RFQuack

The versatile RF-analysis tool that quacks!

With ♥ From Trend Micro Research

Presented at HITB 2019 Armory by:
Federico Maggi, @phretor
This work wouldn't have been possible without the support of my employer.

In particular, I'd like to thank:

- **Managers** and **execs**, who believed in this project and let me work on it
- **Jonathan Andersson**, who inspired and helped me debugging the quirks of the CC1120
- **Philippe Lin**, an early adopter of (the first-ever prototype of) RFQuack
- **Marco Balduzzi**, who never stopped asking me "how's RFQuack going?"
- **Jullienne Yerro**, and the rest of the marketing team for the beautiful logo (kudos to Jojo Mendoza for that) and the media support
Signal Analysis 101
From Symbols to Signal: **Baseband Data**
From Symbols to Signal: **Amplitude Shift**
From Symbols to Signal: Frequency Shift
From Symbols to Signal: Phase Shift
From Signal to Symbols
The Hard Part is not Over
From Bits to Packets
Still, we Haven’t Reverse Engineered the Protocol
From Packets to Application Payload

Custom application protocol
(with security through obscurity baked in, usually)
Software Defined Radios
SDRs - Main Idea: Take Many RF Signal Samples
SDRs: Pros vs. Cons

- Great for signal **reconnaissance**
- Very flexible: you get straight access to the raw **signal**
- **Software** support to assist in writing radios
- You have to **write your own radio** in software
- Radio **accuracy** is up to you
- Serious ones can be **expensive**
Bottom Line

It’s hard to build an accurate and reliable radio
RF Dongles
RF Dongles - Main Idea: Embedded Radio

```
root@edolin ~/rfcat $ ./rfcat -r
'RfCat, the greatest thing since Frequency Hopping!'

Research Mode: enjoy the raw power of rflib

currently your environment has an object called "d" for dongle. this is how you interact with the rfcat dongle:

>>> d.ping()
>>> d.setFreq(4330000000)
>>> d.setMdmModulation(MOD_ASK_OOK)
>>> d.makePktFLEN(250)
>>> d.RFxmit("HALLO")
>>> d.RFrecv()
>>> print d.reprRadioConfig()
```

In [1]: 1
RF Dongles: Pros vs. Cons

- Great to **quickly demodulate** signals
- Very **accurate**: you get reliable access to the demodulated bitstream
- As **fast** as the **hardware** radio

- **Not as flexible** as SDRs
- Demodulation support is **limited** to what the **hardware** can do
Bottom Line
There’s no such thing like a "generic RF dongle"
The Perfect Corner Case
TI CC11xx's in 4-FSK

It's still 4-FSK, but it uses only 2 symbols for preamble and sync.

Then switches to 4 symbols.
But but...the TI CC1111 can do 4-FSK

| 6:4 | MOD_FORMAT[2:0] | 000 | RW
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>2-FSK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>GFSK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>ASK/OOK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>Reserved</td>
<td>4-FSK</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>MSK</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The sync word is a two-byte value set in the SYNC1 and SYNC2 registers. The sync word provides byte synchronization of the incoming packet. A one-byte sync word can be emulated by setting the SYNC1 value to the preamble pattern. It is also possible to emulate a 32-bit sync word by using MDMCFG2.SYNC_MODE set to 3 or 7. The sync word will then be repeated twice.
Principles
Hardware Modularity

- Detachable RF Module
- Supports any embedded radio
- UART
- MCU
- Supports "any" MCU
- Optional Cellular or WiFi Module
Software Abstraction With Full Low-level Control

- **High-level operations**
  - Set frequency
  - Switch mode (TX, RX, IDLE)
  - Reset radio

- **Low-level operations**
  - Set register to value
  - Get register value
  - Upcoming: straight access to make SPI transactions from the Python client
Developer Friendly

- C + Arduino compatible + build system based on PlatformIO
- Simple and clean API: Inspired by, and including MQTT
  - Inbound: `[command]~Base64([Protobuf-serialized blob])`
  - Outbound: `<[command]~Base64([Protobuf-serialized blob])`

- Verbose, configurable logging facility

From RFQuack

- `[RFQ] 156 T: RFQuack data structure initialized: WEMOSD1_CC1120`
- `[RFQ] 464 T: Connecting WEMOSD1_CC1120_6c54 to MQTT broker 192.168.42.225:1883`
- `[RFQ] 2117 T: MQTT connected`
- `[RFQ] 2130 T: Subscribed to topic: rfquack/in/#`

From the radio driver

- `[RHAL] SRES`
- `[RHAL] SCAL`
- `[RHAL] SIDLE`
- `[RHAL] START MARCSTATE.MARC_STATE ==============================`
  - Waiting for MARCSTATE.MARC_STATE == 0b1`
- `[RHAL] END MARCSTATE.MARC_STATE ==============================`
- `[RHAL] IRQ bus clear`
- `_variablePayloadLen = 1`
Cut the Cords

RFQuack Dongle

Detachable
RF Module

UART
MCU

Optional
Cellular or WiFi
Module

WiFi, Cellular
## Comparison Matrix

<table>
<thead>
<tr>
<th></th>
<th>SDRs</th>
<th>YardStickOne</th>
<th>PandwaRF</th>
<th>RFQuack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supported Radios</td>
<td>Any (software)</td>
<td>CC1101</td>
<td>CC1101</td>
<td>Any (even multi radio)</td>
</tr>
<tr>
<td>Client Support</td>
<td>Lots of options</td>
<td>RFCat firmware and client</td>
<td>RFCat client</td>
<td>Developer-friendly API</td>
</tr>
<tr>
<td>Open Software</td>
<td>Depends</td>
<td>Yes</td>
<td>Not the firmware</td>
<td>Yes, Arduino compatible</td>
</tr>
<tr>
<td>Open Hardware</td>
<td>Depends</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Connectivity</td>
<td>USB, Gigabit</td>
<td>USB</td>
<td>USB, BT</td>
<td>USB, WiFi, Cellular</td>
</tr>
<tr>
<td>Price</td>
<td>$20–2000</td>
<td>&gt;= $100</td>
<td>&gt;= $110</td>
<td>&gt;= $40</td>
</tr>
</tbody>
</table>
Getting Started
Get and Assemble the Hardware
What the Hardware!?

● Pick any SPI (Serial Peripheral Interface) embedded radio module
  ○ Available anywhere from Adafruit, Sparkful, eBay, Amazon, AliExpress
  ○ RFM69, CC1111, CC1120, nRF24, nRF51

● Hint: there are pre-made shields for popular radios (e.g., FeatherWing Radio)

● Connect SPI pins
  ○ MOSI
  ○ MISO
  ○ SCLK
  ○ CS

● Plus at least one interrupt line to the MCU's GPIO pin

● Add an antenna
Make it Nicer (and give it a modem)
Check out the Code

$ git clone https://github.com/trendmicro/RFQuack
$ cd RFQuack
$ pip install -r src/client/requirements.pip
$ pio install -g <library name>  # from library.json
$ cd examples/
<table>
<thead>
<tr>
<th>Repository Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFQuack-huzzah-rf69hw</td>
<td></td>
</tr>
<tr>
<td>RFQuack-huzzah-rf69hw-ser</td>
<td></td>
</tr>
<tr>
<td>RFQuack-wemosd1-cc1120-ser</td>
<td>RFQuack-&lt;board&gt;-&lt;radio&gt;-&lt;transport&gt;</td>
</tr>
<tr>
<td>RFQuack-wemosd1-cc1120</td>
<td>&lt;no transport&gt; = MQTT, by default</td>
</tr>
<tr>
<td>RFQuack-wemosd1-rf69hcw-ser</td>
<td></td>
</tr>
<tr>
<td>RFQuack-wemosd1-rf69hcw</td>
<td></td>
</tr>
</tbody>
</table>

Federico Maggi Pre HITB release
Configure the Firmware "src/main.cpp"

```cpp
#include "rfquack.h"

void setup() { rfquack_setup(); } void loop() { rfquack_loop(); }
```

```cpp
#define RFQUACK_UNIQ_ID  "WEMOSD1_CC1120"       // <- unique ID
#define RFQUACK_NETWORK_ESP8266
#include "wifi_credentials.h"                  // <- not committed because it contains secrets

#define RFQUACK_TRANSPORT_MQTT
#define RFQUACK_MQTT_BROKER_HOST "192.168.42.225" // <- MQTT broker IP or hostname (credentials are supported too)

#define RFQUACK_RADIO_CC1120                     // <- Radio chip (CC1120 and RF69 are supported as of now)
#define RFQUACK_RADIO_PIN_CS  15                // <- SPI Slave select PIN
#define RFQUACK_RADIO_PIN_IRQ   4                // <- Interrupt PIN
#define RFQUACK_RADIO_PIN_RST   5                // <- Reset PIN
#define RFQUACK_DEBUG_RADIO true
#define RFQUACK_DEV
#define RFQUACK_LOG_SS_DISABLED // <- Disable SoftwareSerial logging (we're using HardwareSerial)
```
Mind the Serial Port in "platformio.ini"

[env:d1_mini]
platform = espressif8266
board = d1_mini
framework = arduino
upload_port = /dev/cu.wchusbserial14110
monitor_port = /dev/cu.wchusbserial14110
upload_speed = 115200
monitor_speed = 115200
Build the Firmware

$ git clone https://github.com/trendmicro/RFQuack
$ cd RFQuack
$ pip install -r src/client/requirements.pip
$ pio install -g <library name>  # from library.json
$ cd examples/
$ make && sleep 1 && make upload && make monitor
$ mosquitto -v  # if using MQTT transport
Boot and Connect

[RFQ] 152 T: Setting sync words length to 4
[RFQ] 153 T: Packet filtering data initialized
[RFQ] 154 T: Packet modification data initialized
[RFQ] 156 T: RFQuack data structure initialized: WEMOSD1_CC1120
[RFQ] 464 T: Connecting WEMOSD1_CC1120_6c54 to MQTT broker 192.168.42.225:1883
[RFQ] 2117 T: MQTT connected
[RFQ] 2130 T: Subscribed to topic: rfquack/in/#
[RFQ] 2231 T: Setting up radio (CS: 15, RST: 5, IRQ: 4)
[RFQ] 3141 T: Radio initialized (debugging: true)
[RFQ] 3142 T: CC1120 type 0x4823 ready to party 🎉
[RFQ] 3144 T: Modem config set to 5
[RFQ] 3147 T: Max payload length: 128 bytes
[RFQ] 3151 T: Radio is fully set up (RFQuack mode: 4, radio mode: 2)
[RFQ] 3258 T: Transport is sending 26 bytes on topic rfquack/out/status
RFQuack data structure initialized: WEMOSDI_CC1120
RFQuack connected
RFQuack subscribed to topic: rfquack/in/#
Setting up radio (CS: 15, RST: 5, IRQ: 4)

RFQuack serial console output (optional, but very useful)

1557990916: Sending PINGREP to WEMOSDI_CC1120_28e7
1557990920: Received PINGREQ from WEMOSDI_CC1120_28e7
1557990920: Sending PINGREP to WEMOSDI_CC1120_28e7
1557990924: Received PINGREQ from WEMOSDI_CC1120_28e7
1557990924: Sending PINGRESP to WEMOSDI_CC1120_28e7

MQTT broker output (optional)

RFQuack(RFQuackShell, localhost:1883)
Get status of the RFQuack dongle

**Packet statistics**

```python
RFQuack(RFQuackShell, localhost:1883).q.get_status()
```

```python
stats {
  rx_packets: 0
  tx_packets: 0
  rx_failures: 0
  tx_failures: 0
  tx_queue: 0
  rx_queue: 0
}
```

*mode: IDLE*

*modemConfig {
  syncWords: "EDCB"
}*

*tx_repeat_default: 0*

```python
RFQuack(RFQuackShell, localhost:1883).
```

```python
RFQuack(RFQuackShell, localhost:1883).q.set_modem_config(syncWords='\x93\x8B\x51\xD8', txPower=8)
```

```python
RFQuack(RFQuackShell, localhost:1883).q.data
```

**Modem status**

```python
Out[3]:
```

```python
u'status': {stats {
  rx_packets: 0
  tx_packets: 0
  rx_failures: 0
  tx_failures: 0
  tx_queue: 0
  rx_queue: 0
}
```

*mode: IDLE*

*modemConfig {
  syncWords: "EDCB"
}*

*tx_repeat_default: 0}*

```python
RFQuack(RFQuackShell, localhost:1883)
```

**Data exchanged between dongle and client is retained in q.data**
DEMO
Talking Nodes
Main Functionalities
Modem Configuration: \texttt{q.set_modem_config()}

\begin{verbatim}
> q.set_modem_config(
    modemConfigChoiceIndex=0,       # canned RadioHead/RadioHAL modem config
    txPower=14,                     # TX output power (sometimes in dB)
    isHighPowerModule=true,         # required by some radio modules
    syncWords=b'\x43\x42',         # sync words
    preambleLength=4,               # number of bytes of preamble
    carrierFreq=433)                # and of course, carrier frequency
\end{verbatim}
Canned Modem Configuration

- Each RadioHead/RadioHAL driver has canned modem configurations
- It's an enum type, so `modemConfigChoiceIndex` is the index
- Examples:
  - FSK_Rb2Fd5
    - FSK modulation
    - With data whitening
    - Receiver bandwidth: 2kb
    - Frequency deviation: 5kHz
  - GFSK_Rb9_6Fd19_2
  - OOK_Rb1_2Bw75
- More at: [https://www.airspayce.com/mikem/arduino/RadioHead](https://www.airspayce.com/mikem/arduino/RadioHead)
- For RadioHAL: [https://github.com/trendmicro/radiohal](https://github.com/trendmicro/radiohal)
Transmit, Receive

```python
> q.set_packet('\x0d\xa2', 13)  # TX'0x0d 0xa2' 13 times
```
- Accepts any raw binary data
- Data size limited by the radio driver (i.e., size of the TX FIFO)
- Re-transmission times limited by RFQuack's TX queue length

```python
> q.rx()  # put radio in RX mode
```
- Will save packets into `q.data['packet']`
- Receive rate limited by RFQuack's RX queue length
- Maybe obvious: will match data according to modem config.
DEMO
Sniffing a Weird Protocol
Register Access (a.k.a. program the radio chip)

```python
q.set_register(0x2e, 0b01000000)  # register address (8 or 16 bits)
q.set_register(0b01000000)  # register value (you can write in HEX or DEC too)

time.sleep(0.2)  # especially if you set many registers in a row
```

- You could **bypass** any (modem) configuration
- You should **study the datasheet** of the radio chip
- You could easily "hang" the radio and RFQuack (just push reset)
Scripting Up!

```python
q.set_modem_config(txPower=14, syncWords=b'\x43\x42', carrierFreq=433)
my_reg_vals = [
    (0x2e, 0x33),
    (0x2f, 0x32),
    (0x01, 0x8D),
]

for a,v in my_reg_vals:
    q.set_register(a,v)
    time.sleep(0.2)
q.rx()
```

You can create your own "library" of reusable settings.
Packet Filtering and Manipulation
Packet Filtering

- Simple filtering done by the radio
  Configurable via registers

- Complex filtering done by RFQuack
  Configurable via regexes
Packet Manipulation

Content-based byte-level access

Operations:
- XOR
- AND
- NOT
- >>
- <<

Positional byte-level access
Conditional Packet Manipulation

```java
if (match("Payload", filter_pattern)) {
    if (match("Payload", pattern1))
        Payload1 = modify("Payload", rule1)
    if (match("Payload1", pattern2))
        Payload2 = modify("Payload1", rule2)
    if (match("Payload2", pattern3))
        Payload3 = modify("Payload2", rule3)
    :  
    if (match("PayloadN-1", patternN))
        Payload1 = modify("PayloadN-1", ruleN)
}
```
DEMO
Reverse Engineering a Weird Protocol
Architecture
High Level
The Radio and Firmware Side

RF Module

Demodulation
- In-hardware packet filters
  - RSSI quality
  - preamble quality
  - sync word quality
  - length field
  - address field
  - CRC

Modulation
- In-hardware packet processors
  - calculate and append CRC
  - prepend sync word
  - prepend preamble

RX Module

RX Loop
- RX FIFO
- Drop
- Matching
- Mode
- Repeater
- Inbound loop
- Outbound loop
- Request Dispatcher
- Transmit payload
- Register get/set
- Internal statistics

RFQuack Firmware

TX Module

TX Loop
- TX FIFO
- Memory
- Other registers
Future
Performance Improvements

- Interrupt-driven RX function: no polling in the firmware
- Using the radio's TX FIFO buffering when available (reduce SPI traffic for repeated transmissions)
- Make RadioHAL thinner and closer to the radio (less abstract wherever possible)
- Optimize the packet filtering/manipulation engine
Test Other Radios (e.g., 2.4GHz, LoRa)
Testing More Platforms
Hardware Shield and Adapters
Making a FeatherWing SIM800 (no, not the FONA)
Integrations and Other Enhancements

- GNU Radio and URH
- Web app interface
- Expose a SPI API
- Multiple radio modules (shared SPI bus, 1 IRQ and 1 SS line each)