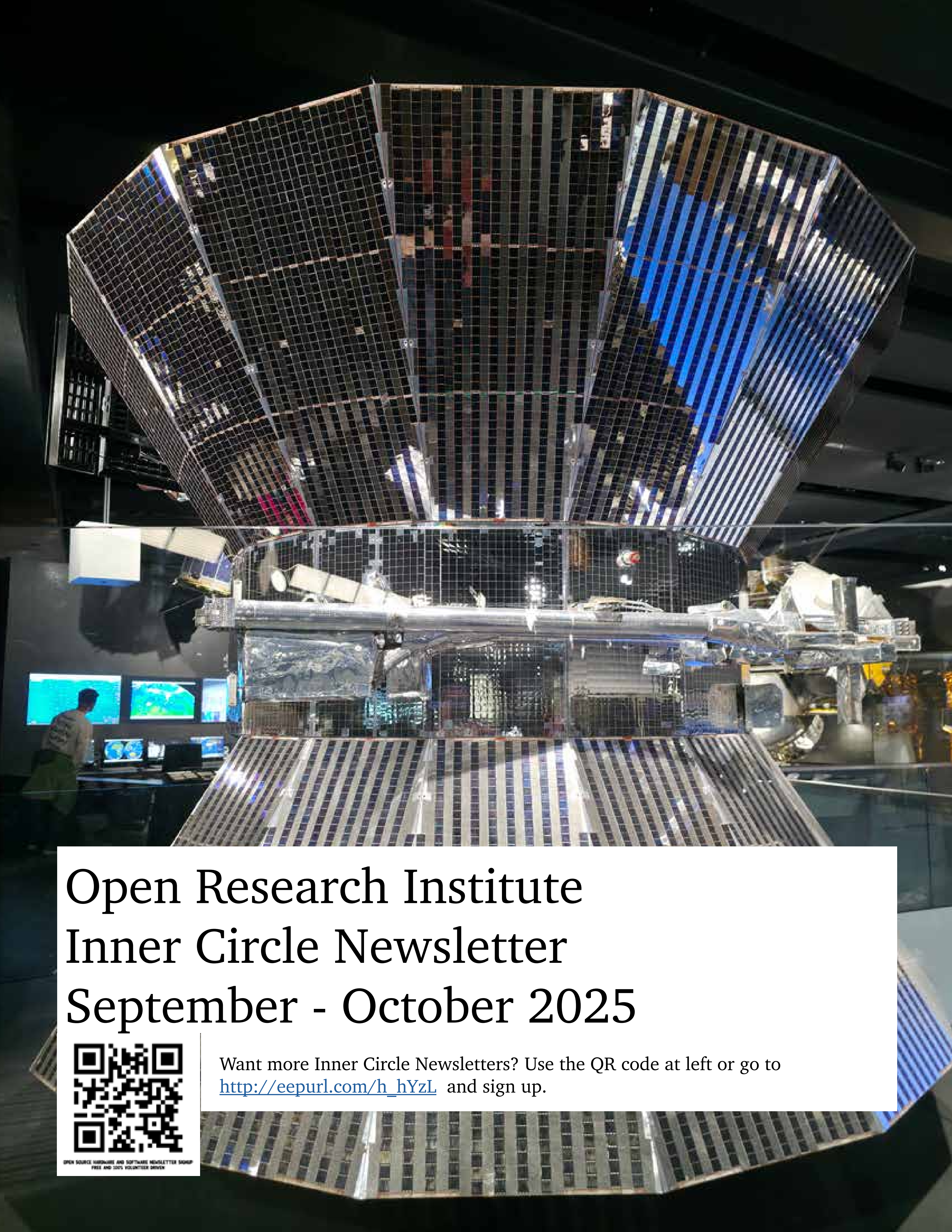


Open Research Institute Inner Circle Newsletter September - October 2025

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

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

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.

This month's puzzle update is a VHDL test bench for the August "bug report". Can you find the root cause using the test bench? Solution in next month's issue.

Link to the VHDL file: https://github.com/OpenResearchInstitute/documents/blob/master/Papers_Articles_Presentations/Articles_and_Announcements/error_starvation_hint.vhd

In addition, the file is listed below:

```
-- PART 2: THE INVESTIGATION
-- VHDL Testbench to expose the Costas Loop Error Starvation
--
-- This testbench will demonstrate the intermittent lock loss
-- and help us discover the root cause through systematic
-- investigation.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.MATH_REAL.ALL;
use STD.TEXTIO.ALL;
use IEEE.STD_LOGIC_TEXTIO.ALL;

entity tb_costas_mystery is
end entity;

architecture testbench of tb_costas_mystery is

    constant CLK_PERIOD : time := 10 ns;  -- 100 MHz
    constant DATA_WIDTH : integer := 16;
    constant PHASE_WIDTH : integer := 12;

    -- Testbench signals
    signal clk : std_logic := '0';
    signal reset : std_logic := '1';
    signal freq_select : std_logic := '0';
    signal rf_input : signed(DATA_WIDTH-1 downto 0);

    -- DUT outputs
    signal i_data, q_data : signed(DATA_WIDTH-1 downto 0);
    signal phase_error : signed(DATA_WIDTH-1 downto 0);
    signal vco_freq : signed(PHASE_WIDTH-1 downto 0);
    signal lock_detect : std_logic;

    -- Stimulus generation
    signal input_phase : real := 0.0;
    signal carrier_freq : real := 10000.0;  -- 10 kHz carrier
    signal noise_level : real := 0.1;       -- 10% noise

    -- Investigation signals
    signal lock_time_f1 : time := 0 ns;
    signal lock_time_f2 : time := 0 ns;
    signal lock_lost_f2 : boolean := false;
    signal test_phase : integer := 0;

    -- Random number generation for noise
    shared variable seed1, seed2 : integer := 1;

    component costas_loop_puzzle is
        generic (
            DATA_WIDTH : integer := 16;
            PHASE_WIDTH : integer := 12;
            F1_OFFSET : integer := 1000;
            F2_OFFSET : integer := 3000
        );
        port (
            clk         : in  std_logic;
            reset       : in  std_logic;
            rf_input    : in  signed(DATA_WIDTH-1 downto 0);
            freq_select : in  std_logic;
            i_data      : out signed(DATA_WIDTH-1 downto 0);
            q_data      : out signed(DATA_WIDTH-1 downto 0);
            phase_error : out signed(DATA_WIDTH-1 downto 0);
            vco_freq    : out signed(PHASE_WIDTH-1 downto 0);
            lock_detect : out std_logic
        );
    end component;

begin

    -- Clock generation
    clk <= not clk after CLK_PERIOD / 2;

    -- Device Under Test
    DUT: costas_loop_puzzle
        generic map (
            DATA_WIDTH => DATA_WIDTH,
            PHASE_WIDTH => PHASE_WIDTH,
            F1_OFFSET => 1000,
            F2_OFFSET => 3000
        )
        port map (
            clk => clk,
            reset => reset,
            rf_input => rf_input,
            freq_select => freq_select,
            i_data => i_data,
            q_data => q_data,
            phase_error => phase_error,
            vco_freq => vco_freq,
            lock_detect => lock_detect
        );

    -- RF Input Signal Generation
    process(clk)
        variable rand_val : real;
        variable current_freq : real;
        variable signal_amplitude : real;
        variable noise_component : real;
    begin
        if rising_edge(clk) then
            -- Select frequency based on freq_select
            if freq_select = '0' then
                current_freq := carrier_freq + 1000.0;  -- F1
            else
                current_freq := carrier_freq + 3000.0;  -- F2
            end if;

            -- Generate phase
            input_phase <= input_phase + (current_freq * 2.0 * MATH_PI *
real(CLK_PERIOD / 1 sec));

            -- Add noise (uniform random)
            uniform(seed1, seed2, rand_val);
            noise_component := (rand_val - 0.5) * noise_level * 32767.0;

            -- Generate clean signal + noise
            signal_amplitude := 32767.0 * 0.8;  -- 80% of full scale
            rf_input <= to_signed(integer(
                signal_amplitude * cos(input_phase) + noise_component
            ), DATA_WIDTH);
        end if;
    end process;

    -- Main Test Sequence
    process
        variable l : line;
        variable lock_start_time : time;
    begin
```

```
        -- Initial setup
        write(l, string'("=== COSTAS LOOP MYSTERY INVESTIGATION ==="));
        writeline(output, l);
        write(l, string'("Testing for intermittent lock loss between F1
and F2..."));
        writeline(output, l);

        reset <= '1';
        freq_select <= '0';
        wait for 100 ns;
        reset <= '0';

        -- TEST 1: Lock to F1 and measure performance
        write(l, string'("TEST 1: Locking to F1 (1kHz offset)..."));
        writeline(output, l);
        test_phase <= 1;

        lock_start_time := now;
        wait until lock_detect = '1' for 50 us;

        if lock_detect = '1' then
            lock_time_f1 <= now - lock_start_time;
            write(l, string'("□ F1 Lock achieved in "));
            write(l, now - lock_start_time);
            writeline(output, l);
        else
            write(l, string'("□ F1 Lock FAILED - timeout"));
            writeline(output, l);
        end if;

        -- Let it settle and verify stability
        wait for 10 us;

        -- TEST 2: Switch to F2 and observe behavior
        write(l, string'("TEST 2: Switching to F2 (3kHz offset)..."));
        writeline(output, l);
        test_phase <= 2;

        freq_select <= '1';  -- Switch to F2
        lock_start_time := now;

        -- Wait a bit and check if lock is maintained or reacquired
        wait for 5 us;

        if lock_detect = '0' then
            write(l, string'("□ Lock lost during F2 transition!"));
            writeline(output, l);
            lock_lost_f2 <= true;
            -- Wait for reacquisition
            wait until lock_detect = '1' for 100 us;
            if lock_detect = '1' then
                lock_time_f2 <= now - lock_start_time;
                write(l, string'("□ F2 Lock eventually reacquired in "));
                write(l, now - lock_start_time);
                writeline(output, l);
            else
                write(l, string'("□ F2 Lock FAILED completely - 
timeout"));
                writeline(output, l);
            end if;
        else
            lock_time_f2 <= now - lock_start_time;
            write(l, string'("□ F2 Lock maintained/acquired in "));
            write(l, now - lock_start_time);
            writeline(output, l);
        end if;

        -- TEST 3: Multiple transitions to expose intermittent behavior
        write(l, string'("TEST 3: Rapid F1/F2 transitions (stress 
test)..."));
        writeline(output, l);
        test_phase <= 3;

        for i in 1 to 5 loop
            -- F1
            freq_select <= '0';
            wait for 2 us;
            write(l, string'("Transition "));
            write(l, i);
            write(l, string'(" F1: Lock="));
            write(l, std_logic'image(lock_detect));
            writeline(output, l);

            -- F2
            freq_select <= '1';
            wait for 2 us;
            write(l, string'("Transition "));
            write(l, i);
            write(l, string'(" F2: Lock="));
            write(l, std_logic'image(lock_detect));
            writeline(output, l);
        end loop;

        -- TEST 4: Add noise and see what happens
        write(l, string'("TEST 4: High noise environment test..."));
        writeline(output, l);
        test_phase <= 4;
        noise_level <= 0.3;  -- Increase to 30%
        freq_select <= '0';   -- Start with F1
        wait for 5 us;

        write(l, string'("F1 with 30% noise: Lock="));
        write(l, std_logic'image(lock_detect));
        writeline(output, l);

        freq_select <= '1';   -- Switch to F2
        wait for 10 us;

        write(l, string'("F2 with 30% noise: Lock="));
        write(l, std_logic'image(lock_detect));
        writeline(output, l);

        -- Final analysis
        write(l, string'(""));
        writeline(output, l);
        write(l, string'("=== INVESTIGATION SUMMARY ==="));
        writeline(output, l);
        write(l, string'("F1 lock time: "));
        write(l, lock_time_f1);
        writeline(output, l);
        write(l, string'("F2 lock time: "));
        write(l, lock_time_f2);
        writeline(output, l);
        write(l, string'("F2 lock lost during transition: "));
        write(l, boolean'image(lock_lost_f2));
        writeline(output, l);
        write(l, string'(""));
        writeline(output, l);
        write(l, string'("CLUE: Notice the dramatically different
behavior between F1 and F2"));
        writeline(output, l);
        write(l, string'("CLUE: Check the loop gains when freq_select
changes"));
        writeline(output, l);
        write(l, string'("CLUE: Error starvation occurs when correction
signals become too weak"));
        writeline(output, l);

        wait;
    end process;

    -- Continuous monitoring for debugging
    process(clk)
        variable l : line;
        variable prev_lock : std_logic := '0';
    begin
        if rising_edge(clk) then
            -- Detect lock transitions for detailed analysis
            if lock_detect /= prev_lock then
                if lock_detect = '1' then
                    write(l, time'image(now));
                    write(l, string'(" LOCK ACQUIRED - F"));
                    if freq_select = '0' then
                        write(l, string'("1"));
                    else
                        write(l, string'("2"));
                    end if;
                    write(l, string'(" Phase Error: "));
                    write(l, integer'image(to_integer(phase_error)));
                    writeline(output, l);
                else
                    write(l, time'image(now));
                    write(l, string'(" LOCK LOST - F"));
                    if freq_select = '0' then
                        write(l, string'("1"));
                    else
                        write(l, string'("2"));
                    end if;
                    write(l, string'(" Phase Error: "));
                    write(l, integer'image(to_integer(phase_error)));
                    writeline(output, l);
                end if;
            end if;
            prev_lock := lock_detect;
        end if;
    end process;

end testbench;

-- WHAT TO LOOK FOR IN THE SIMULATION:
-- 1. F1 locks quickly and reliably
-- 2. F2 takes much longer to lock (if at all)
-- 3. Transitions from F1→F2 often cause lock loss
-- 4. Under noise, F2 performance degrades dramatically
-- 5. Phase error signals in F2 mode are much smaller
--
-- These symptoms point to ERROR STARVATION in F2 mode!
-- The next part will show the solution...
```

“Take This Job”

30 October 2025

Interested in Open Source software and hardware? Not sure how to get started? Here’s some places to begin at Open Research Institute. If you would like to take on one of these tasks, please write hello@openresearch.institute and let us know which one. We will onboard you onto the team and get you started.

Opulent Voice:

- Add a carrier sync lock detector in VHDL. After the receiver has successfully synchronized to the carrier, a signal needs to be presented to the application layer that indicates success. Work output is tested VHDL code.
- Bit Error Rate (BER) waterfall curves for Additive White Gaussian Noise (AWGN) channel.
- Bit Error Rate (BER) waterfall curves for Doppler shift.
- Bit Error Rate (BER) waterfall curves for other channels and impairments.
- Review Proportional-Integral Gain design document and provide feedback for improvement.
- Generate and write a pull request to include a Numerically Controlled Oscillator (NCO) design document for the repository located at https://github.com/OpenResearchInstitute/nco.
- Generate and write a pull request to include a Pseudo Random Binary Sequence (PRBS) design document for the repository located at https://github.com/OpenResearchInstitute/prbs.
- Generate and write a pull request to include a Minimum Shift Keying (MSK) Demodulator design document for the repository located at https://github.com/OpenResearchInstitute/msk_demodulator
- Generate and write a pull request to include a Minimum Shift Keying (MSK) Modulator design document for the repository located at https://github.com/OpenResearchInstitute/msk_modulator
- Evaluate loop stability with unscrambled data sequences of zeros or ones.
- Determine and implement Eb/N0/SNR/EVM measurement. Work product is tested VHDL code.
- Review implementation of Tx I/Q outputs to support mirror image cancellation at RF.

Haifuraiya:

- HTML5 radio interface requirements, specifications, and prototype. This is the primary user interface for the satellite downlink, which is DVB-S2/X and contains all of the uplink Opulent Voice channel data. Using HTML5 allows any device with a browser and enough processor to provide a useful user interface. What should that interface look like? What functions should be prioritized and provided? A paper and/or slide presentation would be the work product of this project.
- Default digital downlink requirements and specifications. This specifies what is transmitted on the downlink when no user data is present. Think of this as a modern test pattern, to help operators set up their stations quickly and efficiently. The data might rotate through all the modulation and coding, transmitting a short loop of known data. This would allow a receiver to calibrate their receiver performance against the modulation and coding signal to noise ratio (SNR) slope. A paper and/or slide presentation would be the work product of this project.

RFBitBanger

HF Digital QRP
HF Digital QRP
HF Digital QRP
HF Digital QRP
HF Digital QRP
HF Digital QRP
HF Digital QRP
HF Digital QRP

Get your kit today
while supplies last!

https://www.ebay.com/usr/openresearchinstitute

$175

Shipping
Calculated
Separately

Looking to Learn more about IQ Modulation?

Basics of IQ Signals and IQ modulation & demodulation – A tutorial by W2AEW

https://www.youtube.com/watch?v=h_7d-m1ehoY

Software Defined Radio For Engineers (free PDF from Analog Devices)

https://www.analog.com/en/resources/technical-books/software-defined-radio-for-engineers.html

These resources will get you well on your way!

https://www.youtube.com/@OpenResearchInstituteInc

FutureGEO Workshop Memo

A FutureGEO Workshop was held 19 September 2025. The event was organized by AMSAT-DL with sponsorship and support from the European Space Agency (ESA). The workshop was immediately before AMSAT-DL's Symposium and Membership Meeting, which was 20-21 September 2025.

Presentations

Peter Gülzow, President and Member of the Board of Directors of AMSAT-DL, opened the workshop, expressing hope for good results.

Peter described the timeline and progression of payloads from SYNCART to QO-100 to FutureGEO. The QO-100 timeline from 2012 to 2018 was explained. QO-100 had some iteration along the way. In the end, AMSAT-DL provided the complete specifications for the custom hardware, which were implemented by Es'hailSat and MELCO. Thus, QO-100 was by no means a "detuned commercial transponder," as ome describe, but was largely composed of tailor-made custom function blocks and circuits. Some commercial off-the-shelf parts were used, in particular both traveling wave tube (TWT) amplifiers used in the NB and WB. These are in fact the "backup" TWT's for the commercial transponders and can be swapped if neccessary.

Volunteer amateurs were unfortunately not directy involved in the building of the satellite. The fact that AMSAT-DL did not build any flight hardware itself was solely due to insurance and liability reasons. With a project volume of US$300–400 million including launch, no insurance company was willing to provide affordable coverage, and in the worst case the personal liability of the AMSAT-DL board members would have been at stake. In the end, it was a win-win situation for AMSAT, because MELCO not only built redundant modules, but also bears full responsibility and guarantees the planned service life of the satellite to the client.
Nevertheless, AMSAT-DL was still deeply involved in Hardware Building blocks: the team defined the requirements and specifications, carried out requirements reviews, the Critical Design Review, final reviews, and participated in quality assurance together with MELCO and Es'hailSat. In addition, AMSAT-DL built the ground station equipment for the spacecraft control center (SCC) consisting of two racks including the beacons, DATV, LEILA, and monitoring. Similar ground stations were also set up in Bochum and at the QARS headquarters.

Peter also explained that the entity that builds the host payload makes a tremendous difference in what the payload looks like, how much power is available, and what policies the hosted payload will be operated under. Es'hail-2 was build by MELCO in Japan, and their decisions set the possibilities and the limitations for the QO-100 hosted payload. We may face similar conditions with a future-GEO. For FutureGEO, multiple designs are expected, covering a range of options and systems design theories.

The 47 degree West footprint shown in slides and online is "a wish". No launch or orbital slot has yet been chosen.

Presentations continued with Frank Zeppenfeldt PD0AP, from the European Space Agency Satellite Communications Group.

Frank explained that there is interest from ESA on a follow-up to QO-100. This lead to the formal solicitation as described in ESA ARTES Future Preparation 1A.126. The FutureGEO outreach and community building process started in 2023. Frank admitted that the communications from ESA have not been as frequent or consistent as desired over the past 18 months. Frank highlighted this workshop as demonstrating an improved FutureGEO community and consensus building state of activity.

AMSAT-DL will help evaluate proposals and is responsible for completing tasks outlined in the ARTES solicitation.

GEO opportunities are hard to get. It's is hard to get a slot. As a way to move the process forward, ESA has indicated that a budget of up to 200,000 € could be allocated in a future phase of ARTES, if the current feasibility study leads to continuation. This would also include funding or building prototypes. However, for the present workshop, only a much smaller preparatory amount was provided, covering infrastructure and moderation costs.

Amateur built payloads are unlikely to fly. But, we must have a number of ideas documented and prototyped so that we are ready to fly something that we want as a community. For example, 18 months ago there was a missed opportunity in the 71-81 GHz band. Ideas need to be developed, and then there will be financing. Peter then explained about attempts to get EU launches. This was very difficult. Peter said that not everyone was welcoming of an amateur payload, and that there were complicated and challenging discussions. Peter reviewed the objective and scope of the ARTES Statement of Work, and then outlined the progress on Tasks 1 through 3.

Task 1 was to identify parties and stakeholders in amateur radio that should be reached out to, and to provide basic storage for data from consultations. Also, to provide background and briefing material about FutureGEO. AMSAT-DL set up a GitLab instance for FutureGEO documents and has provided a chat server for participants.

Task 2 work products are an actively maintained discussion forum on the website, documentation of lessons learned with QO-100, documentation describing requirements formulated by amateur community, documentation describing and analyzing the initial payload proposals from the amateur community, including synergies with other amateur initiatives, and an actively updated stakeholder's database. These parts of the task fall to AMSAT-DL. ESA support in this task will be providing additional contacts of individuals and industry active in the amateur satellite world, initial publicity using ESA's communication channels, and technical support in assessing payload and ground segment options.

Task 3 is to analyze and consolidate 3-4 big/small HEO/GEO designs. The designs need to be interesting for a broad community and also address the need for developing younger engineers. Technologies considered may include software defined radio (SDR), communication theory, field programmable gate arrays (FPGA) and more. A report on the workshop discussions and conclusions is expected along with a report describing the consolidated satellite amateur missions.
Task 1 has been satisfied. This FutureGEO workshop was part of the process of completing Tasks 2 and 3.

Workshop Participants

- Thomas Telkamp PA8Z Amateur Operator
- Danny Orban ON4AOD Amateur Operator
- Bill Slade (with Danny) Amateur Operator

- Nicole Sehrig Bochum Observatory
- Frank Zeppenfeldt PD0AP ESA
- Ken Easton Open Research Institute
- Michelle Thompson W5NYV Open Research Institute
- Schyler Erle N0GIS ARDC
- Brian Jacobs ZS6YZ South African Radio League
- Hans van de Groenendaal ZS6AKV South African Radio League
- Hennie Rheeder ZS6ALN South African Radio League
- Peter Gülzow DB2OS AMSAT-DL
- Thilo Elsner DJ5YM AMSAT-DL
- Matthias Bopp DD1US AMSAT-DL
- Félix Páez EA4GQS AMSAT-EA
- Eduardo Alonso EA3GHS AMSAT-EA
- Jacobo Diez AMSAT-EA
- Christophe Mercier F1DTM, AMSAT-F
- Nicolas Nolhier F5MDY AMSAT-F
- Thomas Boutéraon F4IWP AMSAT-F
- Michael Lipp HB9WDF AMSAT-HB
- Martin Klaper HB9ARK AMSAT-HB
- Graham Shirville G3VZV AMSAT-UK
- David Bowman G0MRF AMSAT-UK
- Andrew Glasbrenner K04MA AMSAT-USA

With audio-visual and logistics support from Jens Schoon DH6BB.
With moderation from Joachim Hecker, a highly-regarded science journalist and electrical engineer.

Brainstorming Session

Workshop participants then focused on answering questions in a moderated session about the satellite system design of a future amateur radio GEO spacecraft. Specific questions were posed at four stations. Workshop participants divided into four groups. Each group spent 20 minutes at each station. All groups visited each station, in turn. The content generated by the participants at the four stations was then assembled on boards. Participants clustered the responses into sensical categories. Then, all participants weighted the clusters in importance with stickers.

Board 1: Mission Services & Overall Architecture

"Your idea for the mission with regards to services offered and overall architecture."
From on-site discussions:
Multiple antennas with beamforming.
Downlink (analog) all antennas? Beamform on the ground.
Physics experiment: measure temp, radiation, etc. Magnetometer.
Downlink in beacon.
For schools.
Something like Astro Pi, coding in space (put data in beacon).
Beacons are very important for reference for microwave. Beacons are good! 24GHz and higher, 10 GHz too.
GEO-LEO multi-orbit communications. In the drawing, communications from LEO to GEO and then from GEO to ground are indicated with arrows.

Doppler and ranging experiments.
Bent pipe and regenerative capability.
Store and forward capability?
Sensor fusion from earth observation, cameras, and data collection.
Multiple bands in and out, VHF to 122 GHz or above.
Inter-satellite links: GEO to GEO, GEO to MEO, GEO to LEO
Education: Still images of Earth, data sent twice in 2-10 minute period. High resolution?
Propagation: greater than or equal to 24 GHz.
Handheld very low data rate text service.
Asymmetric low data rate uplink, traditional downlink.
"Aggregate and broadcast".
Cannot be blocked by local authority.
Use Adaptive Coding and Modulation (ACM) to serve many sizes of stations. Measure SNR, then assign codes and modulations.
Flexible band plan, do not fix entire band, leave room for experimentation.
From online whiteboard:
One sticky note was unreadable in the photograph.
Educational easy experiments requiring receive only equipment.
Same frequency bands as QO-100 to re-use the equipment.
Have a wideband camera as part of the beacon.
Maybe allow nonlinear modulation schemes.
Keep in mind "Straight to Graveyard" as an orbit option. Look not just for GEO but also MEO or "above GEO" drift orbit.
Look at lower frequency bands (VHF, UHF, L-band).
Make the community more digital.
We should have a linear bent-pipe transponder.
Enable Echolink type operations.
For emergency communications it would be useful to be able to pass text-based messages including pictures. Having more bandwidth helps for faster speeds.
We need telemetry, especially for working with students.
Power limitation?
Clustered topics:
Beacons
Coding (like Astro Pi)
Intersatellite links
Telemetry/Sensors
Text and image facilities for emergencies
Near-GEO options for orbits
Adaptive coding and modulation
Bands: transmit 5 GHz 24 GHz and receive 10 GHz 24 GHz 47 GHz 76 GHz 142 GHz
Ranging experiments
Bent pipe or regenerative (including mode changes input to output)

Board 2: Payload & Antenna Subsystem Platform

"Your idea for the payload and antenna subsystem, and their platform accommodation requirements."
From on site discussions:
Phased Arrays:

Simple, fixed, e.g. horn array
Electronically steerable
Switchable
User base:
Large user base at launch with 2.4 GHz uplink 10 GHz downlink.
Good user base at launch with experiments.
Use the frame of the solar cells for HF receive.
Distributed power amplifiers on transmit antennas.
Array of Vivaldi antennas allows multiple frequencies with same array.
Camera at Earth "still" 1 image every 2 to 5 minutes. Camera at satellite for "selfie sat" images. AMSAT-UK coll 2012?
5 GHz uplink and 10 GHz downlink at the same time as 24 GHz uplink and 10 GHz downlink. 76 GHz 47 GHz beacon or switched drive from FPGA core (transponder).
Non-GEO attitude control? GEO 17 degree beamwidth.
ISAC integrated sensing and communications. For example, 24 GHz communications gives free environmental and weather information.
Keep in mind different accommodation models/opportunities. 16U, almost-GEO, MEO, very small GEO.
Optical beacon laser LED.
PSK telemetry.
Spot beam global USA + EU?
Uplink 10 GHz
Downlink 24 GHz, 47 GHz, 76 GHz
132 GHz and 142 GHz?
From online whiteboard:
If we need spotbeams especially at the higher frequencies we should have multiple and/or be flexible.
24 GHz uplink meanwhile is no more than expensive (homebuilt) transverter 2W, 3dB noise figure, approximately 400 Euros, ICOM is meanwhile supporting also 10 GHz and 24 GHz.
Beacons for synchronizing
Cameras: maybe one covering the Earth and one pointing to the stars (for education). Attitude calculation as a star tracker. Space weather on payload. Radiation dose or SEU?
On a GEO satellite we should not only have 12 cm band but for instance also the 6 cm band (WiFi) close by makes the parts cheap. For example, 90 watt for about 400 Euros.
We should have as back at AO40 we should have a matrix. For example, multiple uplink bands at the same time combined in one downlink.
Maybe get back to ZRO type tests.
Inter satellite link would be nice.
Higher frequency bands: use them or lose them.
Linking via ground stations makes sense for Africa being probably in the middle of GEO footprints.
One note was unreadable in the photo, but it starts out "beacon on the higher frequency bands would be interesting as a reference signal for people"
Clustered topics:
Phased arrays.
Large user base at launch from 2.4 GHz uplink and 10 GHz downlink. Good user base at launch from experiments.
Use frame of the solar cells for HF receive.
Keep in mind different accommodation models/opportunities. 16U, almost-GEO, MEO, very small GEO.

Camera for education
Education at lower level such as middle school (12-14 years old).
ISAC: Integrated sensing and communications, for example 24 GHz communications gives free environmental and weather information.
Transponders and frequencies.

Board 3: Ground Segment Operations & Control

"Your idea for the ground segment for operations and control of the payload, and the ground segment for the user traffic."
From on-site discussions:
For hosted payloads, you have less control. For payloads without propulsion, there is less to control. Payloads with propulsion you have to concern yourself with control of the radios and the station position and disposal. There is a balance between being free to do what you want with the spacecraft and the ownership of control operations. This is a keen challenge. Someone has to own the control operations.
If you are non-GEO, then 3 or more command stations are required. Payload modes are on/off, limited by logic to prevent exceeding power budgets. Control ground station receives data (for example, images) and sends them to a web server for global access. This expands interest.
GEO operations may need fewer control operators. HEO (constellation or single) may need more.
Redundancy: multiple stations with the same capabilities.
Clock: stable and traceable (logging). For example, multi band GPSDO with measurement or White Rabbit to a pi maser?
Distribute clock over transponder and/or lock transponder to clock.
Open SDR concept, that the user can also use:
Maybe a Web SDR uplink
Scalable
Remote testing mode for users
Payload clock synchronized via uplink - on board clock disciplined to uplink beacon or GPS. Payload telemetry openly visible to users. Encryption of payload commands?
From online whiteboard:
Allow also an APRS type uplink for tracking, telemetry, maybe pictures, maybe hand-held text based two-way communications, etc. Which suggests low-gain omni-directional antennas.
Ground control for a GEO especially with respect to attitude will most likely be run 24/7 and thus hard for radio amateurs.
If we are using a GEO hosted payload then ground control of the satellite would be provided by the commercial operator.
If we want to have full control on the the payload such as the beacons: on QO-100 on the wide-band transponder we are limited as the uplink is from the commercial provider.
We should have some transponders with easy access (low requirements on the user ground segment) but it is ok to have higher demands on this on some others (like the higher GHz bands).
We will need multi band feeds for parabolic dishes.
Clustered Topics:
We want telemetry!
We should have some transponders with easy access (low requirements on the user ground segment) but it is ok to have higher demands on this on some others (like the higher GHz bands).
We will need multi band feeds for parabolic dishes.
Ownership and planning of payload control.

Redundancy: multiple stations with the same capabilities.
Clock: stable and traceable (logging). For example, multi band GPSDO with measurement or White Rabbit to a pi maser?
Distribute clock over transponder and/or lock transponder to clock.
Open SDR concept, that the user can also use:
Maybe a Web SDR uplink
Scalable
Remote testing mode for users

Board 4: User Segment

"Your idea for the user segment."
From on-site discussions:
Entry-level phased array.
Look at usage patterns of QO-100 for what we should learn or do.
Capability to upload TLM to a central database. The user segment should allow us to start teaching things like OFDM, OTFS, etc.
Easy access to parts at 2.4 GHz, 5 GHz, 10 GHz, 24 GHz.
Broadband RX (HF, VHF, UHF) for experiments
LEO satellite relay service VHF/UHF
Single users: existing equipment makes it easy to get started. Low complexity. Linear.
Multiple access:
Code Division Multiple Access (CDMA), spread spectrum, Software Defined Radio + Personal Computer etc.
Orthogonal Frequency Division Multiplex (OFDM), with narrow digital channels.
Cost per user
Experimental millimeter wave
Something simple, linked to a smartphone, "Amateur device to device (D2D)" or beacon receive.
School-level user station in other words, something larger.
Entry-level off the shelf commercial transceivers.
Advanced uses publish complex open source SDR setups.
Extreme low power access: large antenna on satellite? ISM band? (Educational and school access)?
"IoT" demodulator service
A fully digital regenerating multiplexed spacecraft: Frequency division multiple access uplink to a multi-rate polyphase receiver bank. If you cannot hear yourself in the downlink, then you are not assigned a channel. Downlink is single carrier (DVB-S2).
Mixing weather stations and uplink (like weather APRS).
Dual band feeds exist for 5/10 GHz and 10/24 GHz. We are primary on 47 GHz. Why not a 47/47 GHz system.
Keep costs down as much as possible.
From online whiteboard:
Linear transponder for analog voice most important but we should also allow digital modes for voice and data in reserved segments of the transponder.
Multiple uplinks combined in a joint downlink (as AO13 and AO40).
Experimental days.
Tuning aid for single sideband (SSB) audio.
Segment to facilitate echolink type operation using satellite in place of internet.
Allow low-power portable operations.

Allow experimental operations. For example, on very high frequencies.
Store and forward voice QSOs using a ground-based repeater and frequencies as MAILBOX (codec2 or clear voice + CTSS).
A codec2 at 700 bps/FSK could permit to reuse UHF radios using the microphone input and a laptop. We are exploring this using LEOs.
Depending on the available power we may want to allow non-linear modulation schemes.
We should support emergency operations.
Depending on the platform, we may want to rotate operational modes.
Clustered topics:
Simple (ready made) stations: off the shelf designs and 2.4 GHz 5.0 GHz 10 GHz 24 GHz 47 GHz
Sandbox for experiments and testing new modulations and operations. Just have to meet spectral requirements.

Workshop Retrospective:

A workshop retrospective round-table was held. QO-100 experiences were shared, with participants emphasizing the realized potential for sparking interest in both microwave theory and practice. The satellite enabled smaller stations and smaller antennas. It is unfortunate that people building it were limited in speaking about the satellite performance due to non-disclosure agreements. QO-100 has enabled significant open source digital radio components, such as open source DVB-S2 encoders that are now in wide use. Educational outreach and low barriers to entry are very important. $2000 Euro stations are too expensive. The opportunity for individuals to learn about microwave design, beacons, and weather effects is a widely appreciated educational success. 10 GHz at first seemed "impossible" but now it's "achievable". QO-100 had a clear effect on the number of PLUTO SDRs ordered from Analog Devices, with the company expressing curiosity about who was ordering all these radios. GPSDO technology cost and complexity of integration has come down over time. Over time, we have seen continuous improvements in Doppler and timing accuracy. Building it vs. Buying it is a core question moving forward. A high level of technical skills are required in order to make successful modern spacecraft. Operators clearly benefited from QO-100, but we missed out on being able to build the hardware. Builder skills were not advanced as much due to the way things worked out. "What a shame we didn't build it". No telemetry on QO-100 is a loss. Design cycle was too short to get telemetry in the payload, and there were privacy concerns as well. Building QO-100 was an expensive and long process. Mitsubishi took this workload from us. Moving forward, we should strive to show people that they have agency over the communications equipment that they use. We are living in a time where people take communications for granted. There is great potential for returning a feeling of ownership to ordinary citizens, when it comes to being able to communicate on the amateur bands.

Documentation of the workshop and "Lessons Learned from QO-100" will be done starting now through the end of 2025. Prototypes are expected to be demonstrated in 2026.

Revision History:

2025-09-24 Draft by Michelle Thompson W5NYV - missing one participant name and seeking edits, corrections, and additions.
2025-09-30 Rev 0.1 by Peter Gülzow DB2OS
2025-10-20 Rev 0.1 by Peter Gülzow DB2OS - participants updated

TX IIO Timeline Implementation for Opulent Voice

by Paul Williamson KB5MU

In the August 2025 Inner Circle newsletter, https://www.openresearch.institute/2025/09/12/inner-circle-newsletter-august-2025/ I wrote some design notes under the title **IIO Timeline Management in Dialogus — Transmit**. I came to the tentative conclusion that the simplest approach might be just fine. This time, I will discuss the implementation of that approach. You might want to re-read those articles before proceeding with this one.

In that simplest approach, we set a recurring decision time Td based on the arrival time of the first frame. Ideally, the frames would all arrive at 40ms intervals. We set Td to occur halfway between a nominal arrival time and the next nominal arrival time. No matter what, we push a frame to IIO at Td, and never at any other time. If no frames have arrived for transmission before Td, we push a dummy frame. If exactly one frame has arrived, we push that frame. If somehow more than one frame has arrived, we count an error and push the latest arriving frame. The "window" for arrival of a valid frame is the maximum 40ms wide. In fact, the window is always open, so we don't need to worry about what to do with frames arriving outside the window.

As part of that approach, Dialogus groups the arriving frames into **transmission sessions**. The idea is that the frames in a session are to be transmitted back-to-back with no gaps. To make acquisition easier for the receiver, we transmit a frame of a special preamble sequence before transmitting the first normal frame of the session. Also for the benefit of the receiver, we transmit a frame of a special postamble sequence at the very end of the session. The incoming encapsulated frames don't carry any signaling to identify the end of a session. Dialogus has to infer the session end by noticing that encapsulated frames are no longer arriving. We don't want to declare the end of a session due to the loss of a single encapsulated frame packet. So, when a Td passes with no incoming frame, Dialogus generates a dummy frame to take its place. This begins a **hang time**, currently hard-coded to 25 frames (one second). If a frame arrives before the end of the hang time, it is transmitted (not a dummy frame) and the hang time counter is reset. If the hang time expires with no further encapsulated frame arrival. that triggers the transmission of the postamble and the end of the transmission session.
As it turns out, the scheme we arrived at is not so different from what we intended to implement in the first place. Older versions of Dialogus tried to set a consistent deadline at 40ms intervals, just as we now intend to do. There were several implementation problems.

One problem was that the 40ms intervals were only approximated by software timekeeping, which under Linux is subject to drift and uncontrolled extra delays. A bigger problem was that the deadline was set to expire just after the nominal arrival time of the frame, meaning that frames that were only slightly late would trigger the push of a dummy frame. The late frame would then be pushed as well. The double push would soon exceed the four-buffer default capacity of the IIO kernel driver. After that, each push function call would block, waiting for a kernel buffer to be freed.

The original Dialogus architecture had one thread in addition to the main thread. That thread was responsible for receiving the encapsulated frames arriving over the USB network interface. It used the blocking call **recvfrom()**, so it had to be in a separate thread. That thread also took care of all the processing that was triggered by the arrival of an encapsulated frame, including pushing the de-encapsulated frames to the kernel.
Timekeeping was done in the main thread by checking the kernel's monotonic timer, then sleeping a millisecond, and looping like that until the deadline arrived. Iterations of this loop were also counted

to obtain a rough interval of 10 seconds to trigger a periodic report of statistics. If the timekeeping code detected the deadline passing before the listener thread detected the frame's arrival, the timekeeping code would push a dummy frame and then the listener thread would push the de-encapsulated frame, and either or both could end up blocked waiting for a kernel buffer to be freed.

The duties of the existing listener thread were limited to listening to the network interface and de-encapsulating the arriving frames, placing the data in a global buffer named modulator_frame_buffer. All responsibility for pushing buffers to the kernel was shifted to the new frame timekeeping thread, called the timeline manager. The main thread was left with no real-time responsibilities at all. This architecture change was not strictly necessary, but I felt that it was cleaner and easier to implement correctly.

Timestamping for debug print purposes has so far been done to a resolution of 1ms using the get_timestamp_ms() function. For keeping track of timing for non-debug purposes, we added get_timestamp_us() and keep time in 64-bit microseconds. This is probably unnecessary, but it seemed possible that we might need more than millisecond precision for something.

The three threads share some data structures. In particular, the listener thread fills up a buffer, which the timeline manager thread may push to IIO. To make this data sharing thread-safe, we use a common mutex that covers almost all processing in all three of the threads. The threads release the mutex only when waiting for a timer or pushing a buffer to IIO. This is a little simple-minded, but it should be ok for this application. None of the threads do much in the way of I/O, so there would be little point in trying to interleave their execution at a finer time scale. The exception would be output to the console, which consists of the periodic statistics report (not very big and only occurs every 250th frame) and any debug prints from anywhere in the threaded code. If this becomes a bottleneck, it will be easy enough to do that I/O outside of the mutex.

Each of the three threads is implemented in three functions. One function is the code that runs in the thread, the second function starts the thread, and the third stops the thread. The main function of each thread runs a loop that checks the global variable named *stop* but otherwise repeats indefinitely.

Listener Thread

The listener thread is rather similar to the one thread in previous versions of Dialogus. It still handles the beginning of a transmission session, but then relinquishes frame processing to the timeline manager thread. Pushing of data frames to IIO (after the first frame of a session), pushing dummy frames, and pushing postamble frames are no longer the responsibility of the listener thread.

The listener thread waits blocked in a recvfrom() call, waiting for an encapsulated frame to arrive over the network. When the first encapsulated frame arrives and a transmission session is not currently in progress, the listener thread detects that as the start of a new transmission session. It calls *start_transmission_session*, which takes note of the frame's arrival time, and computes the first Td of the new session. It then creates and pushes a preamble frame, and also pushes the de-encapsulated frame. This creates the starting conditions for the timeline manager to take over for the rest of the transmission session.

Thereafter, for the duration of the transmission session, the listener thread merely de-encapsulates incoming frames into the single shared buffer named *modulator_frame_buffer* and increments a counter named *ovp_txbufs_this_frame*. These are left for the timeline manager thread to handle.

Timeline Manager Thread
The timeline manager has a pretty simple job. It wakes up at 40ms intervals during a transmission session, and pushes exactly one frame to IIO each time. That frame will be a frame received by the listener thread in encapsulated form, if one is available. Otherwise, it will be a dummy frame or a postamble frame.

The timeline manager thread has two main paths, depending on the state of the variable ovp_transmission_active. If no transmission is active, the timeline manager thread waits on a condition, which will be set when the listener thread receives the first encapsulated frame of the next transmission. If a transmission is already active, the timeline manager thread locks the timeline_lock mutex, then checks to see if a decision time has been scheduled. If so, it gets the current time, and compares that with the decision time to see if it has passed. If it has not, the timeline manager releases the mutex, computes the duration until the decision time, and sleeps for that long, so that next time this thread runs, it will probably be time for decision processing. When decision time has arrived, it starts to decide. First it checks to see if one or more encapsulated frames has been copied into the buffer. Each frame beyond one represents an **untimely frame** error, which it simply counts. The buffer contains the one arrived frame, or the last of multiple arrived frames, and this buffer is now pushed to IIO. On the other hand, if no frames have arrived, we will push either a dummy frame (starting or continuing a hang time) or, if the hang time has been completed, a postamble frame. If we push a postamble frame, we go on to end the transmission session normally, and then wait for notification of a new session. Otherwise, we again compute the duration from now until the next decision time, and sleep for that long.

Period Statistics Reporter Thread
The periodic statistics reporter thread has an even simpler job. It wakes up at approximate 10s intervals during a transmission session, and prints a report to the console of all the monitored statistics counts.

Some Optimizations

Ending a Transmission Session
The normal end of a transmission session includes pushing a postamble frame, and then waiting for it to actually be transmitted before shutting down the transmitter. There is no good way to wait for a frame to actually be transmitted, so we make do with a fixed timer. That timer was 50ms, which I believe is just 40ms plus some margin. The new value is 65ms. Computing from the Td that resulted in the postamble being pushed, that’s 20ms for the last dummy frame to go out, then 40ms for the postamble frame to go out, plus 5ms of margin. This will need to be checked against reality when we’re able.

The delay is not desired when the session is interrupted by the user hitting control-C. In that case, we want to shut down the transmitter immediately. This is handled in a new *abort_transmission_session* function.

Special Frame Processing Paths
The preamble is a 40ms pattern of symbols. It does not start with a frame sync word and it does not contain a frame header. Thus, the preamble processing (found in *start_transmission_session*) skips over some of the usual processing steps and goes directly to *push_txbuf_to_msk*.

The dummy frames and postamble frames are less special. They do start with the normal frame sync word, and contain a frame header copied (with FEC encoding already done) from the last frame of data transmitted in that transmission session. This provides legal ID and also enables the receiver to correctly associate these frames with the transmitting station. As a result, these can be created by overlaying the encoded dummy payload or encoded postamble on top of the last payload in *modulator_frame_buffer* and pushing that buffer to IIO as normal.

All three of these special data patterns (preamble, dummy, and postamble) are now computed just once and saved for re-use.

About Hardware/Software Partitioning
Up until this work on timeline management, we have assumed that the modulator in the FPGA would eventually take care of adding the frame sync word, scrambling (whitening), encoding, and interleaving the data in the frames. Under those assumptions, Dialogus just sends a logical frame consisting of the contents of the frame header (12 bytes) concatenated with the contents of the payload portion of the packet (122 bytes), for a total of 134 bytes per frame.

The work described here was done under a different set of assumptions, so that we could make progress independent of the FPGA work. We assumed that the modulator hardware is completely dumb. That is, it just takes a sequence of 1's and 0's and directly MSK-modulates them, without any other computations.

Probably, neither of these assumptions is completely correct. It now seems that the FPGA in the Pluto is not big enough to handle all of those functions. However, a completely dumb modulator in the FPGA is not a very satisfying or clean solution. We may end up with multiple solutions, depending on the hardware available and the requirements of the use case targeted.

About Integrating Receiver Functions
This work has been entirely focused on the transmit chain. This makes sense for a Haifuraiya-type satellite system, where the downlink is a completely different animal from the uplink. However, we also want to support terrestrial applications, including direct user-to-user communications. In that case, it would be nice to have transmitter and receiver implemented together.

We already have taken some steps in that direction, in that the FPGA design we're working with has both modulator and demodulator implementations for MSK. We've been able to demonstrate "over the air" tests of MSK for quite some time already, using the built-in PRBS transmitter and receiver in the FPGA design. Likewise, we've been able to demonstrate two-way and multi-way Opulent Voice communications using a network interface in place of the MSK radios, using Interlocutor and Locus.

Integrating the two demos will bring us significantly closer to a working system. The receive timeline has to be independent of the transmit timeline, and has different requirements, but the same basic approach seems likely to work for receive as for transmit. I expect that adding another thread or two for receive will be a good start.

Finding Optimal Synchronization Words for Digital Voice Protocols

An Exhaustive Search Yields Substantial Improvement Over Classical Sequences for Opulent Voice Protocol from Open Research Institute

By Michelle Thompson W5NYV

Abstract

Synchronization word selection is critical for reliable frame detection in digital communications protocols. While classical sequences like Barker codes are well-known for their autocorrelation properties, are they truly optimal for arbitrary lengths? This article presents a computational approach to finding optimal 24-bit synchronization sequences, achieving a Peak Sidelobe-to-Mainlobe Ratio (PSLR) of 8:1. This is 2.67 times better than concatenated Barker codes. We demonstrate that exhaustive search is feasible for moderate-length sequences. Does a much better PSLR translate to better performance improvements over traditional approaches? The answer depends on how you implement your sync word detector.

Introduction

In digital communications systems, synchronization words (sync words) are bit patterns decided upon and known in advance that mark the beginning of data frames. Receivers correlate, or match, the incoming bitstream against these expected sync words in order to detect frame boundaries. The quality of a sync word is characterized by its autocorrelation properties. Good autocorrelation means that we get a strong bit-to-bit match, or peak number of matching bits, when we exactly align our known sequence with the matching sequence in the incoming bit stream, and we don't get much correlation when it's not aligned exactly.

For example, if we are looking for a sequence, like 11111, and we compare it to our record of what we are looking for (also 11111), then we get the maximum number of 5 matches when it's perfectly lined up. This is what we call the main lobe. But, there's more to the story. We want weak responses at all of the other offsets of the known word to the word in the bitstream. Look again at 11111. We are shifting in our received bitstream one bit at a time, and moving it "past" the known word. So with one bit overlap we have one match. With two bits overlap we have two matches. With three we have three, and with four we have four. Then we get the maximum, which is five of five. Now we start shifting it away, and we get four, then three, then two, then one. In order to use this as a sync word, we'd have to insist on five matches to claim that we have things lined up. If there was any noise in our signal, and there is always noise, then the values that we receive over the air might cause some of these numbers to change, and we might get a false alarm. The number of matches when a sync word is at any other offset other than perfectly lined up are very important to us. Those numbers of matches at all the other offsets are called side lobes. Just like in radio signals, we want a strong main lobe and smallest possible side lobes. We need to suppress our side lobes. 11111 does not have good autocorrelation.

The Challenge

For the Opulent Voice digital voice protocol, we needed a 24-bit synchronization word. The protocol operates most-significant-bit (MSB) first and requires excellent autocorrelation properties to maintain reliable frame synchronization in challenging propagation conditions. Our initial approach was to use the best sequences that we knew about, which are from a family of binary sequences called Barker codes. But, there are no 24 bit Barker codes. The longest Barker code is 16 bits. Therefore, we picked an 11 bit code and a 13 bit code and we stuck them together. This concatenated code seemed entirely reasonable given Barker codes' reputation for optimal autocorrelation properties. But was this truly the best we could do?

Background: Barker Codes and Beyond

What Are Barker Codes?

Barker codes are binary sequences with near-perfect autocorrelation properties. They were discovered by R.H. Barker in 1953. For a Barker code of length *N* bits, the autocorrelation side lobes are at most one bit. Compare that to the 11111 example above, where the max side lobe was four bits in the five bit code. Only 13 Barker codes are known to exist, with the longest being length 13. For arbitrary lengths like 24 bits, no Barker code exists. A common approach if you need something longer than 13 bits is to concatenate two shorter Barker codes, so that the total number adds up to the number of bits you need.

The PSLR Metric

We use Peak Sidelobe-to-Mainlobe Ratio (PSLR) to quantify sync word quality.

```
PSLR = Main Lobe / Peak Sidelobe
```

This is similar to the concept of directionality with an antenna. The main lobe value in PSLR is the autocorrelation result when the sync word has “zero lag”. We slide the received bit stream past the stored copy of the sync word, and measure how many bits match. Zero lag is the point in time where the sync word in the received bit stream is perfectly aligned with the copy of the sync word we’re looking for. Peak side lobe is the maximum value of the autocorrelation at any non-zero lag. This is the maximum number of bit matches at all the other offsets before and after perfect alignment. For our 11111 example, the main lobe was 5 and the peak side lobe was 4, for a PSLR of 5/4.

Higher PSLR indicates better discrimination between true sync and false detections. In the bipolar representation used for signal processing analysis (where we use 1 and -1 instead of binary 1 and 0) the autocorrelation is computed as:

$$R(\tau) = \Sigma\ s(i) \times s(i+\tau)$$

In the Opulent Voice digital hardware implementation, the equivalent operation uses binary representation with logical operations to count bit matches. Bipolar or binary give the same results. Bipolar is traditional to use in signal processing, and binary is used in our Opulent Voice hardware descriptive language implementations. This choice, to match the demodulated bits, turns out to be an important one. Improving the sync word without improving the method of detection means that the increased quality of the sync word doesn't immediately translate to better performance.

Alternative Sequences Considered

Before resorting to an exhaustive search, we evaluated several classical approaches:

1. Single Barker Codes The longest Barker code (length 13) is too short for our 24-bit requirement. Padding with zeros destroys the autocorrelation properties.

2. Concatenated Barker Codes Combining Barker-11 (binary 11100010010) and Barker-13 (binary 1111100110101) yields 24 bits (binary 1110 0010 1111 0011 0101 or hex 0xe25f35).

- PSLR is 3.00:1 (4.7 dB). The main lobe is 24 but the peak side lobe is 8.

The concatenation works reasonably well, but the two constituent codes creates side lobes larger than the ±1 ideal.

3. M-Sequences (Maximal Length Sequences) M-sequences have perfect *periodic* autocorrelation properties. All side lobes equal 1 when treated as circular sequences. However, they only exist at lengths of powers of two minus one (2^n - 1, so 7, 15, 31, 63...). For sync word detection, we need *aperiodic* (linear) autocorrelation since we detect the sequence once, not repeatedly. This is an important distinction, because it's also why we don't use a Zadoff-Chu sequence.

Testing 31-bit m-sequences truncated to 24 bits, the best truncation PSLR: 24:21 (1.2 dB).

The "textbook perfect" periodic property doesn't translate to our one-shot detection scenario and the length constraint ($24 \neq 2^n - 1$) forces truncation.

4. Zadoff-Chu Sequences A common question: "Why not use Zadoff-Chu sequences? They have perfect autocorrelation and are used in LTE/5G!"

Zadoff-Chu sequences are indeed excellent when they are used for their intended application. However, they're fundamentally mismatched for binary sync words. First, they are complex-valued (IQ samples) with constant amplitude. Our sync word must be binary. Quantizing Zadoff-Chu to binary destroys the "perfect" autocorrelation properties.

A 24-bit quantized Zadoff-Chu sequence (root 5) yields only 2.18:1 PSLR with peak side lobe of 11. This is worse than the concatenated Barker code.

Zadoff-Chu sequences have perfect *periodic* (circular) autocorrelation, assuming the sequence repeats infinitely. Sync word detection requires *aperiodic* (linear) correlation. We match the sequence once in a bitstream, not as a repeating pattern. The "perfect" property doesn't apply to one-shot detection.

Another issue is that using Zadoff-Chu properly requires complex baseband processing (I/Q), frequency offset estimation and correction, complex correlation hardware, and magnitude threshold detection

Our binary approach requires a few logic gates, bit counters, and simple threshold comparison.

Zadoff-Chu sequences are brilliant for orthogonal frequency division modulation (OFDM) cellular systems with frequency-domain processing and multiple access requirements. For simple binary sync words in time-domain correlation, exhaustive search of binary sequences provides better PSLR with far simpler implementation.

5. The M17 Case Study: Sync Word Length Tradeoffs

The M17 digital voice protocol provides an instructive comparison. M17 operates at 9600 bps and uses 16-bit sync words (versus our 24-bit requirement):

LSF/Stream frames: 0x3243 (digits of π) - PSLR: 2.67:1
Packet frames: 0xFF5D - PSLR: 2.00:1
BERT frames: 0xDF55 - PSLR: 2.00:1

Why 16 bits instead of 24? Sync word length is a design tradeoff. Opulent Voice has nearly six times faster bitrate, so there's a faster acquisition with 24 bits at the faster bitrate, even though the sync word is longer. The longer the sync word the higher the main lobe, but 16 bits and 24 bits can achieve the same PSLR. 16 bits has less noise immunity, but can be considered lower overhead.

A 16-bit sync word is a reasonable choice for VHF/UHF operation where SNR is typically higher and lower overhead is valuable. However, M17's selection of a sync word without optimization represents a very big missed opportunity.

What if M17 had optimized their 16-bit sync word?

Exhaustive search of all 65,536 possible 16-bit sequences reveals 80 optimal sequences with PSLR = 8.00:1 (peak sidelobe = 2). Example: 0x066b.

M17 could have achieved the same 8:1 PSLR as Opulent Voice's optimal 24-bit sequence, with zero additional overhead, just by running an exhaustive search on their chosen 16-bit length.

Both M17 (16-bit) and OPV (24-bit) made reasonable length choices for their respective applications. The mistake isn't the length. It's failing to optimize for the chosen length. The sync word value was selected from the digits of pi, written out in binary. While irrational numbers appear to be random, we can see from the m-sequence evaluation above that random or pseudorandom numbers do not necessarily have good aperiodic autocorrelation at all. It's just luck that the PSLR is as high as it is for one of the three sync words. Using randomly selected "digits of π" left a lot of performance behind. The packet and BERT sync words have even higher side lobes than the LSF/Stream words, and lower PSLR. This degrades detection performance in multi-path environments by quite a bit, even if the signals are strong. VHF and UHF have multi-path, so this is a real concern.

6. P25 Protocol Case Study

Project 25 (P25) is a suite of standards for digital two-way radio communications widely used by public safety organizations in North America. Developed through joint efforts of APCO International, federal agencies, and standardized by the Telecommunications Industry Association (TIA), P25 Phase 1 represents one of the most mature and widely deployed digital voice protocols in the public safety sector. P25 Phase 1 operates at 9600 bits per second using either C4FM (Continuous 4-level FM) or CQPSK (Compatible Quadrature Phase-Shift Keying) modulation with a symbol rate of 4800 symbols per second.

The P25 protocol uses a 48-bit frame synchronization pattern (a hex value of 0x5575F5FF77FF). This is sent at the beginning of every message and is then inserted every 180 ms during voice messages. The purpose of the frame synchronization pattern in P25 is to enable operators to monitor or join a conversation in progress. This is an important system feature in public safety use cases. While P25 uses twice as many bits for synchronization as Opulent Voice, it transmits these 48 bits at 4800 symbols per second. This takes 5 milliseconds to transmit the complete sync word. Opulent Voice 24-bit sync word transmitted at 54.2 kHz takes approximately 0.44 milliseconds.

P25's design reflects its public safety heritage. The longer sync word and aggressive repetition prioritize reliability and rapid synchronization over spectral efficiency. The protocol accepts higher overhead (48 bits every 180 ms) to ensure first responders can reliably join active transmissions, However, protocols operating at higher data rates (like Opulent Voice at 54.2 kHz) can achieve equivalent time-domain robustness with shorter bit sequences, as a correlation gain comes from processing bandwidth rather than pattern length alone.

When we analyze for the PSLR of P25, we don't compare single bits for the sync word analysis because P25 is not a binary mode. P25 symbols are two bits instead of one bit. P25 receivers operate at 4800 symbols per second. There are two bits (dibits) per symbol, for a bit rate of 9600 bits per second.

The matched filter or correlator in the receiver processes 24 symbols, not 48 bits.

The dibit pattern looks like this:

+3 +3 +3 +3 +3 -3 +3 +3 -3 -3 +3 +3 -3 -3 -3 -3 +3 -3 +3 -3 -3 -3 -3 -3

From dibits of:

01 01 01 01 01 11 01 01 11 11 01 01 11 11 11 11 01 11 01 11 11 11 11 11

Resulting in a PSLR of 3:1 (4.7 dB). This is very reasonable for a manually-designed sync word from the 1990s.

7. D-Star Protocol Case Study

D-Star (Digital Smart Technologies for Amateur Radio) was one of the first digital voice modes widely adopted in amateur radio. It was developed by the Japan Amateur Radio League (JARL) in the late 1990s and early 2000s. D-Star uses GMSK (Gaussian Minimum Shift Keying) modulation and operates at 4800 bits per second for voice communications.

The D-Star frame structure uses a distinctive synchronization pattern that differs from most other digital voice protocols in its design philosophy. Rather than a single long sync word, D-Star employs a composite synchronization approach.

The D-Star sync pattern consists of multiple components. First there is a Frame Start. This is 24 bits consisting of a 10-bit preamble (1010101010) followed by two 7-bit patterns (1101000 + 1101000).

The complete pattern in hexadecimal representation is 0xAA9A9A.

This gives a 24-bit total synchronization sequence. However, the internal structure reveals the design choices. The alternating 1010101010 preamble provides bit timing recovery for the receiver's clock synchronization. Second, it acts as a "warning" that a frame sync pattern is about to arrive. This is conceptually similar to the Ethernet preamble design. In Opulent Voice, we have a separate bit-timing preamble frame that is sent once at the beginning of the transmission. The synchronization words are separate entities that are pre-pended to every frame. D-Star has a preamble and a frame sync for every frame.

When we analyze the full 24-bit D-Star sync pattern for PSLR we get 2.4:1 (7.6 dB).

This PSLR is lower than both concatenated Barker codes (3:1) and our optimal sequences (8:1). The performance gap exists because the D-Star sync word was optimized for a different objective. The repeating 10-bit preamble provides excellent clock recovery characteristics but introduces substantial autocorrelation side lobes. This represents a conscious tradeoff, prioritizing robust bit synchronization over optimal frame detection PSLR.

For D-Star's intended application in the early 2000s, this design made practical sense. Receivers of that era benefited greatly from the explicit bit-timing preamble, and the relatively high SNR of typical VHF/UHF amateur radio operation meant that the lower PSLR wasn't a problem.

However, by modern standards and with current DSP capabilities, D-Star could have achieved better overall performance. A 24-bit optimized sync word (PSLR 8:1) combined with modern digital clock recovery techniques provides both superior timing synchronization and better frame detection in challenging multi-path environments. The D-Star case demonstrates that protocol design reflects the technological constraints and assumptions of its era, and optimization opportunities often exist when reevaluating older designs with current capabilities.

8. NXDN Protocol Case Study

NXDN is a digital radio protocol jointly developed by Icom and Kenwood (JVC Kenwood) in the mid-2000s as a competitor to DMR and P25. NXDN uses 4-level FSK (4FSK) modulation and offers both 4800 bps and 9600 bps variants, with the 4800 bps mode being most common. Like DMR, NXDN operates within 12.5 kHz or 6.25 kHz channel bandwidth.

NXDN's synchronization strategy is notable for using relatively short sync patterns compared to other digital voice protocols. According to the NXDN Technical Specifications (TS 1-A), the Frame Sync Word is specified as 10 symbols (20 bits) rather than the longer patterns used by DMR (24 symbols/48 bits) or P25 (24 symbols/48 bits).

The NXDN sync pattern differs for various frame types. Conventional voice or data frames get a 20-bit pattern at symbol level (10 symbols of 4FSK). For example, 0xCDF5D. However, NXDN's implementation adds complexity through its use of a Link Information Channel Header (LICH). The LICH provides additional information about the frame type and call parameters, and in practice, receivers often use a combination of the Frame Sync Word plus LICH bits for robust synchronization.

When analyzing NXDN at the symbol level (10 symbols for the Frame Sync Word), the PSLR is approximately 2.50:1 to 3.00:1, depending on the specific pattern variant. This is comparable to DMR performance but achieved with fewer symbols.

The shorter sync word in NXDN represents a deliberate design tradeoff. The advantages are lower overhead (20 bits vs 48 bits for DMR/P25), faster frame acquisition in high-SNR environments, more payload capacity for voice or data.

There are some disadvantages. NXDN has lower inherent PSLR due to the shorter sequence. The main lobe can only be equal to the length of the bit sequence. You can’t match more bits than you have in the word. There is reduced multipath rejection capability because of the lower PSLR. It has greater sensitivity to channel impairments that are found in the intended deployment environments.

NXDN compensates for the shorter sync pattern through several mechanisms. First, the protocol includes the LICH which effectively extends the "sync" functionality across multiple bits. Second, NXDN systems typically operate in cleaner RF environments (professional and commercial applications) where the high-SNR assumption holds. Third, the protocol includes robust forward error correction throughout the frame structure.

Could NXDN have optimized better? The constraint is again the multilevel modulation, and yes, an exhaustive search is computationally feasible and would likely yield patterns with better PSLR than the manually selected sequences.

The NXDN case illustrates that sync word length is a system-level design decision. Shorter patterns reduce overhead and increase data throughput, but at the cost of reduced robustness in challenging propagation conditions. The optimal choice depends on the expected operating environment and the receiver's overall detection strategy, including whether additional frame structure elements like LICH are used to augment synchronization.

9. Yaesu System Fusion (C4FM) Case Study

Yaesu System Fusion, introduced in 2013, represents one of the more recent commercial digital voice protocol designs in amateur radio. System Fusion uses C4FM (Continuous 4-level FM) modulation, which is effectively 4FSK. System Fusion operates at 9600 bits per second with a symbol rate of 4800 symbols per second.

The System Fusion sync word design reflects lessons learned from earlier protocols while introducing some unique characteristics. According to the Yaesu Digital Specifications and an analysis of the protocol structure, System Fusion uses a sync pattern encoded as 0xD471C9634D. This is 20 symbols with 2 bits per symbol for a total of 40 bits.

This sync word appears at the beginning of every frame and marks the boundary between frames. Like DMR and NXDN, System Fusion operates with dibit symbols (4FSK), so the synchronization analysis must be performed at the symbol level rather than the individual bit level.

The 40 bit sync pattern in System Fusion represents an intermediate design choice between NXDN's shorter pattern of 10 symbols and DMR's longer pattern of 24 symbols. When analyzed as a 20 symbol 4FSK sequence, the autocorrelation properties give a PSLR of 3:1. System Fusion's sync word choice reflects several design priorities.

First, the protocol strongly emphasizes backwards compatibility with analog FM. The frame structure must support Automatic Mode Select (AMS) where repeaters automatically detect whether incoming signals are digital or analog. This influences sync word design because false detection of sync patterns in analog noise could cause incorrect mode switching. That would be a very negative experience for an operator.

Second, System Fusion prioritizes voice quality and high-speed data transfer capabilities. The protocol allocates substantial bandwidth to AMBE+2 voice encoding and supports transmission of images and GPS data. The sync word overhead must be minimized to maximize payload capacity for these features.

Third, the protocol includes detailed framing information in the FICH (Frame Information Channel Header) that immediately follows the sync word. This 200-bit header provides extensive frame type, mode, and configuration information. The receiver uses both the sync word and FICH in combination for robust frame detection, partially compensating for the moderate PSLR of the sync word itself.

Could System Fusion have achieved better performance? Yes, but with caveats. An exhaustive search would be feasible with a good computer. However, System Fusion's real innovation lies not in sync word optimization but in its overall system architecture. The combination of AMS, high-quality voice encoding, integrated data capabilities, and Internet linking through WIRES-X creates a complicated ecosystem. The sync word is adequate for its role and represents reasonable engineering given the protocol's broader objectives.

The System Fusion case demonstrates that sync word optimization exists within a larger context of protocol design. Sometimes "good enough" for the sync word allows engineering resources to focus on features that provide more significant value to end users. The protocol achieves its goals effectively, even if the sync word itself could theoretically be improved through exhaustive optimization.

The Computational Approach

Is Exhaustive Search Feasible?

For 24 bits, the search space contains 2^{24} = 16,777,216 possible sequences. Modern computers with optimized numerical libraries make this fast and easy. We benchmarked the autocorrelation computation and put a simple test in the Jupyter lab notebook. The rate was ~175,000 sequences/second (NumPy on a newer laptop), and total time was ~96 seconds (1.6 minutes). This is a fully feasible search for ordinary computers.

Search Implementation

The search converts each integer from 0 to 2^{24}-1 into a bipolar sequence, computes its autocorrelation, and tracks sequences with maximum PSLR.

The bit ordering must match your protocol (most significant bit vs. least significant bit).

Results

The exhaustive search completed in 73 seconds and found 6,864 sequences with optimal PSLR of 8.00:1, or 18.1 dB.

Some of the top optimal sequences:

Hex Value	PSLR	Peak Sidelobe
0x00e564	8.00:1	3
0x7006ca	8.00:1	3
0x268b00	8.00:1	3

0x2e9c80	8.00:1	3
0x3c9a80	8.00:1	3

All optimal sequences share the same peak side lobe magnitude of 3, compared to 8 for the concatenated Barker code.

Analysis and Verification

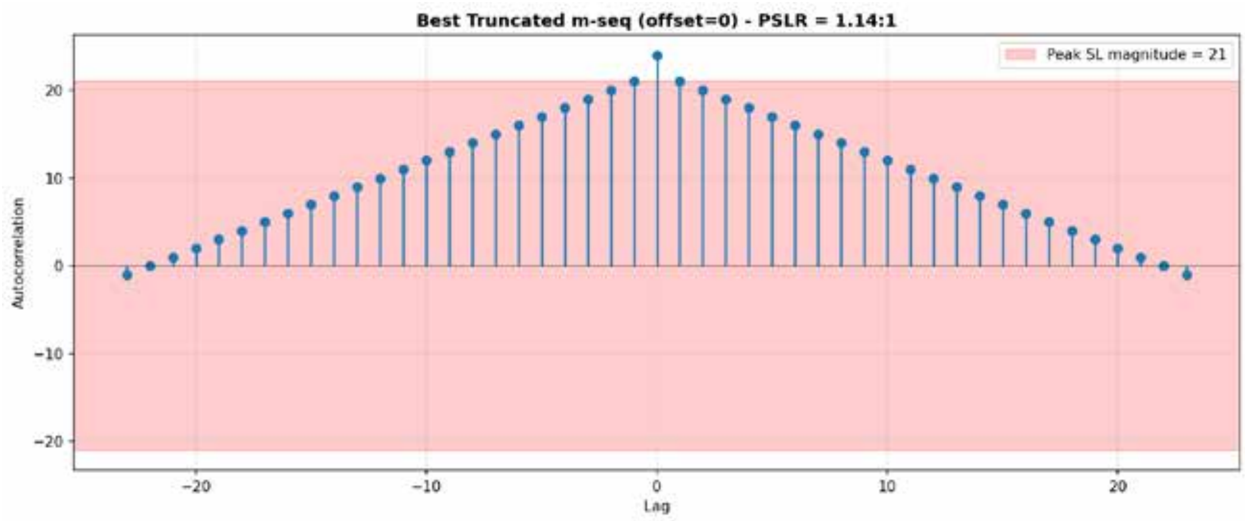
Sequence	PSLR	PSLR (dB)	Peak Sidelobe	Relative Performance
Concatenated Barker (0xe25f35)	3.00:1	9.5 dB	8	Baseline, ok
Best m-sequence truncation	1.14:1	1.2 dB	21	Really bad
Optimal (0x7006ca)	8.00:1	18.1 dB	3	Really good

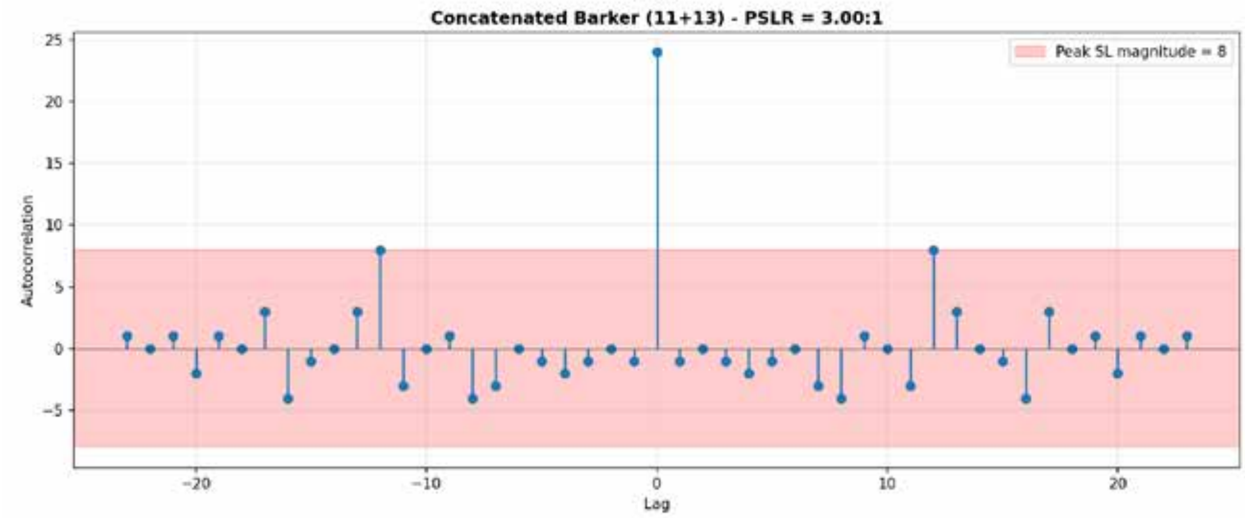
Autocorrelation Visualization

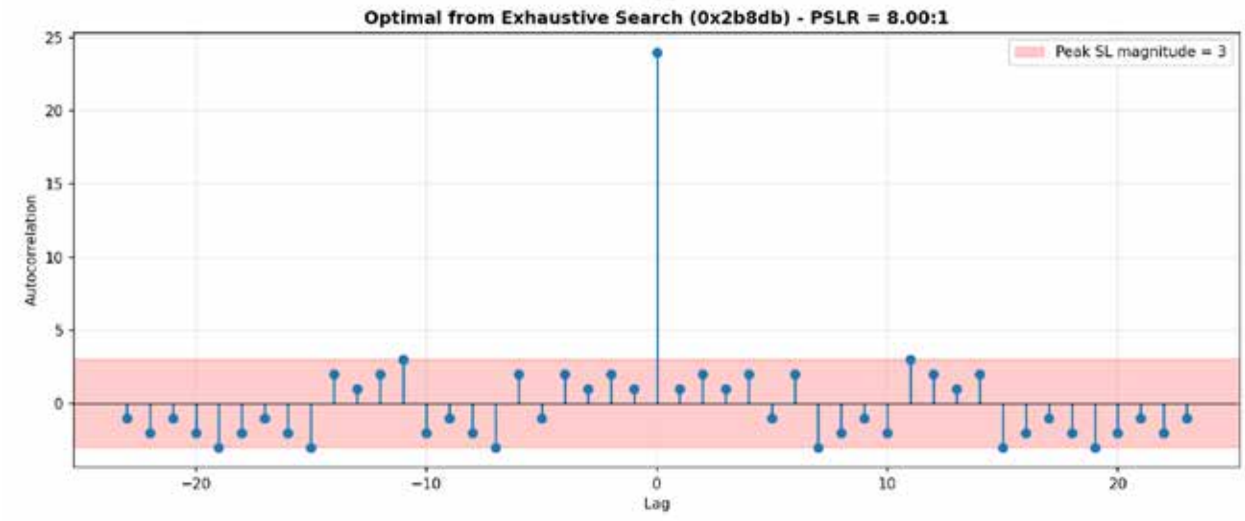
Figure 1 shows the autocorrelation functions for the three sequence types. The optimal sequence exhibits sharp main lobe at lag 0 (amplitude 24), suppressed side lobes (maximum ±3 in bipolar format), and a symmetric structure.

The improvement is dramatic: peak side lobes were reduced from ±8 to ±3, while maintaining the same 24-bit length and full main lobe amplitude.

Figure 1 three_way_autocorrelation_comparison.png







Practical Considerations

Implementation in Digital Hardware

While our analysis uses bipolar (±1) representation for mathematical convenience, hardware implementation typically uses binary logic with XOR operations. See the current implementation of sync word frame detection here: https://raw.githubusercontent.com/OpenResearchInstitute/pluto_msk/refs/heads/main/src/frame_sync_detector.vhd

Choosing a Detection Threshold

With PSLR = 8:1 and peak side lobe = 3, we can tolerate up to 3 bit errors while maintaining reliable sync detection at the bit level. This is using Hamming Distance as a function. Hamming Distance is the number of positions in a sequence that differs from another sequence we are comparing it to. For example, 11111 compared to 11001 has a Hamming Distance of 2.

A Hamming Distance of 3 means that 3 out of the 24 bits might be in error. We want to ensure detection even in this case of three errors. The sync word is not protected by forward error protection, as it is added after the payload is processed through randomization, forward error correction, and interleaving.

Errors	Correlation	Detection
0	24	Strong
1	22	Strong
2	20	Strong
3	18	Practical limit
4+	≤16	Risk of false detection

A threshold of 18 (Hamming distance ≤ 3) provides robust operation while minimizing false positives.

Which Optimal Sequence to Choose?

With 6,864 optimal sequences available, selection criteria include:

DC balance: Choose sequences with as equal numbers of 0s and 1s as possible
Run length: Try to avoid long strings of consecutive identical bits to relieve pressure on the tracking loops
Spectrum: Sequences with more transitions help some modulations in terms of detection
Aesthetics: Round hex values are easier to remember and type

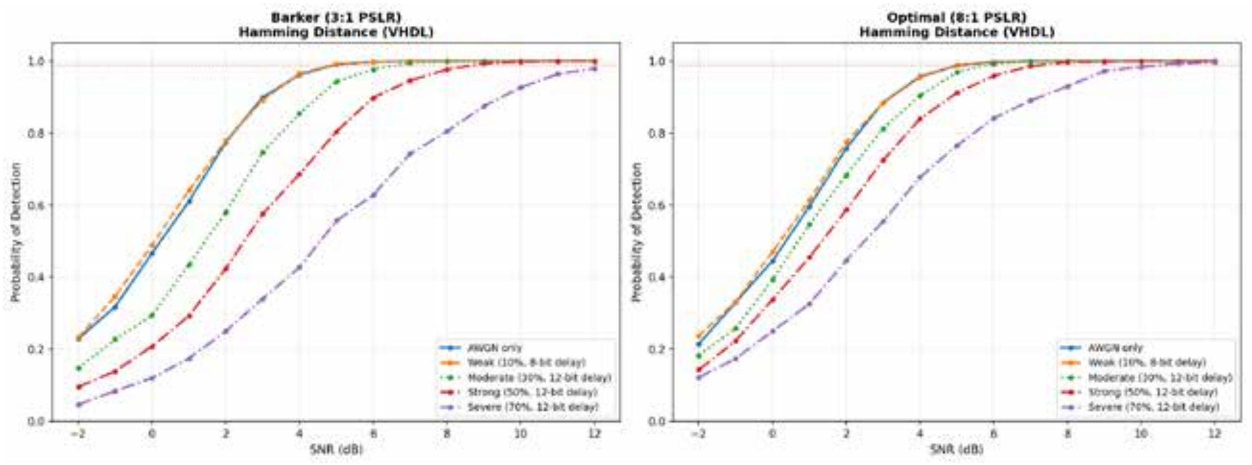
For Opulent Voice, we selected 0x02b8db. It's got ideal PSLR of 8:1, very low DC bias with 11 ones and 13 zeros, and a maximum run of 6 zeros in a row. These are good values, and the mnemonic can be "oh to be eight db".

Performance

Better PSLR provides better performance in two situations.

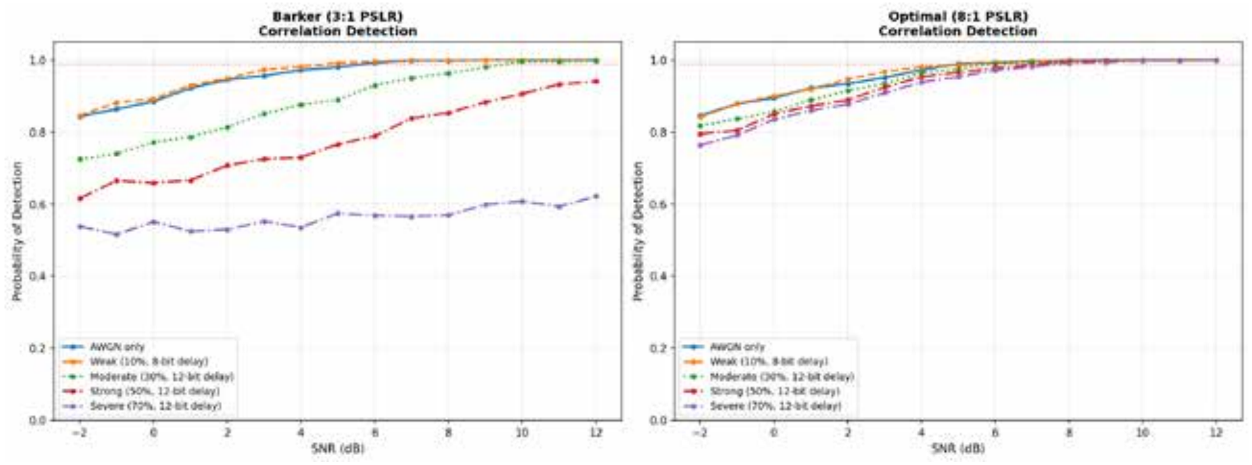
First, with multi-path conditions. Terrestrial features cause a particular type of interference where delayed copies of the signal arrive at the receiver. These echoes are from reflecting off of surfaces and taking longer paths than the line of sight transmission. Delayed copies of signals can destructively add and attenuate a signal. Worst case, they can largely cancel it out. VHF, UHF, and microwave communications systems quite often have to deal with multi-path, and many techniques have been developed to mitigate the damage. The better PSLR sync word, the better the multi-path performance. We show our optimal sync word compared to the concatenated Barker code using Hamming Distance sync detection. We vary the multi-path from none to severe.

Figure 2 hamming_detection_all_conditions.png



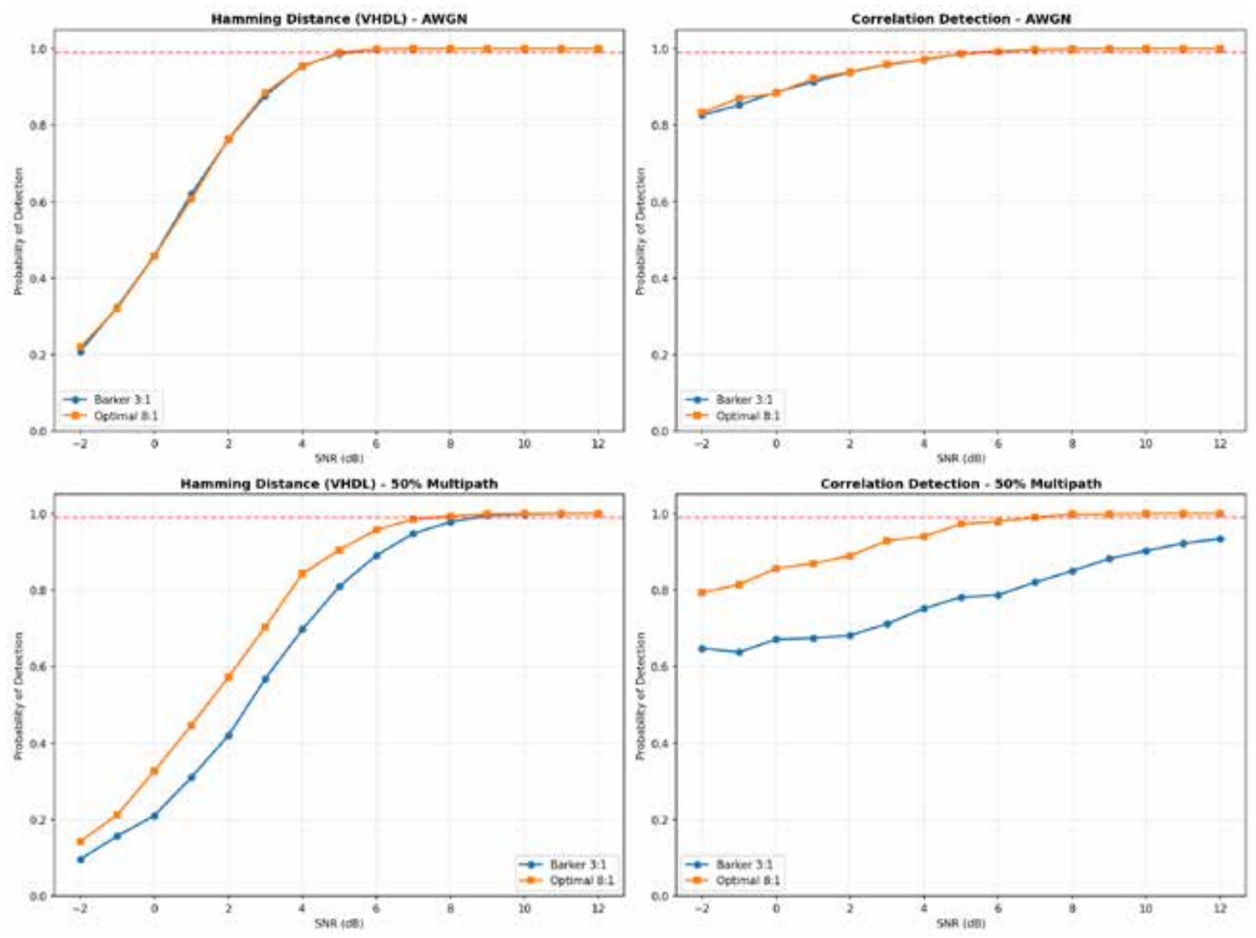
Second, if a correlator is used instead of a Hamming Distance calculator. If we use the more advanced correlation technique, then we see a performance improvement.

Figure 3 correlation_detection_all_conditions.png



Setting an optimized sync word in the protocol makes Opulent Voice future-proof. Multi-path performance significantly improves regardless of whether implementations use a Hamming Distance or a correlator to detect the sync word. If a correlator is used instead of a Hamming Distance calculation, then both classical concatenated Barker code and the optimal code have a significant performance improvement in additive white Gaussian noise (AWGN). The optimal sync word has a slight performance edge in very low SNR conditions. Higher PSLR doesn’t give us much traction against AWGN. We check against AWGN to make sure we don’t lose performance compared to other sequences. PSLR gives a substantial increase in performance when it comes to multi-path, and using a correlator improves both concatenated Barker and optimal sync word performance.

Figure 4 side_by_side_comparison.png



Lessons Learned

1. "Textbook Perfect" Isn't Always Practical

M-sequences and Zadoff-Chu are famous for their perfect autocorrelation properties, but this applies to *periodic* correlation in circular systems (like CDMA). For one-shot sync detection (aperiodic correlation), m-sequences offer no special advantage and are constrained to specific lengths, and Zadoff-Chu offers no special advantage and only works for complex samples. Concatenated Barker codes have drawbacks when it comes to multi-path.

2. Exhaustive Search is Underutilized

For moderate-length sequences, exhaustive search is entirely practical with modern computers. It guarantees finding the global optimum and eliminates guesswork about whether better sequences exist.

3. Classical Sequences Are Not Optimal for Arbitrary Lengths

While Barker codes are optimal for their specific lengths, extending them to arbitrary lengths through concatenation or truncation doesn’t mean they’re optimized for the new length. For custom lengths, dedicated optimization yields real improvements.

4. Bit Ordering Matters

Implementation must carefully match the bit ordering used during optimization. Our MSB-first protocol required specific code changes to ensure the hex value matched the transmitted sequence.

5. Symbol Rate Matters

Always analyze sync words at the level they were designed for. For binary PSK or MSK, analyze bits. For QPSK or 4-ary FSK, if the sync word is defined and transmitted at the symbol rate (dibits for P25), then analyze it at that symbol rate. For 8-ary PSK, if the sync word is defined at the symbol rate, then analyze tribits. And, so on, up the modulation order. If the sync word is defined at the demodulated bit level, then analyze it there.

Conclusion

This work demonstrates that exhaustive computational search can be done to provide optimized synchronization words that can outperform classical sequences in current and future designs. For the Opulent Voice protocol's 24-bit requirement, we achieved an 8:1 PSLR, which was substantially better than concatenated Barker codes and much better than truncated m-sequences.

The methodology was straightforward. We defined the search space (all 2^n sequences). We computed PSLR for each sequence. We selected from the list of all optimal sequences based on practical criteria, including spectral performance and prioritizing more frequent transitions over long runs of zeros or ones in the sequence in order to improve tracking loop behavior.

For sequence lengths up to 32 bits, this approach is entirely practical on modern computers and guarantees finding the global optimum. The resulting sequences provide significantly better performance than classical alternatives against multi-path and when using correlators in the receiver. Don’t assume arbitrarily picked or classical sequences are optimal. Run exhaustive searches, verify independently using different implementations, understand where you are going to see performance wins, document the bit ordering clearly, and test in hardware to confirm that the correlation properties transfer correctly.

The code for this analysis is available at https://github.com/OpenResearchInstitute/interlocutor/blob/main/OPV_sync_word_study.ipynb and can be adapted for other sequence lengths and design constraints. All images in this article are from this Jupyter notebook.

Acknowledgments

Thank you to the amateur radio digital voice community for feedback on sync word selection, and to Matthew Wishek, Paul Williamson, Anshul Makkar, Jonathan Naylor, Wally Ritchie (SK), and Frank Brickle (SK) for helpful discussions about Opulent Voice protocol work.

References

Barker, R.H. "Group Synchronization of Binary Digital Systems." *Communication Theory*, Academic Press, 1953.

Golomb, S.W., and Gong, G. "Signal Design for Good Correlation: For Wireless Communication, Cryptography, and Radar." Cambridge University Press, 2005.

Sarwate, D.V., and Pursley, M.B. "Crosscorrelation Properties of Pseudorandom and Related Sequences." *Proceedings of the IEEE*, vol. 68, no. 5, May 1980.

Skolnik, M.I. "Introduction to Radar Systems," 3rd Edition. McGraw-Hill, 2001. (Chapter on pulse compression and correlation is excellent)

Levanon, N., and Mozeson, E. "Radar Signals." Wiley-IEEE Press, 2004.

Ipatov, V. "Spread Spectrum and CDMA: Principles and Applications." Wiley, 2005.

Opulent Voice Protocol specification https://github.com/OpenResearchInstitute/interlocutor/blob/main/opulent_voice_protocol.md

TIA-102.BAAA-A, "Project 25 FDMA - Common Air Interface"

Daniels Electronics Ltd., "P25 Radio Systems Training Guide," TG-001, 2004

RadioReference.com P25 technical discussions and decoder analysis

APCO International, Project 25 Standards Documentation

About the Author

Michelle Thompson W5NYV@arrl.net is an electrical engineer and amateur radio operator specializing in digital communications. She holds an MSEE in Information Theory from USC and has contributed to a wide variety of open source digital radio projects and amateur radio organizations.

The Dwingeloo Radio Telescope and the Spirit of Amateur Science

by Michelle Thompson W5NYV

In a quiet corner of the northeastern Netherlands, near the edge of the Dwingelderveld National Park, stands a monument to an extraordinary moment in scientific history, and to an even more extraordinary amateur renaissance. The Dwingeloo Radio Observatory, completed in 1956 with a 25-meter dish, was briefly the largest radio telescope in the world. This silver spiderweb helped map the spiral arms of the Milky Way and discover previously hidden galaxies. But this is not simply a story about a telescope. It's a story about what happens when professional science ends and amateur passion begins.

The 1950s saw an explosion of radio astronomy. This was a time when we started seeing the true face of the universe. The Dwingeloo telescope began construction in 1954. When Queen Juliana inaugurated it on April 17, 1956, it represented one of the most advanced efforts to listen to space. Using the 21-cm hydrogen line, astronomers mapped the spiral structure of the Milky Way with increasing detail and discovered high-velocity clouds. These are rivers of gas streaming through space. The hydrogen line is an electromagnetic spectral line at 1420.4 MHz, and is used to map the most abundant element in the universe, hydrogen. The radiation comes from the electron in a hydrogen atom flipping to align with the spin of the proton, which releases a photon. In the 1990s, during a survey of galaxies hidden behind the dust of our own Milky Way, the telescope discovered a rather large galaxy, confirmed by infrared observations and named Dwingeloo I. And, it had a companion, Dwingeloo II.

By 2000, the telescope was no longer in operation. Professional radio astronomy had moved on to interferometers and arrays. The Dwingeloo dish, once the world's largest, stood silent. At over 100 tons and gathering nothing but rust, the future was very uncertain.

This is where many stories of obsolete infrastructure would end. But not ours.
In January 2007, a group of enthusiasts established the C.A. Muller Radio Astronomy Station foundation, known as CAMRAS. The C. A. stands for Cornelis Alexander Muller, one of the Dutch pioneers of radio astronomy. In June 2012, they physically removed the telescope dish for restoration. But CAMRAS didn't restore Dwingeloo to create a museum piece. They brought it back to active life. And, not just life for an insular few, but a place where people could learn and participate in science and technical education.

What makes the Dwingeloo revival remarkable isn't just technical. It's also cultural and philosophical. As we radio amateurs know, the word "amateur" comes from the Latin amator, meaning "lover". To be an amateur in a pursuit is to have a passion for it. The CAMRAS volunteers are radio amateurs, amateur astronomers, engineers, students, and experimenters who have transformed what was once the world's premier professional instrument into what might be called the world's premier amateur instrument.

Here are some recent achievements.

They've detected signals from Voyager 1, nearly 25 billion kilometers away in interstellar space, making Dwingeloo one of only a few telescopes in the world capable of receiving these incredibly faint signals. This achievement had previously been done in the amateur community at Bochum Observatory in Germany, a 20-meter amateur radio telescope with a deep bench of active volunteers. The Deep Space Exploration Society (DSES) in Colorado, USA, with their 18-meter dish, hopes to join this group in the near future.
In October 2021, a team including Thomas Telkamp, Jan van Muijlwijk, and Tammo Jan Dijkema bounced a LoRa IoT message off the Moon. This was the first time that this had been done with a small RF chip, setting a distance record of 730,360 kilometers (https://eandt.theiet.org/2021/11/24/european-scientists-bounce-first-lora-message-moon).

Dwingeloo detected their first Fast Radio Burst (FRB) in March 2024, joining the hunt to understand these mysterious cosmic flashes. What makes this particularly remarkable is that the receivers, built by radio amateurs, were not specifically optimized for receiving broadband signals like those from FRBs.

Dwingeloo achieved their first Venus bounce in March 2025. This involved transmitting radio signals that traveled 42 million kilometers to Venus and back. This was first done by amateurs at Bochum Observatory in 2008. DSES has Earth-Venus-Earth (EVE) plans here too. CAMRAS volunteer Cees Bassa directly supported and documented this work, confirming that the expected Doppler effects matched what was observed and recorded at Dwingeloo and at the collaborating site Astropeiler Stockert. Stockert is another significant amateur radio astronomy site and was involved in planning and executing the EVE 2025 event. Stockert is located in Germany and more information about them can be found at https://www.astropeiler.de/en/.

Radio astronomers, both amateur and professional, use the telescope for Earth-Moon-Earth (EME) experiments and communications. Dwingeloo also conducts pulsar observations, participates in Very Long Baseline Interferometry experiments, tracks spacecraft, and contributes to citizen science projects. The line between amateur and professional has become beautifully blurred.

CAMRAS regularly opens the telescope to visitors, youth, education, amateur astronomers, radio amateurs and other interested parties. What was once a temple of elite science has become a commons. This is a place where anyone with passion and curiosity can reach out and touch the universe. Recently, visitors from Open Research Institute (ORI) were fortunate enough to be able to spend the day at Dwingeloo. This is the story of what we were able to observe. Find out more about ORI at https://openresearch.institute.

Thomas Telkamp bridges the worlds of large-scale networking, amateur radio experimentation, and cutting-edge space technology. He's a pragmatic engineer and a dedicated and confident experimenter, with a career that spans internet backbone work to satellites and deep space. Not only highly competent technically, Thomas was a delightful and generous host for our day at Dwingeloo.

The day's adventure began in Utrecht, in a hotel overlooking the Utrecht Conservatory and a short walk from the Dom Tower. Built in the 1300s, the Dom tower is the symbol of the city of Utrecht, and sets both a historical and cultural tone for the region. Travel to Dwingeloo from downtown Utrecht was accomplished by a combination of bus and then car, but could also have been done by train. The rich structured greens, browns, and grays of the tree-lined urban streets quickly gave way to smoothly variegated greens of farmland and forests.

Arriving at the radio telescope feels a bit sudden. The gradually narrowing country lane bends around one last gentle curve and then ends in a small and tidy parking lot. At the top of the lot is a small house that was occupied in the past by an on-site caretaker. The building is still used at the site to provide a restroom and meeting place for visitors and operators. Much like many modern fire observation and ranger stations in the United States, the modern Dwingeloo site no longer needs a live-in caretaker. However, the infrastructure is still there. At the bottom of the lot, a short pathway leads directly to the telescope.

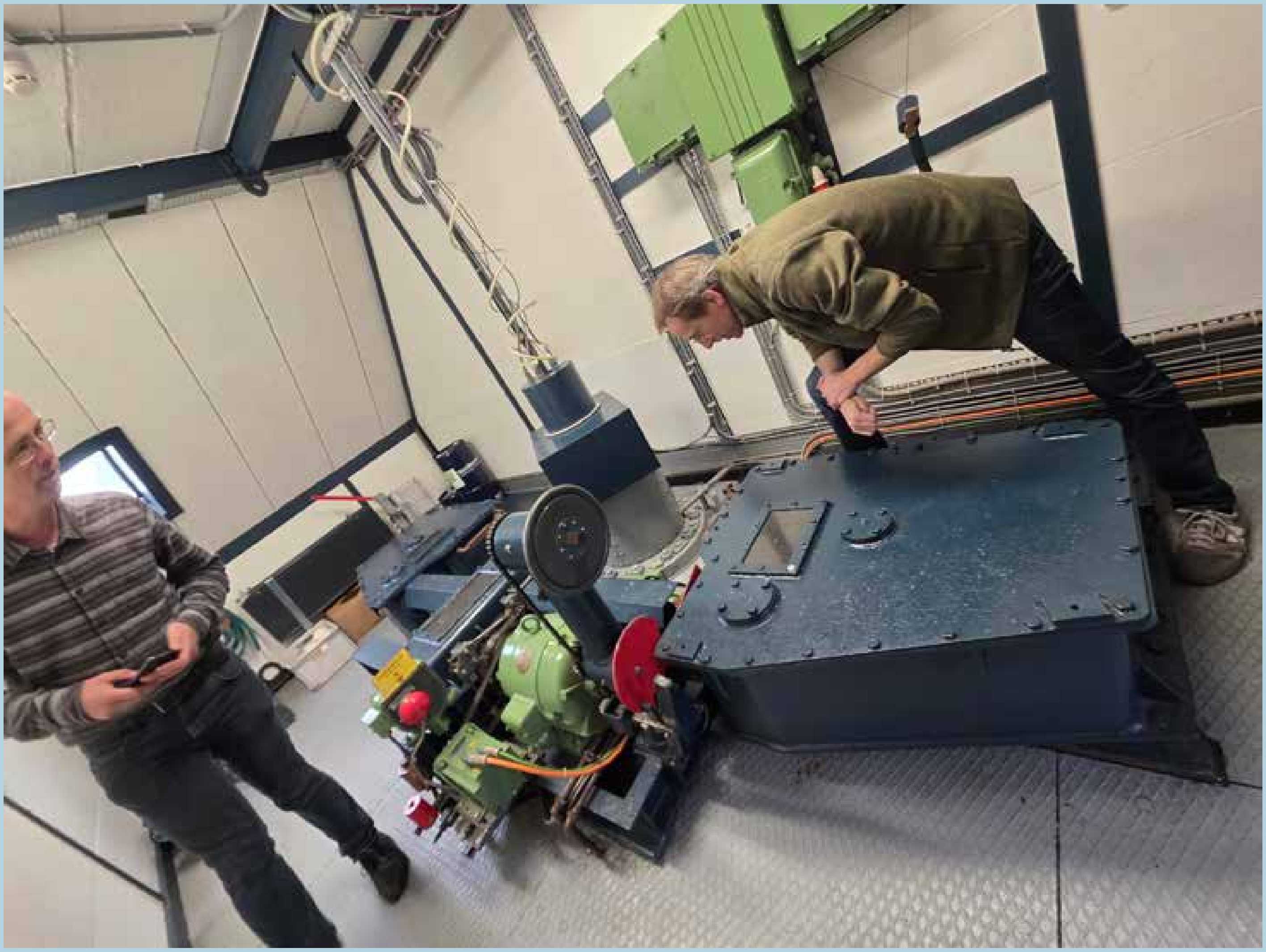
The 25-meter diameter dish support structure rests on a set of large metal wheels that run on a circular track. The telescope building, immediately below the dish, sits up above the track and rotates with the telescope. The motor room is inside this building, off the hallway that connects the front door to the control room.
With the wheels fully exposed at the outside perimeter of the structure, every startup requires a careful inspection and clearing of the track. At 90,000 kg (~200,000 lbs), the scope would easily crush anything on the track. For comparison, this is approximately equal to the weight of a diesel locomotive. In the photo below, note the brushes on either side of the wheel, to sweep away blockage or debris.

We walked around and inspected the track, and then climbed the rain-slick metal stairs to the front door of the telescope building. It felt like boarding a ship or a large airplane. There are enough windows in the control room to see that you are at least a story up off the ground. A skylight in the ceiling gives you a clear view of the mesh dish and feed above.

After setting down our backpacks and the radio gear Thomas brought for the day, the first task was to take off the brake and start up the motors. The engine room is small, with enough space for at most six or seven people to crowd around the motors and gears. There have been changes over the years to the configuration of the motors and gears, with the old infrastructure still bolted into place beside the new. Since the entire building moves with the scope, the power cables can slowly wrap around the stationary main support shaft. Worst case, too many rotations in the same direction could cause the cables to wind up tightly and break. The control room software keeps track of how far around the cables have wrapped and won't let the telescope keep rotating past the point of damage, but it's good to unwind in advance of any long maneuvers or rotations that might be planned for the day. Otherwise, you might be prevented from doing what you want by the software, and have to spend time to unwind the power and communications cables.

We were soon joined by another CAMRAS volunteer, Tammo Jan Dijkema, an engineer and mathematician with a keen interest in radio astronomy.

Below, Tammo at left and Thomas at right inspect the motor after startup. In the upper middle of the image is a vertical shaft that connects the ground to the building and telescope. Cables that are brought in to the building can wrap around this shaft. A monitoring system makes sure that the telescope does not rotate too far in either direction.



With all the setup completed, we moved to the main room of the building. Tammo took one of the operating positions in front of the equipment racks and immediately below a large overhead monitor. Thomas started setting up the radio telescope.

Tammo explained that when demonstrations are done for school field trips and other public open houses over the course of the year, the observations are recorded. These collections and stores of sampled data from the demonstrations allow Dwingeloo to plot Doppler differences of deep space targets. Over the course of a year, objects appear to move away or towards the Earth. These long-term longitudinal studies provided yet another valuable resource for amateur radio astronomy enthusiasts and experimenters.

When the scope slews, or moves quickly between observation settings, one can clearly feel and see the building move and the dish rise or fall. When it settles down to observe at the same rate of rotation as Earth, or at sidereal rate, the movement is much more subtle. One might only notice after a while that the view out the window has changed.

Thomas rewrote some of the software that handles connecting to the telescope and summarizes status and

results. This software is a modern and SDR-friendly update of professional software from the 1990s. There is a user interface of the dish health and safety information that runs on one of the resident computers. The concern on this particular day was high winds throughout the region. The evidence of the inclement weather had been all over the Utrecht city center, with leaves and limbs covering the sidewalks and at least one large branch coming down at daybreak in the middle of a cafe. The size of the Dwingeloo dish means it presents a large wind load, and above a certain wind speed, we would be limited in how we could operate and would have to settle for a much more conservative set of objectives. Any winds higher than the limit would cancel the session entirely. The dish health screen was mostly green indicators, but yellow flashed intermittently. We would have to keep a close eye on that. Fortunately, the software made monitoring dish health as easy as it could be.

Thomas explained how "back in the day" the foundational 21-cm hydrogen line survey was made in this very room. This survey revealed previously unknown Milky Way structures. Against the backdrop of the survey results on a large poster on the wall, we connected the Ettus B210 to the telescope. The B210 is an Ettus Research (National Instruments) software defined radio, and has excellent receiver performance. UHF and 13cm operations are done with a manual patch in the equipment rack. With a laptop, the radio data metadata format SigMF, and a modest amount of cabling, the entire universe can come to life with the B210 and a Python script.

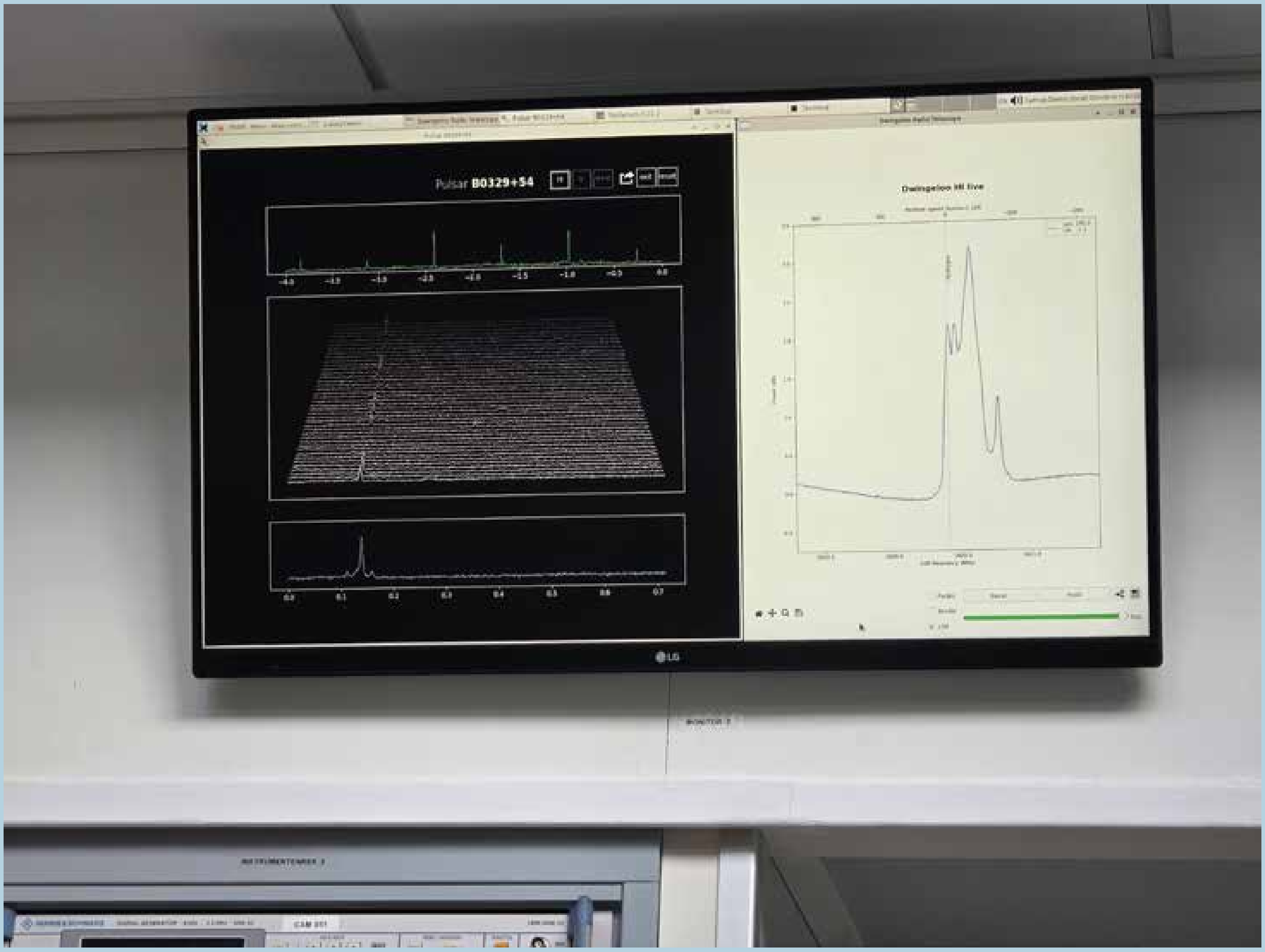
Thomas described efforts earlier in the year to find and receive the carrier signal from the Voyager spacecraft. This was a highlight for CAMRAS, as it's a challenging signal to receive. Another challenge? RFI. Sources of radio-frequency interference range from electric fences, which have been around for decades, to the relatively new "intruder" of electric bicycles, or e-bikes.

Observational activities at Dwingeloo come in categories. Natural objects and phenomena in the universe, human objects in the universe, radio amateur communications (EME, EVE), and educational outreach. Dwingeloo is a SATNOGS station, and regularly contributes observations of satellites to the database under the amateur radio call sign PI9CAM (PI9RD since November 2024).

Due to a happy travel coincidence, Paul Hearn and his wife Pam were able to join us. Paul and Pam were deeply involved with putting on the 2025 EU Conference on Amateur Radio Astronomy (EUCARA25), held just a few weeks prior. They described some behind-the-scenes work and the inspirational keynote from Professor Jocelyn Bell Burnell. She discovered pulsars in 1967, and pulsars were on the list of the day's targets.

We were then joined by Ed Dusschoten, the chair of CAMRAS. He dropped by to check in with Thomas and Tammo and meet Paul and Pam. His enthusiasm for both the site and the role were clear. He made sure the volunteers had everything they needed and shared key historical details with those of us visiting.

Our first observational target was the pulsar in Cassiopeia A. It is in the plane of the Milky Way. We are looking at it through two arms of our galaxy, which presents a complicated and interesting spectral response. Thomas explained the transmission and absorption lines on the spectral display, and how to identify the different arms of the galaxy.



Pulsars are believed to be rapidly rotating neutron stars, which are what stars become when they run out of fuel and collapse. Instead of being a fiery ball of gas, they are believed to be composed of neutrons. Pulsars have very strong magnetic fields which funnel jets of particles out along their two magnetic poles. When these poles happen to point towards Earth, then we can detect the energy as it sweeps by us like the light from a distant lighthouse. The period of pulsars ranges from 2 milliseconds up to many seconds, with the longest discovered so far rotating once every 6 hours.

The science of pulsars is more than enough justification for amateur and professional study. However, the influence of pulsars doesn't stop with science. "Le noir de l'Etoile" by Gérard Grisey, is a modern percussion music piece. It is inspired by the pulsar PSR 1919+21. Dwingeloo is one of the few sites where audio recordings of pulsars can be made and is actively involved with the art as well as the science of space.

Our next goal was to capture the so-called "Giant Pulses" from the Crab Nebula. Since RFI can also come through in these observations, it takes skill, patience, and quality computer programming to weed out the RFI and leave only the true pulses from PSR B0531+21, otherwise known as the Crab Nebula Pulsar. After some hand-editing of the Python script to better capture the 33ms period pulses, Thomas then explained an additional innovative feature of the way Dwingeloo is set up for amateur radio astronomy. Often, a digital radio signal stream of IQ data (in phase and quadrature complex baseband samples) are available to one instrument at a time. The consumer of the radio samples is usually a software program or another piece of hardware. At Dwingeloo, Thomas has modified the software infrastructure to allow multiple consumers of the IQ data stream. This was a big resource improvement over the way it was done before. This allows multiple people to use the data stream at the same time.

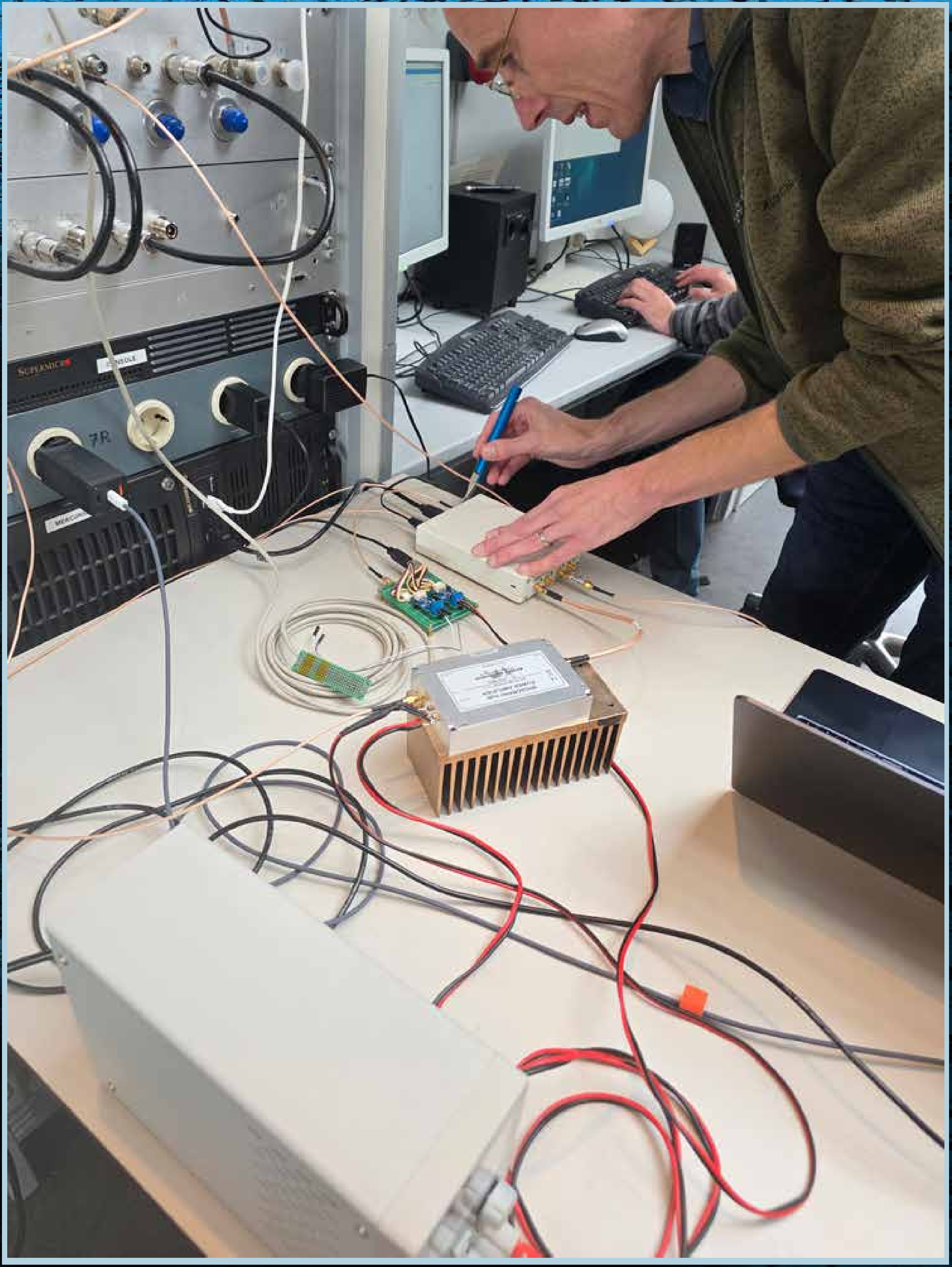
After everything was set up and confirmed to be working as expected, we watched the user interface on the big overhead monitor and waited. The "giant" pulses that we were looking for come every few minutes. We needed to capture and analyze them. If they were a clean spike, then they were probably local noise. If they had the expected and predicted amount of dispersion, then we considered them to be coming from the pulsar. After a while, a small collection of bursts at the right frequency and with the right dispersion showed up. Optimism grew and then became confidence. We'd successfully observed giant pulses from the Crab Nebula. These are radio signals from 6,500 years in the past. Signals produced today won't reach earth until the year 8525.

With Crab Nebula Pulsar bursts successfully observed as a warmup, our next target was something completely different. Instead of passively receiving energy from a naturally occurring phenomenon incredibly far away, we were going to receive energy reflected from an artificial satellite relatively close to us. The target was the radar target sphere LCS-1. This satellite is a hollow sphere of metal 1.12 meters in diameter and has been in space since 1965, making it one of our oldest spacecraft. The orbit isn't expected to decay for another 30,000 years.

The radar cross section of the sphere is very well known. This allows stations on the ground to measure the performance of their radio systems by using it as a reflector. With no extra circuits or transponders or translators or repeaters in the way, using a passive metal sphere to test a reflected signal takes a lot of the guesswork out of characterizing a complicated radio system like Dwingeloo.

The relatively quick motion of a LEO satellite crossing the sky, compared to the essentially stationary location of a deep space object like a pulsar, requires the antenna to move faster in order to track. The higher the speed at which the antenna is moving, then the less margin it has with respect to wind speed. Another limiting factor for Dwingeloo is that the dish can track a LEO orbit only up to about 40 degrees of elevation above the horizon. Above 40 degrees elevation, there are complications and physical limitations. After careful review of the dish health screen, which was still showing green, and the projected orbit, which was below 40 degrees maximum elevation, Thomas decided we would try for LCS-1. And, the attempt was successful. The characteristic return appeared in the spectral display.

Thomas took another hard look at the weather. Windy conditions were happening at both Dwingeloo and Stockert, the other amateur radio site involved in today’s EME attempt. Plans had been made to coordinate with Stockert on an EME bounce attempt with digital signal files. The wind was gusty but the forecast showed conditions improving. After some back and forth with Stockert, Thomas decided to move forward with the EME attempt. The B210 was now connected to a sequencer board, an amplifier, a power supply, and a massive

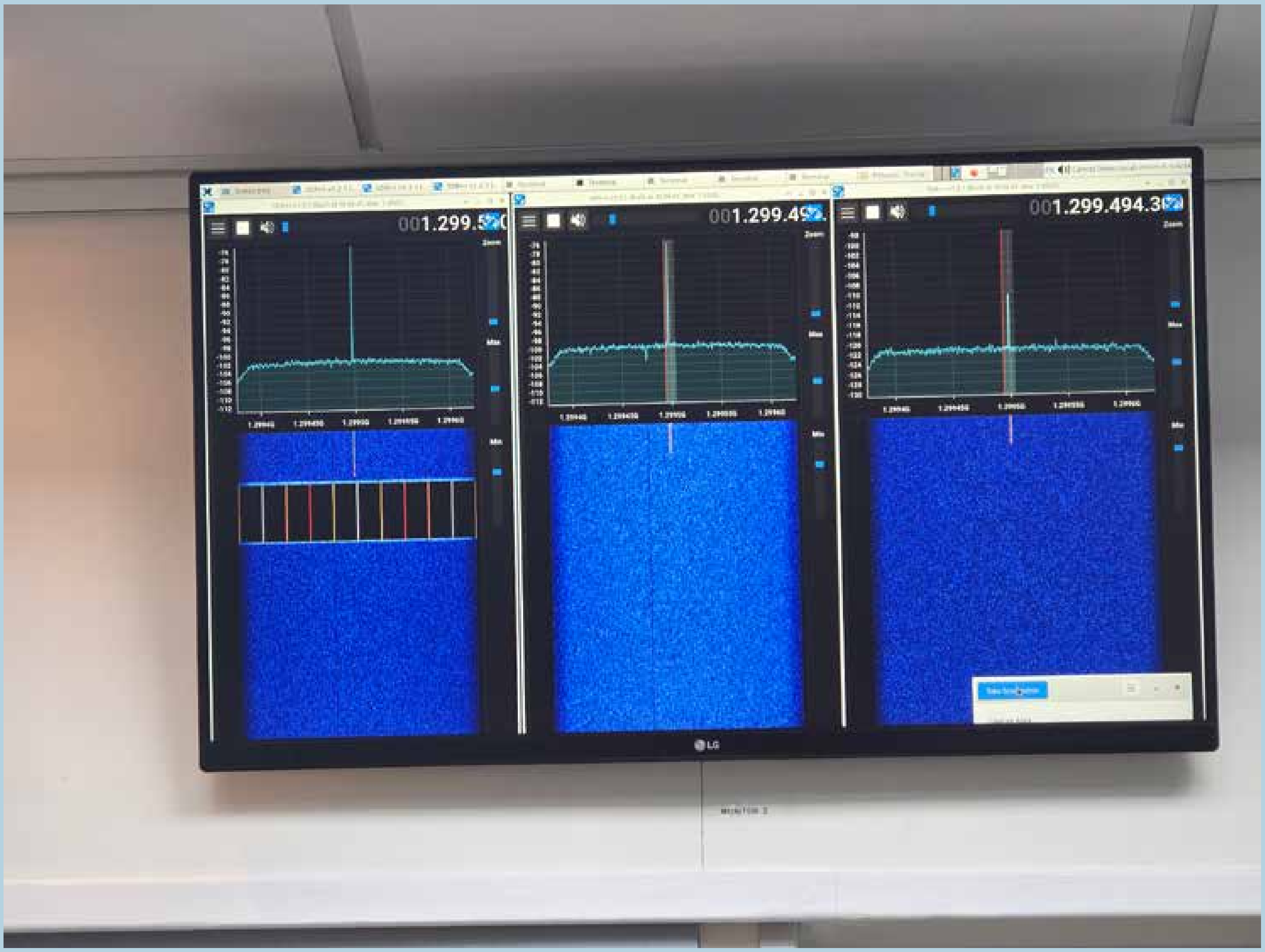


heat sink. The B210, like almost all SDRs, doesn't have enough transmit power on its own to drive radio instruments like the transmitter at Dwingeloo. It needs an amplifier to get into the 1-2 watt range to make things work.

In order to show the signals received from both sites at the same time at Dwingeloo, Thomas put the results up on the large overhead monitor. This showed a side-by-side display waterfall of the radio spectrum from both receivers. In order to get the receiver information from Stockert, Thomas logged in to Stockert over a virtual private network and then opened a secure shell tunnel back to Dwingeloo.

B210 transmit was tested and proven to work. The amplifier for the B210 was switched on. The Dwingeloo amplifier was then switched on. 10 sequences of 30 seconds were transmitted, and loud EME return signals were seen in the waterfall. The sequences sent to the Moon included Zadoff-Chu sequences. These coded transmissions are being constructed and studied as a possible signal type for EVE communication attempts coming up in October 2026. Using EME as a test bed for EVE lets experimenters like Thomas try out different digital signal processing techniques and signal structures without having to wait for Venus to approach close enough for an echo to be heard. Dwingeloo successfully received EVE carrier echoes in March 2025, during the most recent close approach of Venus. Stepping up from a carrier wave detection to communications reception will require advanced digital signal processing techniques and carefully coded signals. Experiments like today's EME bounce provide useful results and experience invaluable for an EVE attempt.

If coordinating two sites for live EME was impressive, then how about three?



Thomas expanded the EME experiment by logging into a citizen science accessible antenna at the Allen Telescope Array (https://www.seti.org/projects/ata/) located at Hat Creek Radio Observatory in Northern California. A third waterfall was configured in software and displayed on the overhead monitor. We now had Dwingeloo, Stockert, and ATA receivers simultaneously watching the Moon for a transmission of a digital amateur radio experiment from rural Netherlands. And, it worked. The return echo from all three sites was clearly seen and recorded at Dwingeloo. This proved that the communications and logistics links between three of the most significant amateur and citizen science radio astronomy sites in the world were working.

Closing down the telescope after so many successful observations was satisfying and a little bittersweet. The Dwingeloo story offers a template for how we might handle aging scientific infrastructure everywhere. Rather than abandoning instruments when professional science moves on, what if we recognized them as educational resources, as platforms for citizen science, as tools that can continue producing valuable data in capable amateur hands?

The volunteers at Dwingeloo have shown that with dedication, creativity, and collaborative spirit, amateurs can operate sophisticated instruments and make real contributions to our understanding of the universe. They've shown that the amateur spirit can be just as powerful a driver of discovery as professional funding and institutional support.

In the shadow of the Dutch pine forests, where a 25-meter dish still turns toward the cosmos, the future and the past of science meet. Professional expertise built Dwingeloo. Amateur passion brought it back to life. And somewhere between those two efforts lies a truth about human curiosity that no amount of rust can diminish.

HI Michelle,

The Sept/Oct 2025 QEX pp. 19-22 has a number of mistakes that might have crept into the article during the editorial process. As a lifelong DSP learner, and as a voracious book reader, I realize how difficult it is to write about DSP! I share as a DSP tutor who helps colleagues unwind things they read, or wrote, that were not quite true. Fire back on my mistakes too. That is how we learn!

(A) **QEX Page 19:** "Let's filter out the lower image, and transmit f_c, which is our modulated signal at the carrier frequency." This statement if false. The problem is that Euler's Identity for cosine expresses the two images you referenced [1]:

$$\alpha\, cos\left(2\pi f_c t\right) = \alpha\, \frac{e^{i2\pi f_c t} + e^{-i2\pi f_c t}}{2}$$

You cannot transmit only the upper image because of Euler's equation [2]. Your "upper image" is only:

$$\frac{\alpha}{2}\, e^{i2\pi f_c t} = \frac{\alpha}{2}\left(cos\left(2\pi f_c t\right) + i\, sin\left(2\pi f_c t\right)\right).$$

The problem is that this signal is <u>complex-valued</u> now. For example:

$$Re\left\{\frac{\alpha}{2}\, e^{i2\pi f_c t}\right\} = \frac{\alpha}{2}\, cos\left(2\pi f_c t\right)$$

$$Im\left\{\frac{\alpha}{2}\, e^{i2\pi f_c t}\right\} = \frac{\alpha}{2}\, sin\left(2\pi f_c t\right)$$

Since neither the real nor the imaginary part is zero, when you deleted your "lower image" you have produced a complex-valued signal. You cannot transmit this signal using a single antenna since it is complex-valued now!

(B) **QEX Page 20:** "Adding the I and Q signals together and transmitting them sends $I\, cos\left(2\pi f_c t\right) + Q\, sin\left(2\pi f_c t\right)$." That is unconventional. Although it may surprise many, most textbooks actually use this form instead [3,4]:

$$I\, cos\left(2\pi f_c t\right) - Q\, sin\left(2\pi f_c t\right).$$

Most books use a negative on the sinusoid component because it is the only form that leads to consistency with other DSP equations — *namely the Fourier transform!* As students learn DSP, eventually they encounter phase modulation where the I & Q values are set according to an angle θ:

$$I = cos\,(\theta) \quad \text{and} \quad Q = sin\,(\theta).$$

Most textbooks use the negative sinusoid since:

$$cos\,(\theta)\, cos\left(2\pi f_c t\right) - sin\,(\theta)\, sin\left(2\pi f_c t\right) = cos\left(2\pi f_c t + \theta\right).$$

One problem — *there are many* — of using the positive sinusoid is that:

$$cos\left(\theta\right)\,cos\left(2\pi f_c t\right) + sin\left(\theta\right)\,sin\left(2\pi f_c t\right) = cos\left(2\pi f_c t - \theta\right).$$ (Bad Convention!)

Students run into trouble with this form when they encounter frequency modulation. A positive change in phase with respect to time should shift the transmitted frequency up, and this is only possible if we use the negative sinusoid of the carrier.

The other problem with your form is inconsistency with Euler and Fourier. For instance, a phase modulated signal in most textbooks is formulated as:

$$Re\left\{\left(I + iQ\right)\left(cos\left(2\pi f_c t\right) + i\,sin\left(2\pi f_c t\right)\right)\right\} = I\,cos\left(2\pi f_c t\right) - Q\,sin\left(2\pi f_c t\right)$$

Notice the negative sign on the right hand side! Now, you will find a few popular textbooks that get this wrong, but we know from algebra that:

$$i^2 = -1.$$

This makes the carrier sinusoid negative in fact! The negative is essential for using all of the famous math results to make the work easier.

(C) **QEX Page 20:** "What happens when we integrate a cosine signal from 0 to T? That value happens to be zero!" This is not true because you stated "the value that we are transmitting is held for a period of time called T" on QEX page 19. Saying this leads to problems for students understanding the DTFT and FFT later. In fact, if we clarify the term T as T$_{sym}$ — *avoiding any ambiguity of reference to a period of the cosine wave* — we have:

$$\int_0^{T_{sym}} cos\left(2\pi\left(2f_c\right)t\right)dt = \frac{sin\left(2\pi\left(2f_c\right)T_{sym}\right)}{4\pi f_c}$$

That is clearly not zero for just any old value of T$_{sym}$. To see this graphically, check out Figure 3-2 in my textbook that hopefully makes it easier [5]. I suggest qualifying the integration period if you present this material in the future.

(D) **QEX Page 22:** "However, using this complex modulation scheme gives us yet another advantage. Because of the math we just did, we eliminate an entire image when compared to a single carrier real signal." This is not true. Your form equals:

$$I\,cos\left(2\pi f_c t\right) + Q\,sin\left(2\pi f_c t\right)$$

Suppose you sent only a "single carrier real signal" and expand using Euler's Identity for cosine:

$$I\,cos\left(2\pi f_c t\right) = I\,\frac{e^{i2\pi f_c t} + e^{-i2\pi f_c t}}{2}$$

Now, suppose you instead send your entire signal and further expand using Euler's Identity for sine:

$$I\,cos\left(2\pi f_c t\right) + Q\,sin\left(2\pi f_c t\right) = I\,\frac{e^{i2\pi f_c t} + e^{-i2\pi f_c t}}{2} + Q\,\frac{e^{i2\pi f_c t} - e^{-i2\pi f_c t}}{2i}$$

Gather terms to see there is no cancellation of images assured by your operations:

$$I\,cos\left(2\pi f_c t\right) + Q\,sin\left(2\pi f_c t\right) = \left(\frac{I}{2} + \frac{Q}{2i}\right) e^{i2\pi f_c t} + \left(\frac{I}{2} - \frac{Q}{2i}\right) e^{-i2\pi f_c t}$$

From this form, arbitrary values for I & Q do not change the number of images versus your other case.

I am sure I made a few mistakes too, but I wanted to share my insights since it seemed a few things crept into an odd place in this article.

-Pete

References
[1]. T. Needham, Visual Complex Analysis, Oxford University Press, New York, 1997, p. 14.
[2]. T. Needham, Visual Complex Analysis, Oxford University Press, New York, 1997, p. 11-12.
[3]. J.G. Proakis, Digital Communications, McGraw-Hill, Boston, 1995, Equation 4-1-12, p. 155.
[4]. R.E. Blahut, Modem Theory, Cambridge University Press, Cambridge, 2010, p. 131.
[5]. P. Wyckoff, Visualizing Signal Processing with Complex Values, KDP, 2023, p. 39.

Response to Pete's Technical Critique

Hi Pete,

Thank you so much for taking the time to provide such detailed technical feedback! I really appreciate your careful reading. You've raised some important points that deserve thoughtful responses. Let me try and address each one.

Filtering Out the Lower Image and Complex-Valued Signals

I was imprecise here. My statement "Let's filter out the lower image, and transmit fc" could be misleading in the context I presented it.

What I was trying to convey to a general audience is the conceptual process that happens in single-sideband modulation when we move to complex baseband representation. What you are saying is that by placing this immediately after showing Euler's identity for a real cosine, I created confusion about what's actually transmittable.

You're correct that if we literally kept only the positive frequency component (α/2)e^(i2πfct), we'd have a complex-valued signal that cannot be transmitted with a single real antenna. In the article's pedagogical flow, I should have either

1. Been more explicit that this is a conceptual stepping stone to understanding quadrature modulation, or
2. Maintained the real signal throughout this section and introduced the complex representation only when discussing I/Q modulation

Your point is well-taken, and this section could be clarified for technical accuracy.

Sign Convention: I·cos(2πfct) + Q·sin(2πfct) vs. I·cos(2πfct) - Q·sin(2πfct)

This is a really interesting point about convention. You're right that most DSP textbooks use the negative sign on the sinusoid: I·cos(2πfct) - Q·sin(2πfct), and you've identified the key reason. It is consistency with Fourier transform conventions and phase modulation.

For the article's target audience (QEX readers who are typically radio amateurs with varying levels of DSP background), I made a deliberate choice to use the positive sinusoid form because it's simpler to explain without being technically wrong.

The positive form I·cos + Q·sin maps more directly to the "cosine on I-axis, sine on Q-axis" geometric intuition that I was building. There is also a hardware convention. The RF engineering texts and datasheets that I have used for I/Q modulators use the positive convention much more often than negative. For readers less comfortable with complex notation, introducing the negative sign requires explaining that it ultimately comes from the complex number properties. This has been a turn-off for people in the past. I did not want it to be a turn off for QEX.

That said, you're absolutely correct that the negative convention gives cos(2πfct + θ) for phase modulation (standard trig identity). The positive convention gives cos(2πfct - θ), which is non-standard. For frequency modulation, the derivative relationship works correctly only with the negative convention. The negative form aligns with Re{(I + iQ)e^(i2πfct)}

In retrospect I could have used the standard negative convention from the start, or explicitly acknowledged the existence of both conventions and explained why I chose one over the other. Since this is the identical treatment that I got from my grad school notes, I'm not the only person to explain it this way. It is, however, a compromise.

Integration of Cosine from 0 to T Being Zero

This is a significant oversimplification on my part. You're absolutely right that ∫₀^Tsym cos(2π(2fc)t)dt = sin(2π(2fc)Tsym)/(4πfc), which is generally not zero.

What I meant to convey (but obviously failed to state clearly) is that when we integrate cos(2π(2fc)t) over many symbol periods, or when the symbol period Tsym is chosen to be an integer multiple of the carrier period, the integral approaches zero. In practical systems, the low-pass filter following the mixer rejects the 2fc components.

I should have written something like: "We use a low-pass filter to remove the 2fc terms, leaving us with the DC component from the integration" rather than claiming the integral itself is zero.

This is an important distinction, especially for readers who might try to work through the math themselves. The pedagogical shortcut I took could definitely cause confusion when students try to verify the math or understand DTFT/FFT concepts later. Thank you for the reference to Figure 3-2 in your textbook (which I have). I am sure that many people will take a look.

Image Elimination

You've identified another important imprecision. When I wrote "we eliminate an entire image," I was trying to express that quadrature modulation produces a single-sideband spectrum rather than the double-sideband spectrum of a real carrier. I wanted to stop there because this is something useful in practical radio designs.

Your math clearly shows that the real carrier: I·cos(2πfct) produces symmetric images at ±fc and the quadrature: I·cos(2πfct) + Q·sin(2πfct) = (I/2 + Q/2i)e^(i2πfct) + (I/2 - Q/2i)e^(-i2πfct)

You're right that this doesn't automatically or magically eliminate an image. Both positive and negative frequency components exist. However, the key advantage is that with independent control of I and Q, we can create single-sideband modulation where the negative frequency component can be zeroed out.

I should have been more precise: quadrature modulation *enables* single-sideband transmission through appropriate choice of I and Q, rather than automatically eliminating an image.

Feedback like this highlights the challenge of writing about DSP for a mixed audience. Trying to build intuition without sacrificing technical accuracy. I absolutely favor paths towards intuitive explanation I absolutely can and do create technical errors in the process. Being more rigorous with the mathematics, and/or explicitlly noting where there's a simplication, is something I will take to heart in any future submissions.

I really appreciate you taking the time to work through these details. Our tradition at ORI is to welcome comment and critique. Thank you!

-Michelle Thompson

FunCube+ Mode Dynamic Transponder Analysis and Simulation Model for AMSAT-UK

The proposed "fast uplink, slow downlink" dynamic transponder concept for AMSAT-UK's upcoming FunCube+ 2U CubeSat represents an interesting power-conservation strategy, but requires careful analysis of buffering requirements, timing constraints, and mode compatibility. Open Research Institute (ORI) was invited by AMSAT-UK to support and propose designs for an upcoming launch opportunity. This article is the first in a series from ORI about this collaboration.

Challenges

Buffer size requirements need to be calculated and tested. We must store potentially entire passes worth of data. We have latency challenges. Communication becomes store-and-forward rather than real-time. FPGA complexity is mainly in the buffering and control logic. We need to define the user experience. We need to define the Doppler on both fast receive and slow transmit.

Mode Analysis for FunCube+

Looking at the operating context of FunCube-1 (AO-73) we have an uplink of 435.150-435.130 MHz (Mode U, 20 kHz bandwidth, LSB) and a downlink of 145.950-145.970 MHz (Mode V, USB). This is a traditional transponder with an inverting linear at 300 mW PEP. Telemetry is 145.935 MHz BPSK, 30/300 mW. What are some reasonable options for FunCube+?

1) Digital Voice, Data, and Keyboard Mode

Fast Uplink of 435.xxx MHz with 81 kHz bandwidth Opulent Voice protocol. Slow Downlink of 145.xxx MHz with a 20 kHz bandwidth BPSK data stream. This is the current bandwidth limit. Bandwidth is traded off against latency and satellite power budget for up to ~4:1 compression.

Open source implementation for FPGA can be found at https://github.com/OpenResearchInstitute/pluto_msk. Advantages include clear quantifiable data rates for buffering calculations and high fidelity voice integrated with keyboard chat, data, and control messages.

2) Digital Data Mode or Pure Store and Forward

Fast Uplink at 435.xxx MHz with FSK/PSK at 9600 bps. Mode such as G3RUH. Slow Downlink at 145.xxx MHz with BPSK at 1200 bps. The optimization of the rates would need to be optimized and analyzed, but we could achieve an 8:1 radio of data rate compression.

This is relatively easy to implement in an FPGA. There are clear quantifiable data rates for buffering calculations. There is potential compatibility with FUNcube Dashboard software ecosystem. This would be suitable for telemetry and educational data collection.

```
┌─────────────────────────────────────────────────────────┐
│                  Ground Station Domain                  │
│  - High Power Transmitter                               │
│  - Fast uplink mode                                     │
│  - Receives slow downlink mode                          │
└─────────────────────────────────────────────────────────┘
                          Fast UP            │
                            │          Slow DOWN
┌─────────────────────────────────────────────────────────┐
│              Satellite Transponder Domain               │
│  ┌───────────────────────────────────────────────────┐  │
│  │ RF Frontend (Mode U/V or S-band)                  │  │
│  └───────────────────────────────────────────────────┘  │
│                           │                             │
│  ┌───────────────────────────────────────────────────┐  │
│  │ Demodulator (Fast Rate)                           │  │
│  │ - Doppler compensation                            │  │
│  │ - Symbol timing recovery                          │  │
│  └───────────────────────────────────────────────────┘  │
│                           │                             │
│  ┌───────────────────────────────────────────────────┐  │
│  │ FPGA Processing Domain                            │  │
│  │  ┌─────────────────────────────────────────────┐  │  │
│  │  │ Rate Buffer (FIFO)                          │  │  │
│  │  │ - Elastic storage                           │  │  │
│  │  │ - Flow control                              │  │  │
│  │  └─────────────────────────────────────────────┘  │  │
│  │                                                   │  │
│  │  ┌─────────────────────────────────────────────┐  │  │
│  │  │ Power Management Controller                 │  │  │
│  │  │ - Battery state monitoring                  │  │  │
│  │  │ - Transmit duty cycle control               │  │  │
│  │  └─────────────────────────────────────────────┘  │  │
│  └───────────────────────────────────────────────────┘  │
│                           │                             │
│  ┌───────────────────────────────────────────────────┐  │
│  │ Modulator (Slow Rate)                             │  │
│  └───────────────────────────────────────────────────┘  │
│                           │                             │
│  ┌───────────────────────────────────────────────────┐  │
│  │ RF Transmitter (Lower power)                      │  │
│  └───────────────────────────────────────────────────┘  │
└─────────────────────────────────────────────────────────┘
```

Jupyter Lab Notebook Python Classes

Link and design analysis was carried out in a Jupyter Lab Notebook in Python. Here is an explanation of the code and the results.

Python imports

We import the libraries that we need to run our code. These are helpful functions for math, plots, code organization, and variable enumeration.

```
In [14]: import numpy as np
         import matplotlib.pyplot as plt
         from dataclasses import dataclass
         from enum import Enum
```

TransponderConfig Dataclass

A dataclass is a object oriented structure. A dataclass is a collection of members (variables or constants). A dataclass is composed only of members. It does not contain any function definitions. This is how we configure our uplink and downlink data rates, buffer sizes, pass duration, radio parameters, battery and solar capacities, and other values. We can have a separate dataclass for different configurations. If we want to change a configuration, or try out a different proposal, then we can simply import a different dataclass that corresponds to a different design. Keeping the data separate from the calculations helps make our analysis flexible and results in easier and better documentation.

```
In [15]: @dataclass
         class TransponderConfig:
             """Configuration parameters for the dynamic transponder"""
             uplink_rate_bps: int = 9600
             downlink_rate_bps: int = 1200
             buffer_size_bytes: int = 65536
             pass_duration_sec: float = 600
             tx_power_watts: float = 0.3
             rx_power_watts: float = 0.1
             battery_capacity_wh: float = 10
             solar_power_watts: float = 4
```

TransponderMode Class

This class sets up an *enumeration* of transponder modes. We can be in receiving, buffering, transmitting, or idle state. When we call TransponderMode(number) then it returns the mode corresponding to that number, as defined inside the class.

```
In [16]: class TransponderMode(Enum):
             RECEIVING = 1
             BUFFERING = 2
```

```
    TRANSMITTING = 3
    IDLE = 4
```

DynamicTransponder Class

At the heart of the simulation are the DynamicTransponder Class methods (functions). This class contains a set of methods that we are going to be using in our mode analysis. The initialization function is listed first. This function is called whenever we create a DynamicTransponder Object. A TransponderConfig object is requried in order to creat a DynamicTransponder object. When we run an analysis, we create a TransponderConfig object first, then we use that list of values to set up a Transponder object.

Separating configuration clases from the transponder class gives us flexibility and clarity. We can look at the values in the configuration file without having to search through a lengthy class full of function defintions for variables. We can have multiple configurations and then simply re-use the DynamicTransponder class with different configurations. This division of labor is very common in scientific Python programming and is a naturally good fit for the analysis of radio systems.

We get many of the members of the DynamicTransponder class by importing all of the members from the TransponderConfig dataclass. This is our "starter set" of values. Each value from the Configuration dataclass is then available in TransponderConfig as self.config.name-of-variable-in-TransponderConfig-dataclass.

Right after that, we also set up additional members for the Transponder object, such as self.buffer_fill_bytes and self.time_sec.

In [17]:
```
class DynamicTransponder:
    def __init__(self, config: TransponderConfig):
        self.config = config
        self.buffer_fill_bytes = 0
        self.mode = TransponderMode.IDLE
        self.battery_level_wh = config.battery_capacity_wh
        self.time_sec = 0

        self.rx_data_total_bytes = 0
        self.tx_data_total_bytes = 0
        self.buffer_history = []
        self.battery_history = []
        self.mode_history = []
    def receive_uplink(self, duration_sec: float, duty_cycle: float = 0.5):
        data_bytes = int(self.config.uplink_rate_bps / 8 * duration_sec * duty_

        if self.buffer_fill_bytes + data_bytes > self.config.buffer_size_bytes:
            overflow = (self.buffer_fill_bytes + data_bytes - self.config.buffe
            data_bytes -= overflow
            print(f"⚠️  Buffer overflow! Lost {overflow} bytes")

        self.buffer_fill_bytes += data_bytes
        self.rx_data_total_bytes += data_bytes
        power_consumed = self.config.rx_power_watts * duration_sec / 3600
        self.battery_level_wh -= power_consumed

        self.mode = TransponderMode.RECEIVING
    def transmit_downlink(self, duration_sec: float):
        data_bytes = int(self.config.downlink_rate_bps / 8 * duration_sec)

        if data_bytes > self.buffer_fill_bytes:
            data_bytes = self.buffer_fill_bytes

        self.buffer_fill_bytes -= data_bytes
        self.tx_data_total_bytes += data_bytes

        power_consumed = self.config.tx_power_watts * duration_sec / 3600
        self.battery_level_wh -= power_consumed

        self.mode = TransponderMode.TRANSMITTING
    def solar_charge(self, duration_sec: float, illumination: float = 1.0):
        power_gained = (self.config.solar_power_watts * illumination * duratio
        self.battery_level_wh = min(self.battery_level_wh + power_gained, self.
    def log_state(self):
        self.buffer_history.append(self.buffer_fill_bytes)
        self.battery_history.append(self.battery_level_wh)
        self.mode_history.append(self.mode.value)
    def simulate_pass(self, pass_duration_sec: float = 600):
        timestep = 10

        for t in np.arange(0, pass_duration_sec, timestep):
            self.time_sec = t

            if t < pass_duration_sec / 2:
                self.receive_uplink(timestep, duty_cycle=0.6)
                self.solar_charge(timestep, illumination=0.5)
            else:
                if self.buffer_fill_bytes > 0:
                    self.transmit_downlink(timestep)
                else:
                    self.mode = TransponderMode.IDLE
                self.solar_charge(timestep, illumination=0.5)

            self.log_state()

        return self.get_statistics()

    def get_statistics(self):
        compression_ratio = self.config.uplink_rate_bps / self.config.downlink_
        buffer_max_utilization = max(self.buffer_history) / self.config.buffer_

        return {
            'compression_ratio': compression_ratio,
            'rx_total_kb': self.rx_data_total_bytes / 1024,
            'tx_total_kb': self.tx_data_total_bytes / 1024,
            'buffer_max_util_%': buffer_max_utilization,
            'battery_end_wh': self.battery_level_wh,
            'battery_used_wh': self.config.battery_capacity_wh - self.battery_l
        }
```

receive_uplink Class Method

The receive_uplink method simulates how the satellite receives data from ground stations. It calculates how many bytes of data arrive during a given time period, checks if there's room in the buffer to store them, and keeps track of power consumption. If the buffer fills up, excess data is lost. When we lose data, we call it an overflow.

The inputs, or arguments, to receive_uplink are the DynamicTransponder object itself (called "self"), a duration in seconds, and a duty cycle.

The first thing we do is find out how many bytes were transmitted. Ground stations transmit at a fixed bit rate. For example, 9600 bits per second. This is defined in the DynamicTransponder object, which we know as "self". The transmitted bit rate is one of those values that was passed in to the DynamicTransponder object from the Configuration dataclass. Specifically, our bit rate is self.config.uplink_rate_bps. The receive_uplink method calculates how many bytes arrive at the satellite during the specified duration, accounting for a duty cycle, which is the percentage of time the ground station is actually transmitting.

The second thing we do is check the available buffer space and add the incoming data. If, after we add the incoming data to the buffer, the result is larger than the size of the buger, then we have an overflow conditions. We report how many bytes were lost. The received data is stored in this buffer until it is retransmitted. We update the number of bytes in our buffer and we update hte number of total bytes that have been transmitted to the satellite.

Receiving consumes power from the satellite's battery. The method then deducts energy based on receiver power draw and how long it took to receive.

We then use our enumeration class to set our mode to RECEIVING.

transmit_downlink Class Method

The transmit_downlink method simulates how the satellite transmits data back down to ground stations. It calculates how many bytes can be sent during a given time period at the slower downlink rate, removes that data from the buffer, and tracks the power consumption of the transmitter.

The satellite transmits at a slower bit rate than it receives. For example, we might have a 20,000 bps downlink vs 54,200 bps uplink. Once we know how many bytes we're transmitting on the downlink, we can update the buffer. The method checks available buffer contents and transmits only what's available. We can't send what we don't have. We update our power consumption. Transmitting consumes more power than receiving, typically. The method deducts energy based on transmitter power draw and transmission duration. This allows us to trade buffer space for power efficiency in order to optimize uplink and downlink bit rates. We then set the state to TRANSMITTING.

solar_charge Class Method

solar_charge calculates the power we get from sunlight on our solar cells. We consume power with receiving and transmitting, and we gain power from solar charging.

The solar_charge method simulates solar panel energy collection. It calculates how much power the panels generate during a time period, scaled by an 'illumination' factor (0.0 = total darkness, 1.0 = full sun). The battery is recharged up to its maximum capacity, but never beyond.

This is a very simple model. There are no orbital mechanics, no eclipse calculations, no Earth shadow geometry, no sun angle (would result in cosine losses), no seasonal variations, and no panel degradations.

So, how can we use this method effectively?

We can calculate illumination based on orbital position.

```
*Example 1: Simple LEO orbit approximation*

orbit_period_min = 97  # ~500 km altitude
eclipse_fraction = 0.35  # ~35% of orbit in shadow

for t in timesteps:
    # Determine if in eclipse based on orbit position
    orbit_phase = (t % (orbit_period_min * 60)) /
(orbit_period_min * 60)

    if orbit_phase < eclipse_fraction:
        # In Earth's shadow
        transponder.solar_charge(timestep, illumination=0.0)
    else:
        # In sunlight
        transponder.solar_charge(timestep, illumination=1.0)
```

We can use an average illumination over many orbits. This is a rough estimate, but gets us in the ball park.

```
*Example 2: Average the Illumination and use that as the
Illumination Parameter*

# LEO satellite: ~35% eclipse, ~65% sun
average_illumination = 0.65

# But also account for sun angle (not always perpendicular)
# Add cosine losses and panel orientation
effective_illumination = 0.65 * 0.7  # ~45% effective
```

```
transponder.solar_charge(duration_sec=600, illumination=0.45)
```

log_state Class Method

log_state updates our buffer, battery, and mode histories. When we append, we add the most recent value to the list. This creates a record over time of the analysis so that we can find patterns, bottlenecks, saturations, or failed optimizatinos. It simply takes the current number of bytes in the buffer, current battery level, and current mode state and appends them to the logs for each value. The logs are used to make visualizations.

simulate_pass Class Method

This is what is known as an "orchestrator method". It's the function call that brings everything together. Many of the other methods are called by this method in order to simulate a pass over a ground station.

simulate_pass divides the pass into small time steps and alternates between receiving data during the first half (uplink) and transmitting during the second half (downlink), while solar panels charge throughout. The default 600 second pass is divided into 10 time steps. We check if we're in the first half of the pass duration. If we are, then we run the receive_uplink and solar_charge methods. If we are in the second half of the pass, we check if we have anything in the buffer than then run the transmit_downlink method. We also run solar_charge method as well. We log_state and then return a printout of the statistics of the pass by calling get_statistics. We'll talk about that method next.

get_statistics Class Method

Here is where we report the results of our analysis. We're interested in something we're calling compression_ratio. This is the ratio between the uplink bit rate and the downlink bit rate. We also report the maximum utilization of the buffer. Higher uplink data rates fill it faster. Higher downlink data rates drain it faster. Higher bit rates cost more power. Finally, we report the ending battery power level and how much power was consumed.

Finding the right balance for the MDT means understanding the trade-offs between bit rate and power consumption for this particular payload. Once we understand the minimums and maximums of the aspects under our control, we can develop a set of optimized numbers for the design. We can also begin to explore an adaptive design. For example, we might lower the downlink bit rate when the battery has lower capacity. Or, we might set it to a particular mode during a particular time period, such as an Experimental Day or a Special Event.

Jupyter Lab Notebook Python Analysis

Now that we've defined our members and methods and organized them into classes, we set up and run an MDT analysis.

Run Simulation

First, we create a TransponderConfig object by calling the dataclass. It doesn't take any arguments. It returns an object that is composed of the list of all the values we assiged as configuration variables for a Transponder design.

Second, we create a transponder object by calling DynamicTransponder. It needs a TransponderConfig object, and we provide the one we just made.

Finally, we call our transponder method simulate_pass. We set the pass duration as 600 seconds. We collect the dictionary of results returned by get_statistics in a variable called stats. A dictionary is a data structure that stores the value in key-value pairs. We fetch the value by calling out the key. This lets us share the results to both humans and other Python coding structures more easily that if we were sharing a bunch of variable names.

In [18]:
```
# Run simulation
config = TransponderConfig()
transponder = DynamicTransponder(config)
stats = transponder.simulate_pass(pass_duration_sec=600)

print("=" * 60)
print("FUNCUBE+ DYNAMIC TRANSPONDER SIMULATION RESULTS")
print("=" * 60)
print(f"Compression Ratio:      {stats['compression_ratio']:.1f}:1")
print(f"Data Received:          {stats['rx_total_kb']:.2f} KB")
print(f"Data Transmitted:       {stats['tx_total_kb']:.2f} KB")
print(f"Buffer Peak Usage:      {stats['buffer_max_util_%']:.1f}%")
print(f"Battery Used:           {stats['battery_used_wh']:.3f} Wh")
print(f"Battery Remaining:      {stats['battery_end_wh']:.2f} Wh")
print("=" * 60)
```

```
⚠️  Buffer overflow! Lost 6464 bytes
⚠️  Buffer overflow! Lost 7200 bytes
⚠️  Buffer overflow! Lost 7200 bytes
⚠️  Buffer overflow! Lost 7200 bytes
⚠️  Buffer overflow! Lost 7200 bytes
⚠️  Buffer overflow! Lost 7200 bytes
⚠️  Buffer overflow! Lost 7200 bytes
⚠️  Buffer overflow! Lost 7200 bytes
⚠️  Buffer overflow! Lost 7200 bytes
⚠️  Buffer overflow! Lost 7200 bytes
⚠️  Buffer overflow! Lost 7200 bytes
⚠️  Buffer overflow! Lost 7200 bytes
⚠️  Buffer overflow! Lost 7200 bytes
⚠️  Buffer overflow! Lost 7200 bytes
⚠️  Buffer overflow! Lost 7200 bytes
⚠️  Buffer overflow! Lost 7200 bytes
⚠️  Buffer overflow! Lost 7200 bytes
⚠️  Buffer overflow! Lost 7200 bytes
⚠️  Buffer overflow! Lost 7200 bytes
⚠️  Buffer overflow! Lost 7200 bytes
⚠️  Buffer overflow! Lost 7200 bytes
============================================================
FUNCUBE+ DYNAMIC TRANSPONDER SIMULATION RESULTS
============================================================
Compression Ratio:      8.0:1
Data Received:          64.00 KB
Data Transmitted:       43.95 KB
Buffer Peak Usage:      100.0%
Battery Used:           0.000 Wh
Battery Remaining:      10.00 Wh
============================================================
```

Make Visualizations for our Data

The first line creates a 2×2 grid of plots (4 plots total) arranged like a window with four panes. We are using matplotlib library for data visualization. What we are doing is much like setting up four blank canvases (subplots) in a grid so we can show four different aspects of the simulation at once. We are labeling each subplot ax1, ax2, ax3, ax4. Since they now have individual names, we can assign different content to each one.

fig is the overall figure. It is a container for the individual plots. ax1, ax2, ax3, ax4 are the four individual plot areas. The double parentheses ((ax1, ax2), (ax3, ax4)) unpack the 2D array into named variables.

The "2, 2, " arguments in the subplots function mean we want 2 rows and 2 columns for 4 plots total. The "figsize=(14, 10)" argument in the subplots function is the width and height in inches.

Breaking this down in computer science terms, plt.subplots(2, 2) returns (fig, ndarray([[ax1, ax2], [ax3, ax4]])). The nested tuple unpacking ((ax1, ax2), (ax3, ax4)) destructures the 2×2 array and provides convenient named access instead of indexing axs[0,0], axs[0,1], etc.

For each timestep, the loop draws a horizontal bar. `barh(0, ...)` draws at y=0. All bars are on the same horizontal line. `10` sets the bar width at 10 seconds. `left=i*10` starts the bar at time `i*10` Timestep 0 starts at 0s, timestep 1 starts at 10s, etc. `color=mode_colors[mode]` sets the color of the bar based on the mode (blue for RX, red for TX, etc.)

The result is a horizontal colored timeline showing mode transitions.

We set the x axis label to Time (seconds). Then we set_yticks with an argument of ([]). This instruction removes the y-axis tick marks. Since y-position is meaningless here, and we are not showing any quantitiative y values, we do this to clear some visual clutter. We then set the title. Finally, we put a limit on the x axis values to equal the maximum value of time_axis.

Now we come to the legend. Here's where it gets interesting. In ax3, the loop creates 60 individual bars (one per timestep). If we just called ax3.legend(), then matplotlib would try to create a legend entry for every single bar. This would be 60 entries and would be mostly duplicates (RX, RX, RX, TX, TX, TX...).

What we actually want is one legend entry per mode type, showing its color. We manually create these entries using list comprehension.

First, we import Patch from matplotlib. Patch is a class that creates generic colored shape objects. Matplotlib will render these as colored squares in the legend. This is exactly what we need.

`pythonlegend_elements = [Patch(facecolor=color, alpha=0.7, label=mode_names[mode_id]) for mode_id, color in mode_colors.items()]`

Breaking this down.

`mode_colors.items()` returns key-value pairs from the dictionary. `[(1, 'blue'), (2, 'yellow'), (3, 'red'), (4, 'gray')]`

The loop then iterates through these pairs. Iteration 1 is mode_id=1, color='blue'.

We then look up the name. mode_names[1] returns 'RX'. We create a blue patch with an RX label with `Patch(facecolor='blue', alpha=0.7, label='RX')`

Iteration 2 is mode_id=2, color='yellow'.

We look up the name. mode_names[2] returns 'BUFFER'. We create a yellow patch with a BUFFER label with `Patch(facecolor='yellow', alpha=0.7, label='BUFFER')`

Iteration 3 is mode_id=3, color='red'. We look up the name. mode_names[3] returns 'TX'. We create a red patch with a TX label with `Patch(facecolor='red', alpha=0.7, label='TX')`

Iteration 4 is mode_id=4, color='gray'. We look up the name. mode_names[4] returns 'IDLE'. We create a gray patch with an IDLE label with `Patch(facecolor='gray', alpha=0.7,` For each timestep, the loop draws a horizontal bar. `barh(0, ...)` draws at y=0. All bars are on the same horizontal line. `10` sets the bar width at 10 seconds. `left=i*10` starts the bar at time `i*10` Timestep 0 starts at 0s, timestep 1 starts at 10s, etc. `color=mode_colors[mode]` sets the color of the bar based on the mode (blue for RX, red for TX, etc.)

The result is a horizontal colored timeline showing mode transitions.

We set the x axis label to Time (seconds). Then we set_yticks with an argument of ([]). This instruction removes the y-axis tick marks. Since y-position is meaningless here, and we are not showing any quantitiative y values, we do this to clear some visual clutter. We then set the title. Finally, we put a limit on the x axis values to equal the maximum value of time_axis.

Now we come to the legend. Here's where it gets interesting. In ax3, the loop creates 60 individual bars (one per timestep). If we just called ax3.legend(), then matplotlib would try to create a legend entry for every single bar. This would be 60 entries and would be mostly duplicates (RX, RX, RX, TX, TX, TX...).

What we actually want is one legend entry per mode type, showing its color. We manually create these entries using list comprehension.

First, we import Patch from matplotlib. Patch is a class that creates generic colored shape objects. Matplotlib will render these as colored squares in the legend. This is exactly what we need.

`pythonlegend_elements = [Patch(facecolor=color, alpha=0.7, label=mode_names[mode_id]) for mode_id, color in mode_colors.items()]`

Breaking this down.

`mode_colors.items()` returns key-value pairs from the dictionary. `[(1, 'blue'), (2, 'yellow'), (3, 'red'), (4, 'gray')]`

The loop then iterates through these pairs. Iteration 1 is mode_id=1, color='blue'.

We then look up the name. mode_names[1] returns 'RX'. We create a blue patch with an RX label with `Patch(facecolor='blue', alpha=0.7, label='RX')`

Iteration 2 is mode_id=2, color='yellow'.

We look up the name. mode_names[2] returns 'BUFFER'. We create a yellow patch with a BUFFER label with `Patch(facecolor='yellow', alpha=0.7, label='BUFFER')`

Iteration 3 is mode_id=3, color='red'. We look up the name. mode_names[3] returns 'TX'. We create a red patch with a TX label with `Patch(facecolor='red', alpha=0.7, label='TX')`

Iteration 4 is mode_id=4, color='gray'. We look up the name. mode_names[4] returns 'IDLE'. We create a gray patch with an IDLE label with `Patch(facecolor='gray', alpha=0.7,`

```
label='IDLE')
```

We loop through mode_colors to get both the mode number and its color, then use that mode number as a key to look up the corresponding name in mode_names. This pairs up the color and name for each mode, so that we can make a list of colored patches. The result is legend_elements, which is a list of 4 Patch objects, ready to display as our legend.

Finally, `pythonax3.legend(handles=legend_elements, loc='upper right')` tells matplotlib "use these specific patches for the legend" instead of trying to auto-generate from the 60 bars.

We're using the mode number (1, 2, 3, 4) as the "glue" that connects colors to names across two separate dictionaries.

If a *picture is worth a thousand words*, then it shouldn't be surprising that creating the picture can often be more involved than the math of the model itself.

Data Flow Rate Visualization

This subplot visualizes the total data transferred over time, showing how the asymmetric uplink/downlink rates affect the satellite pass.

Think of this visualization as a Dungeons and Dragons party. They are looting a dungeon. cumulative_rx tracks the total gold collected room-by-room (fast collection rate when actively looting, zero when not). cumulative_tx tracks total gold carried out of the dungeon (slower rate, can only happen when the party is leaving). The cumulative sum is your running total wealth over time. A powerful party can collect a lot more gold in a day than they can carry out.

We plot both cumulative curves. The blue line is fast uplink data accumulation. We expect to see a steep slope during RX mode. The red line is the slow downlink data transmission. We expect to see a gentle slope during TX mode. We know that this mode will be asymmetric, or unbalanced. We're carrying out this analysis to optimize the difference in rates in order to ensure the communications channel is useful for the operators. Data comes in fast but goes out slow. The gap between the lines in the graph represents data still in the buffer.

Finally, we add labels, title, legend, and a subtle grid. An alpha=0.3 makes it less obtrusive. plt.tight_layout() automatically adjusts spacing to prevent overlapping labels across all four subplots.

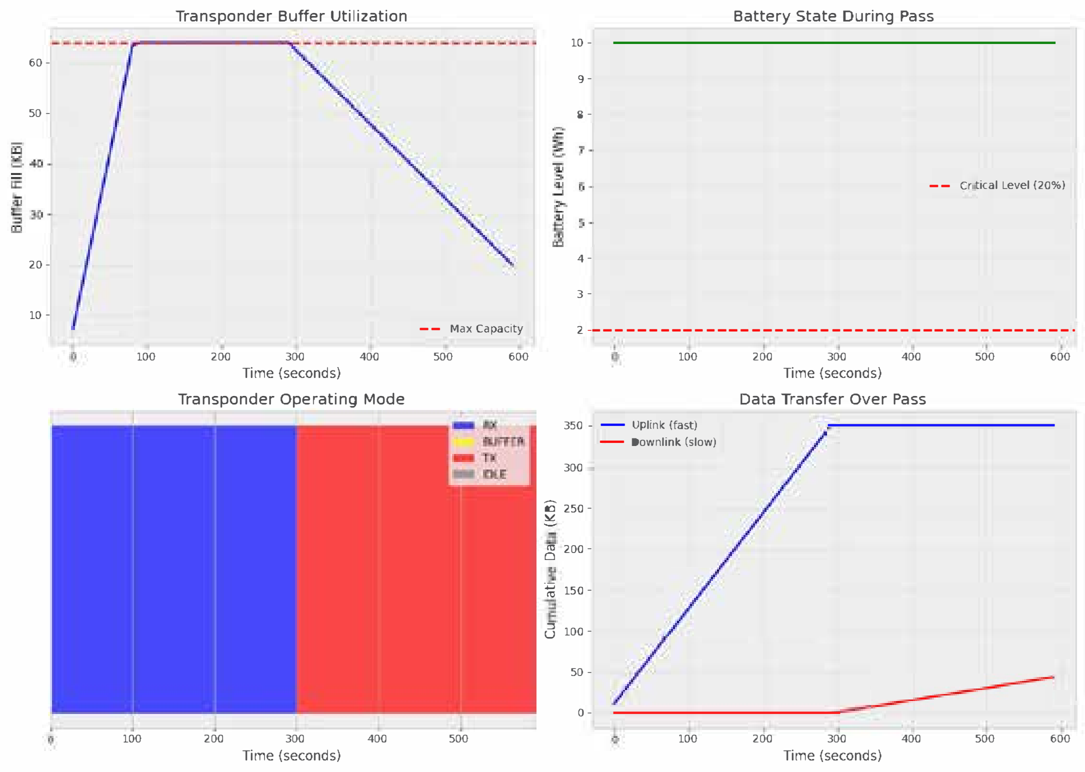
Save the Plots to Current Directory

We use the savefig function to save the entire figure to the current directory. We specify a dots per inch of 150. We then call show() to display the plots to the current display.

In [19]:
```
# Visualization
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(14, 10))
```

```
time_axis = np.arange(len(transponder.buffer_history)) * 10

# Buffer fill over time
ax1.plot(time_axis, np.array(transponder.buffer_history) / 1024, 'b-', linewidt
ax1.set_xlabel('Time (seconds)')
ax1.set_ylabel('Buffer Fill (KB)')
ax1.set_title('Transponder Buffer Utilization')
ax1.grid(True, alpha=0.3)
ax1.axhline(y=config.buffer_size_bytes/1024, color='r', linestyle='--', label='
ax1.legend()

# Battery level over time
ax2.plot(time_axis, transponder.battery_history, 'g-', linewidth=2)
ax2.set_xlabel('Time (seconds)')
ax2.set_ylabel('Battery Level (Wh)')
ax2.set_title('Battery State During Pass')
ax2.grid(True, alpha=0.3)
ax2.axhline(y=config.battery_capacity_wh * 0.2, color='r', linestyle='--', labe
ax2.legend()

# Mode timeline
mode_colors = {1: 'blue', 2: 'yellow', 3: 'red', 4: 'gray'}
mode_names = {1: 'RX', 2: 'BUFFER', 3: 'TX', 4: 'IDLE'}
for i, current_mode in enumerate(transponder.mode_history):
    ax3.barh(0, 10, left=i*10, color=mode_colors[current_mode], alpha=0.7, heig
ax3.set_xlabel('Time (seconds)')
ax3.set_yticks([])
ax3.set_title('Transponder Operating Mode')
ax3.set_xlim(0, max(time_axis))

from matplotlib.patches import Patch
legend_elements = [Patch(facecolor=color, alpha=0.7, label=mode_names[mode_id])
ax3.legend(handles=legend_elements, loc='upper right');

# Data flow rates
cumulative_rx = np.cumsum([transponder.config.uplink_rate_bps / 8 * 10 if m == 
cumulative_tx = np.cumsum([transponder.config.downlink_rate_bps / 8 * 10 if m =
ax4.plot(time_axis, cumulative_rx, 'b-', linewidth=2, label='Uplink (fast)')
ax4.plot(time_axis, cumulative_tx, 'r-', linewidth=2, label='Downlink (slow)')
ax4.set_xlabel('Time (seconds)')
ax4.set_ylabel('Cumulative Data (KB)')
ax4.set_title('Data Transfer Over Pass')
ax4.legend()
ax4.grid(True, alpha=0.3)

plt.tight_layout()

# Save to current directory instead
plt.savefig('funcube_plus_simulation.png', dpi=150)
print("\n✓ Simulation plot saved to current directory")
plt.show()
```

```
✓ Simulation plot saved to current directory
```


Transponder Buffer Utilization
Buffer Fill (KB)
Max Capacity
Time (seconds)
Battery State During Pass
Battery Level (Wh)
Critical Level (20%)
Time (seconds)
Transponder Operating Mode
RX
BUFFER
TX
IDLE
Time (seconds)
Data Transfer Over Pass
Uplink (fast)
Downlink (slow)
Cumulative Data (KB)
Time (seconds)

The Inner Circle
Sphere of Activity

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.

1 September 2025 Our Complex Modulation Math article was published in ARRL's QEX magazine in the September/October issue. See this issue for a comment and critique, and response!

5 September 2025 - Charter for the current Technological Advisory Council of the US Federal Communications Commission concluded.

19-21 September 2025 - ESA and AMSAT-DLworkshop was held in Bochum, Germany.

3 October 2025 - Deadline for submission for FCC TAC membership was met. Two volunteers applied for membership in the next available TAC.

10-12 October 2025 - Presentation given at Pacificon, San Ramon Marriot, CA, USA. Recording available on our YouTube channel.

11-12 October 2025- Presentation given to AMSAT-UK Symposium, and collaboration with AMSAT-UK commenced. See this issue for more details about MDT.

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