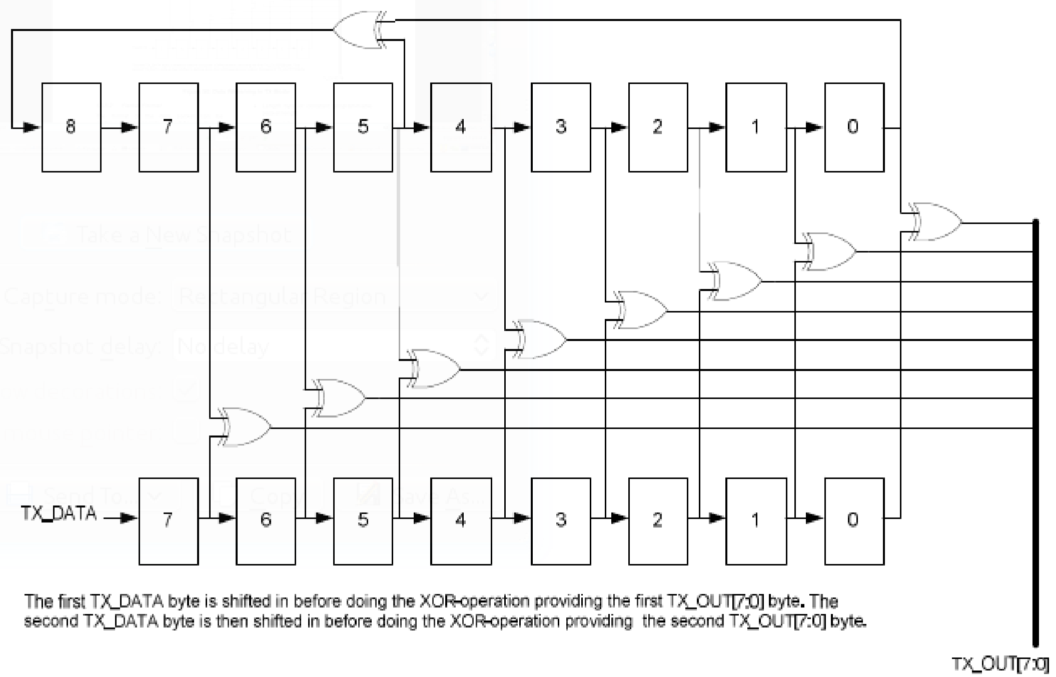


Figure 51: Packet Format

0x30a0 – data whitening – 9 bits of pain

- ideal radio data looks like random data
- real world data can contain long sequences of 0 or 1
- data to be transmitted is first XOR'd with a 9-bit sequence
  - sequence repeated as many times as necessary
- PKTCTRL0

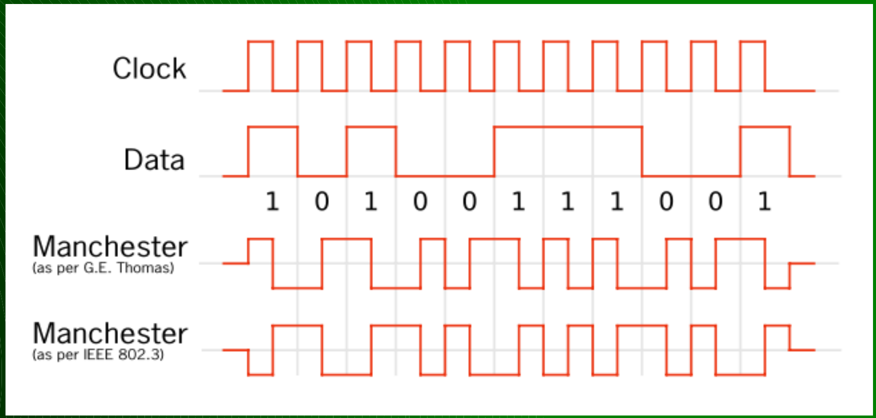


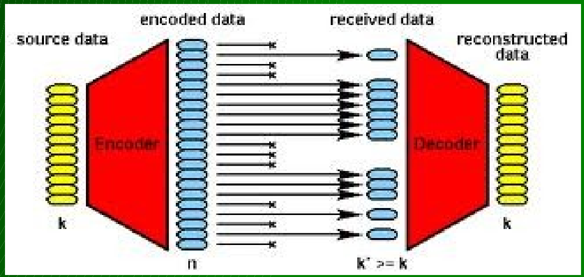
The first TX_DATA byte is shifted in before doing the XOR-operation providing the first TX_OUT[7:0] byte. The second TX_DATA byte is then shifted in before doing the XOR-operation providing the second TX_OUT[7:0] byte.

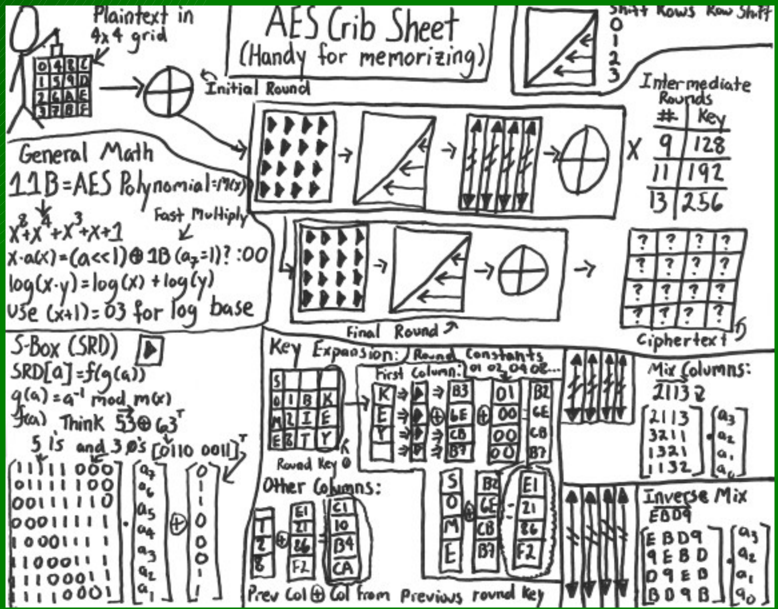
Figure 50: Data Whitening in TX Mode

0x30b0 – encoding

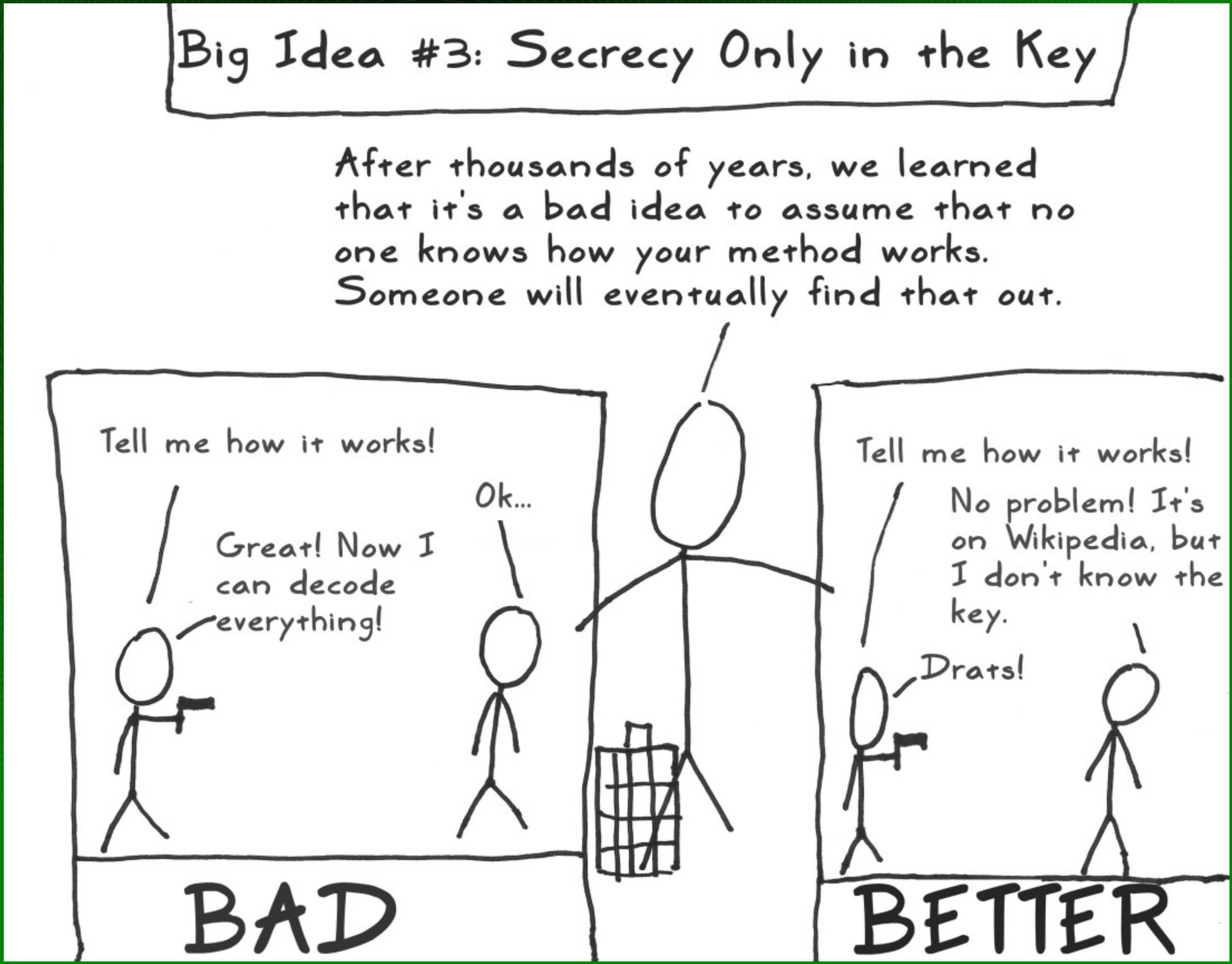
- manchester
  - MDMCFG2
- forward error correction
  - convolutional
    - MDMCFG1
  - reed-solomon (not supported)
- encryption - AES in chip







sorry, couldn't resist



0xE4 - mismatching

- xmitter and recver pick the same config (last ej)
- select random quiet frequency (same as each other)
- xmitter change something and transmit, talk to recver (use your mouth) to discuss results

```
>>> d.setMdmModulation(MOD_*)
 (MOD_ASK_OOK, MOD_GFSK, MOD_2FSK, MOD_MSK)
>>> d.setMdmDRate (baud)
>>> d.setMdmDeviatn(<deviation_number>)
>>> d.makePktFLEN() and d.makePktVLEN()  # vary len too!
>>>
```

0x30c0 – MDMCFG2 register (example)

0xDF0E: MDMCFG2 - Modem Configuration

Bit	Field Name	Reset	R/W	Description
7	DEM_DCFILT_OFF	0	R/W	Disable digital DC blocking filter before demodulator. The recommended IF frequency changes when the DC blocking is disabled. Please use SmartRF® Studio [9] to calculate correct register setting. 0 – Enable – Better Sensitivity 1 – Disable – Current optimized. Only for data rates ≤ 100 kBaud
6:4	MOD_FORMAT[2:0]	000	R/W	The modulation format of the radio signal 000 – 2-FSK 001 – GFSK 010 – Reserved 011 – ASK/OOK 100 – Reserved 101 – Reserved 110 – Reserved 111 – MSK Note that MSK is only supported for data rates above 26 kBaud and GFSK, ASK , and OOK is only supported for data rate up until 250 kBaud. MSK cannot be used if Manchester encoding/decoding is enabled.
3	MANCHESTER_EN	0	R/W	Manchester encoding/decoding enable 0 – Disable 1 – Enable Note that Manchester encoding/decoding cannot be used at the same time as using the FEC/Interleaver option or when using MSK modulation.
2:0	SYNC_MODE[2:0]	010	R/W	Sync-word qualifier mode. The values 000 and 100 disables preamble and sync word transmission in TX and preamble and sync word detection in RX. The values 001, 010, 101 and 110 enables 16-bit sync word transmission in TX and 16-bits sync word detection in RX. Only 15 of 16 bits need to match in RX when using setting 001 or 101. The values 011 and 111 enables repeated sync word transmission in TX and 32-bits sync word detection in RX (only 30 of 32 bits need to match). 000 – No preamble/sync 001 – 15/16 sync word bits detected 010 – 16/16 sync word bits detected 011 – 30/32 sync word bits detected 100 – No preamble/sync, carrier-sense above threshold 101 – 15/16 + carrier-sense above threshold 110 – 16/16 + carrier-sense above threshold 111 – 30/32 + carrier-sense above threshold

0x3100 – how can we figure it out!?

- open / public documentation
  - insulin pump published frequency
- open source implementation / source code
- “public” but harder to find (google fail!)
  - fcc.gov – search for first part of FCC ID
    - http://transition.fcc.gov/oet/ea/fccid/ -bookmark it
  - patents – amazing what people will patent!
    - http://freepatentsonline.com
    - french patent describing the whole MAC/PHY of one meter
    - and another:
      http://www.freepatentsonline.com/8189577.html
      http://www.freepatentsonline.com/20090168846.pdf

0x3101 – how can we figure it out!? -part2

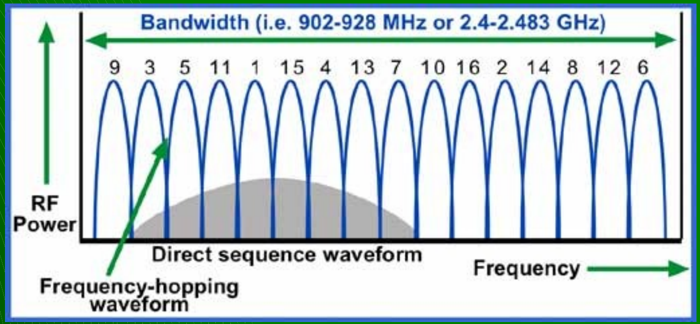
- reversing hw
  - tapping bus lines – logic analyzer
    - grab config data
    - grab tx/rx data
  - pulling and analyzing firmware
- hopping pattern analysis
  - arrays of dongles – space them out and record results
  - hedyattack, or something similar
  - spectrum analyzer
  - USRP2 or latest gadget from Michael Ossman
- trial and error – rf parameters
- MAC layer? - takes true reversing.. unless you find a patent :)

0xE5 – spectrum analysis

```
>>> d.specan(902000000, 50000, 150)
```

<close the spectrum analysis window to end this mode>

0x4000 – intro to FHSS



- FHSS is common for devices in the ISM bands
  - provides natural protection against unintentional jamming /interferance
  - US Title 47 CFR 15.247 affords special power considerations to FHSS devices
    - >25khz between channels
    - pseudorandom pattern
    - each channel used equally (avg) by each transmitter
    - if 20db of hopping channel < 250khz:
      - must have at least 50 channels
      - average <0.4sec per 20 seconds on one channel
    - if 20dB of hopping channel >250khz:
      - must have at least 25 channels
      - average <0.4sec per 10 seconds on one channel

0x4010 – FHSS, the one and only - NOT!

- different technologies:
  - DSSS – Direct Sequence Spread Spectrum
    - hops happen more often than bytes (ugh)
    - typically requires special PHY layer
  - "FHSS"
    - hops occur after a few symbols are transmitted
- different topologies: (allow for different synch methods)
  - point-to-point (only two endpoints)
  - multiple access systems (couple different options)
    - each cell has their own hopping pattern
    - each node has own hopping pattern
- different customers:
  - military has used frequency hopping since Hedy and George submitted the patent in 1941.
  - commercial folks (WiFi, Bluetooth, proprietary stuff like power meters)

0x4020 – FHSS intricacies

- what's so hard about FHSS?
  - must know or be able to come up with the hopping pattern
    - can be anywhere from 50 to a million distinct channel hops before the pattern repeats (or more)
    - some systems use one pattern per cell, others per node
  - must be able to synchronize with an existing cell or partner
    - or become your own master!
  - must know how to discover other nodes (
  - must know channel spacing
  - must know channel dwell time (time to sit on each channel)
  - likely need to reverse engineer your target
  - DSSS (eg. Bluetooth) requires that you have special hardware
- military application will be very hard to crack, as it typically will have hops based on a synchronized PRNG to select channels

0x4030 – FHSS, the saving graces

- any adhoc FHSS multi-node network: (power meters / sensor-nets)
  - node sync in a reasonable timeframe
    - limited channels in the repeated pattern
  - each node knows how to talk to a cell
    - let one figure it out, then tap the SPI bus to see what the pattern is...
- two keys to determining hopping pattern:
  - hop pattern generation algorithm
    - often based on the CELL ID
      - one pattern gets you the whole cell :)
    - others generate a unique pattern per node
  - some sync information the cell gives away for free
    - gotta tell the n00bs how to sync up, right?
    - for single-pass repeating sequences, it's just the channel

0x4040 – FHSS summary

- FHSS comes in different forms for different uses and different users
- FHSS is naturally tolerant to interference, and allows a device to transmit higher power than nonFHSS comms
- getting the FHSS pattern, timing, and appropriate sync method for proprietary comms can be a reversing challenge
- getting a NIC to do something with the knowledge gained above has – to date – been very difficult

0x5000 – intro to RfCat

- RfCat: RF Chipcon-based Attack Toolset
- background...
- goals...
- plans...
- where we're at so far...

0x5010 – rfcat background

- the power grid
  - power meters and the folks who love them (yo cutaway, q, travis and josh!)
  - no availability of good attack tools for RF
- vendor at Distributech 2008:

  "Our Frequency Hopping Spread Spectrum is too fast for hackers to attack."

  - OMFW! really?

0x5020 – rfcat goals

- RE tools - “how does this work?”
- security analysis tools - “your FHSS and Crypto is weak!”
- satiate my general love of RF

- a little of Nevil Maskelyne
- “I will not demonstrate to any man who throws doubt upon the system” - Guglielmo Marconi, 1903
  - lulwut?

0x5030 – this is not HedyAttack

- but leveraged the knowledge from HA...
- cc1111usb is the base code which HedyAttack started
  - forms the USB base for RfCat
- less "researchy"
  - this project won't **find** hopping patterns
  - it's goal is to provide you something to do with that infoz
    - “so, we determined this hopping pattern... now what?”
- more utilitarian
  - give us comms parameters and a hopping pattern, and we'll be a NIC, sniffer, and interact with RF gadgets
  - some devices will require more customization than other

0x5040 – rfcat's interface

- rfcat is many things, but I like to think of it as an interactive python access to the <GHz spectrum!

- rfcat
  - FHSS-capable NIC
    - some assembly may be required for FHSS to arbitrary devices
  - toolset for discovering/interfacing with RF devices
- rfcat_server
  - access the <GHz band over an IP network or locally
  - connect to tcp port 1900 for raw data channel
  - configure on the fly (TCP port 1899)

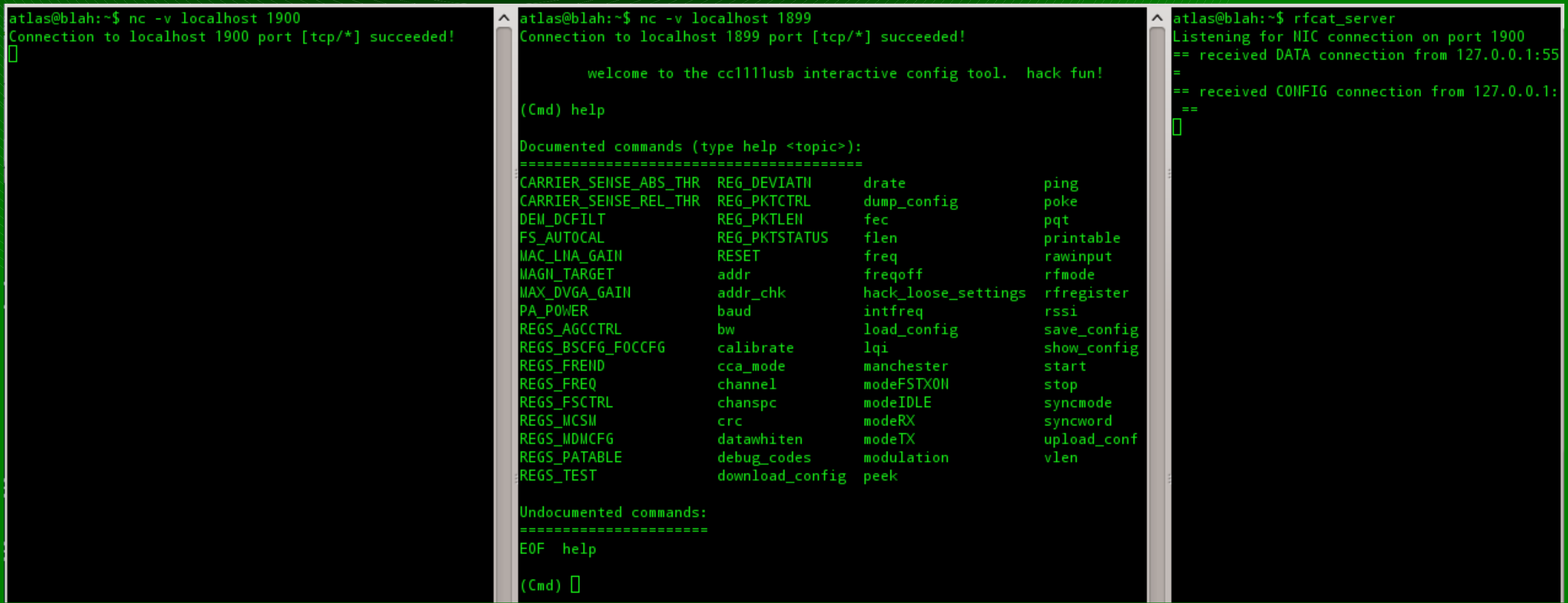
0x5050 – rfcat

- customizable NIC-access to the ISM bands
- lame “default” interface
- “research mode” is where the power is…
- ipython for best enjoyment
- lame spoiler: you get a global object called “d” to talk to RfCat
  - d.RFxmit('blah')
  - data = d.RFrecv()
  - d.discover(lowball=1)
  - d.RFlisten()/d.RFcapture()
  - d.specan()
  - help(d)

```
atlas@blah:~$ rfcat -r
'RfCat, the greatest thing since Frequency Hopping!'

Don't you wish this were a CLI!?  Sorry.  Maybe soon...
For now, enjoy the raw power of rflib, or write your own device-specific CLI!

currently your environment has an object called "d" for dongle.  this is how
you interact with the rfcat dongle, for :
    >>> d.ping()
    >>> d.setFreq(433000000)
    >>> d.setMdmModulation(MOD_ASK_OOK)
    >>> d.makePktFLEN(250)
    >>> d.RFxmit("HALLO")
    >>> d.RFrecv()
    >>> print d.reprRadioConfig()


In [1]: d.ping()
PING: 26 bytes transmitted, received: 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' (0.027489 seconds)
PING: 26 bytes transmitted, received: 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' (0.011954 seconds)
PING: 26 bytes transmitted, received: 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' (0.012381 seconds)
PING: 26 bytes transmitted, received: 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' (0.012189 seconds)
PING: 26 bytes transmitted, received: 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' (0.012411 seconds)
PING: 26 bytes transmitted, received: 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' (0.012139 seconds)
PING: 26 bytes transmitted, received: 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' (0.012379 seconds)
PING: 26 bytes transmitted, received: 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' (0.012392 seconds)
PING: 26 bytes transmitted, received: 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' (0.011946 seconds)
PING: 26 bytes transmitted, received: 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' (0.011591 seconds)
Out[1]: (10, 0, 0.13894200325012207)
```

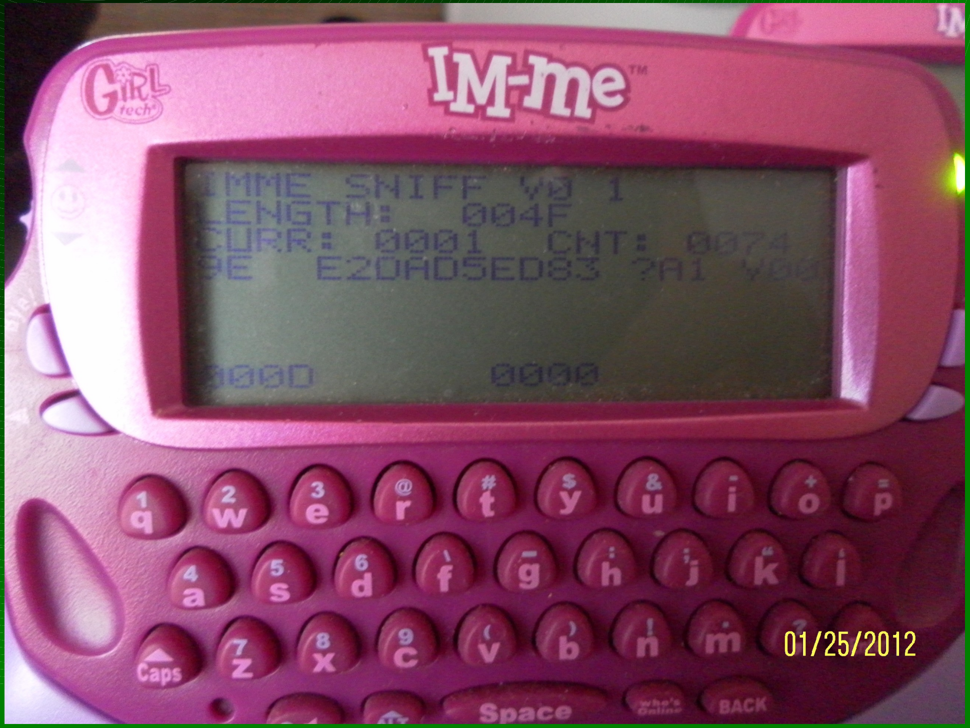
0x5060 – rfcat_server

- bringing <GHz over the IP network!
- connect on TCP port 1900 to access the wireless network
- connect on TCP port 1899 to access the wireless configuration
- created to allow non-python clients to play too
  - stdin is not always the way you want to interact with embedded wireless protocols



0x5070 – rfsniff (pink version too!)

- focused primarily on capturing data from the wireless network
- IMME used to provide a nice simple interface
- RF config adjustment using keyboard!

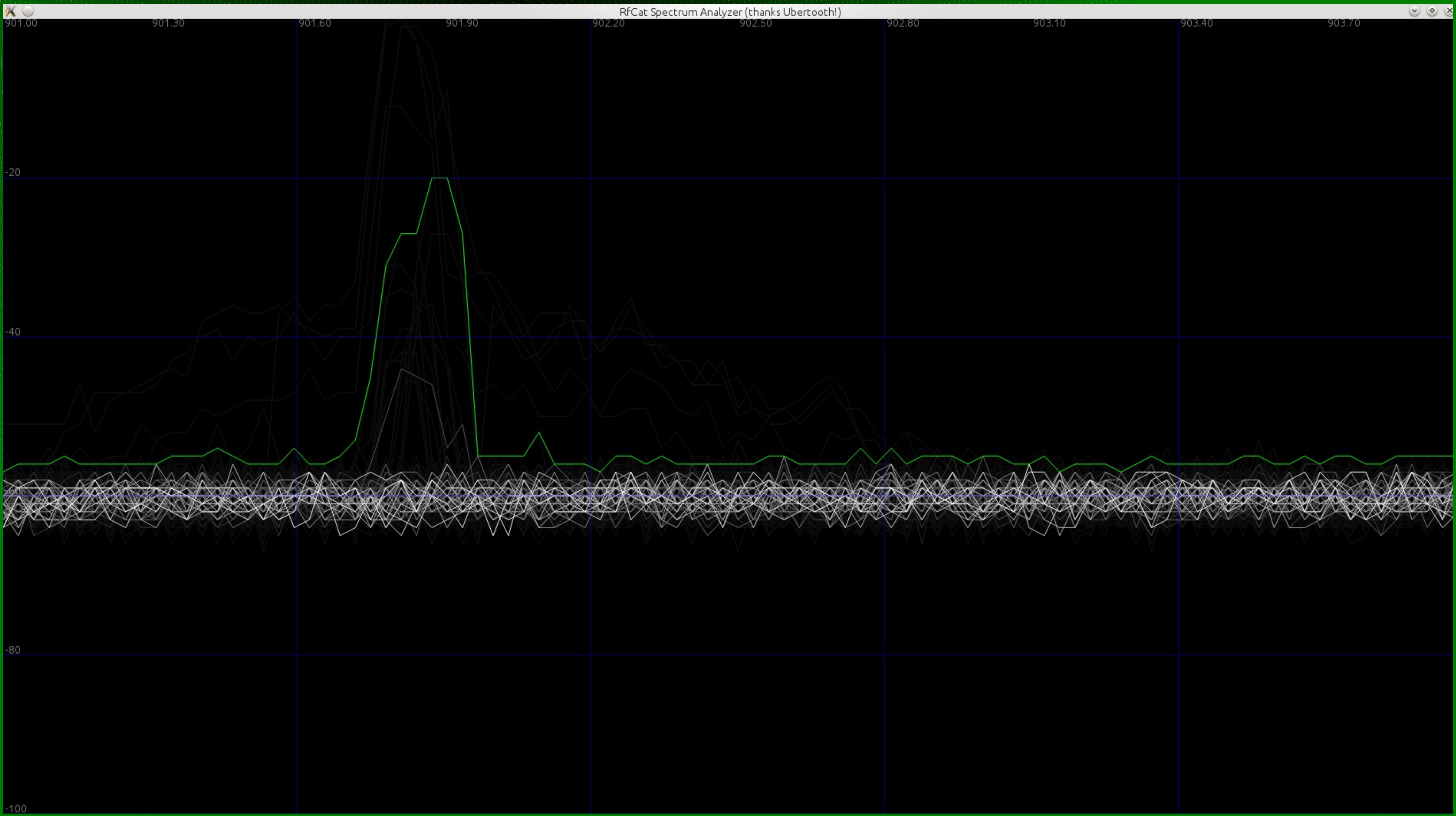


0x5065 – rfsniff – key bindings

```
q, a - inc/dec highest sync word nibble
w, s - inc/dec high-middle sync word nibble
e, d - inc/dec low-middle sync word nibble
r, f - inc/dec lowest sync word nibble
z    - NO sync word matching

menu - inc Modulation type
bye! - dec Modulation type

up   - inc recv bandwidth
down - dec recv bandwidth

right - inc baudrate
left - dec baudrate

p, Enter - inc/dec frequency
o, ',' - faster inc/dec frequency
i, m   - even faster inc/dec frequency
l    - set freq to 915mhz
k    - set freq to 868mhz
j    - set freq to 433mhz
h    - set freq to 315mhz
t, v - inc/dec channels
g    - set channel = 0


SPACE - switch screens
SPKR  - toggle CARRIER TX mode (good for showing up on a SpecAn, or, umm, jamming?)
```

0x5070 – spectrum analaysis (added 7/21/12)

>>> d.specan(901e6, 5e5, 51)



0x5075 – specan limitations

- currently SpecAn uses Channel mode
  - lowest channel spacing: ~23.4khz
  - highest channel spacing: 250khz
- currently SpecAn limits number of sampled freqs to 255
  - recommend 100 or so samples
- still... not bad.
- many thanks to the ubertooth team for the GUI base!

0xE6 – lowball, discovery, and scanning

- enter lowball mode (which stores config)

```
>>> d.lowball()      # (SYNCM_CARRIER)
>>> print d.reprClientState()
>>> d.lowballRestore()   # restore the config
>>> d.RFrecv()  # and again, until you receive a timeout error
```

- now use lowball level 0:

```
>>> d.lowball(0)    # dumps all sorts of crap (SYNCM_NONE)
>>> print d.reprRadioConfig()
>>> d.lowballRestore() # restores original config
>>> d.RFrecv()      # grab raw packet
>>> d.recvAll(APP_NIC, NIC_RECV) # dump all buffered pkts
```

0xE6.1 – discover mode

- now use discover()

```
>>> d.discover()    # press <enter> to leave discover mode
>>> d.discover(lowball=0)    # what do you see?
>>> d.discover(IdentSyncWord=True) # what's that?!
>>> d.discover(SyncWordMatchList=[0x0c4e, 0xf432])
```

0xE6.2 – looking for trouble (kick in a door)

- frequency scanning (based on lowball)

>>> d.scan(basefreq, inc, count, delaysec, drate, lowball)

>>> d.scan(902e6, 250e3, 104, 2, 38400, 1)

```
In [1]: d.scan()
Scanning range:

Scanning for frequency 902000000...
Scanning for frequency 902250000...
Scanning for frequency 902500000...
Scanning for frequency 902750000...
Scanning for frequency 903000000...
Scanning for frequency 903250000...
Scanning for frequency 903500000...
Scanning for frequency 903750000...
Scanning for frequency 904000000...
Scanning for frequency 904250000...
Scanning for frequency 904500000...
Scanning for frequency 904750000...
Scanning for frequency 905000000...
Scanning for frequency 905250000...
```

```
Scanning for frequency 909500000...
Scanning for frequency 909750000...
Scanning for frequency 910000000...
Scanning for frequency 910250000...
Scanning for frequency 910500000...
Scanning for frequency 910750000...
Scanning for frequency 911000000...
Scanning for frequency 911250000...
Scanning for frequency 911500000...
Scanning for frequency 911750000...
Scanning for frequency 912000000...
(1342153528.208) Receiving:  ee97bbb435b8b5a32624414f9fabc96f1f5430f456e1ef87ea
eb2db28113cafb7bf7c73f74e8889dadb3d5e3e11c8ddde9c676431ccd1d1b792ed108677f76fb2
(1342153528.261) Receiving:  62b4ecc96cced1e628498e31282f732085a45ec3567f7f43ec
8ae16bf961975028ba0248aa7fbf99182b9cf6abf54ae209b4f3ed5918e6d74a5e827fdf6ad70eb
```

0x5080 – rfcat wicked coolness

- d._debug = 1 – dump debug messages as things happen
- d.debug() - print state infoz once a second
- d.discover() - listen for specific SYNCWORDS
- d.lowball() - disable most “filters” to see more packets
- d.lowballRestore() - restore the config before calling lowball()
- d.RFlisten() - simply dump data to screen
- d.RFcapture() - dump data to screen, return list of packets
- d.scan() - scan a configurable frequency range for “stuff”d
- d.specan() - spectum analysis (graphics thanks to Ubertooth folks)
- print d.reprRadioConfig() - print pretty config infoz

0x5090 – lowball and discover

- lowball mode stores current radio config

```
>>> d.lowball()      # drops most blocks to pkts (CARRIER)
>>> d.lowballRestore() # returns original config
>>> d.lowball(0)     # dumps all sorts of crap (SYNCM_NONE)
>>> d.lowball(1)     # default... same as no argument
```

- discover() uses lowball mode, adds value

d.discover(lowball, debug, length, IdentSyncWord, SyncWordMatchList)

```
>>> d.discover()     # enters lowball mode, dumps pkts
>>> d.discover(lowball=0)    # dumps way more pkts
>>> d.discover(IdentSyncWord=True)
>>> d.discover(SyncWordMatchList=[0xdead, 0xbeef])
```

0x5100 – one tool to rule them all...

- example RF attack lab setup:
  - dongle “Gina” running RfCat spectrum-analysis
  - dongle “Paul” running RfCat
  - IMME running RfSniff
  - (possibly an IMME's running SpecAn)
  - saleae logic analyzer for hacking of the wired variety
  - FunCube Dongle and quisk/qthid or other SDR
    - sssshhhhh... ignore the “other tools” or i'll have to change the slide title.
    - ok, so it's more like a goal than a title :)

rf attack form

- base freq:
- modulation:
- baud/bandwidth:
- deviation:
- channel hopping?
  - how many channels: channel spacing:
  - pattern and effective sync method? dwell period (ms):
- fixed-/variable-length packets: len/maxlen:
- “address”:
- sync word (if applicable):
- crc16 (y/n): does chip do correct style?
- fec (y/n): type (convolutional/reed-soloman/other):
- manchester encoding (y/n):
- data whitening? and 9bit pattern:
- more complete information: http://atlas.r4780y.com/resources/rf-recon-form.pdf

0xE7 – genie! oh genie!

configure the dongle / determine correct parameters: genie

- start RfCat
  - set Frequency to 315mhz
  - set Modulation to ASK_OOK
  - set SyncWord to AA00 and SyncMode to SYNCM_16_of_16
  - set Packet Length: 30
  - play around with baud rates... end up at 5200 baud
  - d.RFlisten()

0xE8 – car keyless entry

configuring the dongle – Keyless Entry Fob

- start RfCat
  - set Frequency to 315mhz
  - set Modulation to ASK_OOK
  - set SyncWord to FFFE and SyncMode to SYNCM_16_of_16
  - set Packet Length: 12
  - enable Manchester Encoding
  - play around with baud rates... end up at 4761.9 baud
  - d.RFlisten()

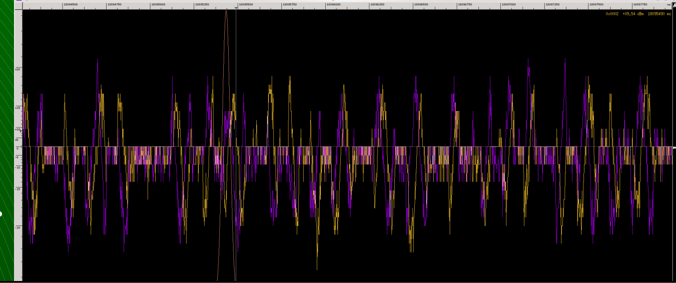
0x6000 – playing with medical devices

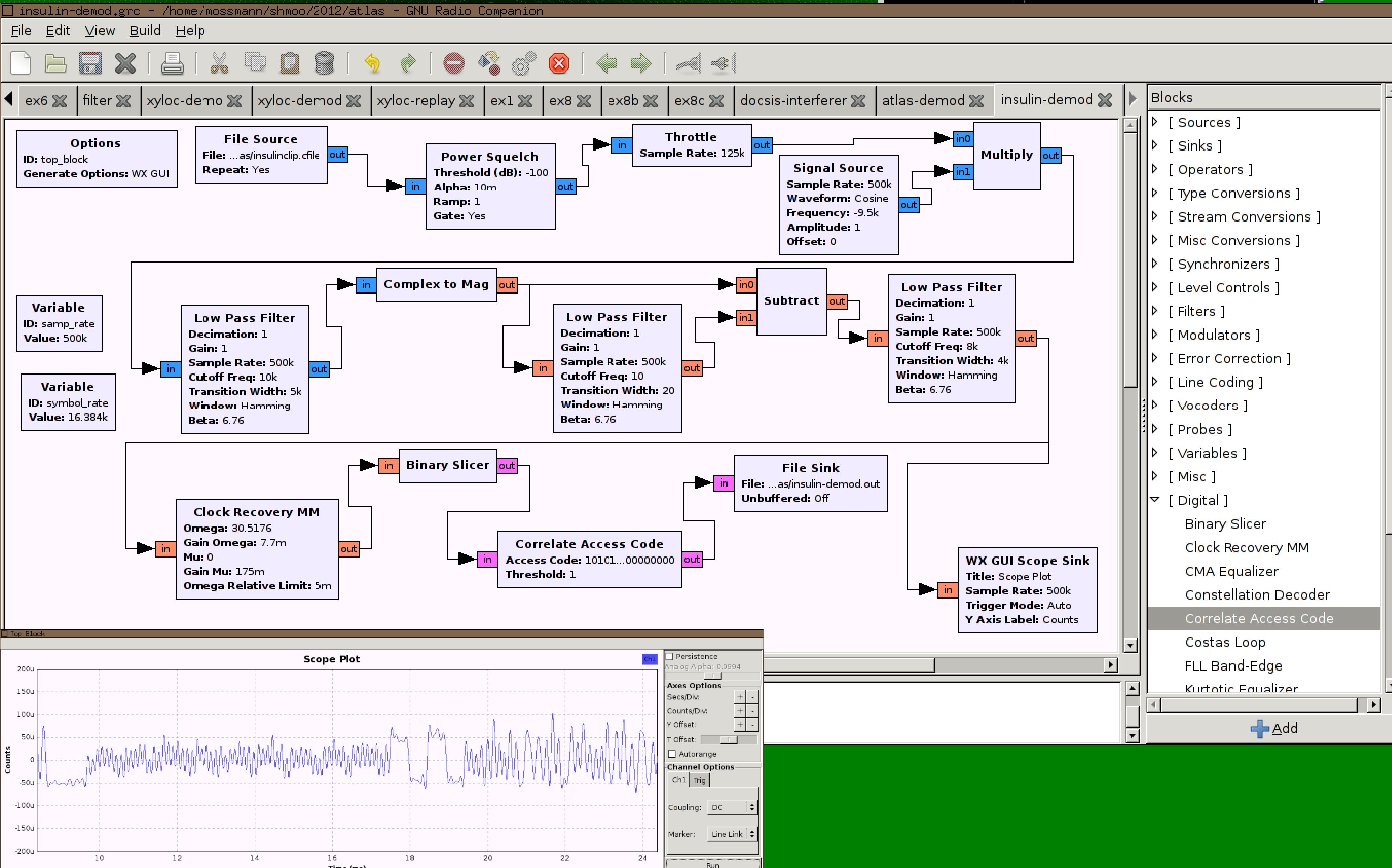
- CAUTION: MUCKING WITH THESE CAN KILL PEOPLE.
  - THIS FIRMWARE AND CLIENT NOT PROVIDED
- found frequency in the pdf manual from the Internet
  - what random diabetic cares what frequency his pump communicates with!? ok, who cares!
- modulation guessed based on spectrum analysis and trial/error
  - the wave form just **looks** like <blah> modulation!
- other characteristics discovered using a USRP and baudline (and some custom tools, thanks Mike Ossman!)

0x6010 – the discovery process

- glucometer was first captured using Spectrum Analyzer (IMME/hedyattack) to validate frequency range from the lay-documentation
- next a logic analyzer (saleae) used to tap debugging lines
- next, the transmission was captured using a USRP (thank you Mike Ossman for sending me your spare!)
- next, the “packet capture” was loaded into Baudline, and analysis performed to identify baudrate and modulation scheme, and get an idea of bits
- next, Mike Ossman did amazing-sauce, running the capture through GnuRadio Companion (the big picture on next slide)
- RF parameters confirmed through RF analysis, and real-life capture.

0x6011 – discovery reloaded





0x6020 –the immaculate reception

- punched in the RF parameters into a RFCAT dongle
  - created subclass of RFNIC (in python) for new RF config
- dropped into “discover” mode to ensure I had the modem right

```
(1327713078.539) Received:  d55555555555555555555555555555557f807f80552d5a399686b38ad8e2c
  possible Sync Dwords: ['0xaaffL', '0xaabfL', '0xaaafL', '0xaaabL', '0xaaaaL', '0x2aaaL']
(1327713078.553) Received:  c0002aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabfc03fc02a96ad1ccb4359c5
  possible Sync Dwords: ['0xaaffL', '0xaabfL', '0xaaafL', '0xaaabL', '0xaaaaL', '0x2aaaL', '0xaaaL']
(1327713078.569) Received:  1c59580002aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabfc03fc02a96ad1ccb4
  possible Sync Dwords: ['0xaaffL', '0xaabfL', '0xaaafL', '0xaaabL', '0xaaaaL']
(1327713078.583) Received:  b38ad8e2cac00015555555555555555555555555555555fe01fe0154b56
  possible Sync Dwords: ['0xaaffL', '0xaabfL', '0xaaafL', '0xaaabL', '0xaaaaL', '0x2aaaL', '0xaaaL']
(1327713078.602) Received:  e65a1ace2b638b2b00005555555555555555555555555555557f807f80
  possible Sync Dwords: ['0xaaffL', '0xaabfL', '0xaaafL', '0xaaabL', '0xaaaaL', '0x2aaaL', '0xaaaL', '0x2aaL']
(1327713078.614) Received:  d2d5a399686b38ad8e2cac00015aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaff
(1327713078.629) Received:  fc02a96ad1ccb4359c56c716560000aaaaaaaaaaaaaaaaaaaaaaaaaaaaa
(1327713078.643) Received:  eff00ff00aa5ab4732d0d6715b1c59580002aaaaaaaaaaaaaaaaaaaaaaaa
(1327713078.658) Received:  aaaaaaff00ff00aa5ab4732d0d6715b1c59580002aaaaaaaaaaaaaaaaaaaa
  possible Sync Dwords: ['0xaaffL', '0xaabfL', '0xaaafL', '0xaaabL', '0xaaaaL', '0x2aaaL', '0xaaaL', '0x2aaL']
(1327713078.674) Received:  aaaaaaaaaaaff00ff00aa5ab4732d0d6715b1c59580002aaaaaaaaaaaaaa
  possible Sync Dwords: ['0xaaffL', '0xaabfL', '0xaaafL', '0xaaabL', '0xaaaaL', '0x2aaaL']
(1327713078.689) Received:  aaaaaaaaaaaaaaaaff00ff00aa5ab4732d0d6715b1c59580002aaaaaaaaa
  possible Sync Dwords: ['0xaaffL', '0xaabfL', '0xaaafL', '0xaaabL', '0xaaaaL', '0x2aaaL', '0xaaaL', '0x2aaL']
(1327713078.704) Received:  aaaaaaaaaaaaaaaaaaaaaff00ff00aa5ab4732d0d6715b1c59580002aaaa
  possible Sync Dwords: ['0xaaffL', '0xaabfL', '0xaaafL', '0xaaabL', '0xaaaaL', '0x2aaaL']
(1327713078.719) Received:  aaaaaaaaaaaaaaaaaaaaaaaaaaff00ff00aa5ab4732d0d6715b1c5958000
  possible Sync Dwords: ['0xaaffL', '0xaabfL', '0xaaafL', '0xaaabL', '0xaaaaL', '0x2aaaL', '0xaaaL', '0x2aaL']
```

- returned to normal NIC mode to receive real packets
- now need the pump to reverse the bi-dir protocol

```
('aa5ab4732d0d2f19ac56558000',
('aa5ab4732d0d2f19ac56558000',
('aa5ab4732d0d2f19ac56558000',
('aa5ab4732d0d2f19ac56558000',
('aa5ab4732d0d2f19ac56558000',
('aa5ab4732d0d2f19ac56558000',
('aa5ab4732d0d2f19ac56558000',
('aa5ab4732d0d2f19ac56558000',
('aa5ab4732d0d2f19ac56558000',
('aa5ab4732d0d2f19ac56558000',
('aa5ab4732d0d2f19ac56558000',
('aa5ab4732d0d2f19ac56558000',
('aa5ab4732d0d2f19ac56558000',
('aa5ab4732d0d2f19ac56558000',
('aa5ab4732d0d2f19ac56558000',
```

0xE9 – hop hop hopping along...

```
>>> d.getFHSSstate()
```

- import friend. decide who will be Sync Master.
- non-Master starts first:

```
>>> d.setFHSSstate(FHSS_STATE_DISCOVER)
```

- now the Sync Master:

```
>>> d.setFHSSstate(FHSS_STATE_SYNCINGMASTER)
```

- once sync'd:

```
>>> d.getMACdata()
>>> print d.reprMACdata()
```

0xE9.1 – FHSS xmit and recv

- now, one of you send (notice, different function)

```
>>> d.FHSSxmit('yo yo! wazgud!?')
```

- and the other of you receive:

```
>>> d.RFrecv()      # nothing special here
>>> d.RFlisten()    # same thing here
```

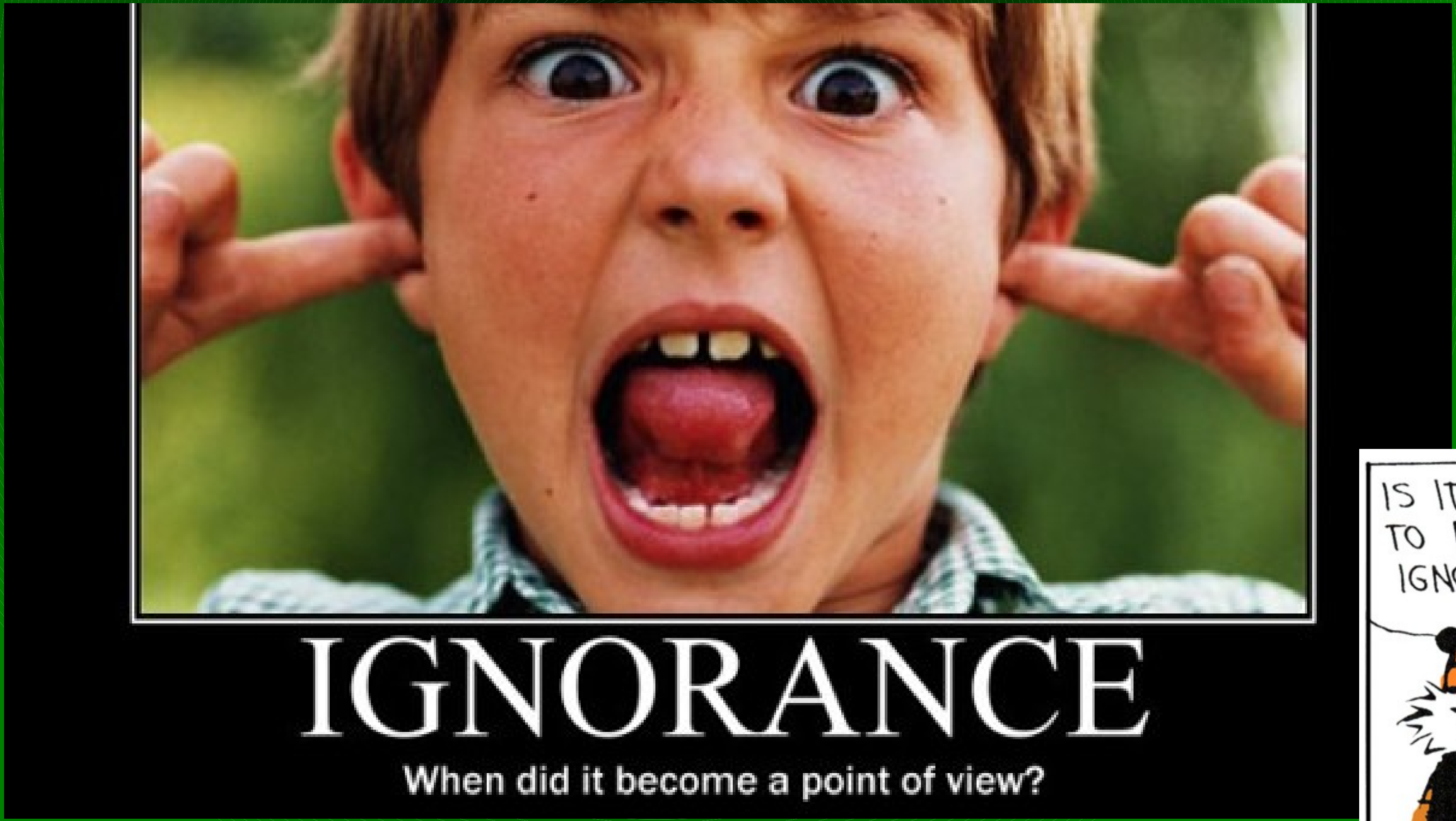
0x6100 – playing with a power meter

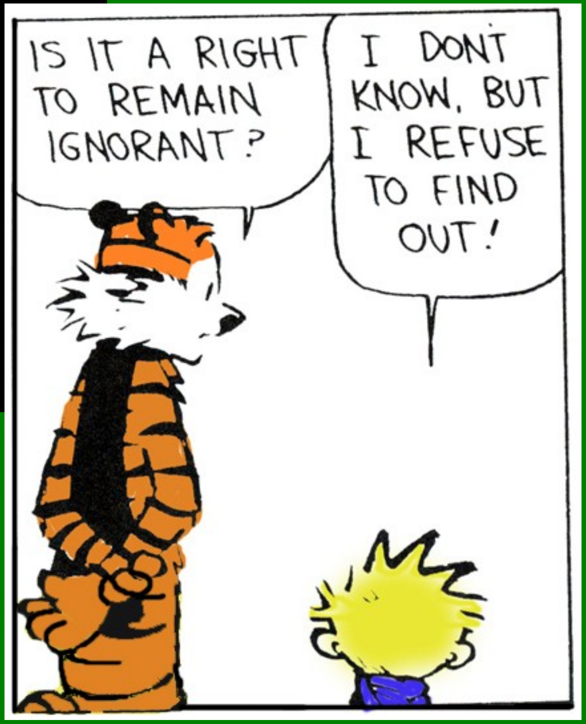
- CAUTION: MUCKING WITH POWER SYSTEMS WITHOUT APPROPRIATE AUTHORIZATION IS ILLEGAL, EVEN IF IT IS ON THE SIDE OF YOUR HOUSE!
- most power meters use their own proprietary “Neighborhood Area Network” (NAN), typically in the 900MHz range and sometimes 2.4GHz or licensed spectrum.
- to get the best reception over distance and gain tolerance to interference, all implement FHSS to take advantage of the Title 47: Part 15 power allowances
- many of the existing meters use the same cc1111 or cc1110 chips, or the cc1101 radio core
- this is the reason I'm here today

0x6110 – as sands through the hourglass

- power meter RF comms have long been “unavailable” for most security researchers
- **some vendors understand the benefits of security rigor by outside researchers**
  - others, however, do not.
- the gear used in my presentation was given to me by one who understands
  - for various reasons, they have asked to remain anonymous, however, their security team has a well founded approach to finding out “how their baby is ugly” I would like to give them credit for their commitment to the improved security of their products.

atlas, tell us what you really think



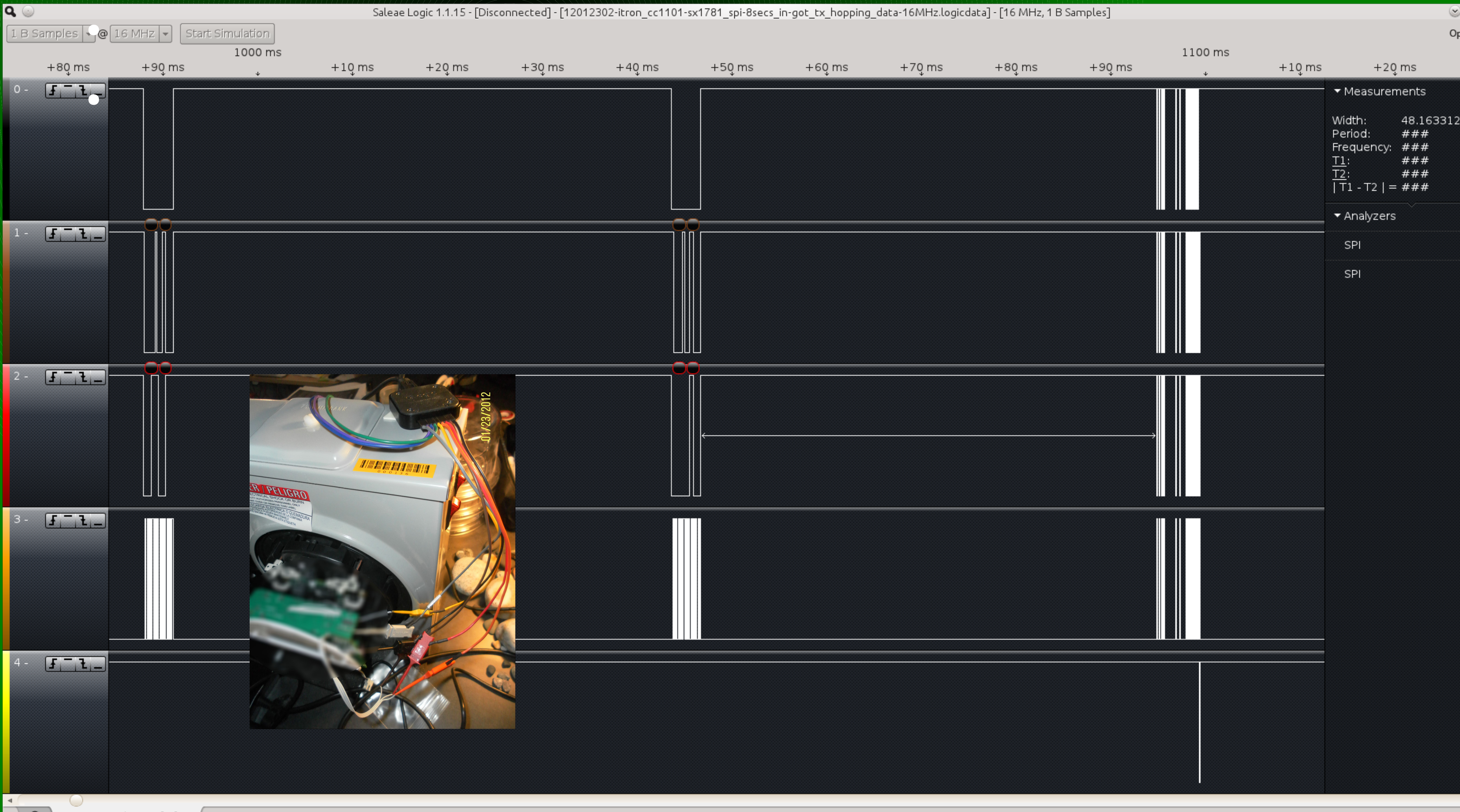


0x6120 – smart meter – the complication

- power meters are not so simple as glucometers
  - proprietary FHSS in a multiple-access topology
  - have to endure the RF abuse of the large metropolis
- complex mac sync/net-registration
- not easy to show with a single meter without a Master node.
- initial analysis was performed via my saleae LA:
- SpecAn code on IMME's and hedyattack dongles
  - good for identifying periods of scanning
- although the dongle can hop along with the meter, we won't be demoing synching with the meter today

0x6130 – the approach

- determine the rf config and hopping pattern through SPI Bus sniffing (and my saleae again)



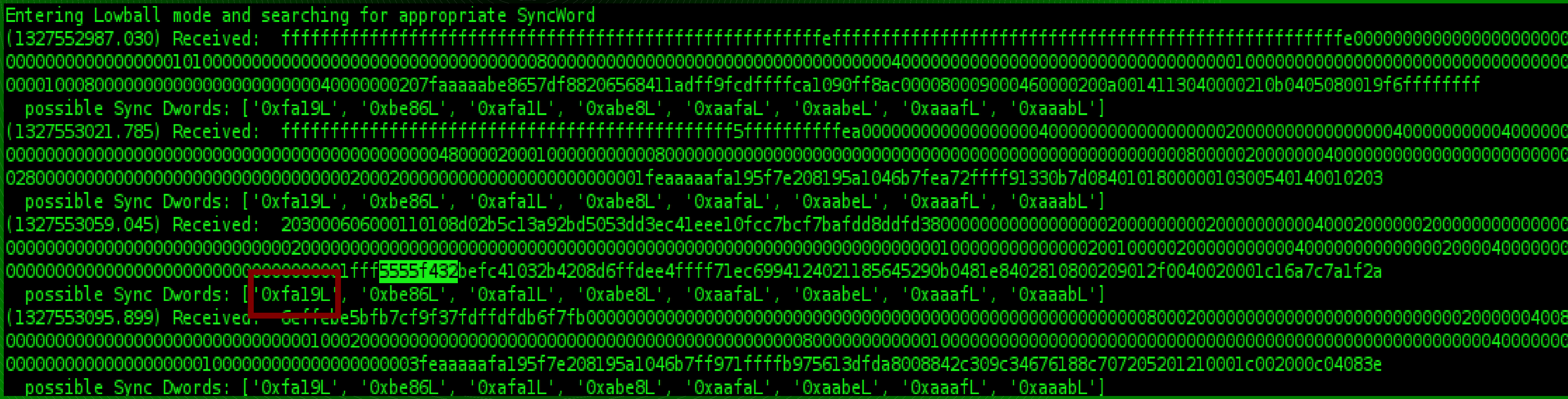
0x6135 – Logic Analyzer

- decoding:
  - custom parser for the target radio--->>>

```
11.840    SET RF channel:  0 - 902250000  [redacted]

11.851     STROBE:      SRES
12.101     WRITE                 IOCFG2                (BURST) 01 [redacted]
12.101     READ                  IOCFG2                (BURST) 01 [redacted]
12.101     WRITE                 FSTEST                (BURST) 59 [redacted]
12.101     READ                  FSTEST                (BURST) 59 [redacted]
12.101     WRITE                 PATABLE               (BURST) 00 [redacted]
12.103     READ                  PATABLE               (BURST) 00 [redacted]
12.103     STROBE:      SIDLE
12.104     STROBE:      SCAL
12.104     STROBE:      SFRX
12.105     STROBE:      SRX
12.107     STROBE:      SIDLE
12.108     STROBE:      SCAL
12.108     STROBE:      SFRX
12.108     STROBE:      SRX
12.123     STROBE:      SRES
12.373     WRITE                 IOCFG2                (BURST) 01 [redacted]
12.374     READ                  IOCFG2                (BURST) 01 [redacted]
12.374     WRITE                 FSTEST                (BURST) 59 [redacted]
12.374     READ                  FSTEST                (BURST) 59 [redacted]
12.374     WRITE                 PATABLE               (BURST) 00 [redacted]
12.375     READ                  PATABLE               (BURST) 00 [redacted]
12.376     STROBE:      SIDLE
12.376     STROBE:      SCAL
12.377     STROBE:      SFRX
12.377     STROBE:      SRX
12.394     STROBE:      SIDLE
12.395     STROBE:      SCAL
12.395     STROBE:      SFRX
12.396     STROBE:      SRX
12.399     STROBE:      SIDLE
12.399     STROBE:      SCAL
12.400     STROBE:      SFRX
12.400     STROBE:      SRX
12.404     STROBE:      SIDLE
12.404     STROBE:      SCAL
12.405     STROBE:      SFRX
12.405     STROBE:      SRX
```

0x6140 – the approach (2)

- discover mode:
  - disables sync-word so radio sends unaligned bits



  - algorithm looks for preamble (0xaa or 0x55)
  - then determines possible dwords
- ummm... but that's not any bit-derivation of the sync word(s) I expect. wut? I am confident those are coming from the meter
  - intro: Bit Inversion (see highlighted hex)

0x6145 – new developments

- vendors filed numerous patents with hopping pattern calculations, communications parameters, etc...
  - WIN!
  - plenty of work to be done! jump right in!
    - http://www.patentstorm.us/patents/7064679/fulltext.html
    - http://www.patentstorm.us/patents/7962101/fulltext.html
    - http://www.patentstorm.us/applications/20080204272/fulltext.html
    - http://www.patentstorm.us/applications/20080238716/fulltext.html

"Abuse is no argument"

- Nevil Maskelyne

0x6150 - conclusions

- rfcat discover mode roxors
- rfcat is a **foundation** for your attack tool
  - way more than just a tool in itself
- we are responsible for ensuring our devices use appropriate security. **do not** simply expect someone else to do it. the first med-device death could be your best friend.

References

- http://rfcat.googlecode.com
- http://en.wikipedia.org/wiki/Part_15_(FCC_rules)
- http://en.wikipedia.org/wiki/ISM_band
- http://www.ti.com/lit/ds/swrs033g/swrs033g.pdf - “the” manual
- http://edge.rit.edu/content/P11207/public/CC1111_USB_HW_User_s_Guide.pdf
- http://www.ti.com/litv/pdf/swru082b
- http://www.ti.com/product/cc1111f32#technicaldocuments
- http://www.ti.com/lit/an/swra077/swra077.pdf
- http://www.newscientist.com/article/mg21228440.700-dotdashdiss-the-gentleman-hackers-1903-lulz.html
- http://saleae.com/
- http://zone.ni.com/devzone/cda/epd/p/id/5150 - FSK details (worthwhile!)
- http://www.radagast.org/~dplatt/hamradio/FARS_presentation_on_modulation.pdf
  - very good detailed discussion on deviation/modulation
- http://en.wikipedia.org/wiki/Frequency_modulation
- http://en.wikipedia.org/wiki/Minimum-shift_keying

0xgreetz

- power hardware folk who play nice with security researchers
- cutaway and q (awesome hedyattackers)
- gerard van den bosch
- travis and mossman
- sk0d0 and the four J's
- invisigoth and kenshoto
- jewel, bug, ringwraith, diva
- Jesus Christ