Dig Out Your Brick Phone!
Bringing AMPS Back with GNU Radio

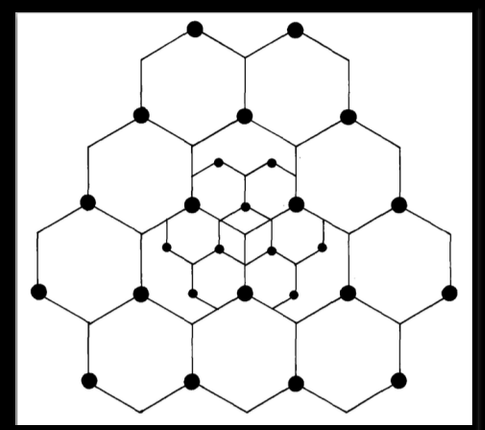
cstone a/k/a Brandon Creighton
Veracode / Ninja Networks
bjc@pobox.com

ShmooCon 2017

thx shmoocon

Goals

- Provide a quick tour of AMPS (**A**dvanced **M**obile **P**hone **S**ystem)
- Show the steps in creating GNU Radio blocks to prototype parts of a base station (gr-amps)
- Have fun!
- Demonstrate that TX > RX



- here to talk about some fun i've had with software-defined radio, building an implementation of the AMPS mobile network base station
- AMPS now off the air, but phones easy to get. i thought it'd be fun to revive them
- picked up a tiny bit of skills in implementing old pager protocols; learned even more here (made some dumb mistakes)
- most of the interesting things that I'd like to do with SDR involve TXing; when I went looking for previous examples, I found relatively few (esp. with GR)

Why AMPS?

- Equipment easily accessible (ebay)
- Simple enough to be a good target
  - Final BS-MS standard (TIA/EIA-553, 1999): 136 pages
- Historical significance
- Anything still out there?

some of you might ask: why are you building this? if you want a cell, use umts/lte/openbts! (i've done that, wanted to build it myself)
made a great target; one person can tackle in a manageable amt of time
history: first widely deployed; first heavily hacked (sheepish: phrack reference); phone calls weren't free

About AMPS

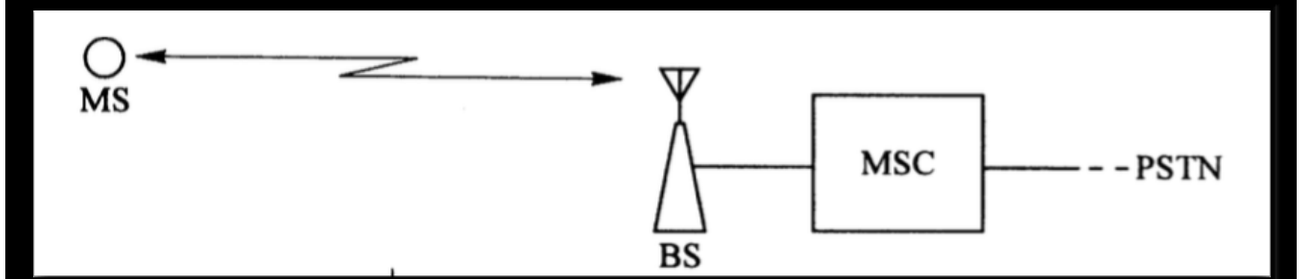
- First cellular network in the US
  - Designed in the 70s; deployed in 1983
- Extremely dumb phones, smarter base stations
- 824-849 MHz (MS->BS); 869->894
- Tradeoffs: power, spectrum use, cost, security
- Looks like a trunking system
- Often nicknamed “analog” since voice is FM

let’s go into the details a bit.
All mobile architectures trade off power efficiency, spectral efficiency, cost, and security; as it was, the initial phones cost thousands
So they left out a lot; the phones are pretty dumb. At the protocol level, AMPS resembles a trunking network more than a modern or even 2G system

A bit about GNU Radio

- GR is a library of DSP/RF components (“***blocks***”) mapped to Python interfaces
- Blocks can be native code (C++ API) or Python
- GUI tool (gnuradio-companion, aka GRC) to let you connect them together in ***flowgraphs***, change variables, and generate Python code to run them
- Watch Mike Ossmann’s video series to learn more!

(ask for show of hands: who’s used this)
I used GR/GRC because I didn’t have a strong background in DSP or RF. Lots of primitives built in already; it’s worth the learning curve and the clunky UI



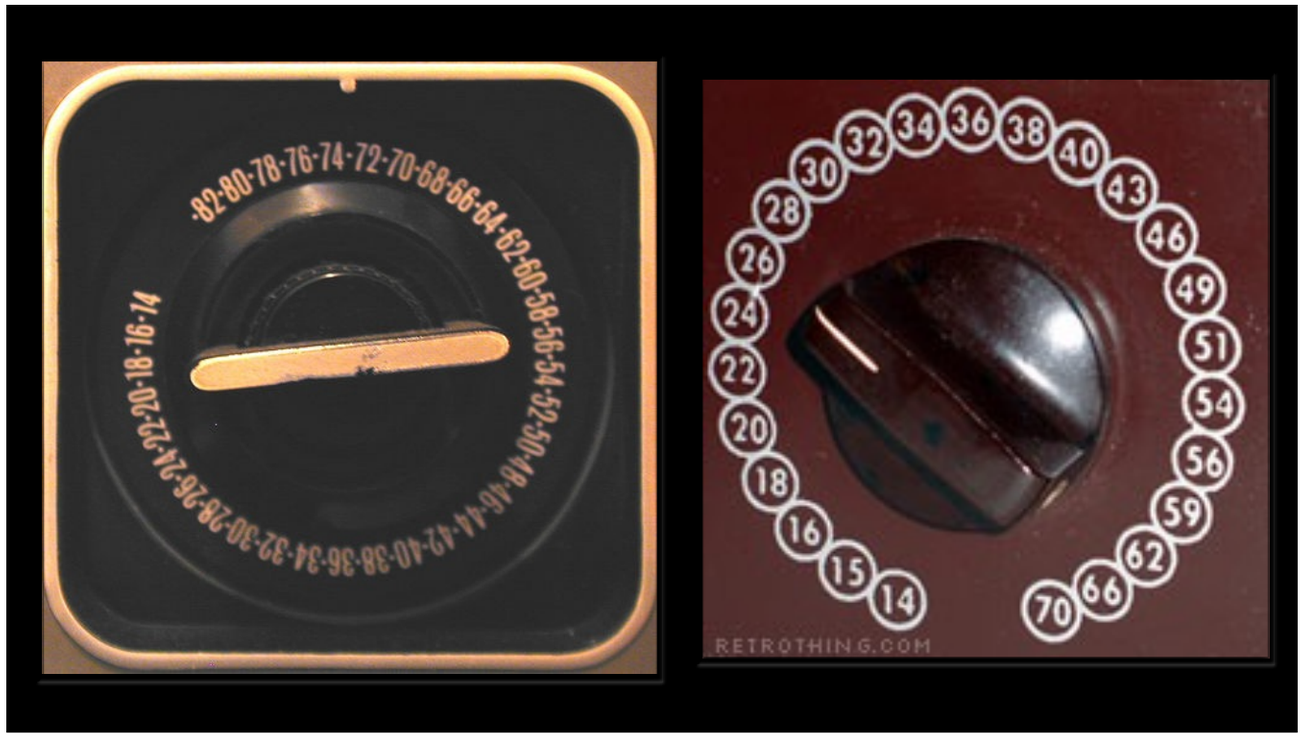
First Steps..

Research!

When I first started, I knew about that much. And that's it.
So I dug into the standards for exactly what I needed to do next.
Some other docs: Phrack, old expensive (now cheap) books

System	Channel Numbers	Forward Center Freq (BS: Base Station)	Reverse Center Freq (MS: Mobile Station)
A''	991–1023 (33 channels)	824.040 – 825.000	869.040 – 870.000
A	1–333 (333 channels)	825.030 – 834.990	870.030 – 879.990
B	334–666 (333 channels)	835.020 – 844.980	880.020 – 889.980
A'	667–716 (50 channels)	845.010 – 846.480	890.010 – 891.480
B'	717–799 (83 channels)	846.510 – 848.970	891.510 – 893.970

The AMPS frequency band is divided into 832 fixed, non-SS usable channels. Each channel number corresponds to a 30kHz range
45 MHz separation between the two (cost, interference)
Those channels were divided equally between the A side and the B side.
Originally, there were only 666, which leads me to a brief interlude



- When North American terrestrial TV was introduced, the UHF band spanned channels 14-83
- In 1983, channels 70-83 were taken away; some of these became AMPS
- These are discrete dials but..
- This is still happening! 2009 (50); Auctions 1001/1002

AMPS Channel Types

Type	Forward (BS -> MS)	Reverse (MS -> BS)
Control (Data only) *Channels 313-333 (A), 334-354 (B)*	**FOCC** (Forward Control Channel)	**RECC** (Reverse Control Channel)
Voice (Data/audio mixed)	**FVC** (Forward Voice Channel)	**RVC** (Reverse Voice Channel)

Anyway, back to AMPS. There are 832 channels

- AMPS has 4 - only 4 - types of channels. Each 30kHz channel is only one of these at any given time.
- There are only 42 channels in the entire system; 21 for A, 21 for B (!)

- 1 channel number identifies a forward/reverse pair

Critical Parameters

Base Station

- **SID:** The 15-bit network ID that the base station belongs to
  - Assigned to telcos; manually programmed in to the device
    - If it doesn't match, the phone is roaming.

Mobile Station

- **ESN:** 32-bit (!) serial number, globally unique to each device
- **MIN**: Phone number of the device
- This pair of IDs uniquely IDs an AMPS phone!

These are the main parameters you need to know.
There are a couple dozen or so different parameters that could be set that affect the behavior of everything to how often to register and how to behave when roaming; most of them are not worth going into detail here.

Control Channels

What does “connected” mean?

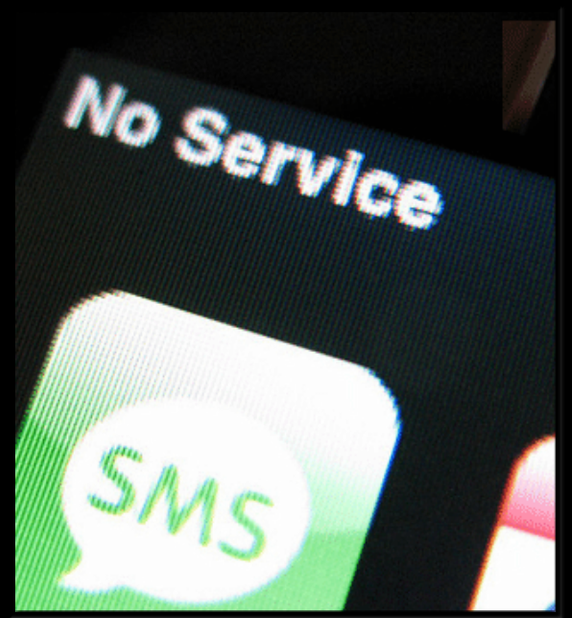
GSM/CDMA/LTE: Registered; acknowledged; authenticated

(exception: emergency calls)

AMPS: Valid control channel received recently, not explicitly disallowing the MS

Registration is unidirectional; acknowledgement not required

(ref: Section 2.6.3.3: “Response to mobile station messages”)



first type: control channels
not to get too philosophical: what does it mean for a phone to be connected?
Turns out: all a phone needs is to **hear** a valid set of FOCC overhead messages that doesn’t explicitly block it
Goal 1: BUILD AN FOCC

Control Channels

- **Forward (FOCC) transmits constantly:**
  1. Broadcast info about the system (Overhead Message)
     - **“Yooo! You’re listening to SID 31337. Here are my settings”**
  2. Direct **all** MSes to do something (Overhead Message)
     - e.g. “**Hey, I’m busy, try only 3 times to get a hold of me before waiting”**
  3. Direct a **specific MS** to do something (Mobile Station Control Message)
     - e.g. “**MS 6668675309! Tune to FVC 42”**
  4. Filler words (Control-Filler Message) — for blank space

What do we need to send?

There are essentially three types of messages: overhead messages, mobile station control messages, and control-filler messages. A FOCC should be transmitting continuously (so MSes can lock on), so when it doesn’t have anything to say, it sends a filler message

Building an FOCC Stream

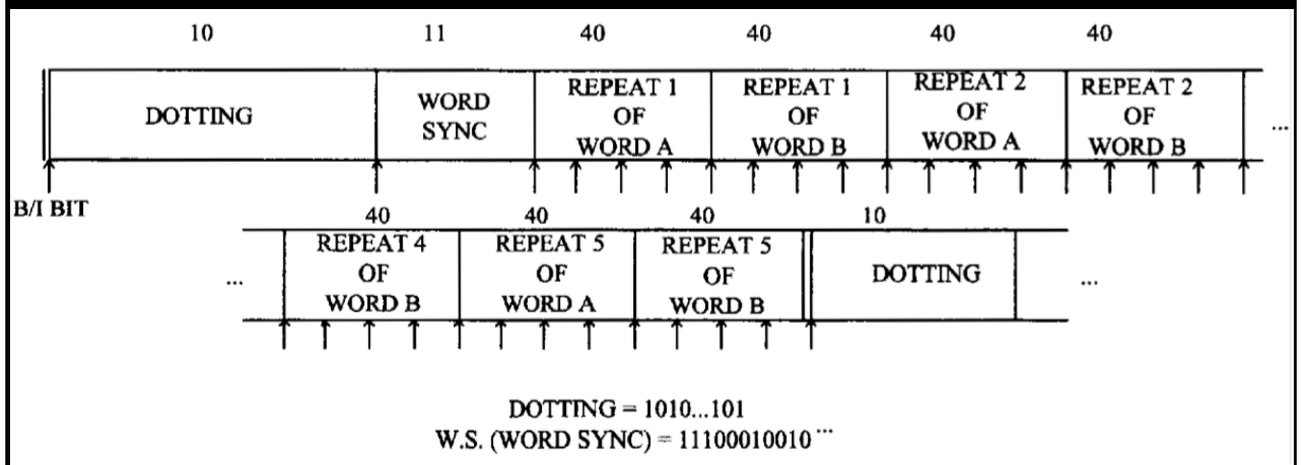
All messages are broken up into 28-bit **words**

Every word includes an additional 12 bits of error correction

Truncated BCH(63, 51); protects against 2 bit flips

Each message contains 28 bits of data plus 12 bits of parity. I used libitpp to give me BCH

FOCC Stream Format (Frame)



Phones where LSB(MIN) = 0 listen to word A

Busy/Idle bit sent before dotting/sync, after sync, and every 10 bits after

Here’s what a single frame (my terminology, not theirs) looks like

Easy-to-lock-on-to 10101010 pattern, followed by a magic sync word sequence (make a note of that one!), followed by data.

Word A / Word B. Why’d they do that? Power!

Building an FOCC Stream

Word 1

T1T2 = 11	DCC	SID1	EP	AUTH	PCI	NAWC	OHD = 110	P
2	2	14	1	1	1	4	3	12

Word 2

T1T2=11	DCC	S	E	REGH	REGR	DTX	
2	2	1	1	1	1	2	

	N-1	RCF	CPA	CMAX-1	END	OHD=111	P
	5	1	1	7	1	3	12

Here’s the diagram for the System Parameter Overhead Message. This is what we need.
Two 28-bit words (The P bits at the end are the BCH-encoded parity bits)
Most of these aren’t important enough to detail; the important ones are SID1 (SID), CPA (combined paging and access; tells MS to listen for paging); and CMAX. Also REGH/REGR tell phones to register

Building an FOCC Stream

- First step: We want to build a static FOCC saying that there's a network out there; everything else can be filler
  - The System Parameter Overhead Message (two words) does this
- Ordering them
  - TIA/EIA-553 3.7.1.2 says that this message must be sent every 0.8 +/- 0.3s
    - Data is sent at 10 kbit/s, which means if we repeat every 18 or 19 words, we're OK:
      - 18 x 463 bits = 8334 bits (0.8334s)
      - 19 x 463 bits = 8797 bits (0.8797s)

We know the message we want to send. We know that we send messages in frames that amount to 463 bits (repeats + BI bits). What exactly does that full stream look like?
We know we need to send those two words every 0.8s; we know a frame lasts 463 bits. So we send two bits plus 16 or 17 filler messages. Then repeat.

0	SYSTEM PARAMETER OVERHEAD MSG, WORD 1
1	SYSTEM PARAMETER OVERHEAD MSG, WORD 2
2	filler
3	filler
4	filler
5	filler
6	filler
7	filler
8	filler
9	filler
10	filler
11	filler
12	filler
13	filler
14	filler
15	filler
16	filler
17	filler

It looks something like this. Over and over.

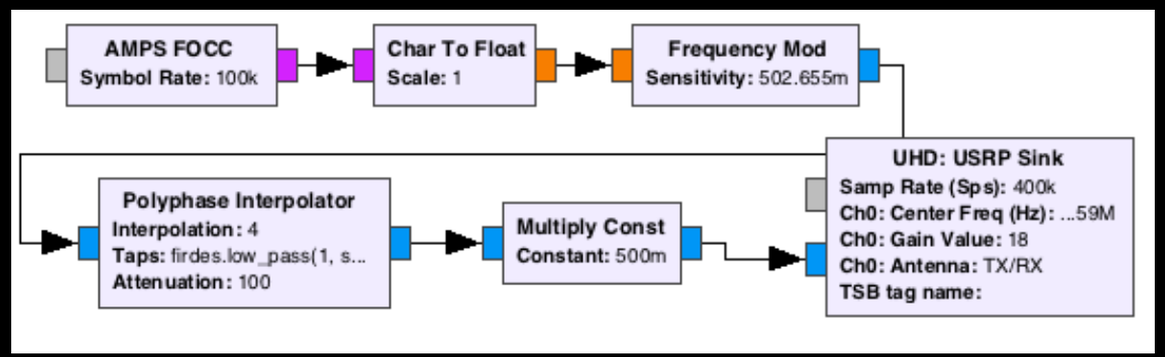
FOCC - Modulation

- **Forward (FOCC):**
  - **Repeated broadcast of BFSK-modulated data**
    - **8kHz deviation**
    - **Phase doesn’t matter (not CPFSK)**
    - **10kbit/sec, but Manchester encoded(!), so 20k symbols/sec**
      - **To send a 0: send 1, then 0**
      - **To send a 1: send 0, then 1**

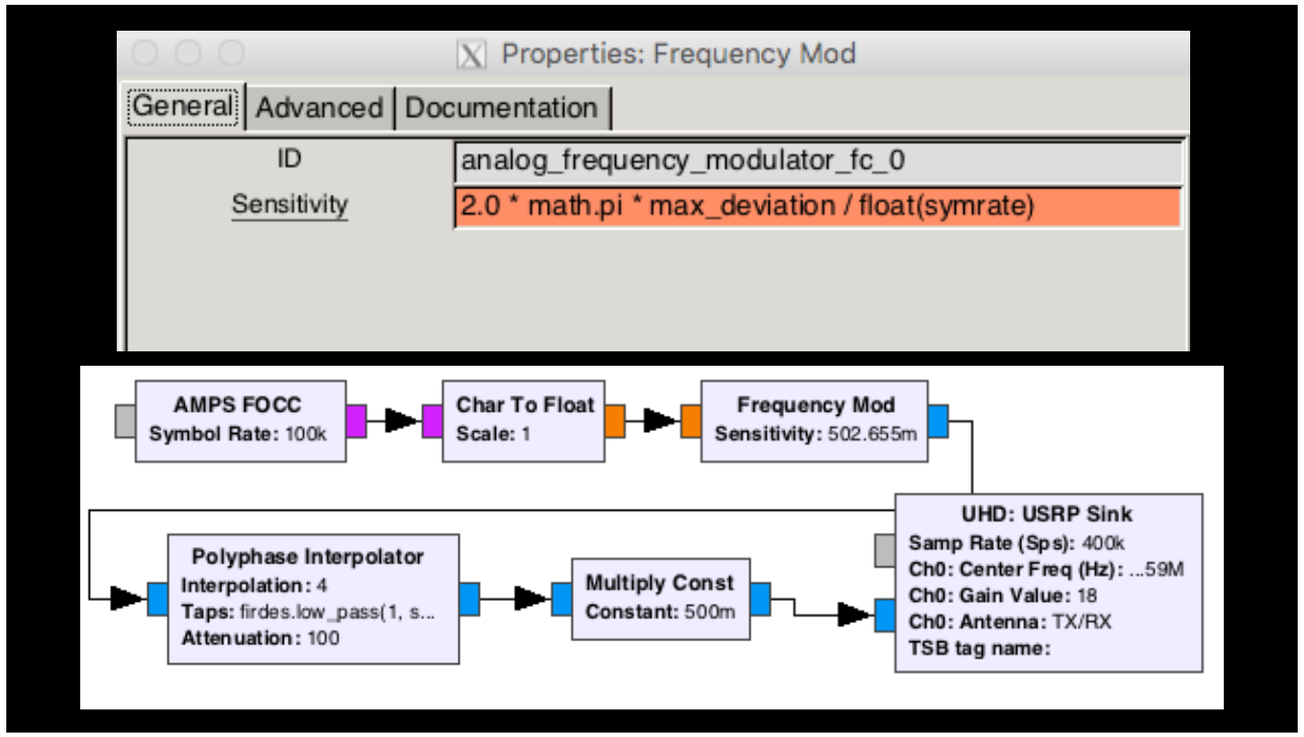
So we know the format of the digital data we want to send. What about the radio part?
FSK: Frequency-shift keying. We’re sending 10k bits a second, but that really means 20k symbols.
(describe ask a bit)

Modulating BFSK (aka 2FSK) in GNU Radio

- **Create a source block that emits values of -1 or 1, each representing a 0 or 1 symbol**
- **Multiply those values by the deviation**
- **Either emit (`sample_rate / 10000`) symbols from the block, or interpolate**



Now that we can see what 2FSK is on a waterfall graph, it's pretty easy to generate. We can use existing GNU Radio blocks to do most of the actual work here.
Start from the sample rate, and figure out how many samples we need to emit per symbol. Or do less, and interpolate later
Is this the right way? I don't know, but it works. DON'T FORGET TO FILTER!



Here you can see the properties for the FM block. In our case, max_deviation is a variable we've set to 8000Hz

Demo - FOCC only

RECC - Reverse Control Channel

Shared channel - phones send bursts of data when they need to

1. Send Acknowledgements (Order Confirmation / Page Response messages)
2. Request something from the BS (Order message)
   - Primarily: registration/authentication
3. Placing a call (Origination messages)

So now we have a base station, at least nominally. Now, let’s create a reverse channel, so we can hear what the phone’s sending to us.
Unlike the forward channel, there isn’t a continuous transmission.

RECC Burst Format

DOTTING	WORD SYNC	CODED DCC*	FIRST WORD REPEATED 5 TIMES	SECOND WORD REPEATED 5 TIMES	THIRD WORD REPEATED 5 TIMES	...
30	11	7	240	240	240	

Seizure Precursor

DOTTING = 1010...010

WORD SYNC = 11100010010

* DIGITAL COLOR CODE - Coded per Table 2.7.1-1.

Same basic idea as the FOCC — repetition

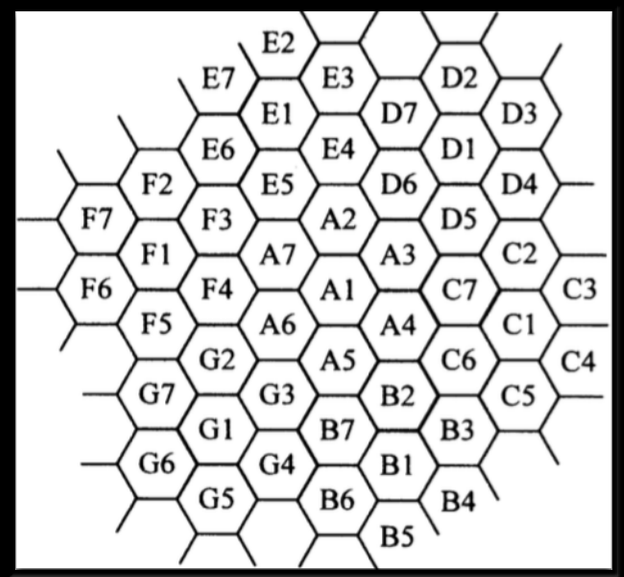
So now we have a base station, at least nominally. Now, let’s create a reverse channel, so we can hear what the phone’s sending to us.
Unlike the forward channel, there isn’t a continuous transmission.
See that DCC value there..

Color Codes?

AMPS is a *cellular* network; there are meant to be other BSes around (on the same channel)!

Each BS sends its DCC in the System Parameter Overhead Message; MSes use that to talk to them

Neighboring BSes have *different* DCC values, and ignore



Looking For a RECC Burst with GR

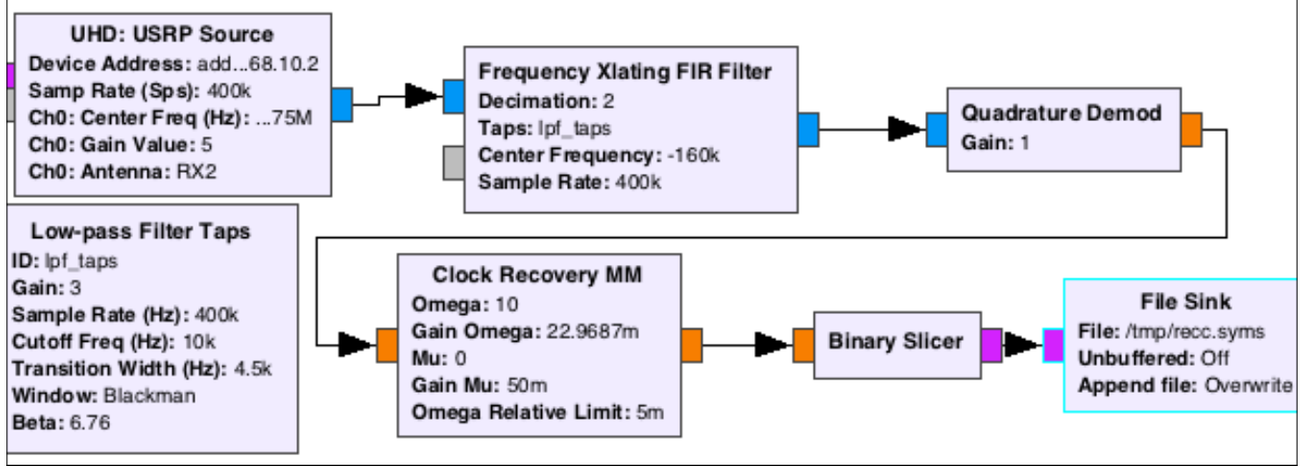
1. Start with an RF source (USRP Source, Osmocon Source, etc).
   - (To avoid hassles, use a frequency offset to avoid the DC spike)
2. Isolate the signal with a filter / demodulate as necessary
3. Demodulate, do clock recovery to align samples with the bitstream
   - Fairly easy if you have a good idea of the symbol rate!
4. Extract bits, collect bits in a custom block

One great part of using GR is that doing something like receiving bursty FSK is pretty well-documented. Numerous examples out there on the internet and in previous talks (NCC's is quite thorough)

Clock Recovery Visualization

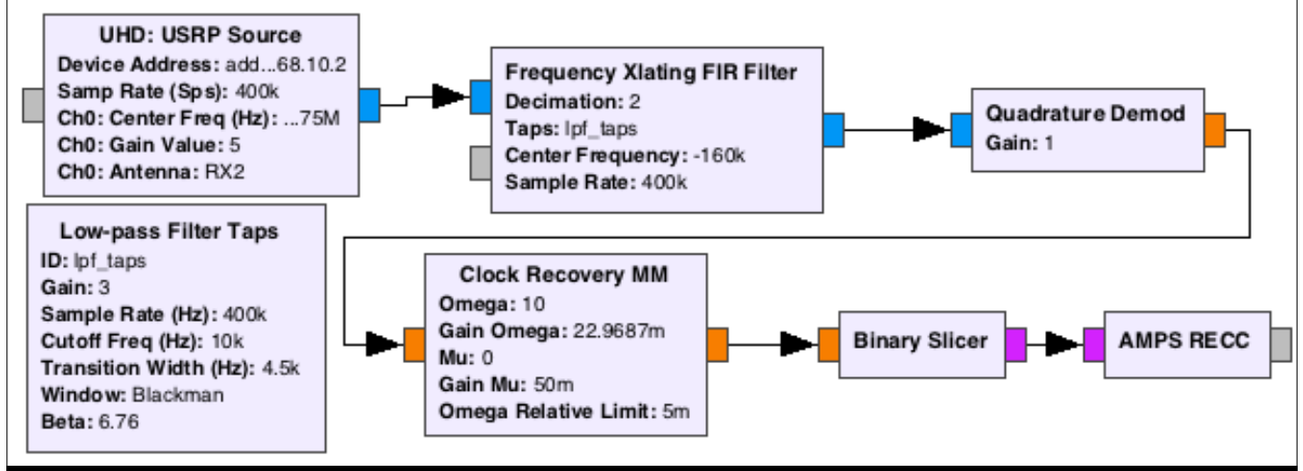
To illustrate what we want out of clock recovery, we’re trying to isolate this much
In a perfect world, we could just listen to X samples and compare values, looking for a pattern - but clocks drift on both sides! A smarter algorithm is needed.
Fortunately GR has the MM block

RECC Demod Flowgraph



GNURadio is great for prototyping this kind of workflow, because you can do all of this with off-the-shelf blocks; here I'm just dumping this data to a file that I can analyze with whatever. It's just a raw file of 0 and 1 bytes.

RECC Demod Flowgraph



Here’s my custom block at the end, receiving a stream of 0s and 1s.

All we have to do is start looking for that preamble

RECC Burst Format

DOTTING	WORD SYNC	CODED DCC*	FIRST WORD REPEATED 5 TIMES	SECOND WORD REPEATED 5 TIMES	THIRD WORD REPEATED 5 TIMES	...
30	11	7	240	240	240	
Seizure Precursor						

DOTTING = 1010...010

WORD SYNC = 11100010010

* DIGITAL COLOR CODE - Coded per Table 2.7.1-1.

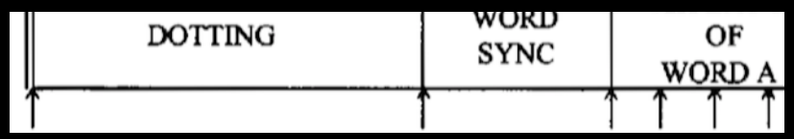
Same basic idea as the FOCC — repetition

(repeated for illustration)

.. and we're good to go. Or so I thought. I was seeing the premable, but not after that. So I decided to actually read more of the standard.

Seizing the RECC

To avoid collisions, MSes monitor the busy-idle bits sent every ~10 bits in the FOCC



MSes transmit when they see idle bits, but continue listening while TXing

- If MS sees that the channel is busy before the first 56 bits are sent, stop
- If MS *fails to see* the channel go busy before the first 104 bits, stop

(Full procedure described in section 2.6.3.5)

Obviously, since it's a shared channel, there's the potential for conflict. Phones can't necessarily hear each other (BS has more/better antennas), so they have to monitor the busy/idle bits being sent on the FOCC
This means that the FOCC has to constantly monitor/change BI bits, with fairly tight tolerance

Building a Better FOCC Block

Original FOCC block's `work()` function just emitted all the bits to a superframe every time it was called with all B/I bits set to idle)

That's not good enough!

New design stores the data as a series of segments (either messages or B/I bits), and `work()` only sends one segment per call (side effect: increased CPU)

Hypothesis: the RECC block could set a global `volatile bool` (yes, this works in GR) when the preamble is detected, and that would be close enough to meet the timing requirements

Results: NOPE

Also tried: Thresholds / AM Demod for carrier detection (but timing is tricky)

This threw a wrench into my plans. I missed this part, and assumed that I could just say the FOCC was idle constantly.
So I had to rewrite most of the FOCC block so that I was only sending a bit at a time. So I had to do some work to make that smarter.

Giving Up On Doing It Right

Access Type Parameters Global Action Message

T1T2 =11	DCC	ACT= 1001	BIS	PCI HOME	PCI ROAM	BSPC	BSCA P	RSVD	END	OHD =100	P
2	2	4	1	1	1	4	3	6	1	3	12

- Turns out, you can just tell the phones to ignore the B/I bits.
- Set BIS = 0 and they won't care.

Eventually, I decided against trying to accurately replicate that behavior for now.

0	SYSTEM PARAMETER OVERHEAD MSG, WORD 1
1	SYSTEM PARAMETER OVERHEAD MSG, WORD 2
2	ACCESS TYPE PARAMETERS GLOBAL MESSAGE
3	filler
4	filler
5	filler
6	filler
7	filler
8	filler
9	filler
10	filler
11	filler
12	filler
13	filler
14	filler
15	filler
16	filler
17	filler

So now the superframe looks like this.

Lazy Preamble Detector

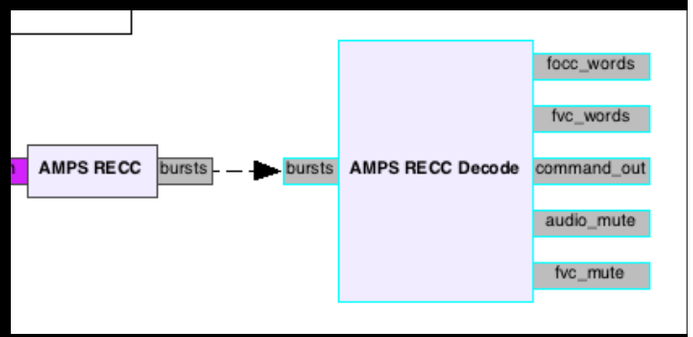
```
unsigned char *starting_ptr;

int recc_impl::work(...) {
    memcpy(&symbol_buf[curidx], input_buf, input_bytes);
    if(starting_ptr == NULL) {
        starting_ptr = memmem(symbol_buf[curidx - search_sz],
                search_sz, preamble, preamble_len);
    }
    if(starting_ptr != NULL
        && (symbol_buf_end - starting_ptr) >= BURST_SIZE) {
        // send the data off
    }
    // Rotate buffers as needed / reset starting_ptr to NULL
}
```

That problem solved, it was time to start looking for preambles in the RECC sink block. Essentially there's a large (64k) buffer big enough to store whatever we might want. When data comes in, we copy it there, then look for the preamble in that new data (starting from a few samples back). If we see it, and we've gathered enough data after that to have a full frame, we send it off. Where do we send it?

GNU Radio Messages

- GNU Radio allows blocks to define message ports, to send bits of data between blocks in an asynchronous way
  - Don't tie up `work()` methods; send messages to a new block instead
  - Used for RECC bursts, outgoing data messages, etc.



Here I've created a new block, AMPS RECC Decode, which decodes the potentially-identified blocks and takes action.
As of today, most of the logic happens here.

Answering Calls

When a call is placed from an MS, it sends an Origination RECC message.

Goal: Answer it, redirect it to a voice channel playing a static looping audio file.

We’ve got a base station and we can receive messages from phones. Let’s answer a call.

Mobile-Originated Call Setup Process

1. **MS sends Origination message over the RECC, including ESN/MIN and dialed number**
2. BS sends a Mobile Station Control Message with SCC != 11 and a voice channel number/RF power level
3. MS tunes to that FVC for incoming audio/messages
4. MS starts broadcasting audio on the corresponding RVC

This is what happens when everything goes right (ignoring error conditions)

Origination Message

All RECC messages start with Word A:

F = 1	NAWC	T	S	E	RSVD = 0	SCM	$MIN1_{23-0}$	P
1	3	1	1	1	1	4	24	12

If T=1, the MS is asking the BS for something; if T=0, it is acknowledging.

E=1 indicates that the next word is Word B

S=1 indicates that Word C is sent

NAWC: Number of Additional Words Coming

RECC messages follow a defined pattern, with words labeled in alphabetical order. Word A is always sent, and flags here indicate that Words B and/or C are sent.

Origination Message

Word B tells us what type of order we're talking about, so it's usually there:

F = 0	NAWC	LOCAL	ORDQ	ORDER	LT	RSVD = 000...0	$MIN2_{33-24}$	P
1	3	5	3	5	1	8	10	12

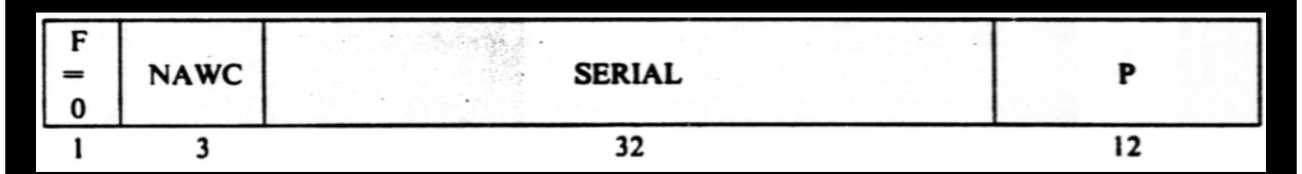
The ORDQ/ORDER/LOCAL fields indicate order type

In the case of an Origination: all three will be 0

This tells us what the MS is asking for or acknowledging.

Origination Message

Word C (sent unless the BS says not to in the parameters) gives the ESN:




```
                unsigned char nawc = worda.NAWC-2;
                if(wordc.NAWC != nawc) {
                    LOG_WARNING("protocol violation!  Word C NAWC does not agree with Word
A's -- continuing anyway");
                }
```

Registration messages send this too. But remember that the BS is not required to acknowledge them, so the phone can’t assume it has seen it. Therefore, you’ll see this sent for almost everything except acknowledgements.

Fun fact: NAWC sometimes lines up (on some DPC550 firmware!) Had to special-case it

Origination Message

For an origination, words D and beyond are just the dialed phone number:

F = 0	NAWC	1st DIGIT	2nd DIGIT	...	...	...	...	7th DIGIT	8th DIGIT	P
1	3	4	4	4	4	4	4	4	4	12

```
Word A (T=1, S=1, E=1)
Word B (ORDER=ORDQ=0)
Word C (ESN)
Word D = 31337
```

This is how we know what we’re calling. And that’s it.

So if we dial 31337, we should expect the burst at the bottom.

Mobile-Originated Call Setup Process

1. MS sends Origination message over the RECC, including ESN/MIN and dialed number
2. **BS sends a Mobile Station Control Message with SCC != 11 and a voice channel number/RF power level**
3. MS tunes to that FVC for incoming audio/messages
4. MS starts broadcasting audio on the corresponding RVC

This is what happens when everything goes right (ignoring error conditions)

Mobile Station Control Message (FOCC)

BS replies to the MS's Origination request with a two-word channel assignment message: (VMAC = power level; CHAN = channel; SCC=color)

T1T2	DCC	$MIN1_{23-0}$	P
2	2	24	12

2	2	10	1	5	3	5	12
T1T2 = 10	SCC = 11	$MIN2_{33-24}$	EF = 0	LOCAL/ MSG_TYPE	ORDQ	ORDER	P
	SCC≠11		VMAC		CHAN		
2	2	10	3		11		12

Here, VMAC is the power level; CHAN is the channel number
SCC is another color code

Mobile-Originated Call Setup Process

1. MS sends Origination message over the RECC, including ESN/MIN and dialed number
2. BS sends a Mobile Station Control Message with SCC != 11 and a voice channel number/RF power level
3. **MS tunes to that FVC for incoming audio/messages**
4. **MS starts broadcasting audio on the corresponding RVC**

This is what happens when everything goes right (ignoring error conditions)
BS FOCC messages are sent using GNU Radio messages

AMPS Voice Channels

Both FVC and RVC can transmit audio or data (also 10kbit 2FSK)

They use ***blank-and-burst*** in-band signaling

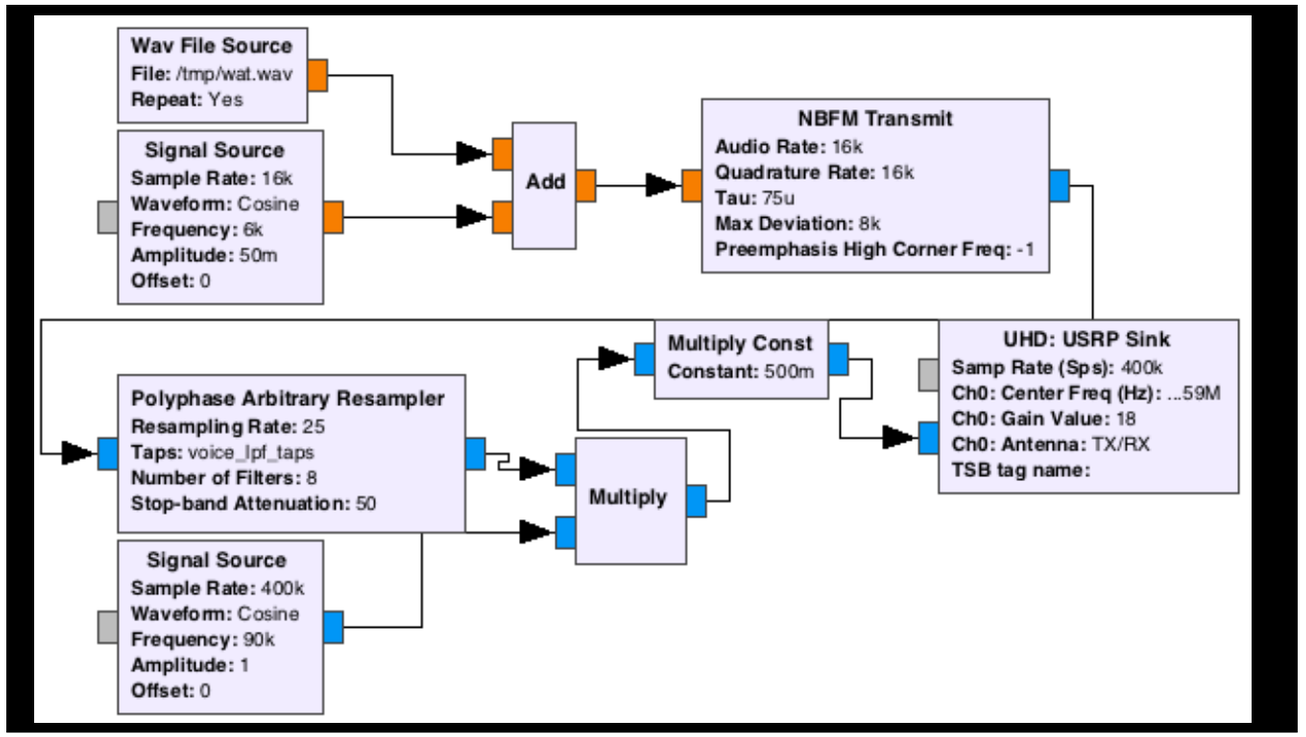
When a ~6kHz audio tone (SAT Tone) is present, then audio is being sent; when it's not, data bursts are being sent

SCC value sent from BS determines the frequency to be used

Also important for defending against audio from interfering cells!

SCC	Freq
0	5970
1	6000
2	6030

SCC serves a dual purpose: signaling and color coding



Now, the RVC (audio from the phone) we don’t really care about listening to at the moment, since we’re just trying to make a call and hear the audio. So we “just” need to get an audio file, modulate in the 6kHz SAT tone, shift the frequency so that we’re not transmitting over the FOCC, and we’re good… uhoh, getting complex

Demo - FVC audio only

Channel Layout

Channel	353	354	355	356
Freq (MHz)	880.59 835.59	880.62 835.62	880.65 835.65	880.68 835.68
Function	FOCC RECC	unused	unused	FVC RVC

Let me illustrate the current channel layout. In the previous slide, we shifted the FVC audio by 90 kHz (3 channels)

Cloning

F = 0	NAWC	SERIAL	P
1	3	32	12

Initially, the RECC had no protection against eavesdropping, making obtaining ESN/MIN pairs easy

Cloning a phone was as simple as changing a phone's ESN

That's all I've got to show, so let's back up a bit.

Cloning Countermeasures

Countermeasure 1: Forbid handset manufacturers from allowing the ESN to be programmable

Some were better at it than others.

Obviously, the carriers didn’t like this.

We all know how this turns out; hackers are gonna hack

Cloning Countermeasures

Countermeasure 2: Bolt-on encryption after the fact

TDMA (IS-54 aka ANSI/TIA/EIA-627), aka D-AMPS included the Cellular Message Encryption Algorithm suite (aka CAVE)

Published in the early 90s

Kept private; document describing the algorithm was leaked to Cryptome in 1997

First severe attacks published by David Wagner, Bruce Schneier, and John Kelsey at CRYPTO 97

Obviously, the carriers didn’t like this.

We all know how this turns out; hackers are gonna hack

Cloning Countermeasures

Countermeasure 3: Make it illegal to make or buy a scanner that covered AMPS ranges

This happened.

Telephone Disclosure and Dispute Resolution Act (1992)

Enforced by the FCC

Cloning Countermeasures

Countermeasure 4: Make users type in a PIN to make a call

Some carriers did this!

It raised the cost only slightly: you had to decode the RECC request, the FOCC response, and the audio on the FVC

BS-Originated Call (Page)

1. BS sends Page Message over FOCC (Mobile Station Control Message)
2. MS sends Page Response over RECC (order ack)
3. BS sends a Mobile Station Control Message with SCC != 11 and a voice channel number/RF power level (same message as before!)
4. MS tunes to designated FVC, awaits commands; starts broadcasting silence (no SAT) on RVC
5. BS sends Alert over FVC; MS rings the phone
6. User Answers
7. MS starts broadcasting SAT and audio

What about when the base station makes a phone ring? How does that work?
I’m not going to bore you with more packet diagrams. Most of these messages are the same as what we’ve covered already. Let’s talk instead about the FVC.

Channel Layout

Channel	353	354	355	356
Freq (MHz)	880.59 835.59	880.62 835.62	880.65 835.65	880.68 835.68
Function	FOCC RECC	unused	FVC RVC	FVC RVC

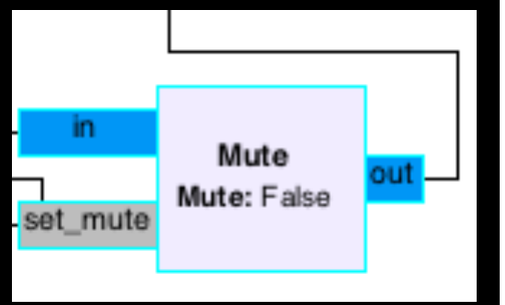
Let’s create a new FVC that can actually send both data and voice, and put it on channel 355.

A basic FVC in GNU Radio

Problem: Need to transmit both data and audio, and choose

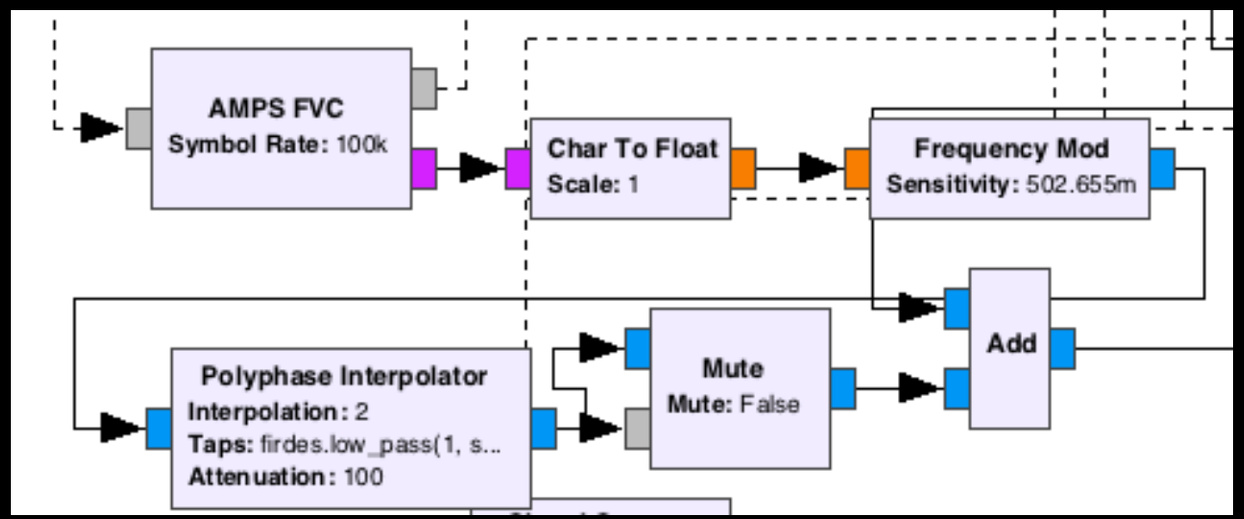
Solution: Two paths transmitting; each flows through a Mute block controlled by messages

(There are probably more elegant solutions.)



The easiest way to hack this up was to transmit audio, like we did before, plus create a new block for the data stream. I’m transmitting both of them

A basic FVC in GNU Radio

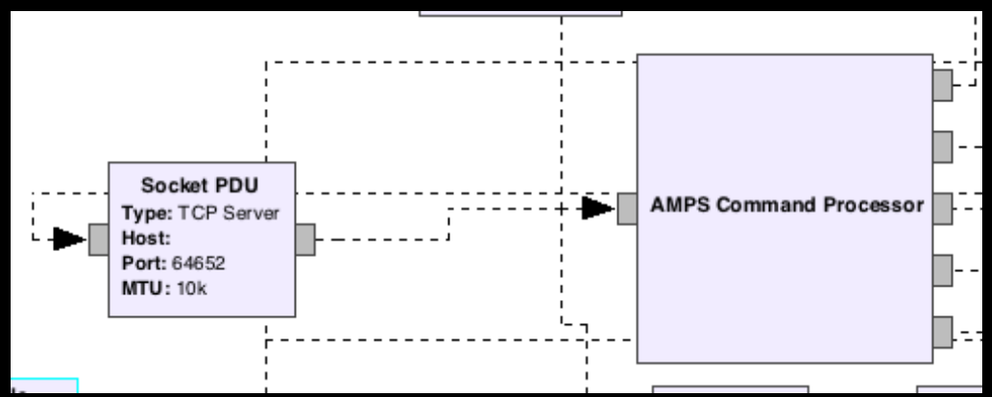


No longer possible to show graphs legibly (if it ever was).
FVC block works just like the FOCC block; modulate using the FM bock and a filter.
Contains an incoming message port for words to send

Build a CLI with Socket PDU

Want a quick and dirty telnet CLI?

Use Socket PDU and a lightweight parser block



I wanted to be able to debug this process more quickly, so I built a special block that does nothing but parse text commands from the Socket PDU GR block.
It can send messages to the FOCC/FVC blocks and toggle the mute blocks, so we can use the “page” command

Demo - FVC Page

Future work - RVC

- For mobile-mobile calls: straight sample copying doesn’t seem to work well
  - Caveat: haven’t tried everything yet
- Is it worth continuing to do this in GR, or is it time to move it out?

Packet-In-Packet Protection!

Orders to certain MINs could not be transmitted on AMPS.

Table 3.7.1-5 Troublesome central office codes

Bit pattern thousands					Central office	
T1T2	Digit DCC	NXX	X	XXX	Code	
00	ZZ	111110(0)0100	10YY	...	007	0,8,9
00	ZZ	111011(1)0001	0010	...	056	2
00	ZZ	111100(0)1001	0ZZZ	...	070	1-7
00	ZZ	000011(1)0001	0010	...	150	2
00	ZZ	000111(1)0001	0010	...	224	2
00	ZZ	000111(0)0010	010Z	...	225	4,5
00	ZZ	001011(1)0001	0010	...	288	2
00	ZZ	001110(0)0100	10YY	...	339	0,8,9
00	ZZ	001111(1)0001	0010	...	352	2
00	ZZ	001111(0)0010	010Z	...	353	4,5
00	ZZ	010011(1)0001	0010	...	416	2
00	ZZ	010111(1)0001	0010	...	470	2

BE CAREFUL

This is infrastructure - deprecated and disabled infrastructure, but nonetheless.

Don’t transmit carelessly.

Recap: Don’t Do These Things

- When RXing/decoding Manchester-encoded data, make sure you’re looking for symbols and not bits
- Read the specs carefully
- Be careful where you click in GRC. (Dragging a connection deletes it.)

Thanks!

https://github.com/unsynchronized/gr-amps

thanks shmoocon
thanks audience
thanks everyone who let me have phones
thanks gnu radio
thanks phrack

References

- The latest BS<->MS standard: TIA/EIA-553-A (1999)
  - Older version published as Canada's RSS 118 Annex A
- Avian's Blog, for practical use/caveats of the Clock Recovery MM block: https://www.tablix.org/~avian/blog/archives/2015/03/notes_on_m_m_clock_recovery/
- Keysight/HP: "Programming techniques for the HP 8920A": http://literature.cdn.keysight.com/litweb/pdf/5964-0159E.pdf

References

- Brian Oblivion: “Cellular Telephony”. Phrack 38:9
- Brian Oblivion: “Cellular Telephony II”, Phrack 40:6
- John G. van Bosse, Fabrizio U. Devetak, Signaling In Telecommunication Networks (2nd ed.)