

Inner Circle Newsletter November/December 2025



This computer that used to run the telescope at Bochum Observatory.



**OPEN
RESEARCH
INSTITUTE**

The Who What When Where Why

Open Research Institute is a non-profit dedicated to open source digital radio work on the amateur bands. We do both technical and regulatory work. Our designs are intended for both space and terrestrial deployment. We're all volunteer and we work to use and protect the amateur radio bands. You can get involved in our work by visiting <https://openresearch.institute/getting-started>

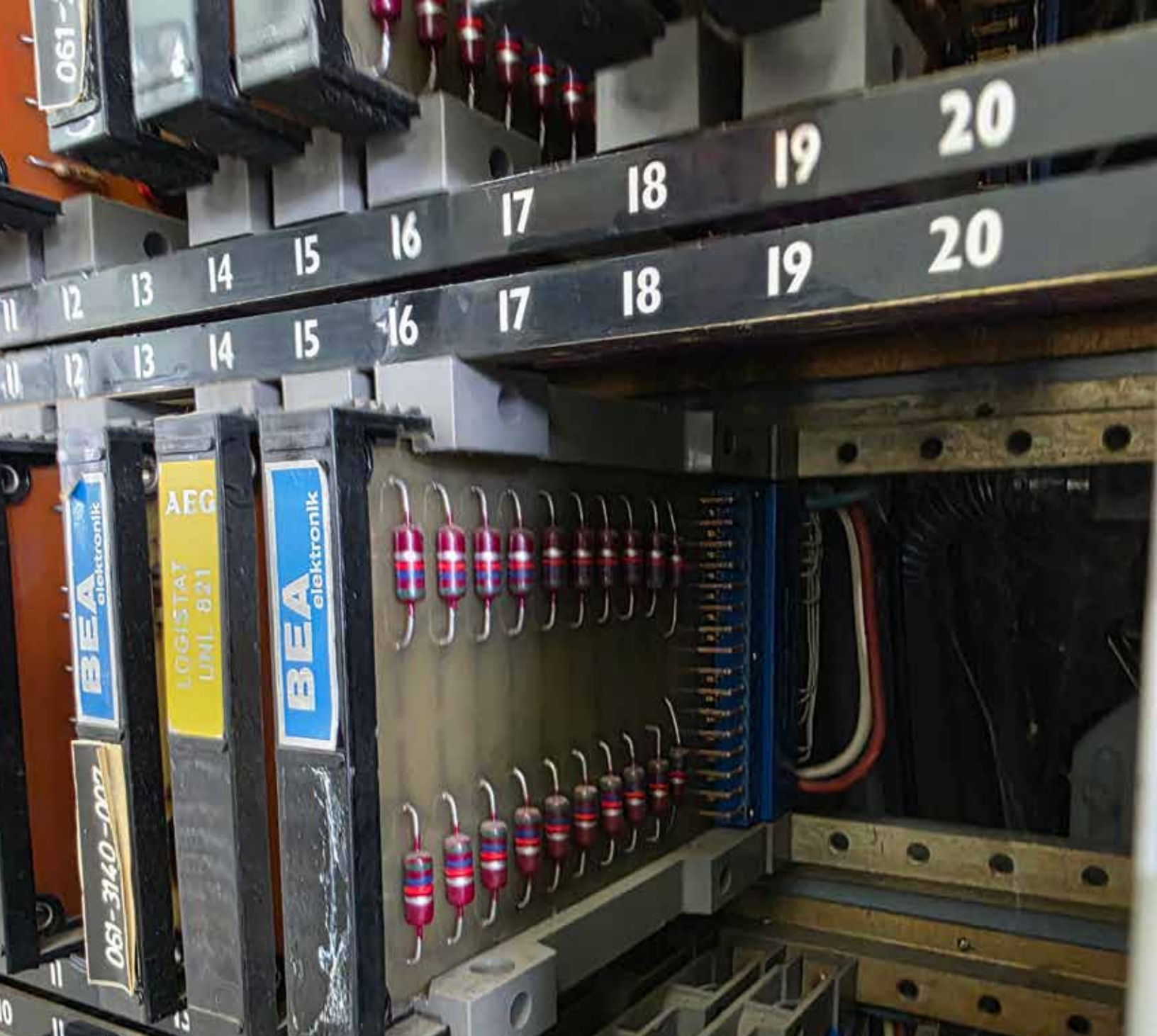
Membership is free. All work is published to the general public at no cost. Our work can be reviewed and designs downloaded at <https://github.com/OpenResearchInstitute>

We equally value ethical behavior and over-the-air demonstrations of innovative and relevant open source solutions. We offer remotely accessible lab benches for microwave band radio hardware and software development. We host meetups and events at least once a week. Members come from around the world.



OPEN SOURCE HARDWARE AND SOFTWARE NEWSLETTER GROUP
FREE AND 100% VOLUNTEER DRIVEN

Want more Inner Circle Newsletters? Use the QR code at left or go to http://eepurl.com/h_hYzL and sign up.



How to Train Your PLUTO

page 4

A Picture is Worth a Thousand Words

page 8

Opulent Voice Update: Abandon Ship!

page 9

Opulent Voice Update: From Pluto to Libre

page 14

Opulent Voice Update: Correlator Upgrade

page 20

November Puzzle Solution

page 22

ORI Regulatory Update: FCC Proposes Deleting BPL Rules

page 23

Inner Circle Sphere of Activity

back page

How to Train Your Pluto

by Paul Williamson KB5MU

Have you ever saved a `pluto.frm` firmware file and then lost track of which build it was? Or perhaps a whole directory full of different .frm files with cryptic names? This little shell script will analyze the .frm file(s) and give you a little report about each one.

```
% ./pluto-frm-info.sh *.frm
Version information for pluto-1e2d-main-syncdet.frm:
device-fw 1e2d
buildroot 2022.02.3-adi-5712-gf70f4a
linux v5.15-20952-ge14e351
u-boot-xlnx v0.20-PlutoSDR-25-g90401c

Version information for pluto-peakrdl-clean.frm:
device-fw abfa
buildroot 2022.02.3-adi-5712-gf70f4a
linux v5.15-20952-ge14e351
u-boot-xlnx v0.20-PlutoSDR-25-g90401c

Version information for pluto-peakrdl-debug.frm:
device-fw d124
buildroot 2022.02.3-adi-5712-gf70f4a
linux v5.15-20952-ge14e351
u-boot-xlnx v0.20-PlutoSDR-25-g90401c
```

The first line gives the `device-fw` *description*. This is created in the Makefile by `git describe --abbrev=4 --always --tags`, based on the git commit hash of the code the .frm file was built with. It may be just the first four characters of the hash, as shown here. It can also include tag information, if tags are in use in your repository, in which case it also inserts the number of commits between the latest tag and the current commit.

Note that if you commit, build a version, make some changes, and build another version without committing beforehand, you will end up with two builds that have the same description. There is no mechanism in the current build procedure to catch that situation, so there's no way to extract that information from the .frm file.

Theory of Operation

The .frm file for loading new firmware into an ADALM PLUTO, as built with the Analog Devices [pluto-sdr-fw](<https://github.com/analogdevicesinc/plutosdr-fw>) Makefile, is in the format of a device tree blob. That is, a compiled binary representation of a Linux device tree. Without specialized tools, this is an opaque binary file (a blob) that is difficult to understand.

So, the first step in understanding the file is to decompile it. Fortunately, the device tree compiler tool, `dtc`, is also able to decompile binary device trees into an ASCII source format.

The decompiled source is in a structured format, and also in a very consistent physical layout. It mostly consists of large image blocks, represented as long strings of two-digit hexadecimal byte values, enclosed in square brackets, each on one very long line of text. There is an image block of this type for each binary resource needed to program a Pluto device. Among other things, this includes an image of the Linux filesystem as it will appear in a RAM disk. The build procedure places certain information about other components of the build into defined locations within the filesystem, in order to make that information available to the Pluto user. We take advantage of the rigid format of the decompiled ASCII format to extract the hexadecimal byte values encoding the filesystem image. This is done in two steps. First, ``sed`` is used to find the header of this image object, identified by its name ``ramdisk@1``, and extract the second line after the line containing that string. Second, the hex data is found between the square brackets using ``awk``. This all depends on the exact format of the decompilation, as created by ``dtc`` in its decompilation mode.

We convert the hexadecimal values into binary using ``xxd``. This binary is gzip compressed, so we use ``gunzip`` to uncompress it. The result is a CPIO archive file containing the Linux filesystem. We use ``cpio`` to extract from the archive the contents of the file that would be ``/opt/VERSIONS`` when installed. This file contains the information we are after; we print it to stdout.

That whole procedure is wrapped in a little loop, so it runs once for each file named (or expanded from a wildcard pattern) on the script's command line.

Prerequisites

This is a bash script, which I've developed on macOS Sequoia and Raspberry Pi OS (Debian) bookworm. It ought to work on any Unix-like system.

You will probably need to install some or all of these utilities before the script will work. These should all be available in your favorite package manager, if not already installed on your system.

- `dtc`
- `sed`
- `awk`
- `xxd`
- `gunzip` (part of `gzip`)
- `cpio`

A special note about ``cpio`` for macOS users

The script uses the ``--to-stdout`` command line flag to avoid writing any files to your host's filesystem. The version of ``cpio`` that Apple ships with macOS does not support this feature, but the version available in [Homebrew](https://brew.sh) does. When you install ``cpio`` using brew, you're warned that brew's `cpio` does not override the Apple-provided `cpio` by default (it's installed "keg-only"). It provides a modification you can install in your shell's startup script to add brew's `cpio` to the path, so you always get brew's `cpio`. If you do that, you can use the standard version of this script, named ``pluto-frm-info.sh``. If you choose not to do that (which is what I recommend), and your brew installation is in ``/opt/homebrew`` as usual, you can use the modified version found in ``pluto-frm-info-macOS.sh``. If your brew installation is elsewhere, modify the path to `cpio` in the macOS version.

Speed

This script does a substantial amount of work on largish data objects, so it can be slow. It takes about 3 seconds per file to run on my 2024 M4 Pro Mac Mini, but it takes about 75 seconds per file on my Raspberry Pi 4B.

Credits

Thanks to all those unknown developers who shared their sed and awk scripts and tutorials on the web, where they could be scraped for LLM training. I don't know enough `sed` or `awk` to have written the respective command lines without a lot of research and struggle. I used suggestions provided by everybody's favorite search engine when I asked how to do those tasks, and in this case the LLM-generated answers were more immediately useful than the first few hits of presumably human-generated content.

Missing Features

- Output build time, building user, and building hostname. This information is in the .frm file somewhere, so this feature might be added with some additional research.
- Describe changes since the commit that appears in the build description. As noted above, I think this information is not contained in the .frm file, so it cannot be added.
- In ORI builds, an extract of the git log for the build is placed (by a command in the Makefile) in `root/fwhistory.txt`. This could be extracted the same way as `opt/VERSIONS`, by just changing the string in this script. However, this history information is better extracted from the repo online, starting from the commit revealed by device-fw line in VERSIONS.

Other Places to Find Information

If you have the Pluto up and running, it has a web server. If you look at [index.html](http://pluto.local/index.html) (on the Pluto) with a web browser (on a connected computer), you'll see all kinds of build information, much more than is returned by this script.

But if you look in the ramdisk image in the pluto.frm file, you'll find the HTML file in /www/index.html contains only placeholders for all of this information. It turns out that these placeholders are replaced with the actual information by a script that runs when the Pluto boots up. That script is located at /etc/init.d/S45msd (or in the Analog Devices repo at [buildroot/board/pluto/S45msd](https://github.com/analogdevicesinc/buildroot/blob/f70f4aff40bcc16e3d9a920984d034ad108f4993/board/pluto/S45msd)). This script has a function `patch_html_page()` that obtains most of its information from resources that are only available at runtime, such as `/sys`, or by running programs like `uname`. Some of the info comes from /opt/VERSIONS, which is where this script looks. Some of that runtime information actually comes out of the hardware, like the serial number, or non-volatile storage on the hardware, like the USB Ethernet mode. The rest, including more build information, must be in the pluto.frm file somewhere, possibly scattered around in multiple places.

The name of the computer the firmware was compiled and the username of the user who compiled it, and more, are buried in the kernel image, and are made accessible at runtime via

```
[pluto3:~]# cat /proc/version
Linux version 5.15.0-20952-ge14e351533f9 (abraxas3d@mymelody) (arm-none-
linux-gnueabihf-gcc (GNU Toolchain for the A-profile Architecture 10.3-
2021.07 (arm-10.29)) 10.3.1 20210621, GNU ld (GNU Toolchain for the
A-profile Architecture 10.3-2021.07 (arm-10.29)) 2.36.1.20210621) #2 SMP
PREEMPT Fri Nov 7 23:04:52 PST 2025
[pluto3:~]#
```

The first part of this output is stored in the kernel in a string named `linux_proc_banner` in `init/version.c`. If I knew how to decompress the Linux zImage, I could probably extract this information.

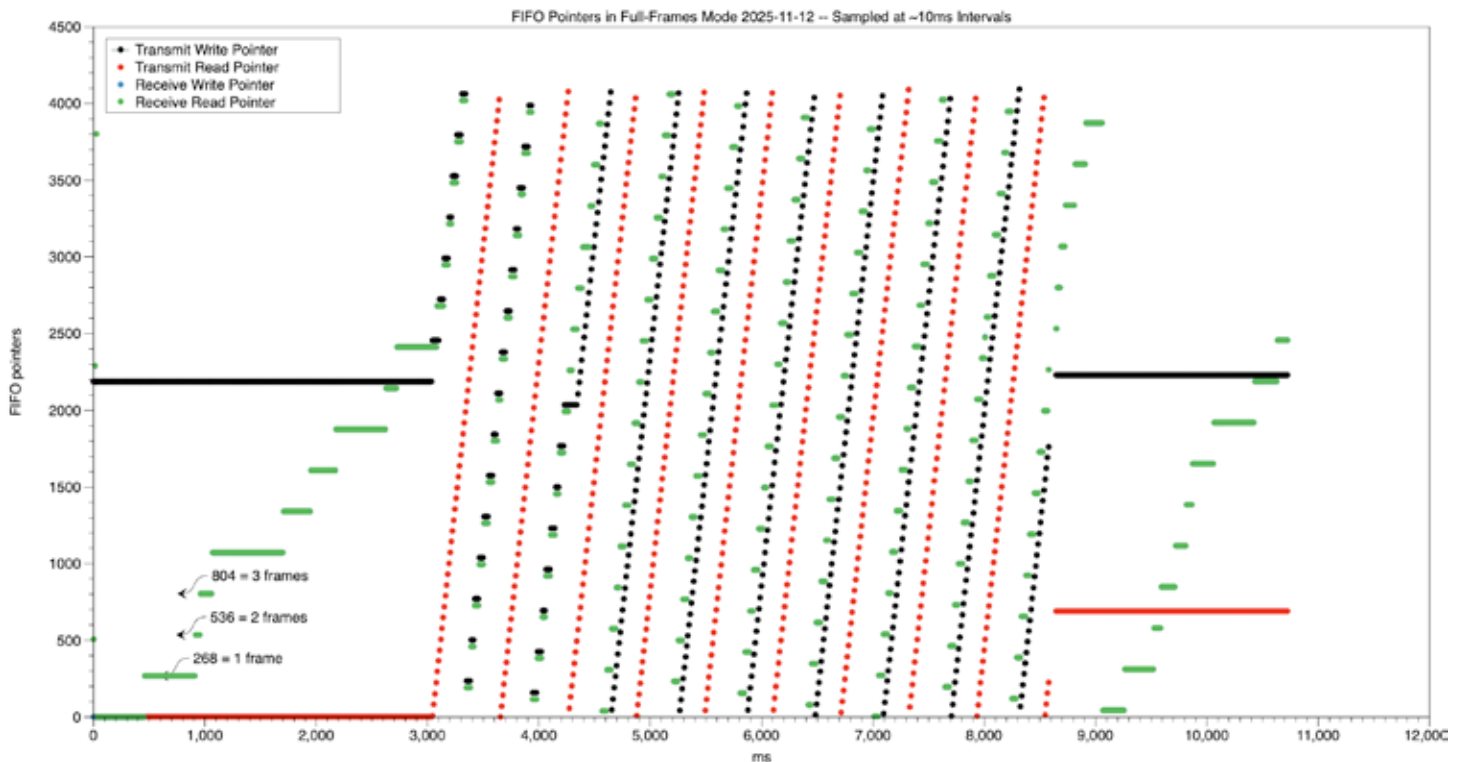
```
#!/bin/bash

# Output version information embedded in one or more
# Pluto firmware files in .frm format.
#
# Specify filename(s) on the command line. Wildcards ok.

for filename in "$@"; do
    echo "Version information for $filename:"
    cat $filename |
        dtc -I dtb -O dts -q |
        # decompile the device tree
        sed -n '/ramdisk@1/{n;n;p;q;}' |
        # find the ramdisk image entity
        awk -F'[][]' '{print $2}' |
        # extract hex data between [] brackets
        xxd -revert -plain |
        # convert the hex to binary
        gunzip |
        # decompress the gzip archive
        cpio --extract --to-stdout --quiet opt/VERSIONS
        # extract the VERSIONS file from /opt
    echo
    # blank line to separate multiple files
done
```

A Picture is Worth a Thousand Words

This image, made with the MacOS DataGraph program by Paul Williamson KB5MU, shows that the Opulent Voice transmit first in first out buffer (FIFO) wasn't properly reset at the start of the program. This means that there was data already in the transmit FIFO at startup. This is not desirable because when we want to transmit, we want to start with the data that we're sending. Whether this is voice, or keyboard, or a file transfer, we don't want extra data at the beginning of the transmission. For voice, this might just mean a very short burst of noise. But, for a text message or data file, it could cause corruption. Visualizations like this help make our designs better, because they reveal what's truly going on with the flow of data through the system.



If a picture is worth a thousand words, then a video is worth a thousand pictures. Visit our YouTube channel at the link below <https://www.youtube.com/@OpenResearchInstituteInc> to see all our presentations, lab demonstrations, and project meetups. Subscribe to get notifications of new content.

Celebrating 700+ Subscribers on YouTube

Subscribe Here

Thank you for the support!

Opulent Voice Report: Abandon Ship!

Opulent Voice, our digital communications protocol, is used as the uplink for our open source satellite program Haifuraiya. Opulent Voice is also perfectly suited for terrestrial communications links.

Development so far has targeted the PlutoSDR. This platform has served us extremely well. However we've driven it as far as it can go. This is the story of how we learned we'd hit the wall, and what we did about it.

The long-term goal for Opulent Voice is an open source ASIC and world-class radio hardware. On the PlutoSDR, Opulent Voice data payloads are delivered from an external source to the modem's network socket (USB). These data payloads have the Opulent Voice header, COBS header, UDP header, IP header, and if voice, RTP and OPUS headers. These data payloads arrive in the modem and are sent in to a transmit first in first out buffer (FIFO). The FIFO absorbs some of the network latency and uncertainties, so that we can support remote radio deployments as well as other challenging real-world timing situations.

The ARM Processor and FPGA in the PLUTO work together in order to send a preamble at the beginning of a transmission, randomize each data payload, apply forward error correction encoding to each data payload, interleave all the bits to take full advantage of the error correction, and then prepend a three-byte synchronization word to the beginning of each frame. The resulting 271 byte frame goes out over the air, modulated as a minimum shift keying signal.

Received signals are demodulated. Preambles help recover bit timing. The synchronization word is used to detect the start of the frame. The resulting payload is deinterleaved, the error correction is decoded, and then the resulting data is derandomized. We now have a data payload frame with Opulent Voice (and other) headers. This is delivered to the human-radio interface so that the data, voice, or text can be presented to the operator.

Up until the point where we fully integrated the forward error correction (FEC), the entire transceiver could fit into the Zynq 7010 in the PlutoSDR. This has 17,600 look-up tables (LUTs), a metric of what we call utilization on an FPGA. The number of LUTs available is similar to the number of shelves in a warehouse. If you fill up all the shelves, then there is no more room for inventory. However, that's not the entire story. Filling up the LUTs with our logic is one aspect of FPGA utilization. Another aspect is how well the different parts of the design are connected together. Data flows through the design, and there are FPGA resources that must be used to make these connections. Some of the connections are only a bit wide, and some are 32 bits wide. The connection resources are like the aisles between the shelving systems in a warehouse. If you can't reach a shelf, then it doesn't matter if you have the inventory or not. Unreachable inventory is not useful.

Below is a visualization of the FPGA utilization from Vivado. The cyan blocks are LUTs that are assigned. Blank spots in the upper 20% or so of the image are unassigned LUTs. Utilization immediately before complete FEC integration was approximately 60%. Why the difference between the visual 80% and the reported 60%? Because each cyan block in the image is not entirely full. What appears to be 80% utilization at this point in development was 60% by LUT count. This is what you want to see, with functions spread out over the resources and not densely packed in to smaller areas.

Don't Fear the ASIC

(Don't Fear the Reaper, by
Blue Oyster Cult)

All our code is open source
Hardware specs are clear
FPGAs got us started here
But ease-of-use draws near

Baby take my hand
Don't fear the ASIC
Baby I'm your friend
Don't fear the ASIC

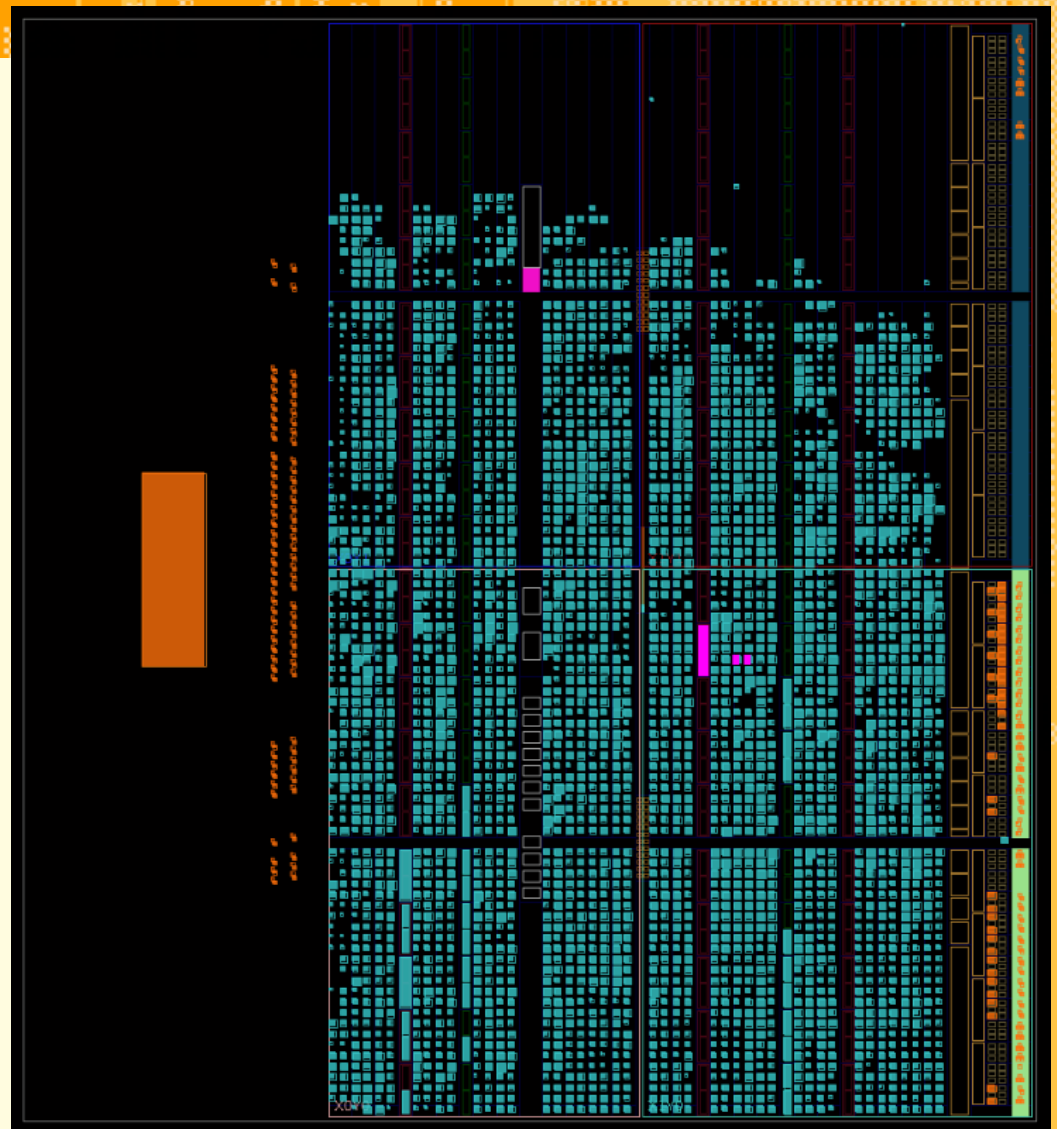
We'll make it simple
Don't fear the ASIC
Baby take my hand
Don't fear the ASIC

La la la la la
La la la la la

Raspberry Pi and PLUTO
board
Installation's rough
Browser and command line
That should be enough

Voice to text is running now
Text to voice as well
Languages translate
between
Stories we can tell

Baby take my hand
Don't fear the ASIC
Baby I'm your friend
Don't fear the ASIC



Radio functions at this point were good. Randomization, the FEC placeholder, and interleaving were all working. Frame sync word was being received and baseband data was being recovered. We expected to integrate “real” FEC and have it fit. There appeared to be enough resources. We decided to go for it.

We'd had a placeholder for the FEC in the design for a while. Since this was a rate 1/2 convolutional encoder, one bit in to the encoder resulted in two bits out. For the placeholder, we simply duplicated every bit and sent it on its way. Once we replaced this placeholder with the much more complicated real convolutional encoder and decoder, the utilization went over the resource limit. After a lot of work, we got it

back down under the limit and it looked like that the design would still fit in the PlutoSDR's relatively small Zynq 7010.

Or did it?

After carefully writing and testing in loopback an open source 1/2 rate constraint length 7 decoder depth 35 convolutional FEC (yes, the time-honored “NASA code”) we integrated the new code into the frame encoder and decoder in the source code. And, we went over budget. Not by much, but enough to where the design simply would not fit. After some work on reducing the

generous allocation to the transmit and receive FIFO to get back some resources, we then came in under the LUT budget, but the failed routing. The next compromise was to drop back to a simpler interleaver. Interleavers reorder the bits in a frame in a way that spreads them out as widely apart from each other's original position as possible. This makes the frame resilient against burst errors. This is a sudden crash of noise or interference or other dropout that lasts for a specific amount of time. The type of forward error correction that we were using wasn't great against burst errors. If we got a burst error, then it would hurt us more than distributed errors. Distributed errors are the type of damage you get from low signal to noise ratios. Burst errors are like someone ripping out 40 pages of a novel you're reading. That's really annoying, but you can still finish the book. You just lose all that storyline. Now, if someone ripped out 40 pages from a book where the pages were all mixed up and not in sequential order, then you could put the pages back together in the right order and you'd just be missing a page every now and then. That's easier to deal with because the damage is now spread out over the whole book. You can infer more of the storyline since contiguous pages were not affected.

Now, imagine that instead of interleaving the pages before you leave your book lying around book vandals, that you interleaved all the paragraphs. Losing 40 pages worth of paragraphs is much less noticeable. Let's keep thinking about this. How about interleaved sentences? Even better! Finally, let's consider the best possible case. Interleaved letters. At this level of book defense, you can figure out almost every word in the book if you're just missing a letter ever so often. This is how interleaving helps our forward error correction. Our FEC can deal really well with burst errors spread out, just like our brains can deal with missing letters spread out over a whole book. Unfortunately, our "interleave the letters" logic was too expensive. We had to drop back to something like "interleave the pages". We had been interleaving each bit and enjoying the benefits. To reduce the size of the interleaver, we first simplified the design so that the buffer could be assigned to block RAM resources instead of LUTs. At one point this did get things under the LUT count, but it wouldn't route the design. We had a full warehouse, but couldn't reach all the shelves. Next, we changed the interleaver to re-order each byte, instead of each bit. This design required a simpler buffer and smaller lookup table for the positions. And, this new smaller design fit under the LUT count and routing worked again.

Utilization went down to 86%. We were thrilled. This was a huge step forward. We made a firmware build for the PlutoSDR and went into the lab to test over the air. However, the transmitter sent exactly one frame, and then quit. We called this bug "the transmitter stall" and started working on fixing it. The immediate blame fell on the encoder. We reasoned that this was probably a broken handshake between the data passing functions of the FIFO to the encoder, or the encoder to the deserializer. Not great, not terrible, just another thing to sort out. Simulation worked flawlessly,

Just plug and play now
Don't fear the ASIC
Baby take my hand

Come on baby
(And the modem learned to speak)

Come on baby
(No central servers that we need)

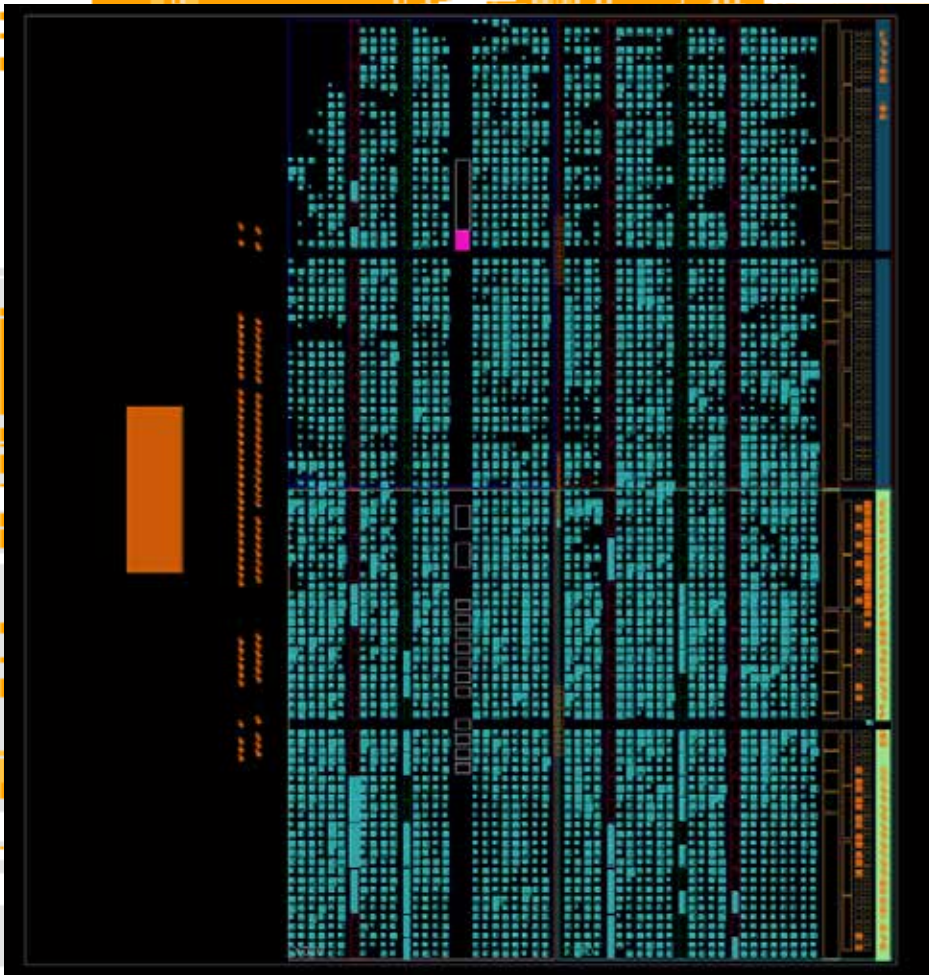
Come on baby
(Took the heavy lifting off your hands)
Come on baby
(Now everyone can understand)

Forty thousand slices more
Made the barrier high
Now the chip does all the work
Radio democratized

Romeo and Juliet
Are talking cross the sea
In different languages they spoke
But now they both run free

Baby take my hand
Don't fear the ASIC
Baby I'm your friend
Don't fear the ASIC

We'll bridge the distance
Don't fear the ASIC
Baby take my hand



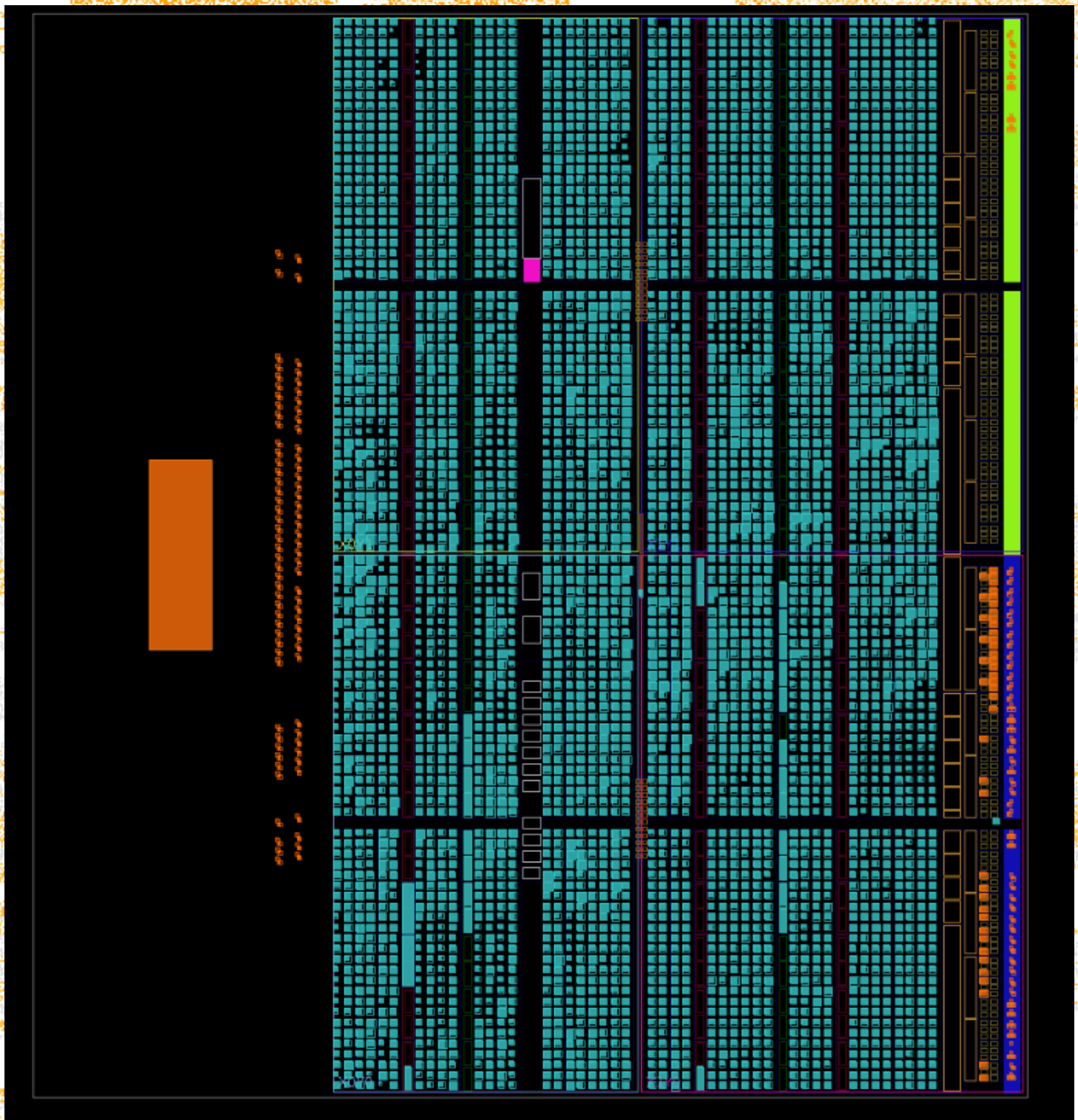
so the problem was only in hardware. Bypassing the encoder resulted in data flowing. It wasn't being received, but the receiver was trying to decode unencoded data of the wrong size, so we didn't think it was much of a clue. But, after combing through the code, and generating a lot of excellent bug fixes and other improvements, the transmitter stall stubbornly remained. Additional signals were brought out to status and control registers, so that we could get a little more visibility into the internals. Unlike in an FPGA simulator, we just can't see most of the signals in the design in the PlutoSDR hardware. We can only see what's exposed to the processor side through registers. We had recently gotten three new registers focused on the FIFO and frame synchronization.

There was plenty of room in two of them, so we took over those bits to tell us what the encoder was up to. And then it got very interesting. The patterns that we were seeing clearly showed a stall. But, not in the forward error correction, which was the new code and therefore getting the suspicion. Instead, the stall was in the interleaver.

The real bug was in the loop for the interleaver. An Opulent Voice data payload has 134 bytes. A forward error corrected data payload has 268 bytes. But, the interleaver was only reordering 134 of the 268 bytes. This was an easy fix, only one line of code. But that one line of code caused utilization to soar above the LUT limit again. This was very curious.

And then the real learning started. The process of turning source code into FPGA hardware involves a process called synthesis. Synthesis figures out how to represent your source code into logic gates. Synthesis is followed by implementation, where we place and route the design in particular hardware targets. Synthesis can and will optimize parts of your design away. Synthesis will remove dead or unreachable code. And, only doing 134 of 268 things in your interleaver will remove quite a bit of unused unreachable code.

Once this became clear, we dug in harder into the design. We knew we had a tricky situation with the pseudo random binary sequencer (PRBS) tricking the synthesizer into not bothering to implement the encoder. We'd already protected the encoder with "don't touch" attributes that told the synthesizer to keep its ambitious little hands off our code. But, we didn't protect the separate module for the "real" FEC. And, we hadn't protected the decoder either. And, now we had this much larger loop in the interleaver. We got to work protecting the design against the optimizer, and then



doing a lot of optimizing ourselves in order to free up more resources. After properly protecting all the new code, which implemented all the missing parts of the encoder and the decoder, we now also had more logic in the design from the proper loop sizing. We removed the (unused) TDD function, I2C peripheral, and SPI peripheral. We simplified anything we could think of that was a buffer. We thought about removing PRBS entirely, but the savings were minimal. For a brief moment, we got under the LUT limit. Here's what that looked like.

It looked like we'd succeeded. But, the table of utilization results broke the bad news. We'd protected the frame encoder, the FEC encoder, and the frame decoder, but the synthesizer had still removed most of the internals from the FEC decoder. It looked good from the top level, but it was

missing vital functions deep inside. Protecting all the signals in the decoder busted our LUT limit hard. There was nothing else to remove without cutting deeply into the quality of the design. We were already settling for hard (instead of soft) decisions in the frame sync word detector and FEC, and we were already running with a compromised byte-level interleaver. We still had symbol lock to integrate, and we didn't want to rewrite the entire design just to fit this one hardware development target.

It was time to move to a different development target. This process of changing from a platform you've outgrown to another with better resources is much like abandoning a sinking ship. You really don't want to jump into the freezing cold ocean unless you can see the lifeboats coming from the other ship. But, we knew this day was coming, and we were prepared.

Opulent Voice Update: From Pluto to Libre

Once the decision was made to find a larger FPGA, we had to decide what development platform we should move to. There are many choices. We have multiple FPGA development boards, ranging from the Basys 3 (33,280 logic units) to the ZCU102 (equivalent to 600,000 logic units). But, in order to continue development, we really needed something with an integrated or connected radio. Something similar to what we were already using, which was an Analog Devices 936x family. We also had experience with the 9009 and 9002 radio chips.

We settled on the LibreSDR, a PlutoSDR clone. See the github repository that we used here: <https://github.com/hz12opensource/libresdr> This SDR had been recommended by Evariste F5OEO, one of the Opulent Voice technical volunteers. Remote Labs had gone ahead and purchased one in anticipation of running out of space on the PlutoSDR. The layout, form factor, and bill of materials was very similar to the PlutoSDR. The FPGA was a Zynq 7020, with 33,200 logic units. At three times the resource capacity of the PlutoSDR's Zynq 7010, but with most other things remaining very similar or the same, this SDR should work for us.

Getting the LibreSDR up and running in the lab for Opulent Voice development had several stages. First, we had to decide how we were going to set up the repository for the source code and firmware creation framework. Most of the mechanisms for this come from either Xilinx (AMD) or Analog Devices. We decided to add the LibreSDR firmware factory in parallel to the PlutoSDR firmware factory in the `pluto_msk` repository. A command line switch would tell the firmware creation scripts what target we wanted. The alternative was a standalone repository.

We gathered the technical differences between the PlutoSDR and LibreSDR designs. We created new constraints, modified the top level source code blocks, and then tackled the firmware creation scripts themselves. This is where we ran into a bit of a headache.

The scripts from Xilinx (AMD) take command line arguments to identify the hardware target. However, these arguments, if given on the command line, cause the variable name to concatenate itself onto directory names. Then, when crucial files are fetched later in the process, the directory doesn't match the place the scripts thought things were. The Xilinx system archive file, which contains the FPGA bitfile created early in the process, came up "missing". This doesn't usually happen to most people because most people simply type "make" for PlutoSDR and not "make TARGET=pluto". Since we were adding the option of making LibreSDR software to an existing

PlutoSDR firmware creation process, we now needed to use the command line argument. And, we ran into the directory names being mangled. We needed a way to tell the scripts that we wanted to use the LibreSDR files and make libre.frm file, and not use the PlutoSDR files and create a pluto.frm file.

Figuring this out and getting around the problem took a combination of carefully reading scripts, cargo-culting a lot of cruft, and making up a new procedure that neither Analog Devices nor LibreSDR folks were using. We'd use the command line switch (make TARGET=libre) but we'd ignore it in later stages. We had tried to clear this variable and then unset this variable, but neither of those tactics worked.

Ignoring the variable after it did its job did work, and a baseline firmware build, with none of our custom code, was produced. This would prove that the basic process of producing the firmware image for the LibreSDR was working. But, was it a usable image? The firmware image was then sent to the lab, installed on the LibreSDR hardware, and it successfully booted up on the device. The first stage of migration from PlutoSDR to LibreSDR was a success.

This modified pluto_msk repository may not be the permanent solution, but it will serve us until a more stable solution comes online in Remote Labs. https://github.com/OpenResearchInstitute/pluto_msk

What is that more stable solution? It's Tezuka, a project from Evariste F5OEO that provides a universal Zynq/AD9363 firmware builder for a variety of SDRs. The current state of this project can be found here: https://github.com/F5OEO/tezuka_fw

This brought us to the second step of the migration process, where we added in our custom logic to the LibreSDR reference design. This brought us to the second step of the migration process, where we added in our custom logic to the LibreSDR reference design, and then attempted to produce a firmware build with our custom code inside. This would reproduce the excellent results we were getting with the Pluto build process.

The PlutoSDR and LibreSDR, and many other radio boards that use Analog Devices radio chipsets, come with a transceiver reference design. This reference design fills in most of the basic system block diagram for the transceiver. This gives designers an enormous head start, since we don't have to design the direct memory access controllers for the transmitter and receiver. We don't have to set up the register access to the microprocessor, or design basic transmit filters. We are also given the digital highways and traffic signals that our data needs to get from memory to the transmitter, and from the received signal back to memory.

The way we integrate our custom design into this existing design has several moving parts. First, we use a file that lists the connections between parts. Each part of the radio block diagram has input and output ports. To insert a new design in an existing pathway, we disconnect that pathway. We break the connections. The unconnected outputs now go to new inputs. The outputs of our new design then go to the newly exposed inputs of the existing design.

Now, this sounds easy enough, and it is. The script we're modifying is a text file. The commands are intuitive and simple. "Connect from here to here with a wire". But this is the beginning of the process, and not the end. Second, we have to tell the software that programs our FPGA the location of the new files that control the behavior of this new block we've dropped on its head, and we have

to make sure that adding a new set of functions in the middle of a busy digital highway doesn't have any repercussions. Spoiler: it almost always does have repercussions!

For example, what we do with Oplulent Voice is take over the pathway that dumps IQ samples from memory to the transmitter, and we take over the pathway that brings IQ samples back to the processor. Instead of IQ samples to and from the processor, which are in a format almost ready to transmit, we instruct the processor to send and receive data bits instead. Our custom FPGA code turns data bits into IQ samples, instead of getting these samples from the processor. We do all the work to prepare, modulate, and encode these data bits into IQ samples inside the FPGA fabric. We are moving more of the work into the FPGA, so that digital signal processing can happen faster and more efficiently. Doing this also frees up the processor to add user interface and user experience functions that a human operator will appreciate. We have the FPGA doing what it does best (DSP) and the processor is much more free to do what it does best (high level human-focused communications tasks). Even better for our future, the FPGA design will then become an ASIC, for compact, efficient, and modern manufactured radios.

After we integrated the design into the FPGA, we created the SD card image for the LibreSDR. There were some hiccups, but they got worked out in short order. The process cleared up, we sent the newly created files over, and power cycled. And, it didn't work!

Now we had a problem on our hands that did not have a clear solution.

Opulent Voice Update: From Pluto to Libre

Porting the Opulent Voice MSK modem from PlutoSDR to LibreSDR hit a hard wall. The PlutoSDR uses a different digital interface internally than the LibreSDR. Part of this new interface (LVDS) is a tuning algorithm. The tuning is needed to get the interface timing calibrated. The transmission tuning algorithm failed consistently during boot. This transmission tuning algorithm doesn't tune the RF transmitter, but refers to how the transmit data from the radio chip is sent out over the bus to the FPGA. Usually, tuning algorithm information is sent to the next block down in the reference diagram, and that block knows how to participate in this tuning algorithm. However, we cut those wires and "soldered in" our own components. We don't do any of this tuning algorithm. What we have done is take over the timing for the radio within our logic. We can handle it, but the radio chip doesn't know this!

The tuning diagnostic showed all failures across the entire timing grid. Here's what it looked like in the logs:

```
SAMPL CLK: 61440000 tuning:
TX 0:1:2:3:4:5:6:7:8:9:ac:d:e:f:
0:#####
1:#####
ad9361 spi0.0: ad9361_dig_tune_delay:
Tuning TX FAILED!
```

This pattern indicates a fundamental problem with the timing not happening at all, and not marginal timing. The system worked fine on PlutoSDR. Stock LibreSDR firmware booted without issues.

What was different? Well, the presence of our design was different. But, how could hardware working perfectly on another platform, and working perfectly in simulation, cause this sort of a failure?

The key insight came from comparing Pluto and LibreSDR at the hardware interface level. Pluto uses a CMOS digital interface to the AD9361 radio chip. No timing calibration needed. LibreSDR uses LVDS, which requires precise timing calibration between FPGA and AD9361. The driver's tuning algorithm sends test patterns through the transmit path and checks what comes back on the feedback clock.

Here's where our MSK circuits caused the problem. In our FPGA design, the MSK modulator sits directly in the TX data path. During kernel boot, before any userspace initialization, MSK outputs zeros. The tuning algorithm expects to see its test patterns reflected back. Instead, it sees nothing but zeros at every timing setting. Every cell fails. Stock LibreSDR firmware passes tuning because its FPGA design has a clean path from the internal DDS to the DAC during boot. The AD9361 driver supports a digital-interface-tune-skip-mode device tree property. That's a fancy way of saying that we have choices for how the driver does these tests. There's a setting that can be 0, 1, 2, or 3.

0 = Tune both RX and TX
1 = Skip RX tuning
2 = Skip TX tuning
3 = Skip both

Setting skip-mode to 2 tells the driver "Don't try to calibrate TX timing because the FPGA handles it." This looked like what would be most correct for our design. MSK owns the transmit data path, and our FPGA timing constraints were already met with 0.932 ns of "slack", or timing margin. RX tuning still runs normally because MSK sits downstream of where this test occurs on the receive path.

The fix was a one-line change in `ori/libre/linux-dts/zynq-libre.dtsi`. Here's that change!

```
adi,digital-interface-tune-skip-mode = <2>; /* Skip TX tuning - MSK owns TX path */
```

This one line removed the block and we were able to boot and confirm transmission over the air. This revealed yet more very interesting problems that will be described in next month's newsletter.

While debugging the boot issue, we discovered the build system was generating a Pluto-centric `uEnv.txt` that lacked SD card boot support for LibreSDR. We had to manually swap in the `uEnv.txt` file to get it to boot off the SD card. This wasn't going to work long-term, so we updated the Makefile. It now automatically adds the `sdboot` command for SD card booting and fixes the serial port address (`serial@e0001000` to `serial@e0000000`). These fixes apply only when `PLATFORM=libre`, keeping Pluto builds unchanged.

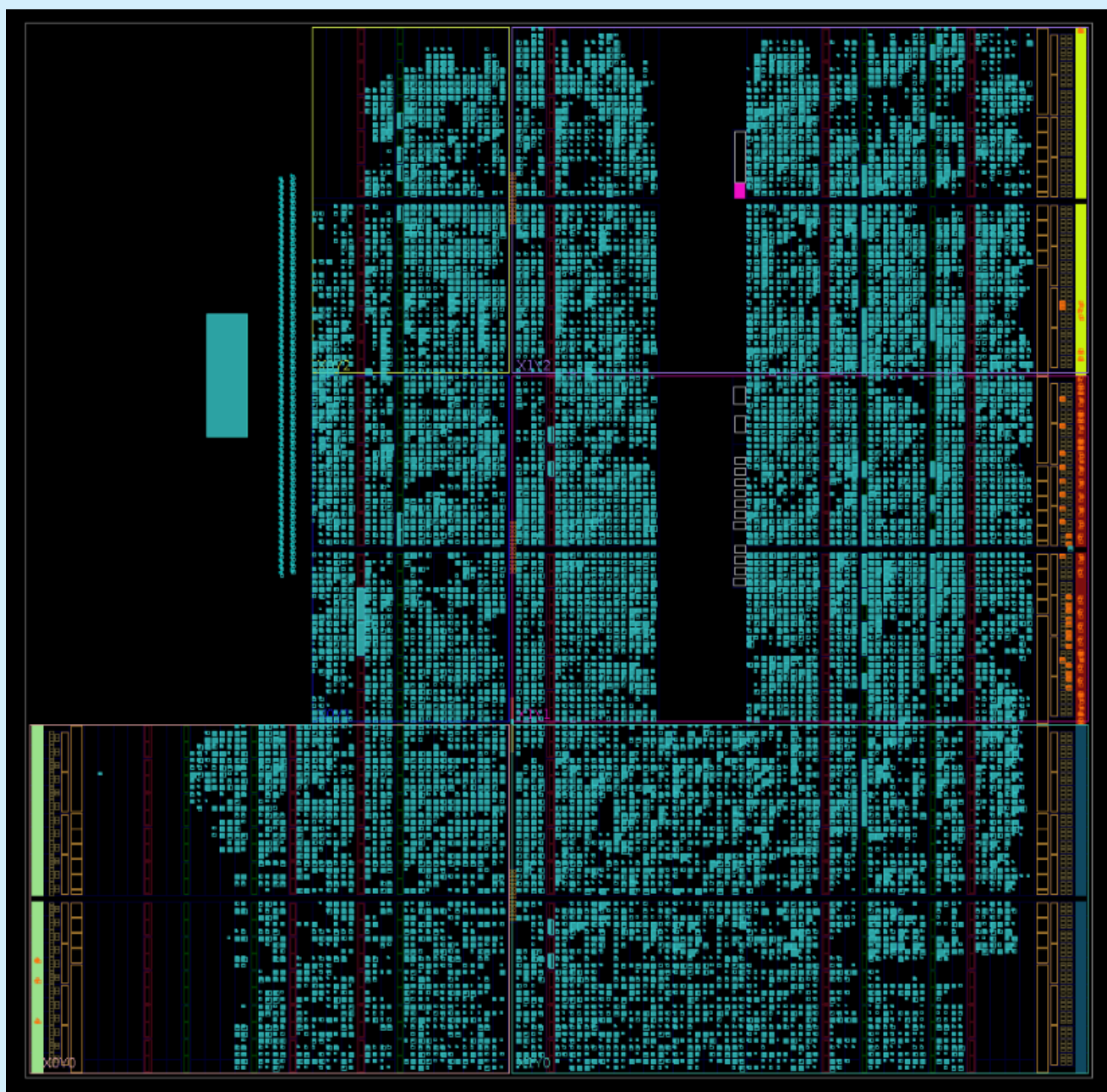
With these changes, LibreSDR booted successfully with the MSK modem. We confirmed TX/RX state machine registers responding via `libiio`, RSSI register readable (custom MSK logic working), frame sync status visible, RF transmission confirmed on spectrum analyzer, and the 61.44 MHz sample clock verified. This was a huge step forward, and gave us valuable experience in porting our design to different FPGAs. We expect to port the design to the `zcu102` development board (with `Ultrascale+` FPGA) in order to demonstrate Haifuraiya HEO/GEO satellite work in 2026. The port process, in order for Opulent Voice to be in the uplink receiver channel bank, will go very similar to what is described here.

In Remote Labs today, we're now debugging actual MSK modem behavior (frame timing and synchronization) rather than fighting boot failures. This represents a significant milestone: the first successful integration of the Opulent Voice FPGA design with LibreSDR hardware.

Lessons learned? CMOS vs LVDS interfaces have different boot-time requirements that aren't obvious until you hit them. When custom FPGA logic sits in the data path, driver auto-calibration may not work as expected. Device tree properties can tell drivers "I know what I'm doing" when appropriate. Build system automation prevents manual copy errors that waste debugging time.





Next steps? Debug the weird 9.42-frame gap appearing after dummy frames. Investigate frame synchronization timing. Loopback testing to verify full transmit and receive chain. And, integration testing with Dialogus and Interlocutor.

Finally, we could close the circle. We had to abandon the PlutoSDR because we ran out of room on the FPGA. What did the FPGA utilization look like now on the LibreSDR?










Well, that's a lot better! The design has a different shape because of the different layout of the programmable logic. And, there's more room. But wait. There's something wrong.

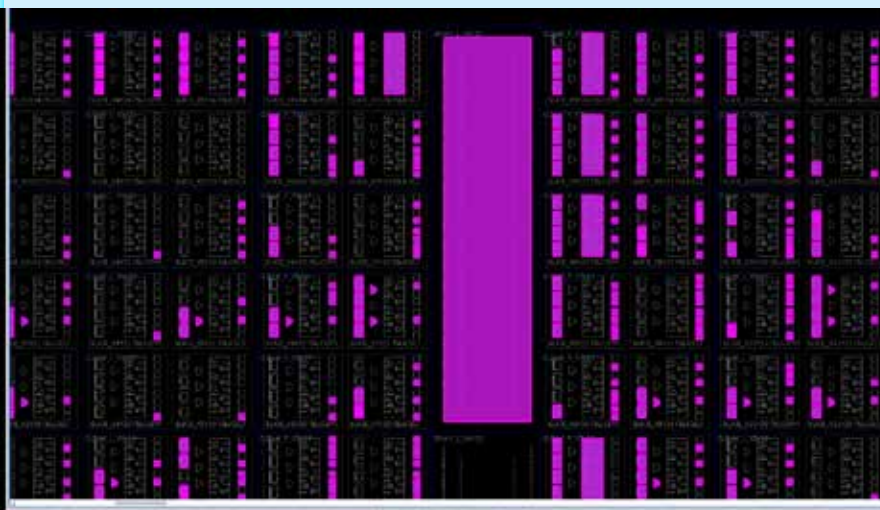
Look at the decoder utilization. Only 3 logic units? That's not even remotely plausible. The Viterbi decoder had been completely optimized out! Our decoder is a hollow shell, just passing data from derandomizer to deinterleaver.

>  u_msk_top_csr (system_msk	2303	1618	0	0	891	2
▼  u_ov_decoder (system_msk	994	5454	144	64	1441	
 U_DECODER (system_msk	3	3	0	0	1	
▼  u_ov_encoder (system_msk	3598	9822	405	197	2645	3
 U_ENCODER (system_msk	2217	5428	0	0	1318	2
▼  u_power_det (system_msk_t	57	41	0	0	21	
 u_ema_1 (system_msk_to	57	41	0	0	21	

Aggressively adding instructions to the synthesis tool reversed the damage. Carefully inspecting the log files for any disconnections or removed logic, and protecting any signals affected anywhere in our design, finally resulted in a completely clean bill of health. Utilization reports were run again, and the true picture of how much logic it takes to place and route our design came into focus.

>  u_msk_top_csr (system_msk	2300	1618	0	0		
▼  u_ov_decoder (system_msk	2670	10280	240	70	2	
 U_DECODER (system_msk	1679	4828	96	6	1	
▼  u_ov_encoder (system_msk	10966	9807	416	192	3	
 U_ENCODER (system_msk	7541	5380	0	0	2	
▼  u_power_det (system_msk_t	56	41	0	0		
 u_ema_1 (system_msk_to	56	41	0	0		

With hard decision Viterbi decoder and a hard decision synchronization word detector, we are at 56% utilization. We now have plenty of room to go back to the bit-level interleaver, upgrade to a soft decision decoder, and get a true correlator for the synch word detector. This is a very satisfying result and gives us a truly good place to be for 2026.



At left, a zoomed-in section of the placed and routed design, showing a successful block RAM assignment.

Opulent Voice Update: Correlator Upgrade

How we went from theory to gates in optimizing the frame synchronization for Opulent Voice.

Finding the beginning of a data frame in a noisy radio channel is like searching for a needle in a haystack. Except, the haystack is constantly shifting, and some of the hay looks suspiciously like needles. For the Opulent Voice digital voice protocol, we've tackled this physics challenge on two fronts. First, we mathematically optimized our synchronization word. Second, we upgraded our FPGA detector from hard-decision matching to soft-decision correlation in order to take advantage of the better mathematics of the optimized codeword.

When we first designed the frame structure for Opulent Voice, we experimented with a familiar tool used in legacy digital voice protocols, called Barker codes. These binary sequences, discovered by R.H. Barker in 1953, have near-perfect autocorrelation properties. This means that when you slide them past themselves, you get a sharp peak at alignment and minimal response elsewhere. The problem? The longest Barker code is only 13 bits, and we needed 24 bits.

The textbook solution is concatenation. Lucky for us, we could stick an 11-bit Barker code together with a 13-bit Barker code, and get 24-bits output. This gives you 0xE25F35, with a Peak Sidelobe-to-Mainlobe Ratio (PSLR) of 3:1. Respectable, but we realized that this wasn't necessarily optimal for 24 bits.

The answer required brute force. With $2^{24} = 16,777,216$ possible sequences, modern computers can exhaustively search the entire space in about 90 seconds. The results were illuminating. 6,864 sequences achieve the optimal PSLR of 8:1. This was nearly three times better than our concatenated Barker code.

We can think of this like antenna directivity. A PSLR of 8:1 means our "main lobe" (the correlation peak when perfectly aligned) is eight times stronger than any "sidelobe" (responses at other alignments). Higher PSLR translates directly to better false-detection rejection, especially in multipath environments where delayed signal copies can trigger spurious sync detections.

For Opulent Voice, we selected 0x02B8DB from the optimal set. Besides having the best possible PSLR, it has good DC balance (11 ones, 13 zeros) and a maximum run length of 6 zeros. This would be important for tracking loop stability in minimum shift keying modulation. The mnemonic is "oh to be eight dB" for the hex digits.

Having an optimal sync word is only half the battle. The detector implementation matters just as much.

Our original frame sync detector used Hamming distance. We counted up how many bits differed between the received pattern when compared to our known sync word. If fewer than some threshold differ, we declared that sync was found. This works fine for strong signals, but there's a fundamental problem buried lower in the noise. By the time bits reach the detector, the minimum shift keying demodulator has already made "hard decisions". That means that each symbol has been quantized to a definitive 0 or 1.

Hard decisions throw away valuable information. The demodulator might have been 99% confident about one bit but only 51% confident about another, yet both become equally weighted in the

Hamming distance calculation. In a D&D analogy, it's like reducing your attack rolls to just "hit" or "miss" without tracking the actual roll. You lose the ability to distinguish a near-miss from a catastrophic fumble. Now, if all we had to work with was hard decisions, then this is the best we could do. But there's something really neat about our demodulator. It also decodes how confident it is of that 1 or 0. The soft decision metric is already produced and already available as a demodulator output.

The solution to our sync word detection optimization problem is to use soft-decision correlation. Instead of binary bits, we work with signed values that indicate both the decision and the confidence. A value of +7 means "definitely a 0 with high confidence," while +1 means "probably a 0 but not very sure." Negative values indicate 1s.

The math is elegant. For each of the 24 sync word positions, we multiply the soft sample by +1 if we expect a '0' or -1 if we expect a '1', then sum all 24 products. Perfect alignment produces a large positive value; misalignment produces values near zero. The peak stands out sharply from the noise floor. We already had the information. We just had to use it.

The new `frame_sync_detector.vhd` in our encoder-dev branch implements soft-decision correlation with several key features:

First, we have parallel data paths. We maintain two shift registers. One is for soft decisions (24 × 16-bit signed values) and one is for hard decisions (24 bits). The soft path feeds the correlator; the hard path handles byte assembly after sync is found. This lets us have our cake and eat it too.

Second, we have polarity-aware correlation. This sounds fancy, but it's a simple process. Our minimum shift keying demodulator uses a convention where positive soft values indicate '0' bits and negative values indicate '1' bits. The correlator accounts for this. When the sync word expects a '1', we subtract the sample (making a negative contribution become positive). This detail matter. If we get the polarity wrong then our correlator becomes an anti-correlator.

Third, we have frame tracking with a flywheel. Once locked, we don't search for sync on every bit. Instead, we count through the known frame length and verify sync where we expect it. This "flywheel" approach dramatically reduces computation and provides robustness against brief interference. We maintain lock through up to two consecutive missed syncs before returning to full search mode. It takes three consecutive successful sync word detections to declare lock. We may update these numbers later on if they are too small or too big. This is a good start and is similar to what commercial communications systems implement. We're on the right track here.

Fourth, we have adaptive thresholds. Our HUNTING mode uses a stricter threshold than LOCKED mode. When searching, we need high confidence to avoid false positives. Once locked and tracking, we can be more forgiving. If sync is in roughly the right place with reasonable correlation, we stay locked. We have to really lose track of our frame boundaries in order to go back to HUNTING, where we search through every single bit we receive with a sliding window and correlator to find our optimized pattern.

Fifth, we have some debug instrumentation. The design exports correlation values and peak tracking signals, essential for threshold calibration. We can't set thresholds blindly; they depend on your ADC scaling, Costas loop gains, and signal levels. We need to know what the correlator calculated and we need to know the peak detected. Otherwise we might be way off on thresholds.

The combination of optimized sync word and soft-decision detection provides measurable improvements.

For pure AWGN channels, correlation detection offers roughly 2-3 dB improvement over Hamming distance at moderate SNR. The optimal sync word provides a slight additional edge at very low SNR compared to concatenated Barker. This means that we can deliver the same performance with about half the signal power. That's not bad. But, the real payoff comes in multipath environments. With delayed echoes from terrain features, the 8:1 PSLR sync word dramatically outperforms the 3:1 concatenated Barker code. The suppressed sidelobes mean echoes are far less likely to trigger false sync detection. Combined with correlation-based detection, we see substantial improvement in frame acquisition reliability under realistic VHF/UHF propagation conditions.

If you're building an Opulent Voice implementation, here's how to calibrate the correlation thresholds.

First, connect the debug correlation output to an ILA or register interface. Second, transmit known sync words and observe the peak correlation value. Third, set `'HUNTING_THRESHOLD'` to 70-80% of this observed peak. Finally, set `'LOCKED_THRESHOLD'` to 40-50% of the observed peak.

The defaults in the VHDL (10,000 for hunting, 5,000 for locked) were conservative starting points. Your actual values will depend on your particular signal chain in your design.

August Puzzle Solution

This month's puzzle update is a VHDL solution for the August "bug report" and September test bench hint.

Link to .vhd file here: https://github.com/OpenResearchInstitute/documents/blob/master/Papers_Articles_Presentations/Articles_and_Announcements/VHDL_newsletter_puzzle_solution.vhd

ORI Regulatory Update: FCC Proposes Deleting BPL Rules

The FCC's "Delete, Delete, Delete" initiative proposes removing the entire Access Broadband over Power Line (BPL) regulatory framework (Part 15 Subpart G) from the Code of Federal Regulations. The reasoning: BPL was never successfully commercialized, so the rules are dead letter. This item is scheduled for the December 18, 2025 FCC Open Meeting.

For those who weren't in the amateur radio trenches in the United States during the mid-2000s, BPL was one of the most contentious regulatory battles in recent ham radio history. The technology promised broadband internet delivery over power lines, but there was a big catch. Power lines make excellent antennas in the HF spectrum. BPL systems operating from 1.7 MHz to 80 MHz range caused substantial interference to amateur radio operations, shortwave broadcasting, and other licensed services. This was documented by radio groups large and small across the US.

ARRL fought this battle all the way to federal court. In 2008, the DC Circuit Court found the FCC had violated the Administrative Procedure Act in its BPL rulemaking. At the time, this was recognized as a significant victory. The court ordered the FCC to reconsider, but the Commission largely reaffirmed its original rules in 2011, leading to continued legal challenges that seemed to promise to drag on for years.

Then a plot twist happened. The market solved the problem before the courts got back around to it. Every major commercial BPL deployment in the United States eventually shut down because they failed their business cases. Fiber, DSL, cable, and wireless broadband simply won. The last significant BPL internet provider (IBEC) closed shop in 2012. Cincinnati's BPL system pulled the plug in 2014.

Part 15 Subpart G contained special provisions for Access BPL devices, including things like exclusion zones, database registration requirements, consultation requirements, mandated measurement procedures, and notching requirements for amateur bands.

Without Subpart G, any future BPL-like device would be regulated under the general Part 15 unintentional radiator provisions. These are the same rules that govern everything from your laptop to your garage door opener.

So, does this matter now? Well, yes. First of all, good riddance. These rules governed a technology that no longer exists in commercial deployment. Removing dead regulations is good regulatory hygiene. If someone wanted to resurrect BPL tomorrow, they'd still need to meet Part 15 emission limits and couldn't cause harmful interference to licensed services. That's the spectrum regulatory reality regardless of Subpart G. But, it's not that simple. The specialized Subpart G rules existed precisely because generic Part 15 limits were inadequate for dealing with how harsh BPL interference really was. NTIA studies showed that BPL systems operating at generic Part 15 limits had essentially 100% probability of interfering with nearby HF operations. Removing the framework means any future power-line broadband technology would start from scratch without the hard-won protections built into Subpart G.

This is being processed as what is known as a Direct Final Rule. This means that the FCC believes it's non-controversial and doesn't require the traditional notice-and-comment process. However,

the agency is accepting input. If adverse comments are filed, the rule would convert to a standard rulemaking requiring public comment.

Parties who have views on this deletion (like ARRL, which invested significant resources fighting these battles) have an opportunity to weigh in before the December 18 meeting.

FCC Document: DOC-415572A1 (Delete, Delete, Delete - Direct Final Rule)

Current regulations: 47 CFR Part 15 Subpart G (§§15.601–15.615)

Background: ARRL v. FCC, DC Circuit Court of Appeals (2008)

For ORI members interested in the regulatory history, the ARRL's BPL archive at arrrl.org/broadband-over-powerline-bpl contains extensive documentation of the interference measurements, court filings, and technical studies from this era.

BPL Regulatory Throwback

From ARRL Bulletin ARLB003, in February 2005:

ARRL CEO David Sumner, K1ZZ, called Powell's performance 'a deep disappointment' after some initial optimism--especially given his unabashed cheerleading on behalf of the FCC's broadband over power line (BPL) initiative.

'It's no secret that we thought Chairman Powell was going entirely in the wrong direction on BPL and dragging the other commissioners and FCC staff along--willing or not--because he was, after all, the chairman,' Sumner said. 'A new chairman might be a chance for a fresh start.'

When the FCC adopted new Part 15 rules for BPL last October, Powell called it 'a banner day.' While conceding that BPL will affect some spectrum users, including 'all those wonderful Amateur Radio operators out there,' Powell implied that the FCC must balance the benefits of BPL against the relative value of other licensed services.

Inner Circle Sphere of Activity

10 December to 5 January - Holiday Hours, Remote Lab cleanup, inventory, renovation, and repairs

31 December - our fiscal year ends and non-profit tax bureau report preparation begins

31 January and 1 February - FODEM Presentation by Evariste F5OEO

If you know of an event that would welcome ORI, please let your favorite board member know at our hello at openresearch dot institute email address.

Thank you to all who support our work! We certainly couldn't do it without you.

Anshul Makkar, Director ORI
Keith Wheeler, Secretary ORI
Steve Conklin, CFO ORI
Michelle Thompson, CEO ORI
Matthew Wishek, Director ORI