It’s 8051s all the way down

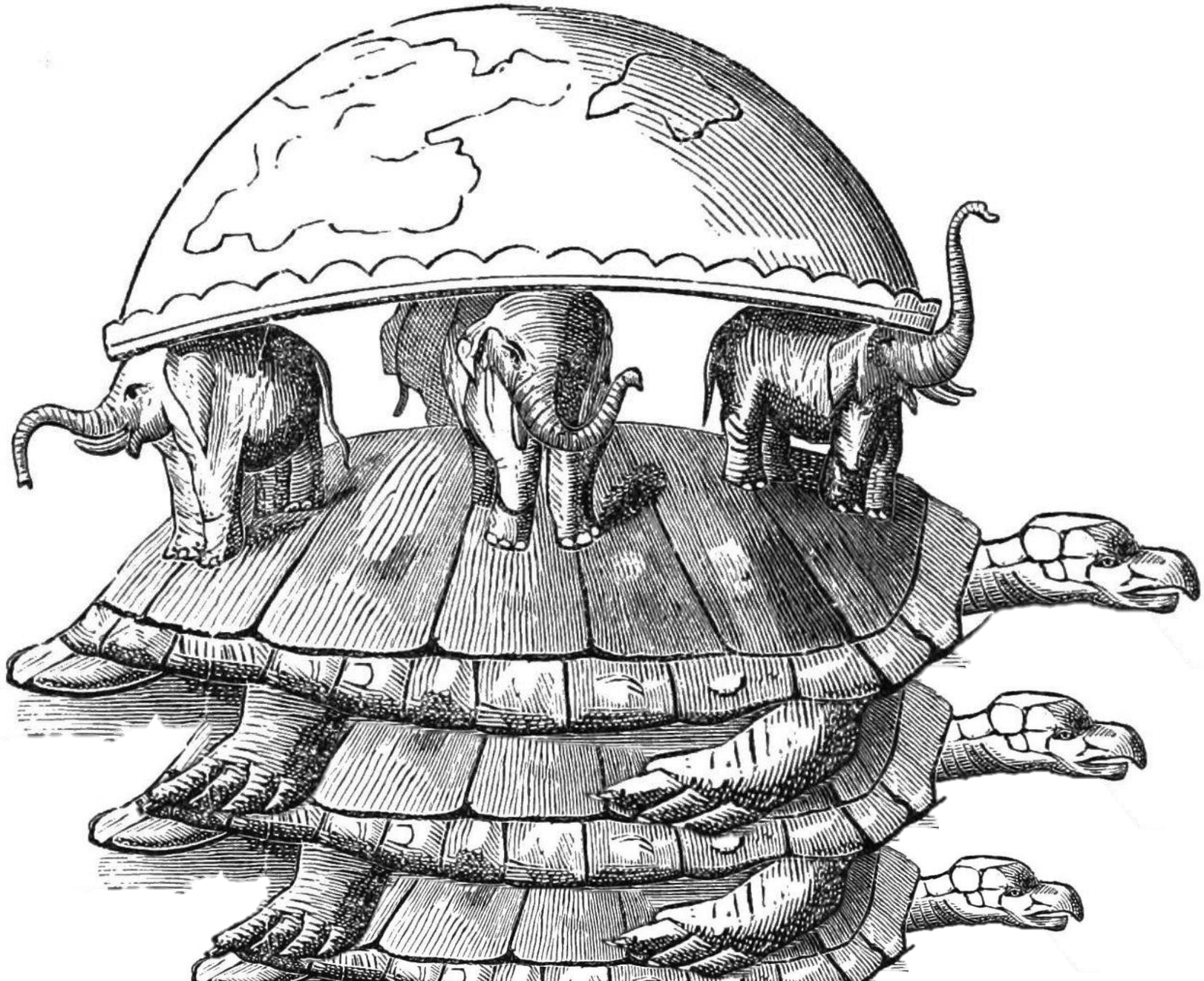
Reverse engineering the SiLabs EZRadioPRO with radare2

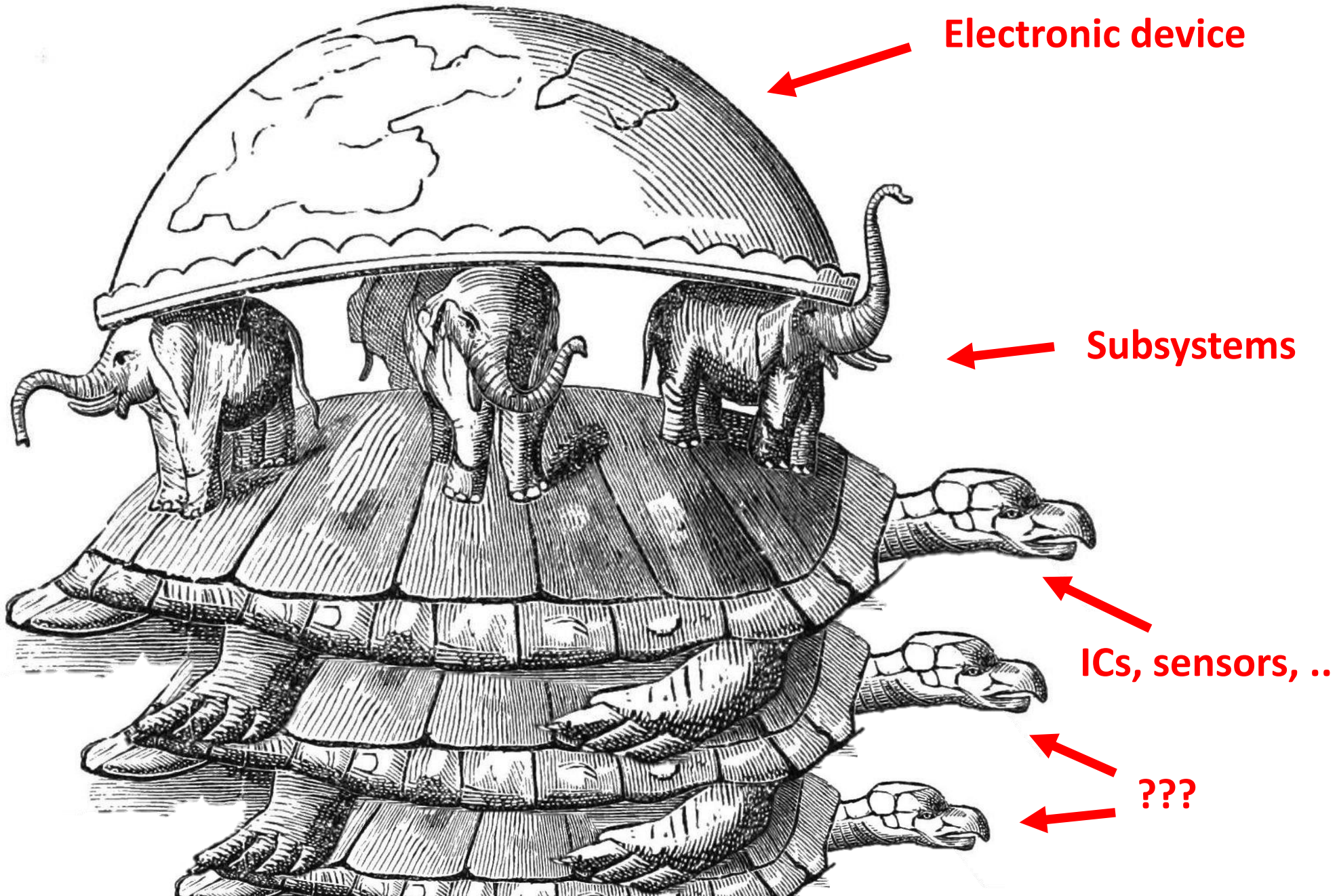
https://github.com/astuder/Inside-EZRadioPRO

@adistuder

About Me

- 1990s
  - Demoscener
  - Author of PPLD (PCBoard PPE decompiler)
- 2000s
  - Too busy at work …
- 2010s
  - Hardware hacking
  - Creator of dAISy AIS receiver



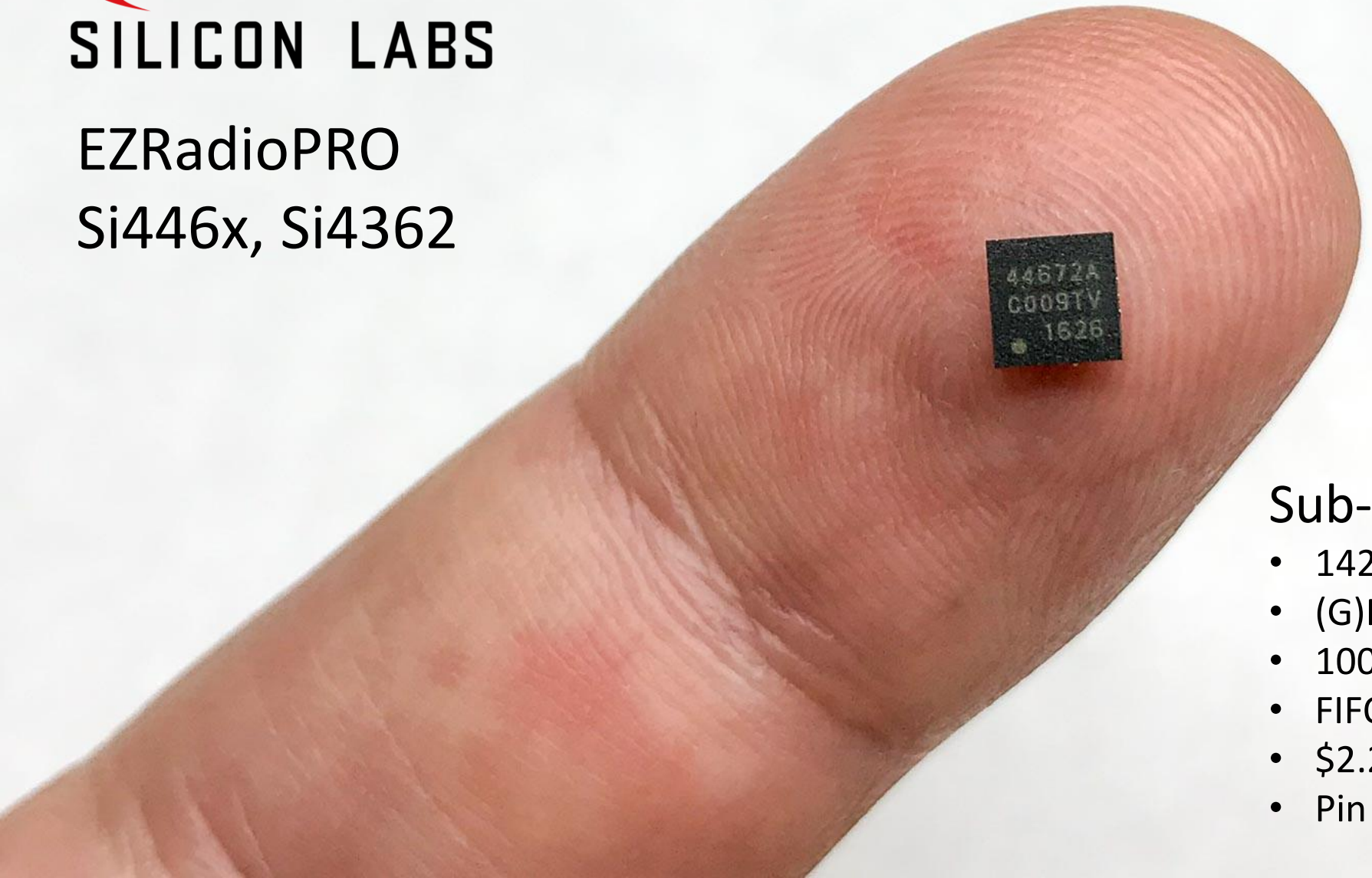

Electronic device
Subsystems
ICs, sensors, ..
???

It’s MCUs all the way down

- Intel Management Engine (IME)
- Hard disks (SpritesMods @OHM2013)
- SD Cards (talk by Bunnie & xobs @30c3)
- eMMC (eMMC sudden death @xdadevelopers)
- Radio & MCU combos (WiFi, BT, baseband..)
- ..

SILICON LABS

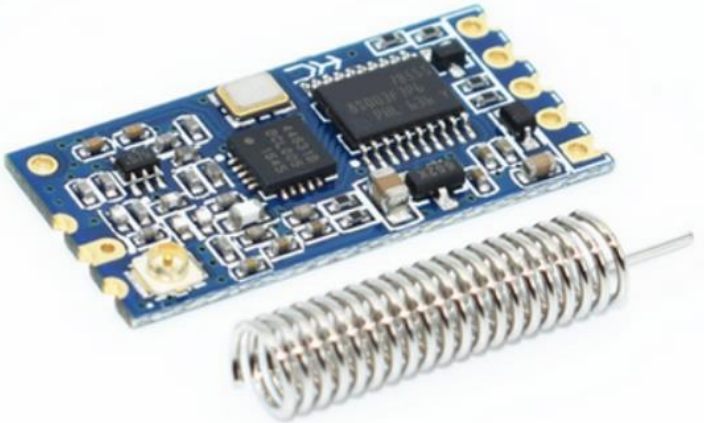
EZRadioPRO
Si446x, Si4362



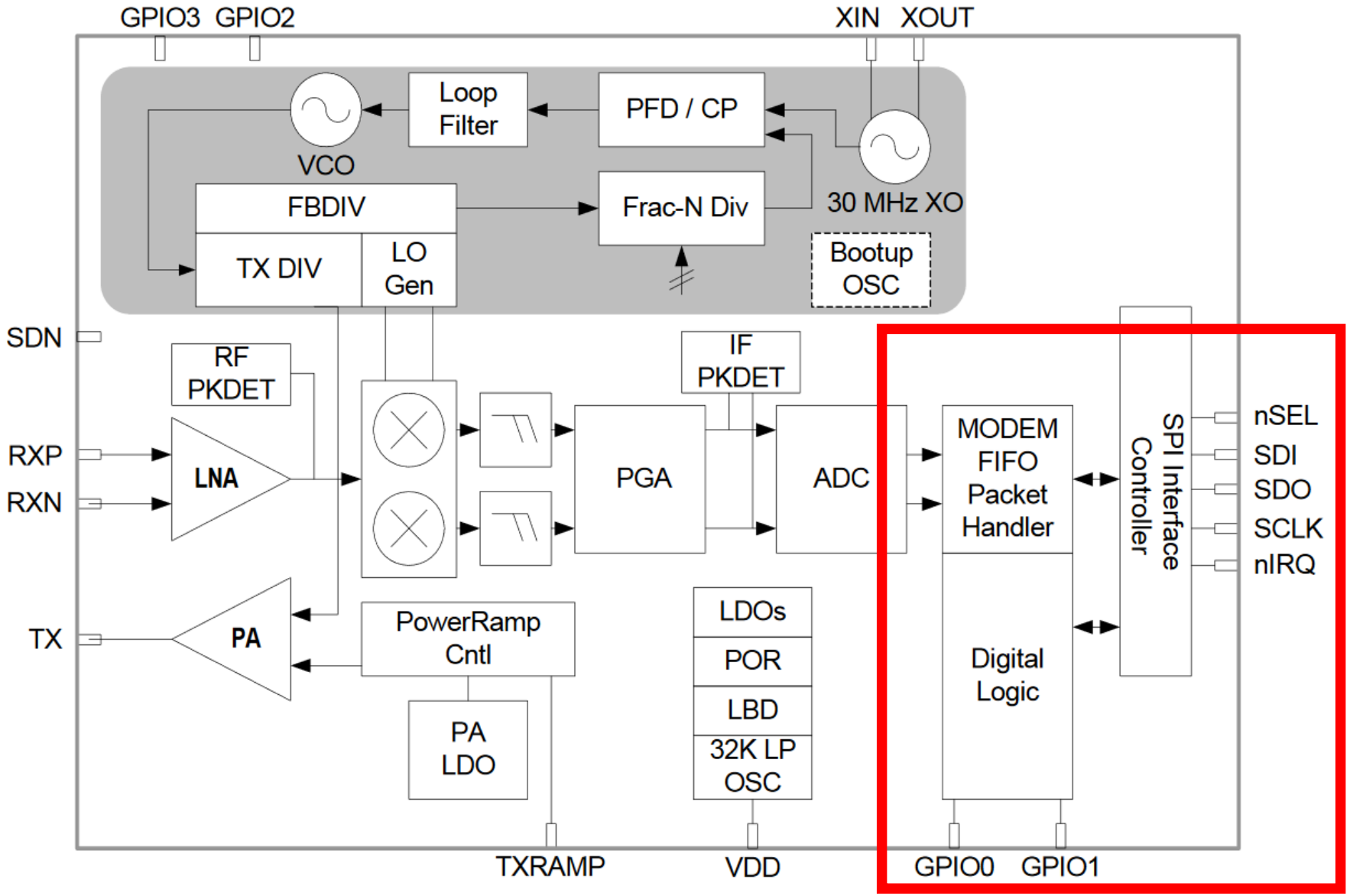
Sub-GHz radios

- 142-1040 MHz contiguous
- (G)FSK, 4(G)FSK, (G)MSK, OOK
- 100-1M bps
- FIFO, packet handler, CRC, …
- $2.20-$2.50 in single quantity
- Pin compatible across whole family

- Xyloband
- goTenna, goTenna MESH
- Sigfox modules (TD next TD120x)
- Cheap AIS receivers
- High-altitude balloon trackers (APRS transmitters)
- 433 MHz modules (“Si4463 module” or HC-12 on Ebay, AliExpress)



Architecture according to the datasheet



Architecture according to SiLabs patent

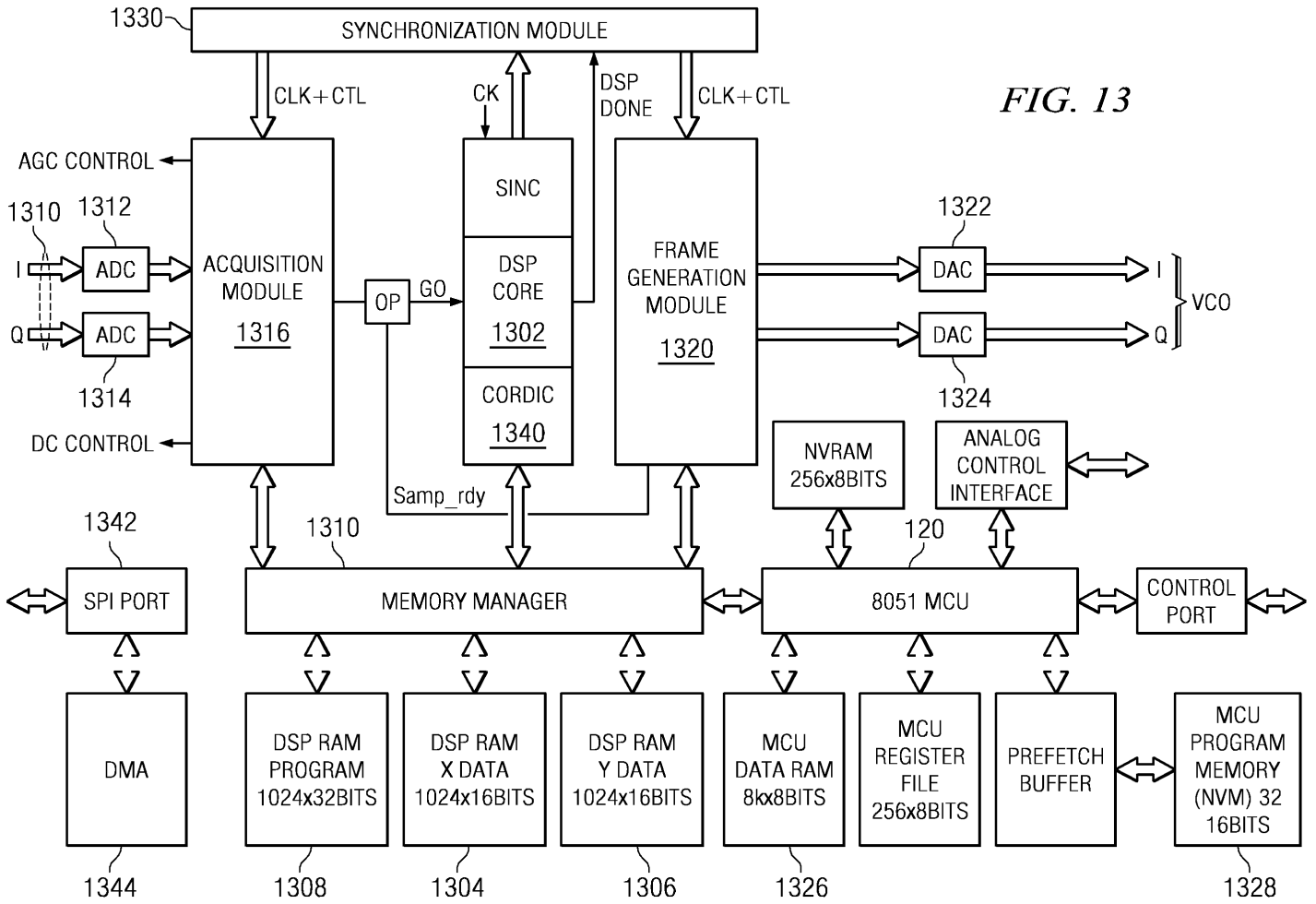


FIG. 13

Single chip low power fully integrated 802.15.4 radio platform
https://patents.google.com/patent/US8050313B2

In a related patent…

image. Note that in various implementations, multiple firmware versions may be generated such that different devices formed of a single wafer or within one or more wafer lots can each be programmed with different firmware versions. These firmware versions may thus provide different functionality, allowing a single mask set to be used to produce devices having different functionalities, which may provide for sales at different price points.

Still referring to FIG. **3**, during marketing of a chip, which may occur over a number of months or years, it may be

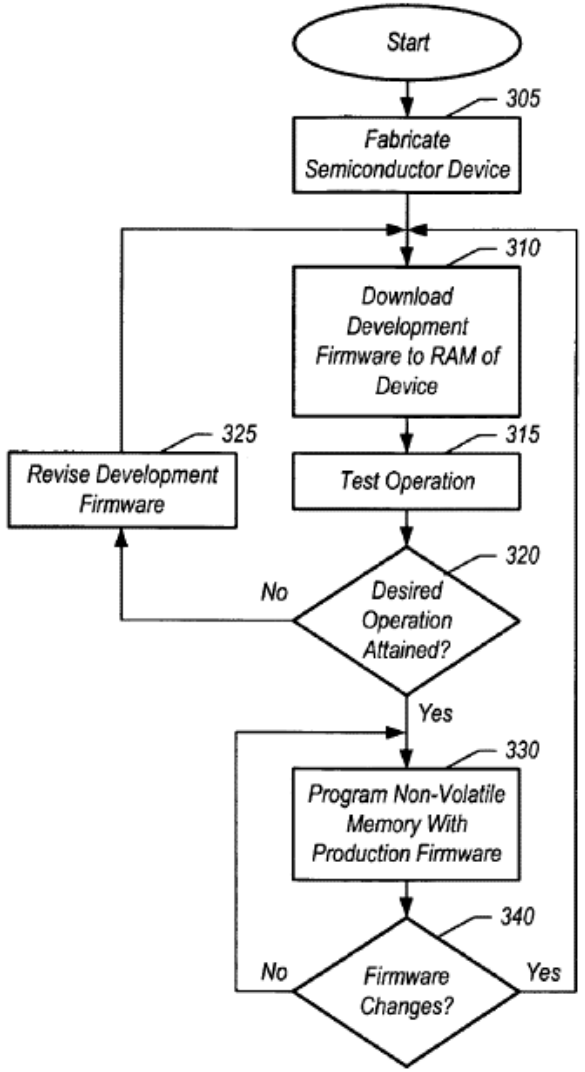


FIG. 3

Digital architecture using one-time programmable (OTP) memory
https://patents.google.com/patent/US7613913B2

The are patches!

"If a bug is identified it is usually fixed with a patch on the current revision, and the change will be implemented on the next revision. Also, we can test future features of the chip by patching the radio."

https://www.silabs.com/community/wireless/bluetooth/forum.topic.html/apply_patch_to_si446-c7pE

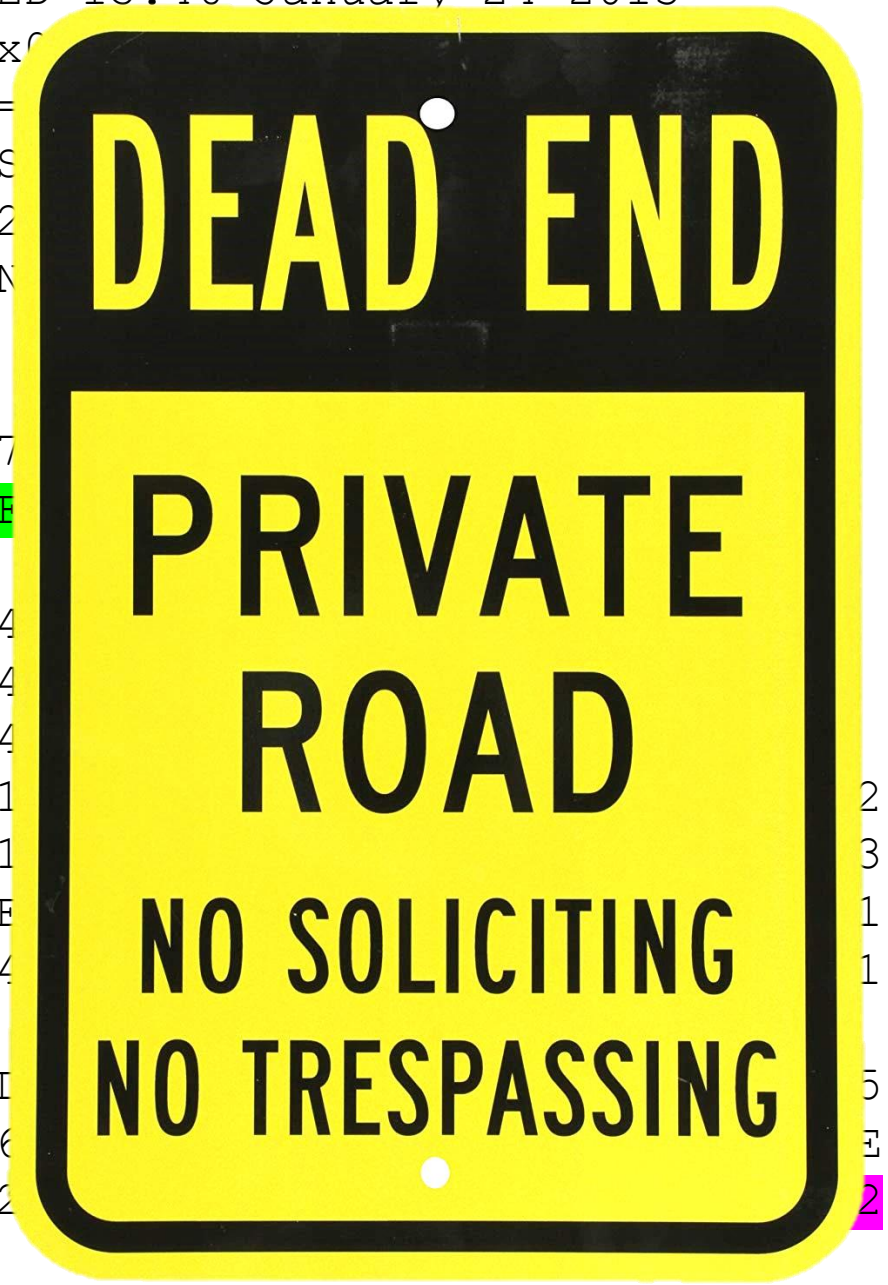
```
// GENERATED=15:46 January 24 2013
// ROMID=0x03
// PATCHID=0xF692
// REQUIRES=NONE
// SIZE=2128
// FUNCTION=MAIN
// MAJOR=3
// MINOR=0
// BUILD=27
// CRCT=0xF776

#define SI446X_PATCH_ROMID  03
#define SI446X_PATCH_ID     27
#define SI446X_PATCH_CMDS                          \
{ 0x04,0x11,0xF7,0x76,0x00,0x00,0xA6,0x82 }, \
{ 0x05,0x61,0xE6,0x82,0x5E,0xB7,0xFB,0x93 }, \
{ 0x05,0x1E,0x12,0xBD,0x5A,0xC2,0x52,0x41 }, \
{ 0xE7,0xF4,0xDF,0x6A,0x24,0xD9,0xBA,0x31 }, \
…
{ 0xEF,0x7D,0x0D,0xB5,0xCF,0x00,0xC5,0x75 }, \
{ 0xE3,0xC6,0x0E,0x0B,0x10,0x44,0x10,0xEE }, \
{ 0x05,0x12,0x86,0x0D,0xC0,0xA5,0xF6,0x92 }
```

265 lines

```
// GENERATED=15:46 January 24 2013
// ROMID=0x
// PATCHID=
// REQUIRES
// SIZE=212
// FUNCTION
// MAJOR=3
// MINOR=0
// BUILD=27
// CRCT=0xE

#define SI4
#define SI4
#define SI4                                                    \
{  0x04,0x11                                                2 }, \
{  0x05,0x61                                                3 }, \
{  0x05,0x1E                                                1 }, \
{  0xE7,0xF4                                                1 }, \
...
{  0xEF,0x7D                                                5 }, \
{  0xE3,0xC6                                                E }, \
{  0x05,0x12                                                2 }
```

PATCH_IMAGE

PATCH_ARGS,
PATCH_DATA?



265 lines
= 1-2k of code?

SiLabs knowledge base

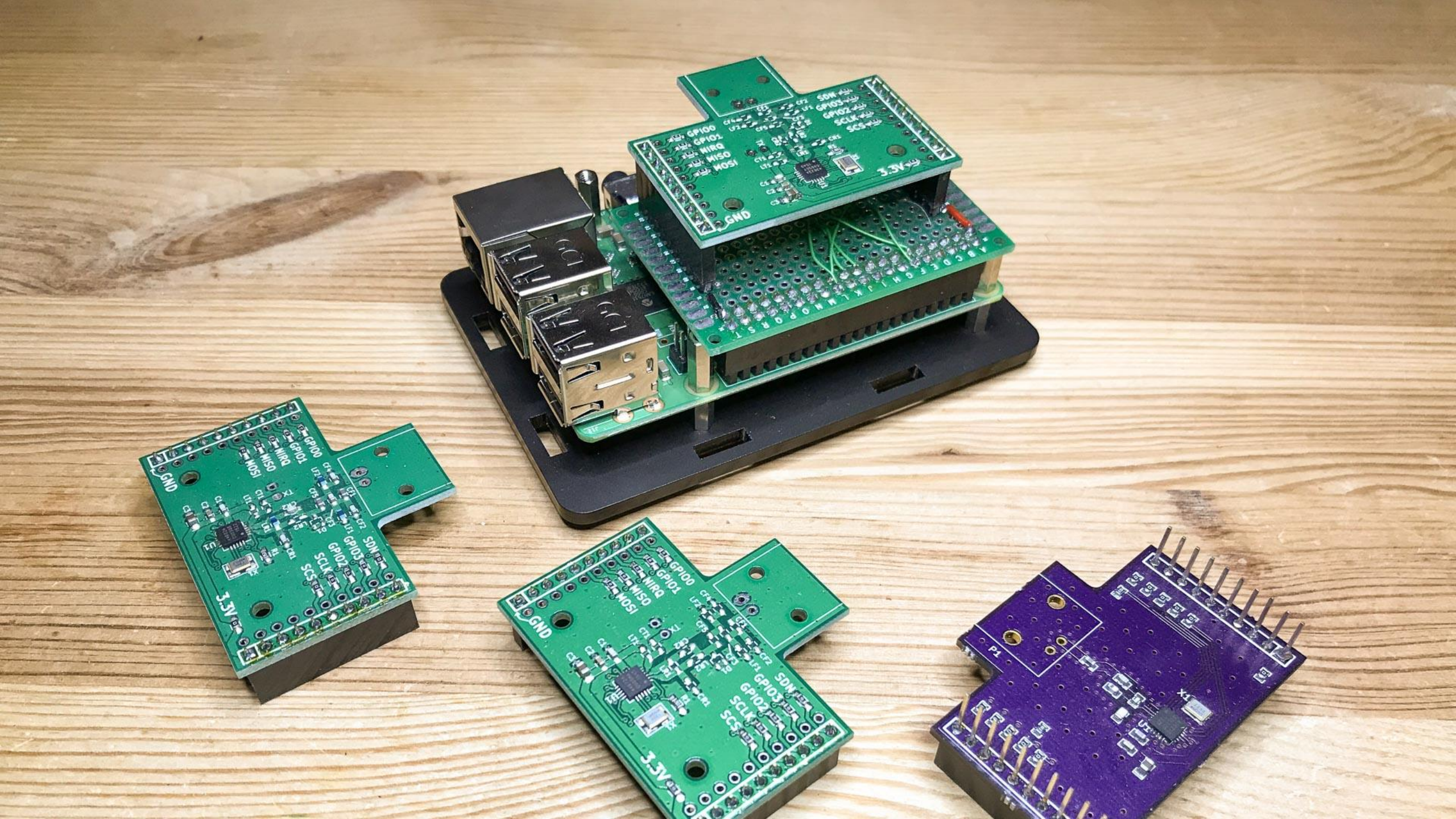
Q: How do I retrieve and re-apply the results of IQ calibration on Si446x?

A: [..] special API command that allows direct register access is referred to as POKE for writing registers and PEEK for reading [..]

The command code for PEEK is 0xF0 and for POKE it is 0xF1. The two byte addresses of the registers of concern are the following: iq_calamp – 0x00 0xD6; iq_calph – 0x00 0xD7.

https://www.silabs.com/community/wireless/proprietary/knowledge-base.entry.html/2014/05/12/si446x_iq_calibratio-ybHg


**** COMMODORE 64 BASIC V2 ****
64K RAM SYSTEM 38911 BASIC BYTES FREE
READY.
10 A=0
20 PRINT PEEK(A)
30 A=A+1
40 IF A<65536 GOTO 20
RUN


GPIO0
GPIO1
NIRQ
MISO
MOSI
GND
SDN
GPIO3
GPIO2
SCLK
SCS
3.3V
P1

Dumping full 16bit address range

- Raspberry Pi
- Breakout boards for radio ICs
- Python, rpi.gpio, py-spidev

```
resetRadio()
sendCommand(POWER_UP)
for addr in range(0,65535):
    print sendCommand(PEEK,addr)
```

A first look at the dump (Si4362-C2A)

astuder

Hi guys! I’m new to radare2 and have issues disassembling 8051 firmware. Half the opcodes are wrong or not implemented.

pancake

send a PR

astuder

…

A closer look at the dump (Si4362-C2A)

- 0x0000 data
- 0x0100 jump table
- 0x02eb code
- 0x051e data
- 0x0800 code
- 0x4000 vector table
- 0x4100 mirror of 0x0100
- 0x4800 data
- 0x4900 empty
- 0x5000 data?

Finding the entry point

- 0x0100 jumps to mostly above 0x8000 (ROM starts at 0x8000?)
- 0x4000 looks like 8051 interrupt vector table (mirror of 0x0000?)
- RESET vector points to 0xcc43
- No code at 0xcc43 ☹
- Find candidates for reset code
  - Search for setup of stack
  - r2: /c mov sp,
  - Two entries, offset matching with RESET and EINT0
  - 0x0b98 = ROM 0xcc43? Nope ☹
- Large parts of ROM are missing (14K vs 32K expected)

Finding entry point for API commands

- EZRadioPRO API format:

Argument Stream:

FIFO_INFO Argument Stream									
	Index	Name	7	6	5	4	3	2	1	0
	0x00	CMD	*0x15*							
	0x01	FIFO	0	0	0	0	0	0	RX	TX

Reply Stream:

FIFO_INFO Reply Stream										
	Index	Name	7	6	5	4	3	2	1	0
	0x00	CTS	CTS							
	0x01	RX_FIFO_COUNT	RX_FIFO_COUNT							
	0x02	TX_FIFO_SPACE	TX_FIFO_SPACE							

Finding entry point for API commands

- PEEK: cmd 0xf0, POKE: cmd 0xf1
- /c #0xf0 /c #0xf1

```
[0x00000000]> /c #0xf1
0x000012c9   # 2: anl a, #0xf1
0x00001e41   # 3: cjne a, #0xf1, 0x1e46
0x00001e9a   # 3: anl 0xe4, #0xf1
0x00002eb2   # 2: mov r0, #0xf1
0x00002fbe   # 2: mov r7, #0xf1
0x00003125   # 3: anl r7, #0xf1
0x00003ed3   # 2: mov r0, #0xf1
```

- CJNE a, imm, addr: compare and jump if not equal

Finding entry point for API commands

```
0x00001e3c      b4f002          cjne a, #0xf0, 0x1e41
0x00001e3f      a12a            ajmp 0x1d2a
0x00001e41      b4f102          cjne a, #0xf1, 0x1e46
0x00001e44      8064            sjmp 0x1eaa
0x00001e46      b4f202          cjne a, #0xf2, 0x1e4b
0x00001e49      801e            sjmp 0x1e69
0x00001e4b      b41303          cjne a, #0x13, 0x1e51
0x00001e4e      020258          ljmp 0x0258
0x00001e51      b43502          cjne a, #0x35, 0x1e56
0x00001e54      8021            sjmp 0x1e77
0x00001e56      b43803          cjne a, #0x38, 0x1e5c
0x00001e59      02014d          ljmp 0x014d
0x00001e5c      b41a03          cjne a, #0x1a, 0x1e62
0x00001e5f      02019e          ljmp 0x019e
0x00001e62      7002            jnz 0x1e66
0x00001e64      e15b            ajmp 0x1f5b
0x00001e66      020132          ljmp 0x0132
```

0xf0 = PEEK

0xf1 = POKE

Some cmds go via jmp table

Reversing PEEK and POKE

- Argument stream passed in XREG 0x70-0x7f
  - 0x70 Command
  - 0x71-0x7f Arguments
- Reply stream data passed in XREG 0x70-0x7f
  - Reply byte 0x01 at 0x70 -> CTS byte generated somewhere else
- RET to finish command

Reversing PEEK and POKE

- PEEK and POKE are guarded by bits in 0x07f9
  - Only in C2A/A2A, not in B1B -> Silabs could turn them off in future revs
- PEEK accepts up to 7 addresses
  - Returns 1 byte per address
- POKE accepts up to 5 address, value tuples

Reversing PEEK and POKE

- PEEK address translation
  - 0x00 – 0xff: read via SFR 0xa6 (“registers” in PEEK/POKE article)
  - 0x0100 – 0x51ff: read 0x0100-0x51ff in XDATA address space
  - 0x5100 – 0x52ff: read 0x00-0xff in IDATA address space, ignores MSB
  - 0x5500 – 0xdfff: read via function calls
  - 0xe000 – 0xf0ff: read via function call, ignores MSB
  - 0xf0ff – 0xffff: call function with r7=0x11
- POKE address translation similar, but
  - 0x07f0 – 0x07ff: protected, region contains part number
  - Writes to XDATA above 0x0800 have no effect

Arbitrary code execution

Goal: Dump complete CODE and/or XDATA address space

Attack plan:

- POKE our code into unused RAM
- Hijack existing API command by patching jump table

Constraints:

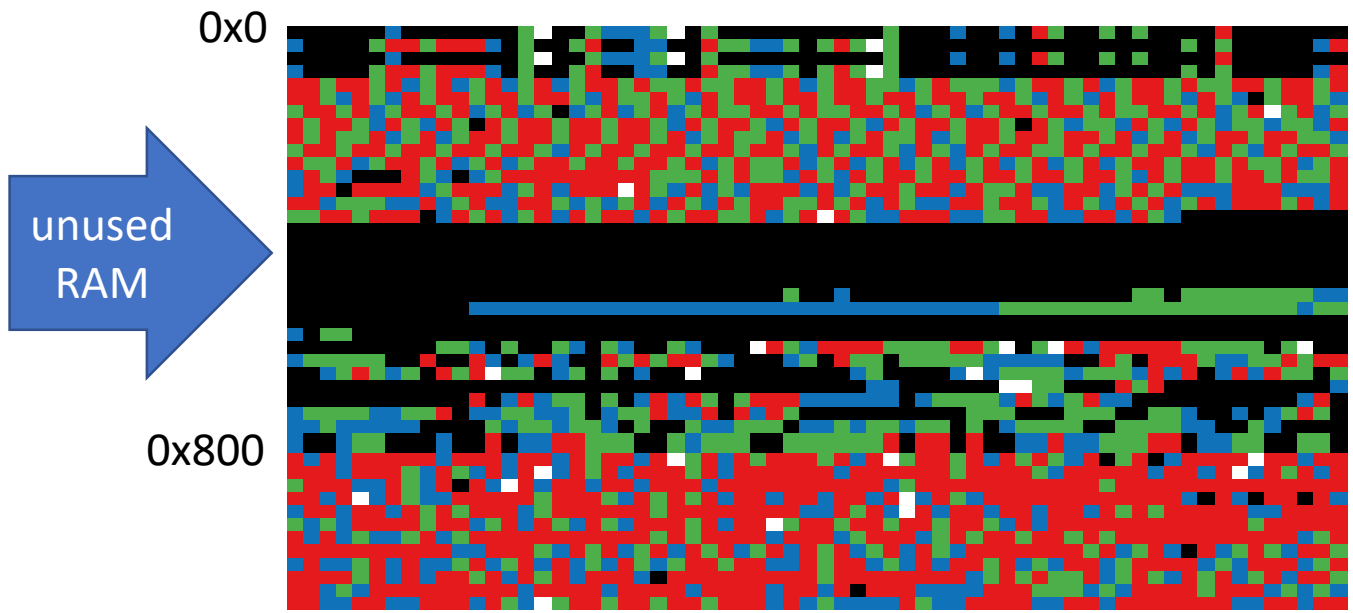
- POKE only works for 0x0100-0x07f0
- Support B1B, C2A and A2A chip revisions

Arbitrary code execution – find target

- Command to hijack
  - CJNE present in dump
  - Jump table entry above 0x0100
  - ➢ 0x12 GET_PROPERTY (B1B: 0x0225; C2A/A2A: 0x0168)

```
cjne a, #0x12, 0x1e24
ljmp 0x0168
```

- Unused RAM
  - B1B: 0x0484 – 0x0560
  - C2A: 0x03b6 – 0x051e
  - A2A: 0x0466 – 0x051e
- Write patch at 0x0488
- 96 bytes max



Arbitrary code execution – dump code space

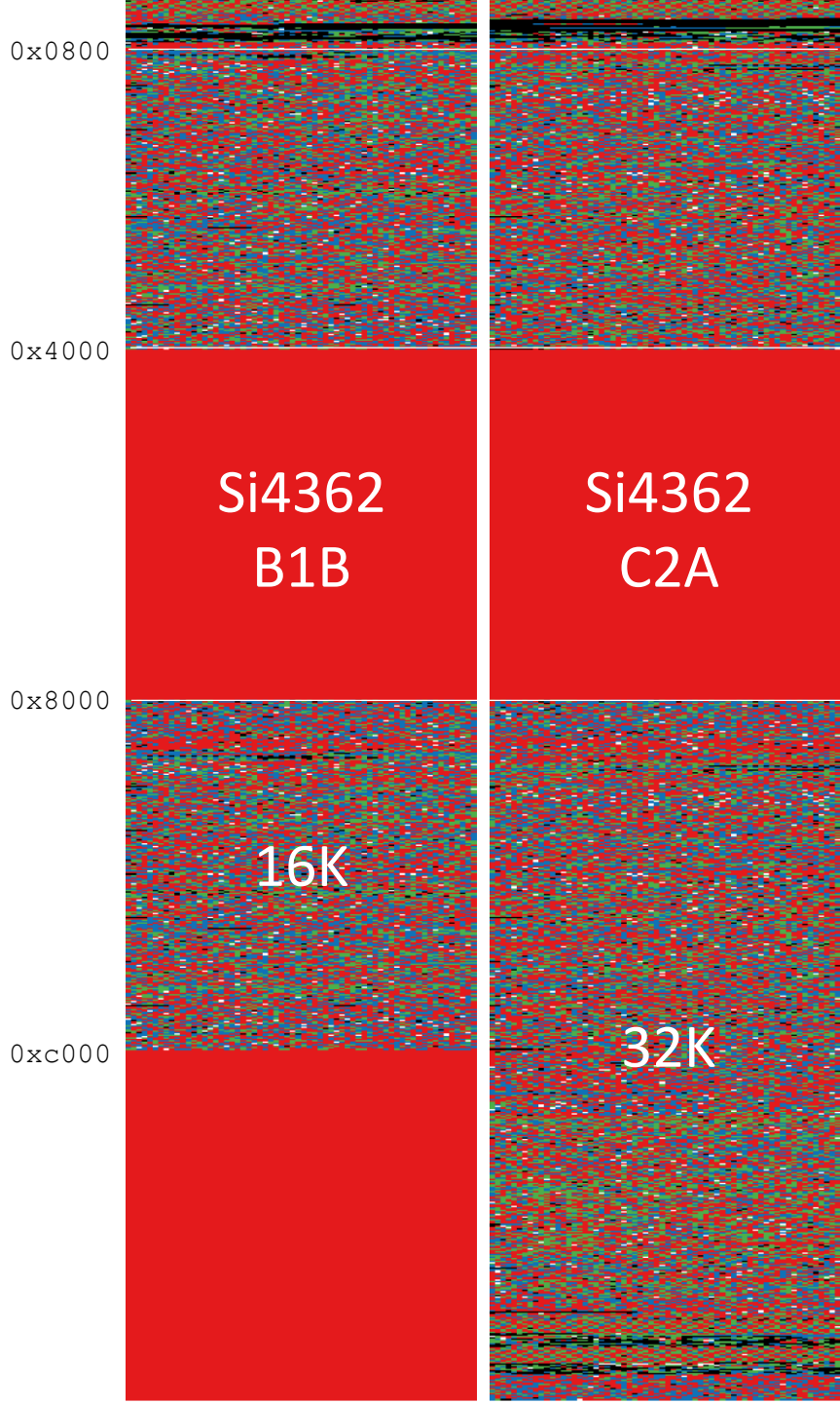
```
hack.memory_dump ();
        ; CODE XREF from map.cmd_get_property (0x168)
        0x00000488              7871            mov r0, #0x71
        0x0000048a              e2              movx a, @r0
        0x0000048b              f583            mov dph, a
        0x0000048d              08              inc r0
        0x0000048e              e2              movx a, @r0
        0x0000048f              f582            mov dpl, a
        0x00000491              7870            mov r0, #0x70
        ; CODE XREF from hack.memory_dump (0x498)
    ┌─> 0x00000493              e4              clr a
    ¦   0x00000494              93              movc a, @a+dptr
    ¦   0x00000495              a3              inc dptr
    ¦   0x00000496              f2              movx @r0, a
    ¦   0x00000497              08              inc r0
    └─< 0x00000498 [1]          b880f8          cjne r0, #0x80, 0x0493
        0x0000049b              22              ret
```

Patch procedure – in pseudo python

```
patch_code = [0x78,0x71,0xe2,..]
patch_loc = 0x488
patch_jmp = 0x168
patch_cmd = 0x12

resetRadio()
sendCommand(POWER_UP)

addr = patch_loc
for data in patch_code:
     print sendCommand(POKE,[addr,data])
     addr = addr+1

sendCommand(POKE,[patch_jmp,patch_loc])

sendCommand(patch_cmd,[params])
```

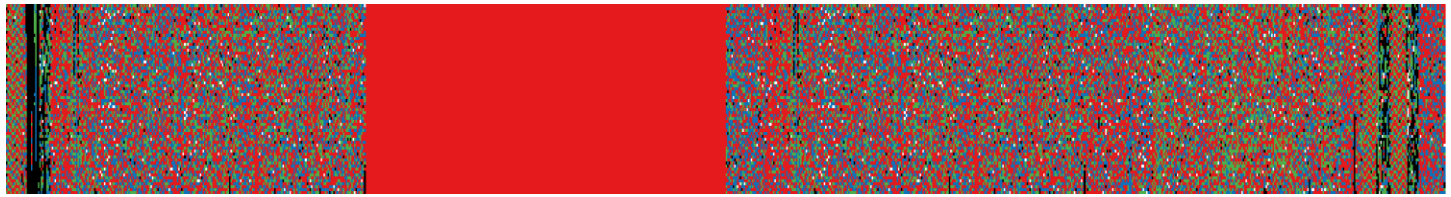
Analyzing CODE dumps

- 0x0000 code – RAM 2K
  - 0x0000 vector table
  - 0x0057 jump table
  - 0x02eb code
  - 0x051e data
- 0x0800 code – mirror of ROM @0x8800
- 0x4000 empty
- 0x8000 code – ROM 16-32K



Comparing CODE dumps

Si4362-C2A with binvis.io



Si4362-B1B vs. Si4362-C2A

Si4362-C2A vs. Si4460-C2A

Si4460-C2A vs. Si4467-A2A

Strategies for reversing embedded firmware

- Strings!

Only few strings

```
[0x00000000]> b 4k
[0x00000000]> p=z
0x00000000 00 0004 |
0x00001000 01 0000 |
0x00002000 02 0000 |
0x00003000 03 0001 |
0x00004000 04 0000 |
0x00005000 05 0000 |
0x00006000 06 0000 |
0x00007000 07 0000 |
0x00008000 08 0002 |
0x00009000 09 0000 |
0x0000a000 0a 0000 |
0x0000b000 0b 0001 |
0x0000c000 0c 0001 |
0x0000d000 0d 0000 |
0x0000e000 0e 0000 |
0x0000f000 0f 0005 |
[0x00000000]>
```

```
[0x00000000]> izz ~ascii
376 0x0000f410 0x0000f410   7   8 () ascii \a\b\t\n\v\f\r
377 0x0000f429 0x0000f429  32  33 () ascii  !"#$%&'()*+,-./0123456789:;<=>?
378 0x0000f4a5 0x0000f4a5   5   6 () ascii g`M6!
379 0x0000f552 0x0000f552   6   7 () ascii `\f\b\f\fK
380 0x0000f597 0x0000f597   5   6 () ascii  \t@<B
382 0x0000f9cc 0x0000f9cc   5   6 () ascii g`M6!
383 0x0000fa79 0x0000fa79   6   7 () ascii `\f\b\f\fK
384 0x0000fab6 0x0000fab6   5   6 () ascii  \t@<B
386 0x0000fb2f 0x0000fb2f   5   6 () ascii Cyy<l
387 0x0000fb40 0x0000fb40  10  11 () ascii filler.txt
390 0x0000fbf9 0x0000fbf9   5   6 () ascii v8u.f
391 0x0000fc16 0x0000fc16   6   7 () ascii .O3\f9+
393 0x0000fcc6 0x0000fcc6   4   5 () ascii nvDc
394 0x0000fcd6 0x0000fcd6   4   5 () ascii *M[c
398 0x0000fdd8 0x0000fdd8   4   5 () ascii S.fD
399 0x0000fdf9 0x0000fdf9   4   5 () ascii mS\rX
400 0x0000fe16 0x0000fe16   4   5 () ascii Ec\b!
401 0x0000fe3e 0x0000fe3e   6   7 () ascii wEKl,v
402 0x0000fe52 0x0000fe52   4   5 () ascii .!c_
403 0x0000fe8d 0x0000fe8d   4   5 () ascii ="BM
404 0x0000fea8 0x0000fea8   4   5 () ascii U`\b.
407 0x0000ff67 0x0000ff67   4   5 () ascii B.cc
408 0x0000ff89 0x0000ff89   4   5 () ascii =lmg
409 0x0000ffc8 0x0000ffc8   4   5 () ascii aw7\e
410 0x0000ffdb 0x0000ffdb   4   5 () ascii l0\rA
411 0x0000fff6 0x0000fff6   6   7 () ascii si4440
[0x00000000]>
```

Strategies to reverse the firmware

- ~~Strings!~~
- aaaaaa

Limitations of r2 anal for 8051

- Size-optimized code breaks af
  - JMP at end of functions
  - CALL into middle of functions
  - Detect broken functions with afl~-
- Function calls via jump table not handled well
- Switch constructs not recognized (JMP @a+dptr)
- Indirect addressing modes not recognized

Limitations of r2 anal – workarounds

- Manually declare all functions
  - f name = location; afu end @ location
  - Macro: "(fcn start end name,f $2 = $0; afu $1 @ $0)“
- Use axr to find code and data references
- Use r2pipe to automate some analysis tasks
  - e.g. create xrefs for indirect register access

Strategies to reverse the firmware

- ~~Strings!~~
- ~~aaaaaa~~ manually construct functions, run axr for references
- Reverse documented API
  - Commands
  - Properties

0x33 REQUEST_DEVICE_STATE

Argument Stream:

REQUEST_DEVICE_STATE Argument Stream										
	Index	**Name**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	0x00	**CMD**	*0x33*							

Reply Stream:

REQUEST_DEVICE_STATE Reply Stream										
	Index	**Name**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	0x00	**CTS**	CTS							
	0x01	**CURR_STATE**	X	X	X	X	MAIN_STATE			
	0x02	**CURRENT_CHANNEL**	CURRENT_CHANNEL							

0x33 REQUEST_DEVICE_STATE

```
adrian@radare:~/oidarze$ r2 -a 8051 dumps/Si4362-C2A-code.bin
 -- Control the height of the terminal on serial consoles with e scr.height
[0x00000000]> /c cjne a, #0x33
0x00001e24   # 3: cjne a, #0x33, 0x1e2a
0x00009e24   # 3: cjne a, #0x33, 0x9e2a
[0x00000000]> pd 2 @0x9e24
        :   ;-- hit0_1:
        ┌──< 0x00009e24      b43303          cjne a, #0x33, 0x9e2a          ; '3'
        │└─< 0x00009e27      020264          ljmp 0x0264
[0x00000000]> pd 1 @0x0264
        ┌─< 0x00000264      029fea          ljmp 0x9fea
[0x00000000]> pd 10 @0x9fea
            0x00009fea      12c84a          lcall 0xc84a
            0x00009fed      7870            mov r0, #0x70                  ; 'p'
            0x00009fef      e51d            mov a, 0x1d                    1d:1]=0
            0x00009ff1      f2              movx @r0, a
            0x00009ff2      90002a          mov dptr, #0x002a              ; '*' ; [0x2000002a:1]=0
            0x00009ff5      e0              movx a, @dptr
            0x00009ff6      08              inc r0
            0x00009ff7      f2              movx @r0, a
            0x00009ff8      22              ret
        ┌─< 0x00009ff9      02cabf          ljmp 0xcabf
[0x00000000]>
```

SPI reply buffer

IDATA 0x1d = curr_state

XDATA 0x2a = current_channel

0x33 REQUEST_DEVICE_STATE

- Assign variable names with flag command
  - Includes virtual registers to emulate 8051 memory layout
  - f var.current_state @ _idata+0x1d
  - f var.current_channel @ _xdata+0x2a

- Rinse and repeat
  - Cover all documented functions and properties, or
  - Focus on functionality of interest

Strategies to reverse the firmware

- ~~Strings!~~
- ~~aaaaaa~~ manually construct functions, run axr for references
- Reverse documented API commands
- Save project

Strategies to reverse the firmware

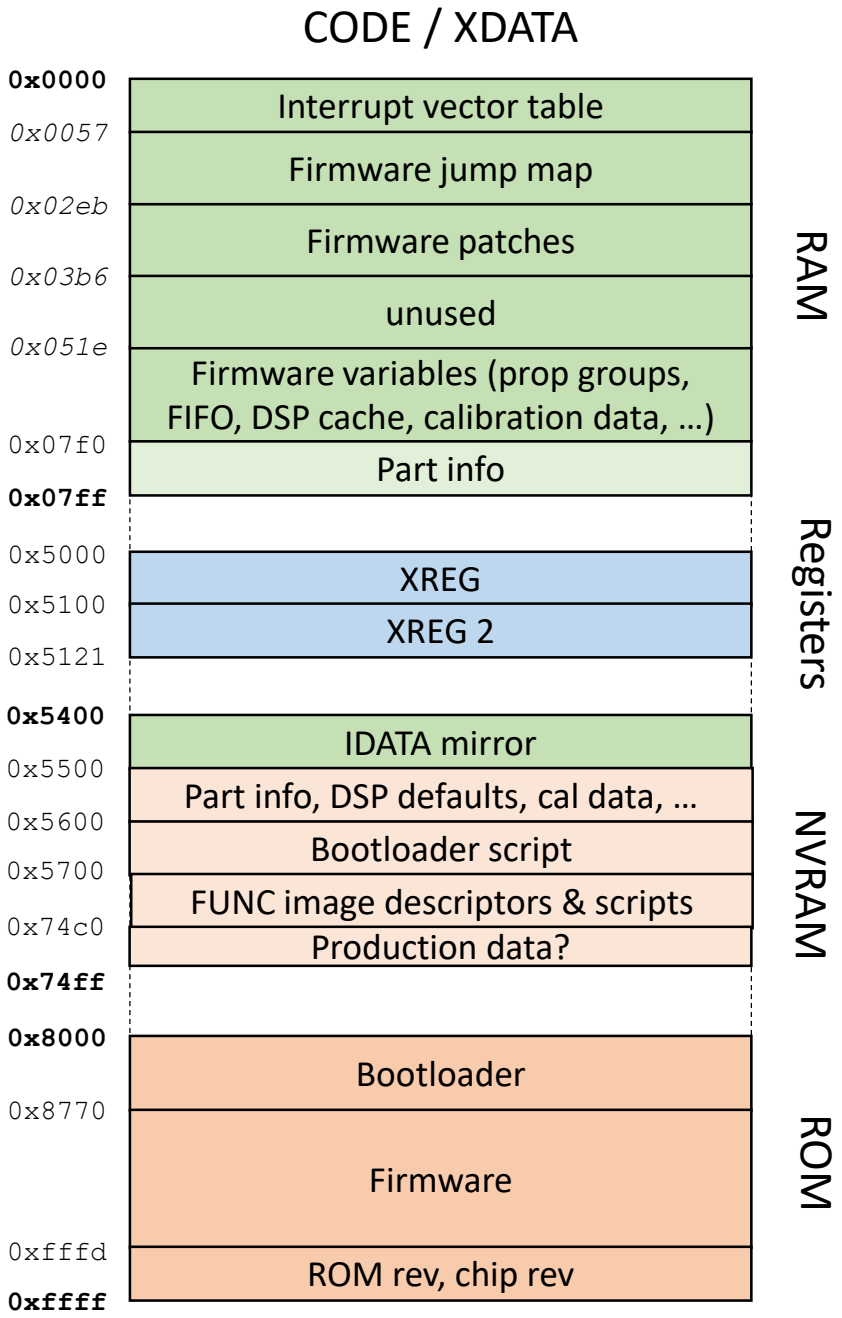
- ~~Strings!~~
- ~~aaaaaa~~ manually construct functions, run axr for references
- Reverse documented API commands
- ~~Save project~~ use r2 scripts
  - common.r2 – config r2, standard 8051 stuff
  - rom-[rev].r2 – ROM
  - registers-[rev].r2 – SFR, other registers
  - xdata-[rev].r2 – external RAM
  - func1-[rev].r2, boot-[rev].r2 – root script, calls other scripts, IDATA

What I learned about the EZRadioPRO

(so far)

Memory map (C2A, FUNC 1)

- Shared CODE and XDATA address space
- 2K XDATA + 256 bytes IDATA RAM
- 32K ROM
- 3 sets of registers
  - SFR
  - XREG/XREG2 (XDATA 0x5000-0x51ff)
  - “DSP” via SFR 0x94/0x95
- 4-8K NVRAM
  - Access via DMA



Bootloader

- Runs after reset / SDN
- Entry point 0x8000
- Determine programming state (factory, user, run)
- Copy factory data from NVRAM to XDATA
  - 0x07f0-0x07ff device identification and capabilities
  - 0x0761-0x0797 calibration data
  - 0x0798-0x07ef register (DSP?) defaults
- Run reset script from NVRAM
- Enable SPI and process API calls, until POWER_UP
- Load selected FUNC image from NVRAM and JMP to 0x0003

Undocumented bootloader commands

- PATCH_CLEAN (0x03)
- PATCH_IMAGE (0x04)
  - Also called by POWER_UP to load FUNC image from NVRAM and ROM
- PATCH_ARGS (0x05)
- PATCH_COPY (0x06, 0x0a)
- PATCH_DATA (0xe0-0xef)

- READ_NVRAM_74C0 (0x09)
- TAIL_ROM (0xfe)

Undocumented runtime commands (FUNC1)

- PROTOCOL_CFG (0x18)
  - Config for IEEE802.15.4g, documented in programming guide rev 0.5
- EZCONFIG_CHECK (0x19)
  - Verify image CRC, similar command documented for Si4455
- OFFLINE_RECAL (x038)
  - Partially documented in SiLabs knowledge base
- CMD_1B (0x1b), CMD_35 (0x35), CMD_D0 (0xd0), CMD_F2 (0xf2)
  - Function unknown
- PEEK (0xf0), POKE (0xf1)

Undocumented property groups

- CAL_DATA (0xf2)
  - 0x37 bytes
  - Copied from NVRAM at boot
  - Probably calibration data
- UNK_0x24 (0x24)
  - 9 bytes
  - Unknown function, some byte related to XO_TUNE, MOD_TYPE

Undocumented FUNC images

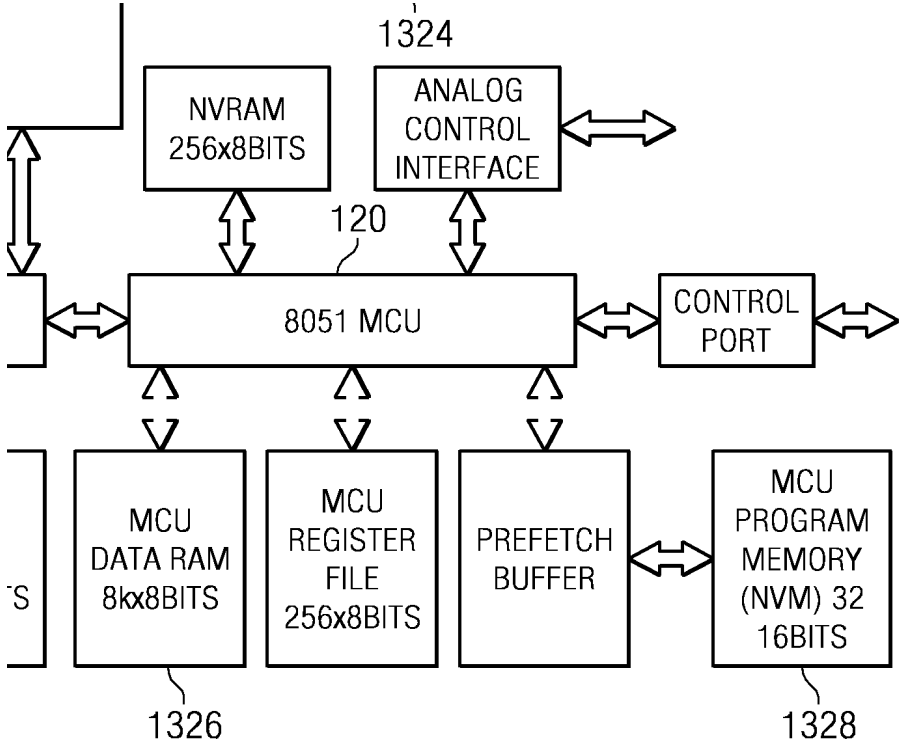
- NVRAM and ROM includes additional FUNC images

- FUNC2
  - Shares most config API commands with runtime (FUNC1), incl PEEK/POKE
  - Undocumented commands 0x80-0x85, 0x87, 0x8a, 0x8c, 0x8d
  - No START_RX/TX commands

- FUNC3
  - More reversing required

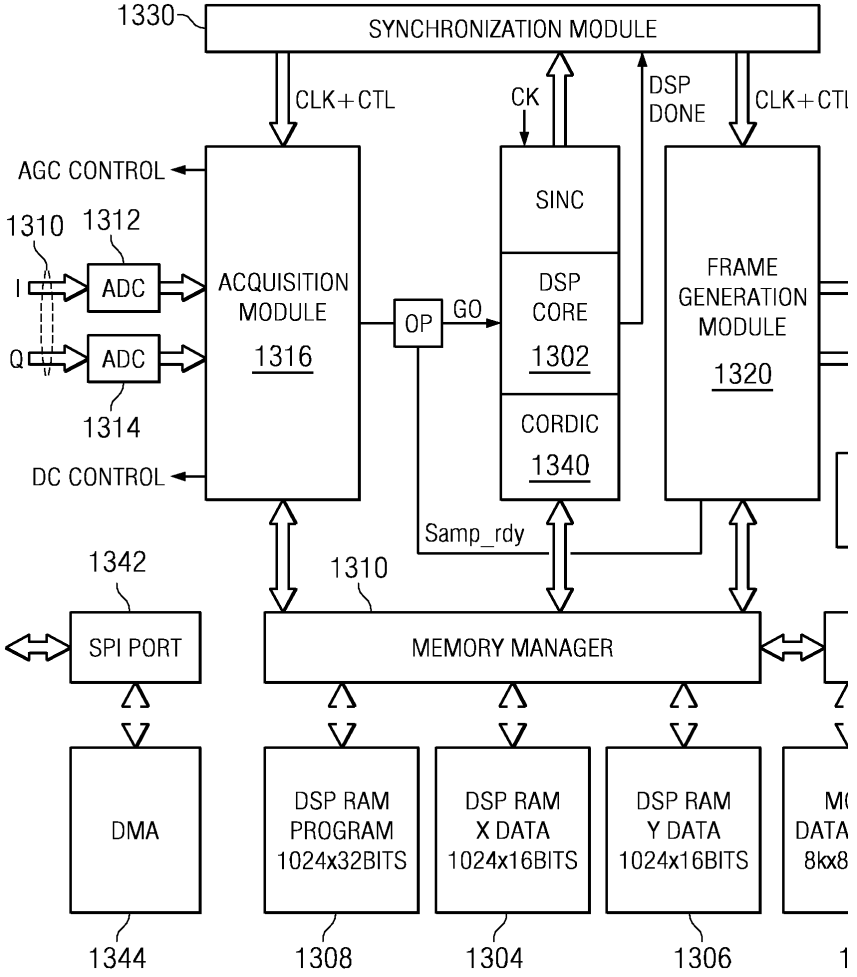
What is done by the 8051?

- Configuration
- Radio state machine
- Packet handler
  - ISR when byte received / sent
  - FIFO in RAM (XDATA)
  - Non-std preamble
  - Software CRC & whitening
- Image rejection calibration
- RSSI latch, antenna diversity (RX)
- Power amplifier control (TX)
- ADC for temperature & voltage



What is done by the DSP or in hardware?

- SPI communication
- Some API command
  - FRR_x_READ, READ_RX_FIFO
- Modem
- Standard preamble
- HW CRC & whitening
- IF ADC / DAC
- GPIO (requires more research)



Future research

- Decrypting and creating valid firmware patches
- Understand Undocumented cmds and FUNC images (factory tests?)
- Access to DSP (RAM? ROM?)
- Writing to NVRAM
- Explore other SiLabs radios (e.g. Si46xx/Si47xx FM transceivers)
- Decap a few ICs

Project Inside-EZRadioPRO

- https://github.com/astuder/Inside-EZRadioPRO
- What I will share:
  - Python code to dump firmware
  - R2 scripts
  - Reversed documentation (like this deck)
- What I won't share:
  - Firmware binaries (use Python script to dump your own)
  - SiLabs documentation (use Google)
- Pull requests are welcome

Thank you!

Questions?

@adistuder

https://github.com/astuder/