OFDM Packet Receivers in GNU Radio

Martin Braun (Ettus Research / GNU Radio)

FOSDEM 2014

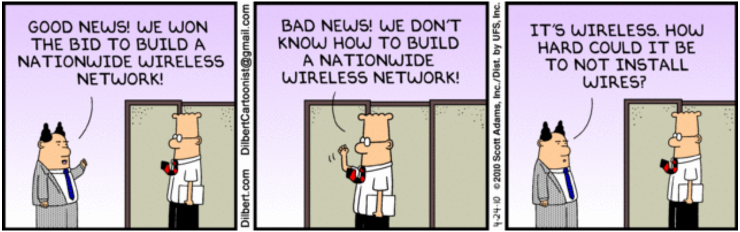
Introduction

- Who is this guy:
  - `mbr0wn`
  - GNU Radio contributor since 2008
  - KIT graduate
  - Now full-time SDR developer for Ettus Resarch LLC

Part I – OFDM PHY Development

Part I – OFDM PHYs

- What is OFDM?
- How can we build OFDM-based PHY layers in GNU Radio?



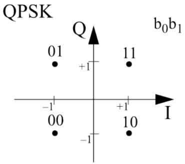
Outline

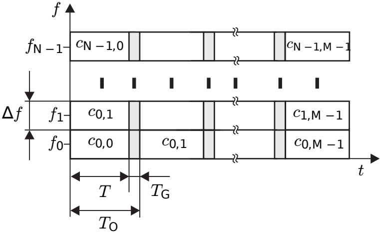
What is OFDM?

- Orthogonal Frequency Division Multiplexing: Transmit many narrow-band signals in parallel on orthogonal frequencies
- “Good way to transport lots of digital data over the air”
- Used in many standards (LTE, Wifi, DVB, DAB, . . .)

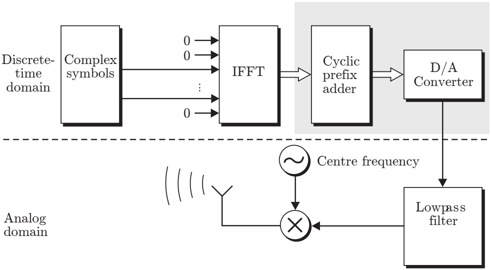
Anatomy of an OFDM signal

- Complex modulations symbols (BPSK, QPSK, . . .)
- OFDM symbols: Set of complex modulation symbols transmitted at once
- Subcarriers: Discrete frequencies on which data are transmitted
- Frame: Set of OFDM symbols
- Header: Carries info on frame, helps synchronization. . .
- Pilot symbols: Special symbols, known a-priori





An OFDM transmitter

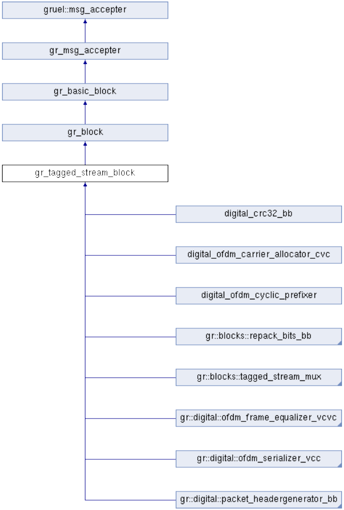


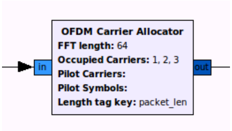
- Efficient sub-carrier modulation via IFFT (creates baseband signal)
- Cyclic prefix: Creates space between OFDM symbols
- ...so how do we make on of these in GNU Radio?
- No states!

Outline

gr_tagged_stream_blocks

- Handle stream boundaries
- Input-driven
- Uses tags
- Not really the same category as sync, decimator, interpolator
- Tag on the first item defines packet length
- Examples:
  - CRC32
  - OFDM-Frame operations
  - More to follow





Outline

GNU Radio OFDM Codes

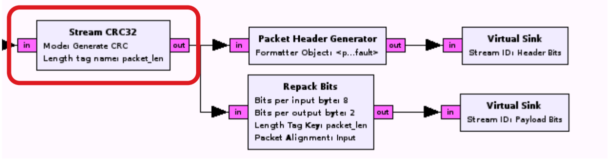
- Disclaimer: There are two versions of OFDM codes in GNU Radio
- All of this depends on the new codes!
- Where to start:
  - `gr-digital/examples/ofdm/*.grc`
  - “OFDM Transmitter” and “OFDM Receiver” hierarchical blocks
  - In Python: `digital.ofdm_rx` and `digital.ofdm_tx`
- Many recent developments have gone into this (tags, message passing, tagged stream blocks. . .)

OFDM – Wishlist

- Fully configurable frame configuration (pilot tones, occupied carriers. . .)
  - Can we reconfigure the whole thing to do 802.11a *and* DAB?
- Any part of the flow graph should be exchangeable
- . . . and individually useful

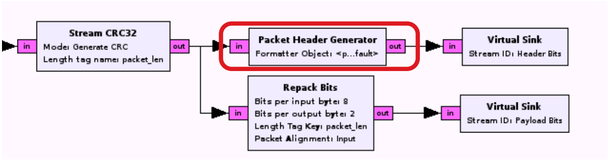
Outline

The OFDM Transmitter



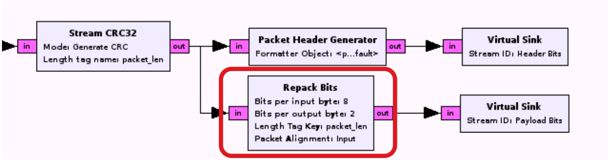
- CRC block: Output is always 4 bytes longer than input

The OFDM Transmitter



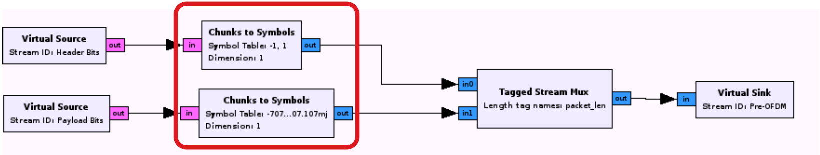
- CRC block: Output is always 4 bytes longer than input
- Packet header generator: Evaluates payload and metadata to generate custom payload

The OFDM Transmitter



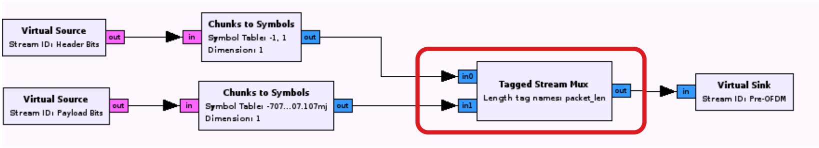
- CRC block: Output is always 4 bytes longer than input
- Packet header generator: Evaluates payload and metadata to generate custom payload
- Bit repacker: Prepare for modulation, handles odd numbers of bits

The OFDM Transmitter



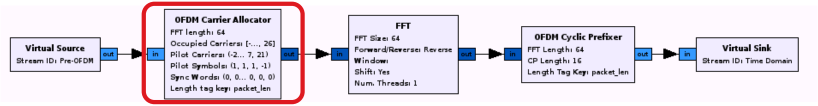
- Symbol mappers: Regular blocks
- None of the code after the mappers cares about the actual complex values (enforce boundaries!)

The OFDM Transmitter



- Symbol mappers: Regular blocks
- None of the code after the mappers cares about the actual complex values (enforce boundaries!)
- Multiplexer: Respects tag positions and boundaries

The OFDM Transmitter



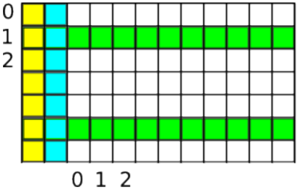
- Carrier allocator: Distributes symbols in time and frequency, adds pilot symbols and headers

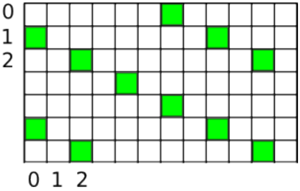
The OFDM Transmitter

- Carrier allocator: Distributes symbols in time and frequency, adds pilot symbols and headers
- Cyclic prefixer: Includes rolloff

Pilot allocation

- Pilot symbols: Known symbols to aid the receiver



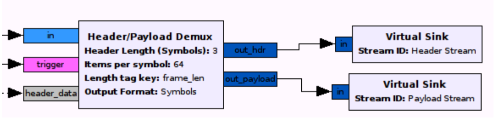


- Pilot symbols can be allocated in any manner
- “Wifi-style:” `( (1, 5), )`
- “DRM-style:” `( (1, 5), (), (2, 6), (), ...)`
- Constant header can be injected

Outline

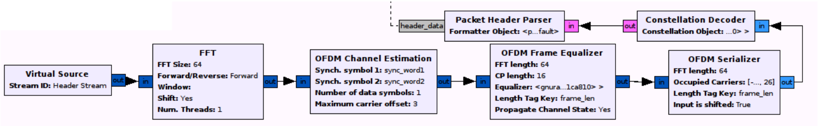
The OFDM Receiver

- How can we find a packet, decode its header and then act depending on the configuration?



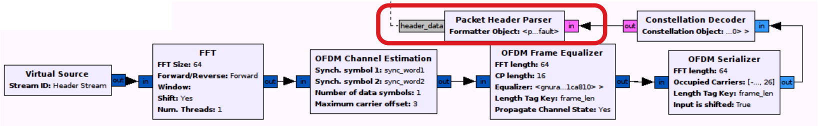
- Waits for packet detection
  - “High” signal at the trigger input denotes start of packet
  - Tags can also denote start of packet
- Pipe header to first sub-flow graph
- Wait for decoding, use header info to determine length of payload
- Pipe payload to second sub-flowgraph

The OFDM Receiver



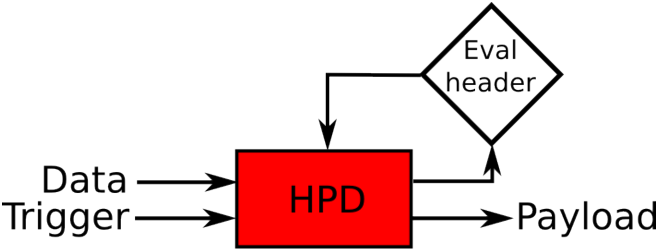
- Channel estimator / equalizer: Reverse the effects of the radio channel

The OFDM Receiver



- Channel estimator / equalizer: Reverse the effects of the radio channel
- Header parser: Uses the same object as the header generator
- Passes information to the HPD as an asynchronous message (“feedback”)

The OFDM Receiver

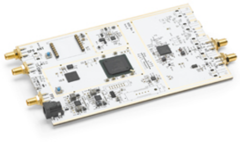


- Waits for packet detection
  - “High” signal at the trigger input denotes start of packet
  - Tags can also denote start of packet
- Pipe header to first sub-flow graph
- Wait for decoding, use header info to determine length of payload
- Pipe payload to second sub-flowgraph

Outline

Over the air

- My setup:
  - RTLSDR Dongle (gr-osmocom)
  - USRP B210
  - GNU Radio (current version)
  - gr-osmosdr + dependencies
  - That’s it – no magic extra libraries



Getting it running

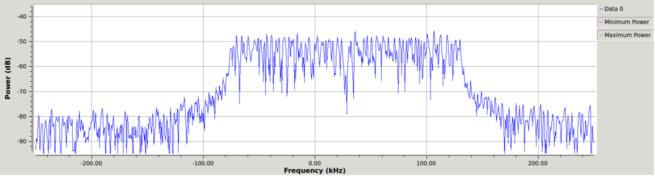
- Use hierarchical blocks (“OFDM Transmitter”, “OFDM Receiver” in GRC)
  - Let’s try that!

Getting it running

- Use hierarchical blocks (“OFDM Transmitter”, “OFDM Receiver” in GRC)
  - Let’s try that!
- Make sure signal amplitude is in valid range (PAPR!)
- Play around with gains
- Add rolloff
- Avoid DC spurs

Getting it running

- Use hierarchical blocks (“OFDM Transmitter”, “OFDM Receiver” in GRC)
  - Let’s try that!
- Make sure signal amplitude is in valid range (PAPR!)
- Play around with gains
- Add rolloff
- Avoid DC spurs
- This is what you want at the receiver:



Demo

- 250 kHz bandwidth
- QPSK
- max. 375 kbps
- Downsides:
  - Heavy CPU usage
  - No FEC

How do I build my own OFDM transceivers?

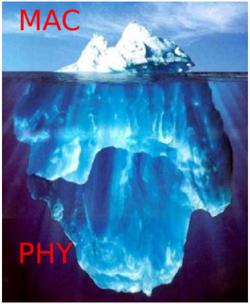
- Fastest dev path: Change as little as possible
- Critical components:
  - Synchronization / Detection
    - Find begin of packets
    - Correct fine frequency offset
  - Header formatter
    - Generate and parse headers (let's have a look at them. . .)
  - (Equalizer)
    - Stock equalizers might be enough
  - See also Bastian's talk!

Packet Header Object

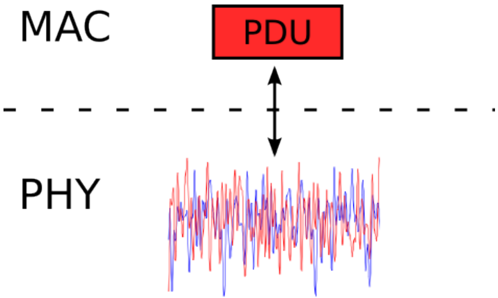
gr-digital/include/gnuradio/digital/packet_header_default.h

```
/*!
 * \brief Encodes the header information in the given tags into bits and places them into \p out
 */
virtual bool header_formatter(
    long packet_len,
    unsigned char *out,
    const std::vector<tag_t> &tags=std::vector<tag_t>()
);

/*!
 * \brief Inverse function to header_formatter().
 */
virtual bool header_parser(
  const unsigned char *header,
  std::vector<tag_t> &tags);

static sptr make(
        long header_len,
        const std::string &len_tag_key="packet_len",
        const std::string &num_tag_key="packet_num",
        int bits_per_byte=1);
```

7 Asynchronous operation: Messages



Synchronous vs. asynchronous operation

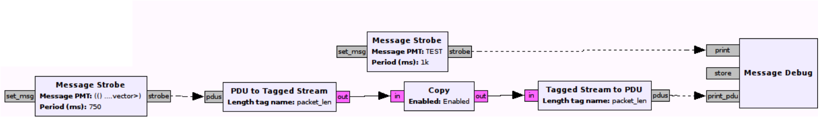


- PHY layer: streaming-oriented (samples)
- MAC layer: packet-oriented, timing constraints
- How do we traverse this boundary?

Outline

7. Asynchronous operation: Messages

Message passing interface



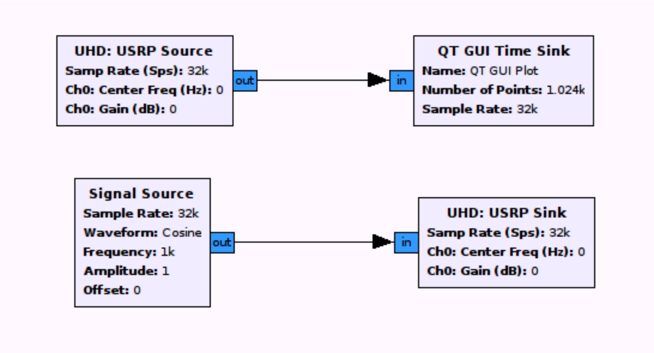
- Remember the header/payload demultiplexer?
- Dotted lines mean asynchronous data passing
- We can switch between domains!
- Both domains support metadata transport (tags)

What metadata are understood?

- Looking at gr-uhd documentation:
  - `rx_freq`, `rx_time`, `tx_time`
- Header/payload demuxer can be told about these items!
- Seems like it's all there to start implementing!

Transceivers

- If you have a device that supports it, you can set up half-duplex transceivers without any additional efforts



Simplest Possible MAC

```python
import numpy
import pmt
from gnuradio import gr

class mac(gr.sync_block):
    def __init__(self):
        gr.sync_block.__init__(self, name="mac", in_sig=None, out_sig=None)
        self.message_port_register_in(pmt.intern('pdus'))
        self.message_port_register_out(pmt.intern('data'))
        self.set_msg_handler(pmt.intern('pdus'), self.receive_pdu_from_phy)

    def receive_pdu_from_phy(self, msg):
        meta = pmt.to_python(pmt.car(msg)) # This is a dictionary!
        vect = pmt.cdr(msg)
        self._evaluate_metadata(meta)
        self._send_pdu_to_higher_layer(meta, vect)

    def send_pdu_to_phy(self, data):
        meta = {'tx_time': self._calculate_next_tx_time()}
        pdu = pmt.cons(meta, data)
        self.message_port_pub('data', pdu)
```

That's all, folks!

- Check us out on `www.gnuradio.org`!