Unguruşan Fabian-Adrian

PROIECT DE DIPLOMĂ

Software Defined Radio and Antenna Switching Application using HackRF One and OperaCake

SPECIALIZAREA TEHNOLOGII ȘI SISTEME DE TELECOMUNICAȚII

Marchează dacă ai studiat în limba engleză: ÎN LIMBA ENGLEZĂ

☑

FACULTATEA DE ELECTRONICĂ, TELECOMUNICAȚII
ȘI TEHNOLOGIA INFORMAȚIEI
UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

2025

Facultatea de Electronică, Telecomunicații și Tehnologia Informației

Str. George Barițiu, nr. 26-28, 400027, Cluj-Napoca, tel: 0264-401224, tel/fax: 0264-591689, etti.utcluj.ro

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA
FACULTATEA DE ELECTRONICĂ
TELECOMUNICAȚII ȘI TEHNOLOGIA INFORMAȚIEI
Specializarea: Tehnologii și Sisteme de Telecomunicații (în limba engleză)

Software Defined Radio and Antenna Switching Application using HackRF One and OperaCake

Proiect de diplomă

Absolvent,
Ungurușan Fabian-Adrian

Decan,
Prof.dr.ing. Ovidiu POP

Președinte comisie,
Prof.dr.ing. Emanuel PUȘCHIȚĂ

2025

Această pagină a fost lăsată goală în mod intenționat.

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA
FACULTATEA DE ELECTRONICĂ
TELECOMUNICAȚII ȘI TEHNOLOGIA INFORMAȚIEI

Departamentul Comunicații

Titlul proiectului de diplomă:

Software Defined Radio and Antenna Switching Application using HackRF One and OperaCake

Descrierea temei:

This project aims to implement close to real-time radio frequency spectrum monitoring and antenna switching application by combining the HackRF One software defined radio with the OperaCake switch. A virtual machine environment is first created with Ubuntu 22.04 and the software part was developed and implemented using GNU Radio for digital signal processing using VS Code. For the graphical interface the QtGUI and PyQt5 were used. Several customizable modes of operations can be used such as manual switching, time switching and wideband frequency switching.

Locul de realizare:

Str. Observator, nr.1, laborator 410

Adaugă consultant: ☐

Data emiterii temei: 03.10.2024

Data predării temei: 03.09.2025

Absolvent,

Unguruşan Fabian-Adrian

Director departament,

Prof.dr.ing. Virgil DOBROTĂ

Conducător,

Ș.l.dr.ing. Cristian CODĂU

Această pagină a fost lăsată goală în mod intenționat.

Declaraţie pe proprie răspundere privind autenticitatea lucrării de diplomă

Subsemnata/ul **Unguruşan Fabian-Adrian**
legitimat cu CI seria SX nr. 605388 (conform copiei anexate prezentei declarații, copie semnată și certificată "conform cu originalul"), autorul lucrării

Software Defined Radio and Antenna Switching Application using HackRF One and OperaCake

elaborată în vederea susţinerii examenului de finalizare a studiilor de licență,
la Facultatea de Electronică, Telecomunicații și Tehnologia Informației
Programul de studiu Tehnologii și Sisteme de Telecomunicații (în limba engleză)
din cadrul Universității Tehnice din Cluj-Napoca
sesiunea septembrie a anului universitar 2024-2025

declar pe proprie răspundere că această lucrare este rezultatul propriei activităţi intelectuale, pe baza cercetărilor mele şi pe baza informaţiilor obţinute din surse care au fost citate în textul lucrării şi în bibliografie.

Declar că această lucrare nu conţine porţiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislaţiei române şi a convenţiilor internaţionale privind drepturile de autor. Declar, de asemenea, că această lucrare nu a mai fost prezentată în faţa unei alte comisii de examen de absolvire/licenţă/diplomă/disertaţie.

În cazul constatării ulterioare a unor declaraţii false, voi suporta sancţiunile administrative, respectiv anularea examenului de diplomă.

Sunt de acord ca, pe tot parcursul vieții, în cazul în care este necesar și se va dori verificarea autenticității lucrării mele să fiu identificat și verificat în baza datelor declarate de mine și conform copiei documentului de identitate menționat mai sus.

Data
03.09.2025

Nume și prenume
Unguruşan Fabian-Adrian

Semnătura


Această pagină a fost lăsată goală în mod intenționat.

Absolvent: **Unguruşan Fabian-Adrian**
Conducător: **Ş.l.dr.ing. Cristian CODĂU**

SINTEZA PROIECTULUI DE DIPLOMĂ

Software Defined Radio and Antenna Switching Application using HackRF One and OperaCake

Software Defined Radio (SDR) has a lot of applications, implemented especially in GNU Radio. Antenna switching with SDR enhances an RF spectrum monitoring system. As such, the aim of this project is the implementation of a custom intuitive RF spectrum application combined with antenna switching, using the HackRF One SDR and the OperaCake antenna switch from Great Scott Gadgets. The hardware implementation consists of just the OperaCake mounted on top of the HackRF One. On the software side two steps are needed. Installation of a virtual environment Ubuntu 22.04 and the software implementation in VS Code. The latter step involves developing the digital signal processing flowgraphs in GNU Radio and integrating them within a graphical interface built with the help of PyQt5 and QtGUI components. This integration provides intuitive and custom controls over SDR's parameters, enabling OperaCake antenna switching, with several modes of operation such as manual switching, time switching and wideband frequency switching. The user can map various RF applications in the range 10MHz - 4GHz on any of the OperaCake's ports.

Aplicație cu Software Defined Radio și Antenna Switching folosing HackRF One și OperaCake

Software Defined Radio (SDR) are o mulțime de aplicații, implementate în special în GNU Radio. Antena switching cu SDR îmbunătățește sistemul de monitorizare a spectrului RF. Ca atare, scopul acestei lucrări este implementarea unei aplicații intuitive și personalizabile pentru spectrul RF, combinată cu antenna switching, utilizând SDR-ul HackRF One și OperaCake antenna switch de la Great Scott Gadgets. Implementarea hardware constă doar în conectarea OperaCake direct peste HackRF One. Din punct de vedere software doi pași sunt necesari. Instalarea unui mediu virtual Ubuntu 22.04 ș??i implementarea software în VS Code. Cel din urmă pas implică dezvoltarea diagramelor de flux de procesare a semnalului digital în GNU Radio și integrarea acestora într-o interfață grafică alcătuită cu ajutorul componentelor din PyQt5 și QtGUI. Această integrare oferă controale intuitive și personalizabile asupra parametrilor SDR, permițând OperaCake antenna switching, cu mai multe moduri de operare, cum ar fi manual switching, time switching și wideband frequency switching. Utilizatorul poate mapa diverse aplicații RF în intervalul 10MHz - 4GHz pe oricare dintre porturile OperaCake.

Avizul conducătorului
Conducător,

Absolvent,

Această pagină a fost lăsată goală în mod intenționat.

Contents

1. Rezumat în limba română

1.1 Introducere

Lucrarea de față detaliază proiectarea și implementarea unui sistem de analiză spectrală și management dinamic al antenelor bazat pe conceptul de Software Defined Radio (SDR). Scopul este de a construi o platformă capabilă să monitorizeze și să proceseze semnale radio utilizând echipamente cu costuri reduse, cum ar fi HackRF One și comutatorul de radiofrecvență OperaCake RF switch. Această abordare reprezintă o alternativă viabilă la analizoarele de spectru convenționale, care sunt adesea prohibitive ca preț și limitate la benzi de frecvență fixe.

Sistemul implementează prin software funcționalități avansate, adaptându-se la dinamica actuală a spectrului radio, care este din ce în ce mai aglomerat. Metodologia de bază include utilizarea tehnicilor de procesare a semnalelor digitale (DSP), cum ar fi transformata rapidă Fourier (FFT), pentru a converti semnalele din domeniul timp în cel al frecvenței. Alături de aceasta, sunt integrate ferestrele de ponderare pentru a minimiza efectele de scurgere spectrală (spectral leakage) și a optimiza rezoluția în frecvență. Platforma permite monitorizarea în timp real, detecția de interferențe și a comportamentelor anormale, precum și vizualizarea benzilor largi de frecvență prin tehnici de sweep și concatenare.

Proiectul combină teoria fundamentală a semnalelor, incluzând modelul semnalului complex și eșantionarea, cu implementarea practică. Sistemul software, dezvoltat într-o interfață grafică PyQt5, asigură o gestionare intuitivă a datelor și un control eficient asupra hardware-ului. Rezultatele experimentale demonstrează că un sistem compus dintr-un dispozitiv SDR cu cost redus, completat de un comutator de antene și de o interfață software personalizată, poate funcționa ca o platformă performantă de analiză spectrală. Astfel, se confirmă potențialul acestei soluții pentru aplicații educaționale și de cercetare, oferind o metodă accesibilă de a studia și de a interacționa cu mediul electromagnetic.

1.2 Fundamentare teoretică

Baza teoretică a proiectului se sprijină pe mai multe concepte esențiale din domeniul procesării semnalelor și al comunicațiilor digitale:

Semnal complex în bandă de bază (complex baseband). În loc să lucrăm cu semnale radio tradiționale, care sunt centrate pe o frecvență purtătoare înaltă (*fc*), le „mutăm” digital la 0 Hz. Procesul se numește conversie în cuadratură și implică multiplicarea semnalului inițial cu un semnal local (LO) cosinus (pentru a obține componenta In-phase sau I) și un LO sinus (pentru componenta Quadrature sau Q). Combinând aceste două componente obținem un semnal complex, $x(t)=I(t)+jQ(t)$, care capturează absolut toată informația de amplitudine și fază a semnalului original.

Eșantionare și aliasing. Teorema Nyquist-Shannon ne zice că, pentru a eșantiona corect un semnal de lățime de bandă B, rata de eșantionare (fs) trebuie să fie cel puțin 2B. Pentru semnalele reale, banda de frecvențe exploatabilă e doar de la 0 la fs/2. Dar șmecheria cu eșantionarea complexă (I/Q) este că putem folosi atât frecvențele pozitive, cât și pe cele negative, adică de la −fs/2 până la +fs/2. Practic, dublăm lățimea de bandă utilă fără să creștem rata de eșantionare. În cazul HackRF One, rata maximă este de 20 MS/s, deci lățimea de bandă instantanee e limitată la 20 MHz. Pentru a scana o bandă mai largă, trebuie să „lipim” segmente de 20 MHz la rând, o tehnică numită step-tuned stitching.

Analiza spectrală cu FFT și ferestre de analiză. Pentru a trece din domeniul timp în domeniul frecvență, folosim Transformata Fourier Discretă (FFT). Aceasta descompune un set de eșantioane (N la număr) în componente de frecvență (numite bins). Fiecare bin k corespunde unei frecvențe $fk = k/N*fs$. Cu cât folosim mai multe eșantioane (adică N mai mare), cu atât rezoluția spectrală ($\Delta f = fs/N$) este mai bună. Problema e că o FFT directă pe o fereastră dreptunghiulară provoacă scurgeri spectrale, adică energia de la o frecvență se „scurge" și pe frecvențele vecine. Pentru a rezolva asta, aplicăm o funcție de fereastră înainte de FFT. Am folosit fereastra Blackman-Harris pe 4 termeni (BH4) pentru că reduce la minimum acele scurgeri, chiar dacă lățește puțin vârful principal. Lățimea acestui vârf este măsurată prin Equivalent Noise Bandwidth (ENBW). Pentru BH4, ENBW este aproximativ 2 bins. Rezoluția reală în Hz, numită Resolution Bandwidth (RBW), este dată de formula $RBW = ENBW \cdot (fs/N)$.

Tehnici avansate și hardware auxiliar. Pentru a acoperi benzi de frecvențe foarte largi, cum ar fi 200 MHz, step-tuned wideband stitching e o necesitate. Practic, scanam banda în bucăți de 20 MHz și le asamblăm ulterior. E important să gestionăm artefactele de la margini (cunoscute ca "seam effects") și să eliminăm zgomotul de la 0 Hz (DC offset) folosind filtre de tip notch. Pentru a ne ușura munca, am folosit OperaCake, un comutator RF care ne permite să conectăm până la 8 antene la HackRF. Acesta poate comuta automat între ele în funcție de frecvența setată (modul Frequency), după un anumit timp (modul Time) sau manual.

Algoritmi de detecție a anomaliilor. Pentru a identifica automat probleme precum bruiajul (jamming), am dezvoltat câțiva indicatori cheie. De exemplu, ΔdB occupancy ne arată cât de mult din spectru depășește un anumit prag de zgomot. Spectral flatness measure (SFM) ne ajută să distingem semnalele de bruiaj de un zgomot de fond uniform. În plus, am monitorizat ridicările mediei sau medianei spectrale peste un nivel de referință. Combinația acestor metrici cu timpi de așteptare (hysteresis, hold/cooldown) ne permite să detectăm cu încredere și rapiditate anomaliile spectrale.

1.3 Implementare

Implementarea proiectului a urmat o abordare incrementală, pornind de la configurarea riguroasă a mediului de dezvoltare și avansând către dezvoltarea unei aplicații software cu interfață grafică complexă. Această metodologie a asigurat reproductibilitatea rezultatelor și validarea funcționalității la fiecare etapă.

1.3.1 Pregătirea mediului de dezvoltare

Pentru a asigura un mediu de lucru controlat și izolat, s-a optat pentru o mașină virtuală bazată pe Oracle VirtualBox cu sistem de operare Ubuntu 22.04 LTS. Această soluție a facilitat passthrough-ul USB pentru conectarea directă a hardware-ului și a asigurat o bază curată și replicabilă pentru dezvoltare.

În această etapă, au fost instalate și compilate din surse următoarele pachete software esențiale:

- GNU Radio: Un toolkit software-defined radio (SDR) fundamental pentru procesarea semnalelor.
- gr-osmosdr: O interfață unificată pentru diverse dispozitive SDR, inclusiv HackRF.
- HackRF Tools: Setul de utilitare specific pentru controlul dispozitivului HackRF One.

Compilarea din surse a fost necesară pentru a garanta compatibilitatea cu noile funcționalități ale comutatorului OperaCake. Validarea inițială a recunoașterii corecte a dispozitivului a fost realizată cu succes prin intermediul comenzii hackrf_info.

1.3.2 Prototipuri cu GNU Radio Companion (GRC)

Pentru a valida funcționalitatea hardware și a testa conectivitatea, au fost dezvoltate prototipuri inițiale folosind GNU Radio Companion (GRC). Au fost create flowgraph-uri minimale, utilizând blocurile osmosdr.source pentru recepția semnalului, și qtgui.freq_sink_c și qtgui.waterfall_sink_c pentru vizualizarea spectrală. Aceste prototipuri au permis confirmarea funcționalității prin detectarea clară a posturilor de radio FM din spectrul local, validând astfel recepția datelor.

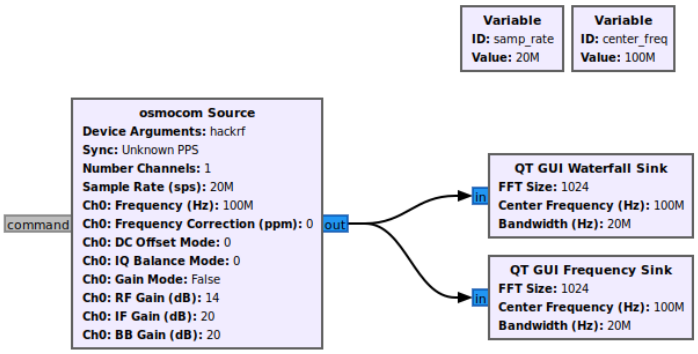

Figure 1. Diagrama bloc GRC

1.3.3 Integrarea OperaCake

Înainte de a trece la dezvoltarea interfeței grafice, controlul hardware al modulului OperaCake a fost verificat independent. Prin intermediul utilitarelor de linie de comandă hackrf_operacake, s-au testat toate modurile de comutare: manual, frequency și time. Această etapă a permis nu doar validarea comportamentului de comutare, ci și estimarea preliminară a pierderilor de inserție introduse de comutator.

1.3.4 Dezvoltarea aplicației software cu interfață grafică (GUI)

Aplicația finală a fost dezvoltată, urmând o arhitectură modulară pentru o mai bună mentenabilitate. Structura de bază a constat în două module principale:

- GUI.py: Acționează ca lansator principal, gestionând interfața și selecția modurilor de operare.
- live_spectrogram.py: A găzduit logica de bază pentru fiecare mod, inclusiv procesarea semnalului și controlul comutatorului OperaCake.
- LED.py: Vizualizează dynamic starea comutatorului OperaCake prin intermediul unor indicatori LED și etichete actualizând culorile și starea pe baza portului selectat.

Au fost implementate mai multe moduri de operare distincte:

- Manual Switching: Permite selecția directă a unui port de antenă de către utilizator.
- Frequency Switching: Comutare automată a porturilor în funcție de frecvența centrală curentă.
- Time Switching: Ciclare automată a porturilor la un interval de timp prestabilit.
- Delay Mode: Un mod de analiză specializat, care utilizează un waterfall cu istoric scurt (~100 ms) pentru a măsura latența de comutare.

- Wide Spectrum Switching & Wide Spectrum Frequency Switching: Implementarea tehnicii de step-tuned stitching pentru a vizualiza benzi largi, cu sau fără asocierea porturilor OperaCake.
- Event Detection and Switching: Integrează algoritmi de detecție a anomaliilor, bazându-se pe metricile ΔdB occupancy, Spectral Flatness Measure (SFM) și Median/Mean lifts.

1.3.5 Validări și optimizări finale

Proiectul s-a încheiat cu o etapă de validare și optimizare a codului. Au fost adăugate mecanisme să verifice intrările utilizatorului, funcții de debounce pentru comutările hardware, mesaje de eroare clare în interfața grafică și funcții de cleanup pentru eliberarea resurselor la închiderea fiecărui mod. Aceste optimizări au contribuit la creșterea robusteții și fiabilității aplicației finale.

1.4 Teste și rezultate

1.4.1 Pregătirea mediului de lucru

S-a utilizat un sistem Ubuntu 22.04 într-o mașină virtuală Oracle VirtualBox, pentru a facilita passthrough-ul USB și a menține mediul curat și reproductibil.
Au fost instalate: GNU Radio, pachetul gr-osmosdr și HackRF Tools, toate compilate din surse pentru a asigura compatibilitatea cu OperaCake.

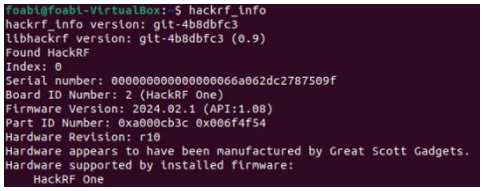

Figure 2. Captură terminal hackrf_info care confirmă recunoașterea dispozitivului

1.4.2 Primele prototipuri cu GNU Radio Companion (GRC)

Au fost create flowgraph-uri minimale cu blocuri ‚*osmosdr.source*', ‚*qtgui.freq_sink_c*' și ‚*qtgui.waterfall_sink_*c' pentru a valida recepția. Rezultatele au confirmat detecția posturilor FM locale și funcționalitatea hardware.

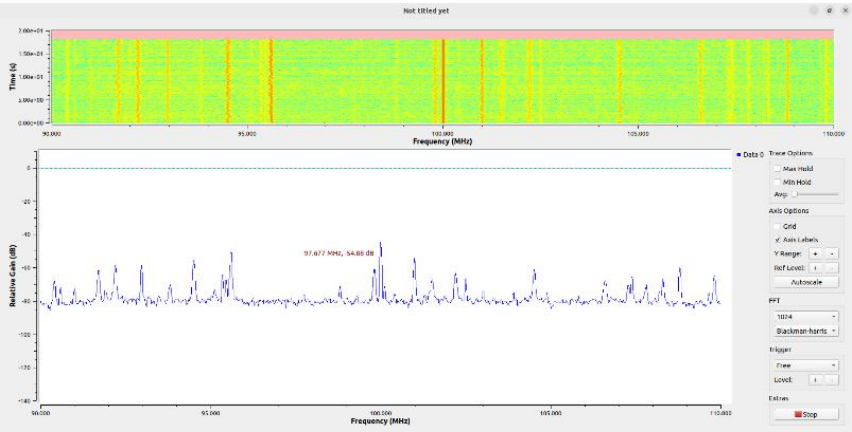

Figure 3. Captură spectru FM

1.4.3 Integrarea OperaCake prin terminal

În faza inițială, OperaCake a fost controlat prin comenzi ‚hackrf_operacake', verificând modurile manual, frecvență și timp. Acest lucru a permis confirmarea comportamentului mutării.

Figure 4. HackRF One conectat cu OperaCake

1.4.4 Dezvoltarea interfeței grafice

Au fost dezvoltate moduri distincte pentru rutare manuală, în funcție de timp și de frecvență, pentru calcularea timpului de rutare, bandă larga și bandă largă cu schimbare în frecvență și încă un mod ce detectează evenimente nedorite de tip anomalie și schimbă portul.

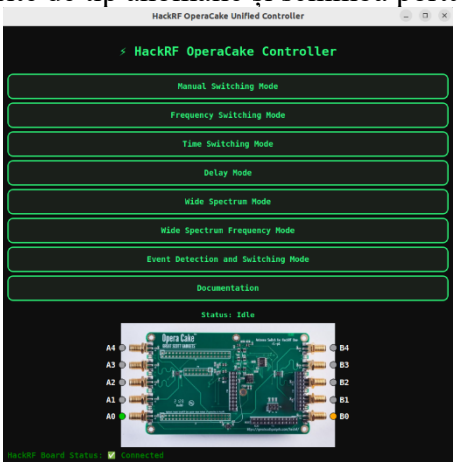

Figure 5. Ecran principal aplicație

1.4.5 Validări și optimizări

S-au adăugat verificări de tip input, debounce la comutări, afișare mesaje de eroare și rutine de închidere și eliberare a resurselor folosite.

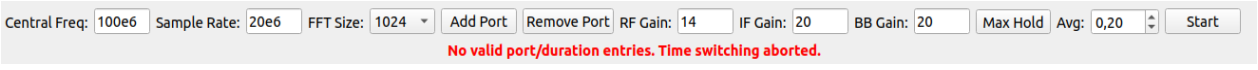

Figure 6. Captură exemplu eroare la pornirea modului rutare în timp

1.5 Concluzii

1.5.1 Principalele concluzii

Integrarea unui software avansat, care utilizează tehnici precum eșantionarea complexă și concatenarea spectrală, a permis extinderea capacităților de analiză ale unui sistem SDR low-cost, rivalizând cu echipamente comerciale costisitoare. Suplimentar, utilizarea comutatorului

OperaCake și a unei interfețe grafice intuitive a facilitat monitorizarea multi-bandă și detecția eficientă a anomaliilor spectrale prin metrici simpli, eliminând necesitatea unor algoritmi complecși de învățare automată.

1.5.2 Îmbunătățiri viitoare

Un domeniu viitor de cercetare include dezvoltarea de metode pentru calibrarea absolută a puterii în dBm, esențială pentru măsurători precise și aplicații de conformitate, precum și sincronizarea mai multor SDR-uri pentru a crea o rețea distribuită capabilă de triangulație pentru localizarea surselor de semnal.

2. Work Planning

No.	Task	Start date	No. of days	End date
1	Selection of topic and initial research	7.04.2025	8	15.04.2025
2	Install Ubuntu VM, configure GNU Radio, gr-osmosdr, HackRF tools, test basic flowgraphs	16.04.2025	14	30.04.2025
3	Connect HackRF One, validate device recognition, test manual switching	1.05.2025	3	4.05.2025
4	Connect OperaCake, validate device recognition, test frequency and time switching	5.05.2025	7	12.05.2025
5	Spectrum visualization with GNU Radio Companion, basic delay estimation with python scripts	13.05.2025	12	25.05.2025
6	Implememt Pyt5 launcher, manual/frequency/time switching modes	26.05.2025	20	15.06.2025
7	Add delay mode, wide spectrum mode and event detection logic	16.06.2025	35	21.07.2025
8	Test all modes, measure latency, error handling improvements	22.07.2025	5	27.07.2025
9	Main GUI development	28.07.2025	2	30.07.2025
10	Comparative evaluation of results with different antennas and ports	31.07.2025	4	4.08.2025
11	Thesis writing	5.08.2025	23	28.08.2025
12	Final edits, proofreading, formatting and submission	29.08.2025	5	3.09.2025

3. State of the Art

In modern wireless systems, spectrum availability has become increasingly congested, pushing engineers and researchers to explore more dynamic, adaptive, and efficient communication technologies. At the heart of this transformation lies Software-Defined Radio (SDR), a clever approach that replaces rigid hardware radios with programmable platforms. This means you can implement all sorts of protocols, signal processing, and monitoring techniques just by changing the software [1], [2].

Developing SDR fits right in with the broader trend of reconfigurable computing. Instead of fixed hardware functions, we're moving towards programmable environments that let us quickly prototype, deploy, and experiment [1]. SDR systems are key for things like sensing the spectrum in real-time, adapting how signals are sent, making systems resilient to jamming, and operating across multiple frequency bands. These capabilities are crucial not just for research and education, but also for vital applications in public safety, telecommunications, aerospace, and electronic warfare [2], [3].

Plenty of tools and frameworks have popped up to support SDR development. Among them, GNU Radio stands out as the top open-source signal processing toolkit [4]. When you pair it with affordable hardware like HackRF One [5] and add accessories such as the OperaCake RF switch [6], the SDR ecosystem becomes incredibly accessible for experimentation. This chapter will look at the core technologies, available platforms, and recent research that set the stage for our work within the evolving world of SDR systems.

3.1 Introduction to Software-Defined Radio

Software-Defined Radio (SDR) lets you implement traditional radio components – think mixers, filters, demodulators, and modulators in software, running on standard processors, DSPs, or FPGAs [1], [2]. This architecture offers unmatched flexibility for designing, deploying, and testing wireless systems. Compared to old-school radio hardware, SDR makes it possible to operate across multiple standards (like GSM, LTE, or Wi-Fi), update firmware and protocols remotely, cut down prototyping costs, and reconfigure the system in real-time [7].

Historically, SDR technology first emerged for military and aerospace uses, where versatile communication was a must [1]. Today, SDR is a core enabler for civilian tech like 5G, IoT, and satellite monitoring, showing just how widely applicable programmable radios have become [2], [8].

A key enabler for SDR is digitizing analog RF signals as close to the antenna as possible using high-speed analog-to-digital converters (ADCs) [1]. Once the signal is digital, you can build signal chains using software libraries to perform filtering, demodulation, decoding, and analysis [9], [10]. This approach supports the development of advanced systems such as cognitive radios, dynamic spectrum access solutions, and AI-driven communication protocols [11]. Modern SDR systems also support wideband signal capture, meaning a single device can monitor or interact with multiple frequency bands at once [12]. This is increasingly vital for things like spectrum monitoring, satellite communication, and surveillance [2], [13]. The combination of USB 3.0 or PCIe-based SDR frontends with multi-threaded DSP libraries running on CPUs or GPUs has further boosted real-time performance across various platforms [9].

3.2 GNU Radio Companion and Open-Source Signal Processing

GNU Radio Companion (GRC) is one of the most widely adopted open-source frameworks for developing and executing SDR applications [4]. First released in the early 2000s, it has evolved into a modular toolkit that handles the full digital signal processing pipeline, from manipulating raw IQ stream to high-level modulation and protocol decoding.

GRC applications are composed of flowgraphs-networks of interconnected processing blocks. These blocks can be defined in Python or created using GRC's graphical interface [14]. The framework's modularity and extensibility allow for integration with external libraries, enabling real-time processing, protocol emulation, and RF analytics [4].

One GRC's major strengths is its community-driven design, which encourages continual improvements, new block contributions, and hardware compatibility through modules such as gr-osmosdr. Devices like HackRF One, USRP, RTL-SDR, and LimeSDR are natively supported, making GNU Radio a powerful foundation for cross-platform, multi-vendor SDR development [13].

While powerful, GRC's default interface isn't really built for rich GUI interactivity beyond its basic QT GUI sinks [4]. For more advanced applications involving live spectrum navigation, antenna switching, or user-driven frequency hopping, you'll need external interfaces. These are usually developed using GUI frameworks like PyQt5 or PySide2, which embed GRC blocks into fully interactive software systems [11]. Our current work follows this strategy: we integrate real-time SDR streaming with PyQt5 GUI elements to give users accessible controls for things like FFT size, gain, waterfall parameters, antenna switching, and event detection and switching thresholds. Projects like QSpectrumAnalyzer [15] and GQRX [16] show similar integrations, though they often don't have multi-mode antenna control or live switching logic.

3.3 HackRF One and OperaCake: Capabilities and Use Cases

HackRF One, developed by Great Scott Gadgets, is an affordable half-duplex SDR transceiver operating across a frequency range from 1 MHz to 6 GHz. It supports a sampling rate up to 20 MS/s, USB 2.0, and integrates seamlessly with GNU Radio [6]. Its open-source design and extensive documentation have made it widely adopted in education, research, and cybersecurity [3].

To expand its utility, the OperaCake RF switch provides dynamic routing of RF signals between HackRF and up to eight antenna ports (four on bank A and four on bank B) [6]. You can control OperaCake via USB, external scripts, or GUI-based software, enabling high-level logic for automated frequency band scanning or directional monitoring [17].

OperaCake has three main switching modes:

- Manual Mode: User manually selects the antenna port.
- Frequency Mode: Port selection is driven by the tuned center frequency.
- Time Mode: Ports cycle through a configured sequence based on dwell times [6].

These modes enable applications such as:

- Multi-band spectrum surveillance (FM, ISM, ADS-B, etc.).
- Time-segmented monitoring of RF environments.
- Antenna performance comparison across frequency ranges [5], [8].

Despite these powerful capabilities, no existing open-source GUI application fully supports all three switching modes with built-in safety logic, dynamic visuals, and runtime reconfigurability. The system we're proposing tackles this gap by developing a PyQt5-based GUI application that includes real-time waterfall and frequency plots, antenna control widgets, safe mode switching, and logic validation [11].

3.4 Existing GUI Tools and Unmet Requirements

Several third-party tools have emerged to enhance GNU Radio's usability and interface capabilities. Among these, GQRX, QSpectrumAnalyzer, and SDRAngel are the most referenced in open-source SDR development. Each tool brings unique strengths but falls short in certain key aspects:

- GQRX is built with Qt and GNU Radio, offering real-time waterfall and frequency displays. However, it lacks runtime configurability and is limited when it comes to live reconfiguration SDR parameters or antenna logic [16].
- QSpectrumAnalyzer provides frequency sweep visualization with simple GUI controls built using PyQt and Python. While it supports frequency tuning and basic parameter updates, it doesn't support dynamic antenna switching or logic integration [15].
- SDRAngel supports multiple platforms and advanced signal chains but has a steep learning curve, a less intuitive plugin architecture, and limited extensibility for GUI-based experimentation or educational use [3].

Common limitations among these tools:

- No multi-mode antenna switching logic.
- No real-time reconfiguration of FFT size, gain, or waterfall properties.
- No jamming detection or anomaly response logic.
- Poor support for modular expansion in research contexts [11], [3].

Additionally, many tools don’t implement error handling or debounce logic when reconfiguring SDR frontends. This can lead to crashes or undefined behavior during streaming, particularly when switching FFT sizes, updating frequency ranges, or interacting with hardware ports [4],[10].

3.5 Addressing the Gap

To address these limitations, this thesis introduces a modular, PyQt5-based graphical interface layered over a GNU Radio backend. The system includes:

- A unified GUI interface with live waterfall and frequency spectrum visualization
- Runtime control of FFT size, SDR gain, waterfall parameters, and antenna port
- A logic engine for safety, seamless switching between Manual, Frequency, and Time modes
- Integrated event detection with automated switching logic based on threshold analysis

The PyQt5 frontend communicates with GNU Radio through well-defined configuration bridges, enabling tight synchronization between user input and DSP processing blocks [11]. This design also ensures that updates to visual parameters or SDR state do not destabilize the application. In addition to manual control, the logic engine enforces safety by validating port entries and implementing duration checks, frequency thresholds, and safe default states [6].

Overall, this work bridges a critical gap in the current SDR GUI ecosystem. It offers a platform suitable not only for educational demonstrations and lab experiments but also for research in dynamic spectrum management, real-time scanning, and intelligent RF control [11], [8]. Future directions may include the integration of AI/ML-based classification, RF fingerprinting, or autonomous reconfiguration protocols further extending the capabilities of low-cost SDR systems [9], [13].

3.6 Outlook on SDR Evolution

As SDR hardware continues to become more affordable and capable, the need for robust, real-time, and user-friendly graphical interfaces will only increase [1], [3]. Human-in-the-loop systems will require intelligent visual interfaces that not only show RF activity but also suggest or automate reconfiguration actions based on the current context and needs [9].

Emerging tools such as web-based SDR interfaces, AI-assisted spectrum analysis, and remote-controlled RF environments will likely become standard [11]. Integration with cloud platforms or distributed sensing networks is also expected to appear, especially for collaborative spectrum monitoring, interference analysis, and wireless security [7], [8].

The system developed in this work represents a cornerstone step toward such future systems merging usability with technical robustness and providing a testbed for next-generation SDR research and education [11].

4. Theoretical Fundamentals

Modern spectrum monitoring hinges on three realities: heterogeneous signals crowd wide frequency spans; front-ends offer only finite instantaneous bandwidth and dynamic range; and interactive tools must stay responsive while remaining metrically sound. The theory in this chapter formalizes the complex-envelope model, sampling/aliasing limits, windowed FFT estimation (with our display normalization), RBW/ENBW relations, wide-span coverage by step-tuned stitching, OperaCake switching policies, and robust deviation metrics. Throughout, symbols and defaults match the running application (GNU Radio + HackRF One + OperaCake + PyQt5), so that each control: sample rate, FFT size, averaging, gains, switching mode maps directly to a well-defined quantity [1], [2], [4] – [6], [9], [11], [13], [17].

4.1 Why SDR: problem setting and constraints

Monitors are expected to scan diverse standards (FM, cellular, ISM/Wi-Fi, GNSS, etc.), detect activity over wide spans, and react deterministically. Fixed, per-band analog receivers rarely meet these needs economically or flexibly. Software-defined radio pushes mixing, filtering, measurement, and even control policy into software after early digitization, so that one signal chain can be retuned, resized, and repurposed on demand [1], [2].

4.2 Complex baseband model and frequency translation

4.2.1 From reall passband to complex baseband

We adopt the standard complex-enveloped mode. Let the real RF signal be $x_{RF}(t)$ centered at fc. After quadrature mixing and low-pass filtering (LPF) we obtain the complex envelope $x(t)$:

$$x(t) = x_{RF}(t)e^{-j2\pi fct} * LPF \quad => \quad x_{RF}(t) = \Re\{x(t)e^{j2\pi fct}\} \tag{1}$$

Processing is performed on x(t) (I/Q) which preserves amplitude and phase [1], [2].

4.2.2 Discrete-time model, sampling and aliasing

With sampling period $T=1/fs$,

$$x[n] = x(nTs) \tag{2}$$

For a complex (I/Q) srteam, the usable baseband span is $|f| \leq fs/2$ (full two-sided fs); for a real stream it would be $[0, fs/2]$. Anti-alias analog filtering should therefore pass the occupied bandwidth B and reject images beyond $\pm fs/2$ [1], [2]. If the desired spectrum does not fit in fs (HackRF practical limit $\approx 20MS/s$ in our application), wider coverage is achieved by step tuning.

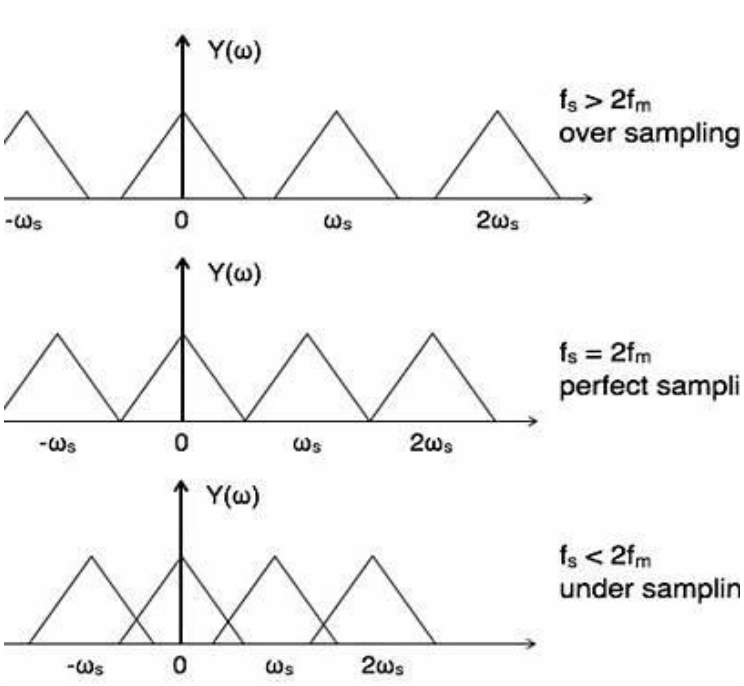

Figure 7. Spectrum replicas under different sampling conditions

4.2.3 Numerical translation

Software frequency translation is multiplication by a complex exponential:

$$y[n] = x[n]e^{-j2\pi f \Delta nTs} \tag{3}$$

Which re-centers content by f Δ without analog LO feedthrough. We use this concept implicitly via the FFT's frequency axis (no separate NCO block is required in our current modes) [1], [2] [4].

4.3 FFT mapping, windowing and normalization

4.3.1 Bin-to-frequency map

For an N-point frame x [0], …, x [N-1] the windowed DFT is:

$$X[k] = \sum_{n=0}^{N-1} x[n]\, w[n]\, e^{-\frac{j2\pi kn}{N}}, \qquad k = 0, \dots, N-1 \tag{4}$$

the (fftshifted) baseband frequency of bin k is

$$f[k] = \frac{k - \frac{N}{2}}{N} f_S \;\Rightarrow\; f_{RF}[k] = f_c + f[k] \tag{5}$$

This mapping is used to build stitched wideband axes.

4.3.2 Window choice and leakage

We proceed with the 4-term Blackman-Harris window w[n] (BH4) for high sidelobe suppression. Two window constants govern amplitude and noise scaling. The coherent gain C_w used for tone amplitude, and the equivalent noise bandwidth (ENBW) are:

- Coherent gain

$$C_w = \frac{1}{N}\sum_{n=0}^{N-1} w[n] \tag{6}$$

- ENBW (in bins)

$$ENBW_{bins} = \frac{N\sum_n w^2[n]}{(\sum_n w[n])^2} \tag{7}$$

The RBW is the ENBW expressed in Hz. For BH4, $ENBW_{bins}$ is approximately 1.9-2.0 for N ≥ 256. For ENBW we defined the resolution bandwidth (RBW):

$$RBW = ENBW_{bins} * \frac{f_S}{N} \tag{8}$$

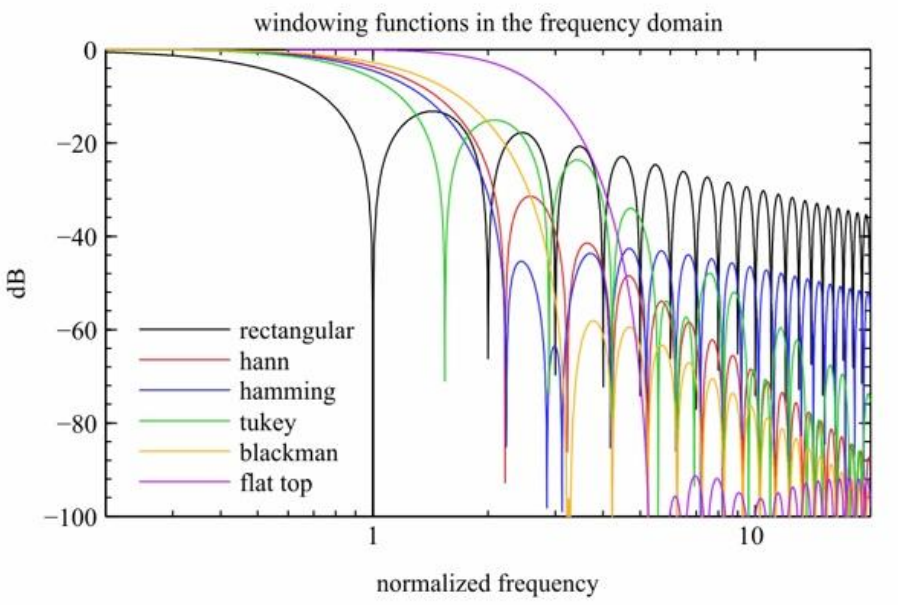

Figure 8. Comparison of window functions in the frequency domain

4.3.3 Normalization

The spectra shown (and fed to the detectors) uses a relative magnitude scale in decibels rather than an ENBW/coherent-gain-corrected power scale. For each bin k we take the windowed FFT magnitude, normalize by the FFT size, smooth with an exponential moving average, convert to dB, and add a user offset:

$$S_{dB}[k] = 20\log_{10}\left(EMA_\alpha\left\{\frac{|X[k]|}{N}\right\}\right) + k \tag{9}$$

Where N is FFT length, $EMA_\alpha\{.\}$ uses the UI 'Avg" factor α, and k is the UI 'Relative Gain' offset. For plotting stability, we set $S_{dB}[k]$ to a fixed lower floor. Coherent-gain and ENBW corrections are intentionally omitted.

4.4 Variance control

Periodograms are noisy, we expose:

- EMA (magnitude domain):

$$\widehat{m_t}[k] = \alpha m_t[k] + (1-\alpha)\,\widehat{m_{t-1}}[k], \qquad m_t[k] = \frac{|X_t[k]|}{N} \tag{10}$$

  With α € [0, 1] set by the 'AVG' UI controller.
- Max hold: a running per-bin maximum to capture intermittent peaks [1], [10], [11].

The waterfall stacks successive FFT slices, GUI timers (not only N/fs) set the visible cadence.

4.5 Dynamic range and gain staging

A simplified linear model is

$$y[n] = Gx[n] + v[n], \qquad G = G_{RF}G_{IF}G_{BB} \tag{11}$$

where v[n] aggregates thermal, front-end, and quantization noise. Gains lift signals above the floor but reduce headroom, overload manifests as compressed peaks and intermodulation spurs [2], [8]. The UI clamps valid gain ranges and encourages a moderate RF gain, then IF/BB recipe, with band-selected antennas/filters per port when available.

4.6 Wideband coverage by step-tuning and stitching

4.6.1 Step plan and frequency axis

When (fmax – fmin) > fs, the app scans by steps of size ≈ fs and concatenates FFT slices [8], [13]:

$$S = \left\lceil \frac{f_{max} - f_{min}}{f_s} \right\rceil, \qquad f_c{}^{(i)} = f_{min} + \left(i + \frac{1}{2}\right) f_s, \quad i = 0, \dots, S-1 \tag{12}$$

For each step, we compute an N-point FFT frame and map bins to absolute RF using; the stitched axis is:

$$\{f_c{}^{(i)} + f[k]\} \tag{13}$$

4.6.2 Junction artifacts and mitigations

Seams appear because adjacent steps meet at their -3 dB window skirts. We apply:

- DC notch: replace bins $|k - k_0| \leq K$ around DC with neighbor mean (implementation used K=2).
- Edge taper: fade first/last bins (small Tukey) to smooth joins.
- Settle delay: small LO-settle delay before capture.
- Stale-capture guard: identical successive frames are retried/skipped.

4.6.3 Update rate and plotting

Per-refresh time ≈ S (t_{settle} + t_{FFT} + t_{USB}). We cap plotted points by decimating only the display vectors, internal arrays keep full S * N resolution for analytics.

4.7 OperaCake switching logic: manual, frequency, time

OperaCake is a dual-bank RF switch: two primary ports (A0, B0) can each be routed to any of four secondary ports (A1–A4, B1–B4). Each bank behaves like a 1→4 electronically controlled RF switch. In typical SDR workflows this enables rapid selection among antennas, preselectors, filters, or DUT paths while keeping the radio front-end unchanged [5], [6], [17].

4.7.1 Base operating modes

- Manual (static routing)
  - Explicitly connect a primary to one secondary (e.g., A0 – A3) and hold that state until a new command arrives.
  - Use cases: A/B antenna comparisons, controlled calibration runs, directed measurements.
- Frequency-keyed (LO-tracked) routing
  - Provide set of non-overlapping frequency intervals, each mapped to a port. Whenever the receiver center frequency fc enters an interval, the switchs connects the corresponding path.
  - Use cases: Band-specific antennas (FM, LTE, 5G etc.).
- Time-keyed routing
  - Visit ports in a programmed sequence with dwell Tp per port, then repeat.
  - Specify dwell in samples: Tp = N_{dwell}/fs
  - Use cases: Periodic multi-antenna surveys, long-term monitoring across bands.

4.8 Event detection and switching deviation test

We track a masked, per-bin baseline

$$\overline{P_t}[k] = \beta\overline{P_{t-1}}[k] + (1-\beta)P_t[k], \qquad k \in M \tag{14}$$

With β ≈ 0.95, mask M excluding DC ± k and edges.

$$k = \max\left(2, \left\lfloor \frac{N}{256} \right\rfloor\right) \tag{15}$$

4.8.1 ΔdB occupancy (fraction of bins above threshold Δ)

$$occ_t = \frac{1}{|M|} \sum_{k \in M} 1\{P_t[k] - \bar{P}_t[k] \geq \Delta\} \tag{16}$$

4.8.2 Spectral flatness (SFM)

In dB, computed on linearized powers p[k] derived from (global scale cancels).

$$SFM_t = 10 \log_{10} \left(\frac{\exp\left\{\frac{1}{|M|} \sum_k \ln p[k]\right\}}{\frac{1}{|M|} \sum_k p[k]} \right) \tag{17}$$

Values near 0 dB indicate noise-like/OFDM, negative values indicate peaky/tonal [9],[11].

4.8.3 Median/band-mean

Lifts and run-length (longest contiguous span above Δ) provide robustness to tones and bursts.

4.8.4 Decision layer

- Hysteresis on median/mean gates, dwell ≈ 0.4s, cooldown ≈ 0.4s, and a smoothed score must improve over the last switch score to avoid ping-pong.
- Defaults: Δ=6.5 dB, adaptive occupancy gate (scaled with |M|), SFM ≥ -5 dB with slowly tracked reference, median/band lifts 2.0/1.5 dB, span fraction ≥ 0.008.
- On confirmation: red banner, port rotation, brief baseline freeze, then tracking resumes.

This behavior is consistent with CFAR-style thinking but lighter-weight for real-time GUI use [9].

4.9 Time–frequency resolution and latency

To be able to say with close precision the real delay of the OperaCake we conducted two tests to help us.

- Timestamped OperaCake switch: toggle ports using the 'hackrf_operacake' default time switching, noting the time before the command is done, and the time after the command succeeded, for this test the results were a little too slow for our please (10-32ms for Linux and 41-49ms for Windows)

- Visual delay mode: Display a single narrowband tone on one port and broadband/noise on another. Trigger a port change (manual or time-based) while the waterfall runs. Measure the row index difference between the click/trigger and the first visible change, multiply by the configured GUI update time. The visible change always occurs within the next GUI frame; thus, the visual delay is bounded by the GUI timer (≈100 ms in our sinks). By zooming in and indexing points we could observe that the switch was

still instantly made in 1ms range, confirming that hardware switching is effectively instantaneous at GUI time scales.

4.10 Mapping theory to the implementation

This section maps each theoretical element to the concrete objects in the code, and clarifies any deliberate deviations.

Signal path & FFT:

- Blocks: osmosdr.source > block.stream_to_vector(N) > fft_vcc(N, forward=True,window.blackmanharris(N)) > blocks.complex_to_max(N); for the stitched and event detection paths we also attach blocks.probe_signal_vf(N) to read vectors back into Python/Qt.
- Frequency map: Bin-to-Hz mapping follows fftshifted relation and absolute RF. The QtGUI sinks receive fc, fs to label axes consistently.

RBW/ENBW and display normalization

- RBW. The BH4 window implies $ENBW_{min} \approx 1.9\text{-}2.0$ for practical N, so RBW = $ENBW_{min}$ * fs/N.
- Live sacle. The app intentionally uses relative magnitude dB: window > FFT > |X|/N > optional EMA > $20\log_{10}(.) + k$ with a fixed lower floor for plotting stability. Coherent-gain and ENBW corrections are not applied to live displays; this keeps ΔdB spectral-flatness metrics scale-invariant and matches the GUI's semantics.

Averaging & max hold

- QtGUI path. freq_sink.set_fft_average(α) implements a magnitude-domain EMA for the live trace; toggling 'Max hold' enables a running per-bin maximum.
- Stitched patch. For wide scan we maintain a separate linear-domain EMA (sweep_prev_linear, factor sweep_avg_alpha) before converting to dB, so decimation affects only the plot, not the internal trace.

Stitching details

- Step plan. We compute centers fc over [f_{min}, f_{max}] in steps of fs (one FFt per step). The stitched frequency axis is formed by concatenating the baseband bin grid around each center.
- Stream control. DC notch, edge taper and settle & stale-guard
- Decimation. For drawing only, we subsample the stitched array.

OperaCake switching

- Manual. update_antenna(port) calls src.set_antenna(port) and updates the board panel.
- Frequency-based. Port intervals are parsed (MHz) and validated in validate_port_ranges(): ranges must be disjointed and ordered; overlaps or touching endpoints are rejected with inline UI errors. When fc enters an interval, auto_switch_port_if_needed() selects that port deterministically (first match wins).
- Time-based. The UI accepts dwell seconds; we convert with Ndwell=Tpfs. Switching uses non-blocking QTimer; per-port elapsed time persisted so pause/resume preserves proportions.
- During wide sweeps. In 'wide spectrum frequency' mode, selecting a port re-seed the sweep bounds to that port's interval and restarts the step plan after a short settle.

Event detector wiring

- Baseline & mask. check_jamming() builds a mask excluding DC ± k (k = max (2, ⌊N/256⌋) and the two edge bins. A per-bin baseline in dB is tracked with EMA (jam_alpha≈0.05), with a freeze window around candidate events to avoid chasing them.
- ΔdB occupancy: fraction of masked bins above a per-bin baseline by jam_thresh_db (default 6.5 dB) with an adaptive occupancy gate scaled by |M|.
- Spectral flatness (SFM): computed on linearized powers, compared to a slowly tracked reference; gates around −5 dB by default.
- Median/band-mean lifts and longest run (contiguous span) complement occupancy for robustness to tones and bursts.
- Anti-flap logic. Hysteresis (separate on/off thresholds), dwell (~0.4 s hold), cooldown (~4 s), and an improvement constraint: a smoothed score must exceed the score at the last switch before rotating to the next port. On confirmation the banner flashes and update_antenna() advance the port; baselines are briefly frozen and re-armed.
- to avoid accidental edits; PyQtGraph remains zoomable for offline inspection.

5. Implementation

The implementation of the diploma thesis followed a bottom-up exploration and top-down refinement of multiple techniques and technologies. While the final version of the software represents a polished, stable application, the path to this point involved considerable experimentation, debugging, hardware integration, and architectural redesigns. This chapter documents all the implementation attempts, including those that did not result in the final version but were instrumental in shaping the direction of the project.

5.1 Initial Setup and Virtual Machine Environment

The first stage of the implementation involved preparing a suitable software environment for development and testing of Software-Defined Radio (SDR) functionalities using the HackRF One device. Given the extensive compatibility of GNU Radio Companion and HackRF tools with Linux distributions, the decision was made to create a Linux-based development workspace within a virtual machine.

An Ubuntu 22.04 LTS [18] virtual machine was installed using Oracle VM VirtualBox [19], chosen for its widespread support, ease of USB device passthrough, and efficient management of snapshots and system configurations. This setup allowed rapid testing and iteration without risky modifications to the host operating system and provided the ability to restore the system to a clean state in case of misconfiguration or driver conflicts.

Once the operating system was installed, the following essential packages and libraries were configured:

System Update and Core Dependencies

Before installing any packages, the system's package index was updated to ensure access to the latest repository information, using the commands shown in Figure 9:

```
foabi@foabi-VirtualBox:~$ sudo apt update
sudo apt install -y cmake g++ git python3-dev python3-pip \
libboost-all-dev libgmp-dev swig qtchooser qtbase5-dev qtbase5-dev-tools \
libfftw3-dev python3-mako python3-numpy python3-scipy \
python3-apt python3-click python3-click-plugins python3-zmq
```

Figure 9. Terminal command for system preparation to install core build dependencies

Python and Required Libraries

Python 3.10 was already pre-installed on Ubuntu 22.04. Additional Python packages were installed using pip:

```
foabi@foabi-VirtualBox:~$ python3 -m pip install --upgrade pip
python3 -m pip install PyQt5 pyqtgraph matplotlib QDarkStyle
```

Figure 10. Terminal command to install needed python packages

- **GNU Radio Framework**

The main digital signal processing (DSP) framework used to build and test signal processing pipelines in both GRC and Python was installed:

```
foabi@foabi-VirtualBox:~$ sudo apt install gnuradio
```

Figure 11. Terminal command to install DSP framework GNU Radio

- **gr-osmosdr**

To ensure compatibility with HackRF and OperaCake, the gr-osmosdr extension was built manually from the official GitHub source, shown in Figure 12:

```
foabi@foabi-VirtualBox:~$ git clone https://github.com/osmocom/gr-osmosdr.git
cd gr-osmosdr
mkdir build && cd build
cmake ..
make -j$(nproc)
sudo make install
sudo ldconfig
```

Figure 12. Terminal commands for specific version installation of gr-osmosdr extension

- **HackRF Tools**

The command-line utilities used to interact with the HackRF One and the OperaCake RF switch were compiled from source to guarantee support for all hardware features:

```
foabi@foabi-VirtualBox:~$ git clone https://github.com/greatscottgadgets/hackrf.git
cd hackrf/host
mkdir build && cd build
cmake ..
make -j$(nproc)
sudo make install
sudo ldconfig
```

Figure 13. Terminal commands installation of HackRF One Tools

This provides tools like: hackrf_info (device info/debugging), hackrf_transfer (raw I/Q capture and TX), hackrf_operacake (antenna port switching).

The USB passthrough feature in VirtualBox was configured to ensure the HackRF device could be accessed directly by the Ubuntu VM. This required installing the VirtualBox Extension Pack and adding the user to the vboxusers group for permission handling.

After completing all installations, it could be seen that HackRF One SDR and the Opera Cake switch were properly recognized using the 'hackrf_info' command. Figure 14 shows the output of this command, with various parameters of the HackRF One SDR, such as serial number, firmware version, hardware revison and so on. With HackRF recognized and all tools installed, development could now proceed into practical signal experiments and interface prototyping.

```
foabi@foabi-VirtualBox:~$ hackrf_info
hackrf_info version: git-4b8dbfc3
libhackrf version: git-4b8dbfc3 (0.9)
Found HackRF
Index: 0
Serial number: 0000000000000000066a062dc2787509f
Board ID Number: 2 (HackRF One)
Firmware Version: 2024.02.1 (API:1.08)
Part ID Number: 0xa000cb3c 0x006f4f54
Hardware Revision: r10
Hardware appears to have been manufactured by Great Scott Gadgets.
Hardware supported by installed firmware:
    HackRF One
Opera Cake found, address: 0
```

Figure 14. Terminal hackrf_info log

5.2 GRC Flowgraph Prototyping

At the early stages of development, before implementing a custom graphical interface, several flowgraphs were created and tested in GNU Radio Companion (GRC) to validate the basic functionality of the SDR environment and the HackRF One device. These initial prototypes were minimal but essential.

They typically included:

- An Osmocom Source block, configured to interface with HackRF hardware.
- A QT GUI Frequency Sink, providing a real-time frequency spectrum display.
- A QT GUI Waterfall Sink, allowing visualization of the time-varying frequency components.

At this point, the only available hardware consisted of HackRF One paired with a basic antenna, without the OperaCake RF switch or additional modules, connected to a laptop running the Ubuntu operating system and the GRC software program.

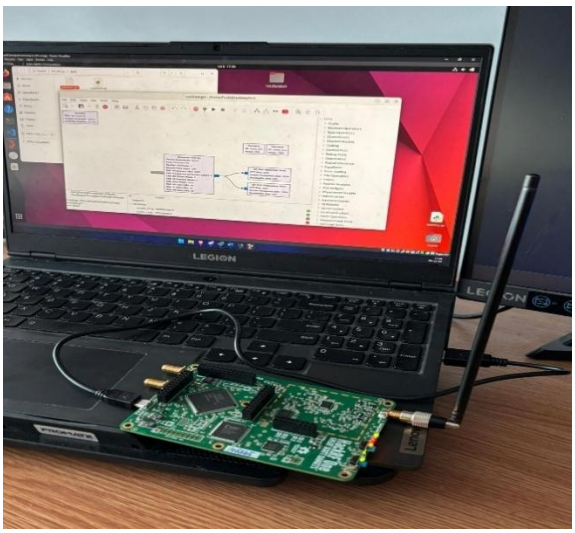

Figure 15. HackRF One connection logic implementation with antenna connected directly to its antenna port

These flowgraphs confirmed that the HackRF One device was correctly recognized and functional, and that the environment was correctly capturing live RF signals from the antenna.

5.2.1 Basic Spectrum Visualization Flowgraph

To visualize live spectrum activity in the environment, a simple GRC flowgraph was constructed, and one antenna was connected directly to HackRF One's SMA Antenna port. This flowgraph included:

- Two variable blocks for sample rate (samp_rate) and central frequency (center_freq).
- An osmocom Source block with the hackrf device argument.
- QT GUI Frequency Sink and QT GUI Waterfall Sink.

Both GUI sinks were configured to reflect the central frequency and sample rate through the variable blocks. The frequency sink control panel was also enabled for real-time adjustments. Figure 16 presents the flowgraph implementation.

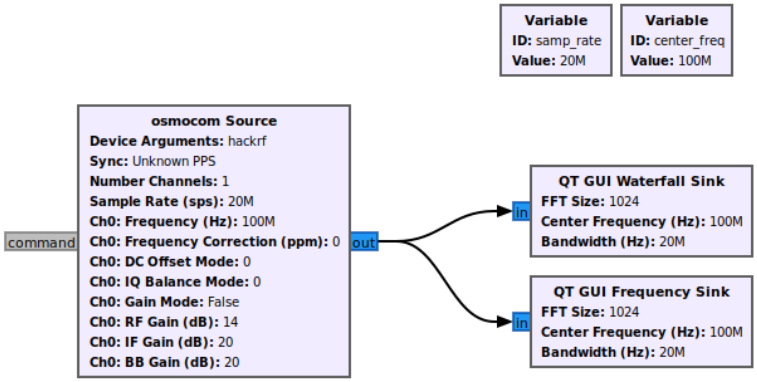

Figure 16. GRC osmocom source + Waterfall and Frequency Spectrum initial blocks

This setup allowed for the initial confirmation of HackRF reception and real-time spectral visualization. Local FM stations and ambient RF signals were clearly visible, as illustrated in Figure 17.

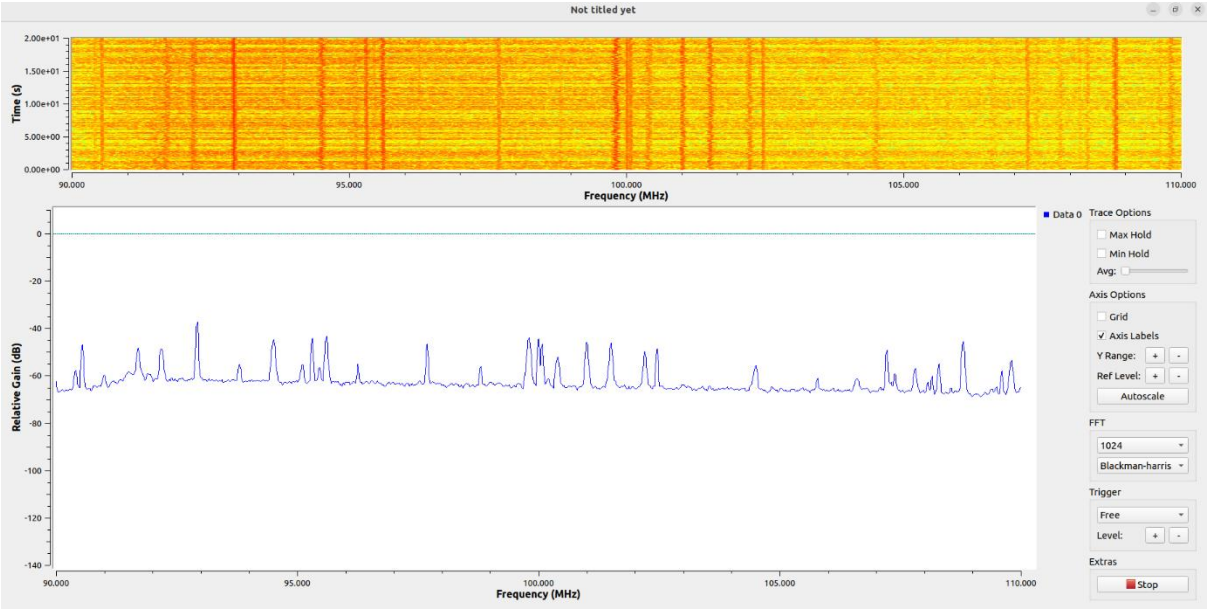

Figure 17. Initial GRC results

By successfully observing these signals, the experiments validated both the hardware setup and the core software components (GNU Radio, gr-osmosdr, HackRF drivers). This stage served as a hands-on introduction to GNU Radio's block-based architecture and provided a reference point for building more complex custom applications in later stages.

5.3 OperaCake Integration

After verifying HackRF One signal reception, we physically integrated the OperaCake RF switch. This is done by removing the HackRF case and mounting the OperaCake RF switch on top of the SDR board, as can be seen in Figure 18. OperaCake provides two primary inputs (A0 and B0) feeding two banks (A and B) of secondary outputs (A1–A4, B1–B4). This lets us dynamically route a primary input to a chosen secondary and automated multi-antenna selection, useful for tasks like delay analysis and event detection and switching.

Figure 18. OperaCake integrated with HackRF One

Initially, OperaCake control was performed via the terminal using the 'hackrf_operacake' utility. The device supports three operating modes: manual, frequency, and time, each offering distinct switching behavior suited to different applications [6].

5.3.1 Manual Switching

In manual mode, the user explicitly specifies the active secondary port using the -a or -b option.

hackrf_operacake -m manual -a A4 (connects A0 to A4 and B0 to B1, based on OperaCake's internal logic). This is the default operating mode upon startup. Alternatively, ports can be specified by number:

- Ports 0-3 correspond to: A1, A2, A3, A4
- Ports 4-7 correspond to: B1, B2, B3, B4.

Both naming conventions were tested and worked as expected. For example:

hackrf_operacake -m manual -a 3 (connects A0 to A4 since index 3 = A4)

Manual switching is useful for static testing scenarios or initial verification of individual antenna paths. However, one limitation observed was that when launching GNU Radio flowgraphs simultaneously, port switching could not be performed dynamically during signal acquisition, a constraint that motivated automation in later modes.

5.3.2 Frequency Switching

Frequency mode allows the OperaCake to switch ports automatically based on the tuned central frequency of the HackRF. This is ideal for scenarios involving antennas or filters optimized for different frequency bands. To use frequency mode, the user specifies multiple port–frequency-band mappings, such as:

hackrf_operacake -m frequency -f A4:100:200 -f B4:300:400 -f A1:250:280

This instructs OperaCake to switch to port A4 when the SDR's central frequency is between 100-200MHz, to port B4 for 300-400MHz and to port A3 for 250-280MHz. Only the A0 connection is explicitly specified, the B0 port is automatically mirrored to match the secondary port on the opposite side. For example, when A0 connects to B2, B0 connects to A2.

Priority is determined by the order of definitions: the first matching band is used. Once configured, frequency switching continues autonomously until explicitly changed or the HackRF is reset.

5.3.3 Time Switching

In time-based switching mode, OperaCake automatically cycles through a sequence of ports based on dwell times defined in units of samples by the user. For example:

hackrf_operacake -m time -t A4:100000000 -t B4:100000000

This instructs OperaCake to switch to port A4 for 100 million samples, then to B4 for another 100 million samples, looping indefinitely. The actual duration in seconds depends on HackRF's sample rate, given by the formula:

$$Duration\ (s) = \frac{Sample\ Count}{Sample\ Rate}$$

If using a 20MS/s sample rate, 100 million samples equal to 5 seconds of dwell time. When all ports share the same duration, the -w option can simplify the syntax:

hackrf_operacake -m time -w 100000000 -t A1 -t A2 -t B3 -t A4

As with frequency mode, only the A0 port is specified, and B0 mirrors it to the secondary port opposite of A0.

5.4 Delay Estimation

Before implementing a real-time GUI for antenna switching and RF monitoring, several methods were tested to estimate system-level delay introduced during switching events. Although the goal was to approximate hardware latency, early approaches primarily measured delays induced by the software stack, including command execution, buffering, and GUI rendering.

5.4.1 Delay Calculation with GRC Flowgraphs

A pair of GNU Radio Companion flowgraphs were built to attempt a visual method of analyzing delay after port switching:

- Flowgraph 1 (Capture Mode):

  Used an Osmocom Source block set to HackRF and record live I/Q data into a file named result.iq using File Sink block. This data captured environmental RF signals during antenna switching moments.

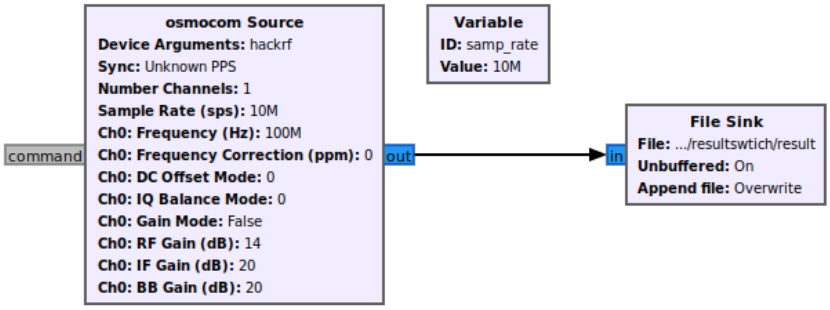

Figure 19. GRC capture flowgraph for I/Q data

- Flowgraph 2 (Playback Mode):
  Read the saved I/Q file with a File Source block, throttled the output to simulate real-time playback, and visualized it using a QT GUI Waterfall Sink.

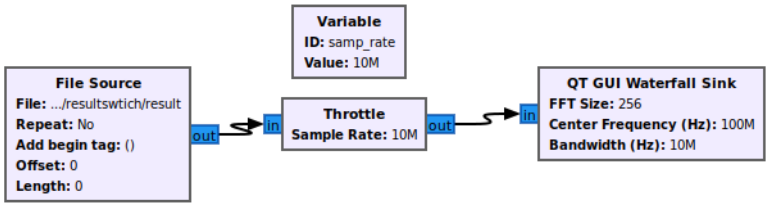

Figure 20. GRC playback flowgraph to replay I/Q data into a waterfall spectrogram

These flowgraphs provided a coarse method for estimating switching delay by visually inspecting when bursts appeared. However, this method lacked synchronization and quantitative accuracy:

- The time between switching command and visible burst was ambiguous.
- Waterfall renders delay further distorted the results.
- No timestamping or event logging was integrated.

5.4.2 Python-Based Switching and Timestamp Logging

To achieve finer-grained delay analysis, a pair of Python applications were developed:

- latency_calc.py (Linux/Unix)
- latency_calc_wind.py (Windows)

These scripts used Python's subprocess and datetime modules to:

- Send manual switching commands to OperaCake
- Log precise timestamps before and after each switch call command
- Compute the duration of each switch and store the results for later inspection

Each script ran for 50 iterations with 1.5-second intervals between commands to ensure isolation of each measurement. The switching logic was encapsulated in a switch_port() function, which captured the elapsed time for each individual command execution in milliseconds.

Although both scripts provided a consistent method for collecting software-induced delay data, the Windows version (latency_calc_wind.py) yielded highly variable results. USB stack behavior and the overhead of launching subprocesses under Windows led to larger and inconsistent delays. These discrepancies further motivated the decision to conduct all further development and experimentation under Linux, where timing behavior proved more stable and reproducible.

Ultimately, these Python-based measurements demonstrated that software-only timestamping was insufficient for capturing the true RF-level or hardware-level delay involved in antenna switching. There was no access to synchronized timestamps between the switching command and the actual RF response in GNU Radio. This realization led directly to the next phase of the project: the development of a custom PyQt5-based GUI, which would allow real-time switching control, signal visualization, and more precise integration between command execution and spectral observation.

5.5 Graphical Interface Foundation

Following the initial testing phase using GRC flowgraphs and command-line switching via hackrf_operacake, it became clear that these approaches were insufficient for the real-time requirements of the project.

While GRC successfully validated HackRF reception and basic spectrum visualization, it lacked the flexibility to dynamically switch antennas during active streaming. Additionally, no accurate hardware delay analysis could be achieved at this point.

5.5.1 Initial QtGUI-Based GNU Radio Integration

The first implementation used GNU Radio's built-in QtGUI sink blocks directly within a custom Python script. A top_block class flowgraph was defined to instantiate and connect the HackRF One source to two primary GUI components:

- qtgui.freq_sink_c - for continuous visualization of the frequency domain
- qtgui.waterfall_sink_c - for continuous visualization of spectral domain through a cascading display

These GUI sinks provided a visual foundation for observing RF signals in the environment and confirming the correct reception pipeline. The parameters for central frequency and sample rate were manually configured in code or using predefined variables.

At this stage, antenna switching was not yet integrated into the runtime environment. The switching logic had to be executed externally through terminal commands using hackrf_operacake, which meant any port changes required pausing the SDR stream or restarting the flowgraph. This

limitation made it impractical to analyze real-time signal variation across antennas or to automate switching behavior within the same pipeline.

5.5.2 Antenna Port Integration in Osmocom Source

A breakthrough occurred when a contributor updated the gr-osmosdr GitHub repositoyory to support the antenna parameter for HackRF devices [20]. With this update real-time control of the active antenna port became possible directly through the osmosdr.source block using the command:

self.antenna select = 'A4'

This eliminated the need for external hackrf_operacake command-line calls and enabled real-time antenna switching without interrupting the SDR stream. With this improvement, I began experimenting with switching logic integrated directly into the flowgraph, opening the way for more interactive control structures.

To make this functionality more accessible, the next step was to develop a basic GUI launcher that allows users to use different operating modes as detailed in the following section.

5.5.3 Basic GUI Launcher Window

To provide a centralized control interface for all operational modes, a minimal PyQt5-based launcher window was implemented. This program, named GUI.py, serves as the frontend that lets users select which operating mode to run. Upon selection, GUI.py calls and instantiates the core spectrum analyzer logic implemented in live_spectogram.py. The initial version included:

- A vertical layout with three buttons: Manual Switching Mode, Frequency Switching Mode, and Time Switching Mode
- A status label to indicate whether a mode was currently active or idle
- Basic signal-slot connections using *QPushButton.clicked.connect()* for launching different signal visualization modes

Each button triggered the creation of a new LiveSpectrogramWindow instance with a corresponding mode argument. For example, the manual switching mode was defined as shown in Figure 21:

```
self.live_spec_btn = QtWidgets.QPushButton("Manual Switching Mode")
self.live_spec_btn.clicked.connect(lambda: self.launch_mode("manual"))
self.layout.addWidget(self.live_spec_btn)
```

Figure 21. Creation of Manual Switching Mode button

The launch_mode() method includes a guard clause to prevent launching multiple overlapping windows:

```
def launch_mode(self, mode):
    if self.active_window is not None:
        return
    self.active_window = LiveSpectrogramWindow(mode=mode)
    self.active_window.closed.connect(self.on_window_closed)
    self.active_window.show()
```

Figure 22. Prevent opening multiple modes at the same time

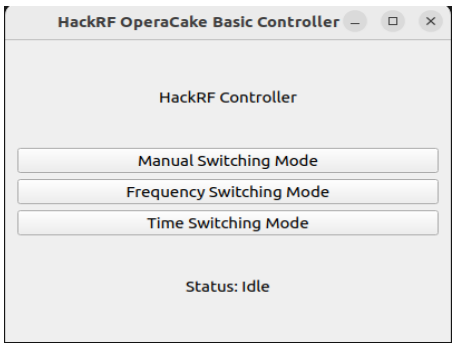

Figure 23. Initial GUI for the application's main menu

This initial GUI design was visually minimal and focused on functionality over aesthetics, serving as a reliable way to launch specific configurations of the live SDR application. This separation allowed each component to evolve independently.

5.5.4 Unified SDR Parameter Configuration

To make the application adaptable for different scanning scenarios and RF conditions, a unified parameter control panel was implemented. This control interface allowed the user to configure essential SDR settings before initiating a signal acquisition session, including:

- Central frequency
- Sample rate
- FFT size
- Gain settings (RF, IF, BB)
- Antenna port selection (A1-A4, B0-B4)
- Max hold
- Averaging

These parameters were exposed through editable QLineEdit fields, grouped at the top of each mode's interface. Values entered by the user were parsed when the Start button was pressed, triggering dynamic configuration of the SDR stream via a central method named: *self.apply_sdr_config().*

This method was responsible for:

- Applying central frequency and sample rate to osmosdr.source
- Setting antenna via *update_antenna()*
- Updating FFT size for both spectrum and waterfall sinks
- Synchronizing all SDR elements before execution

An excerpt from the SDR reconfiguration function:

```
def apply_sdr_config(self):
    self.src.set_sample_rate(self.samp_rate)
    self.src.set_center_freq(self.center_freq)
    self.src.set_gain(self.rf_gain, 0)
    self.src.set_if_gain(self.if_gain, 0)
    self.src.set_bb_gain(self.bb_gain, 0)
    self.src.set_antenna(self.antenna_select, 0)
```

Figure 24. Toolbar function to synchronize SDR elements

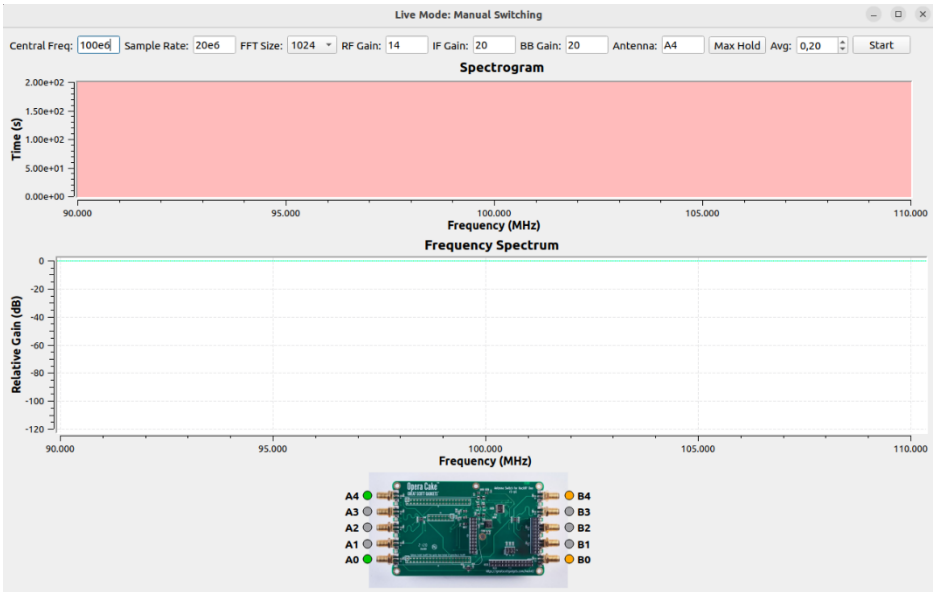

Figure 25. Manual Switching mode window

At this stage, there was no validation or error handling, values were assumed to be correct and within acceptable ranges.

In addition, to improve input flexibility and ensure compatibility with common engineering notation, the following parser function was included in the program. This design gives users the freedom to use various representations such as:

- 1e6 or 1000000
- 1MHz or 1mhz
- 1000khz or 1e3*1e3

```
_FREQ_REGEX  = re.compile(r"^\s*([+-]?(?:\d+(?:\.\d+)?|\.\d+)(?:e[+-]?\d+)?)\s*(hz|khz|mhz|ghz)?\s*$", re.I)
_FLOAT_REGEX = re.compile(r"^\s*[+-]?(?:\d+(?:\.\d+)?|\.\d+)(?:e[+-]?\d+)?\s*$")
_INT_REGEX   = re.compile(r"^\s*\d+\s*$")

def parse_freq_input(text: str) -> float:
    """Strict: number (allows sci notation) + optional unit {Hz,kHz,MHz,GHz}."""
    if text is None:
        raise ValueError("Empty frequency")
    m = _FREQ_REGEX.match(text)
    if not m:
        raise ValueError(f"Invalid frequency format: '{text}'")
    val = float(m.group(1))
    unit = (m.group(2) or "").lower()
    mult = {"":1.0, "hz":1.0, "khz":1e3, "mhz":1e6, "ghz":1e9}[unit]
    return val * mult
```

Figure 26. Regex validation to accept numbers in scientific notation and common units

5.6 Manual Switching Mode

The manual switching mode was the default mode launched by the application and is visually represented in section 2.5.4. This interface allows the user:

- Manually change the antenna port via a text field
- Start/Stop the SDR stream
- Observe real-time spectrum and spectrogram updates with chosen parameters

Internally, port changes were performed using the built-in *update_antenna()* method, avoiding any external command-line switching tools. Because it offered full control over the SDR while maintaining a stable signal path, this mode was foundation for:

- Real-time manual inspection of different antennas
- Early delay and jamming behavior comparisons
- Validating the effectiveness of live spectrum switching

Although later modes automated the switching logic (e.g., frequency-based and time-based), this manual setup remained essential for debugging and verifying signal transitions.

5.7 Frequency Switching Mode

The Frequency Switching Mode was developed to automatically switch the active antenna port based on the SDR's central frequency. This approach simulated use cases where specific frequency bands were tied to designated antennas, enabling intelligent port selection without user intervention during real-time signal monitoring.
The mode was implemented using the following core components:

- Eight editable fields corresponding to the OperaCake ports: A4, A3, A2, A1, B4, B3, B2, B1
- Each field accepted a frequency range in MHz format (100:300) corresponding to when that antenna should be active
- The current central frequency of the SDR was continuously monitored during execution

```python
def auto_switch_port_if_needed(self):
    if not self.validate_port_ranges():
        print("[ABORT] Port ranges are invalid — please fix them first.")
        return

    current_mhz = self.center_freq / 1e6
    selected_port = None

    for port, line in self.port_inputs.items():
        text = line.text().strip()
        if not text:
            continue
        try:
            parts = text.replace(" ", "").split(":")
            low = float(parts[0])
            high = float(parts[1])
            print(f"[DEBUG] Checking port {port} for range {low}–{high} MHz")

            if low <= current_mhz <= high:
                selected_port = port
                break
        except Exception as e:
            print(f"[WARN] Bad format in {port}: '{text}' — {e}")

    if selected_port:
        if selected_port != self.antenna_select:
            print(f"[AUTO-SWITCH] Switching to {selected_port} for {current_mhz:.2f} MHz")
            self.antenna_edit.setText(selected_port)
            self.update_antenna()
        else:
            print(f"[SKIP] {selected_port} already selected.")
    else:
        print(f"[INFO] No matching port for {current_mhz:.2f} MHz")
```

Figure 27. Frequency-based automatic port selection (parsing, validation, match and switch)

In the method shown above, the application continuously compared the current central frequency against the user-defined thresholds during active operation. If the frequency fell within the range assigned to a port, a match was found, and the system would call:

update_antenna() which in turn calls *self.src.set_antenna()*

This ensures that the antenna switch occurs immediately and seamlessly, without interrupting the SDR stream or requiring manual intervention.

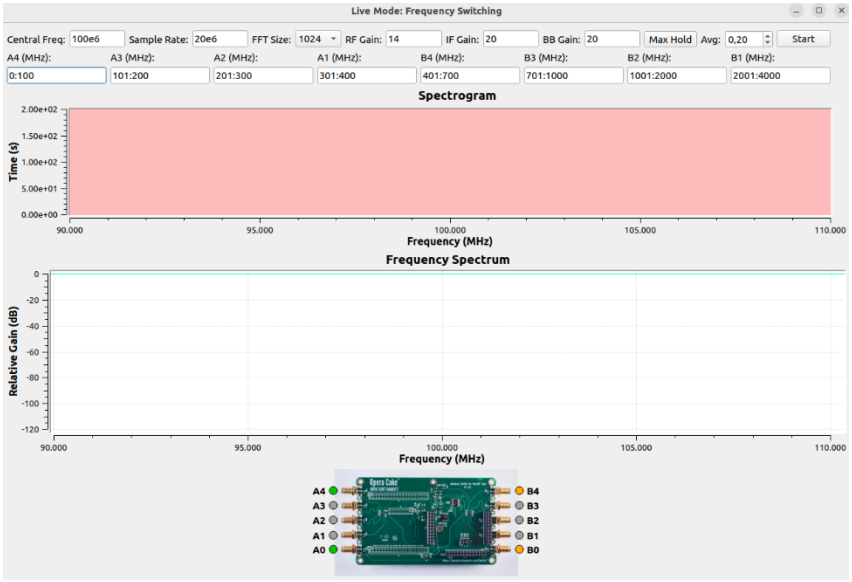

Figure 28. Frequency Switching mode window

The visual layout is similar between modes, but with added fields for port-range mapping and disabled manual antenna selection. This design made a frequency-based control intuitive and fully integrated within the live SDR GUI environment.

5.8 Time Switching Mode

The Time Switching Mode was developed to automatically cycle through a predefined sequence of antenna ports, each active for a user-specified duration. This mode was especially useful for comparing signal conditions across multiple antennas without manual intervention, enabling continuous round-robin scanning. The user interface consisted of:

- Editable port-duration pairs (A4:3, B1:2) defined in seconds
- An internal list structure to store valid tuples like [("A4", 3), ("B1", 2)]
- A Start/Stop toggle button to initiate or halt timed switching

Once the stream was active and the user pressed Start, the application entered a loop managed by a QTimer object. Each time the timer expired, the system automatically advanced to the next port in the sequence.

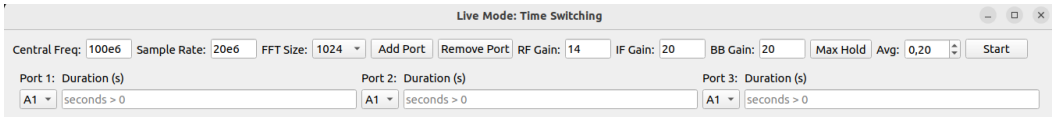

Figure 29. Time Switching mode window

Compared to the other modes here we have 2 new button options: Add Port and Remove Port which in the following figure you can see we can add as many ports as you wish for the cycle to pass through and you can even remove all of them if you want to do something else.

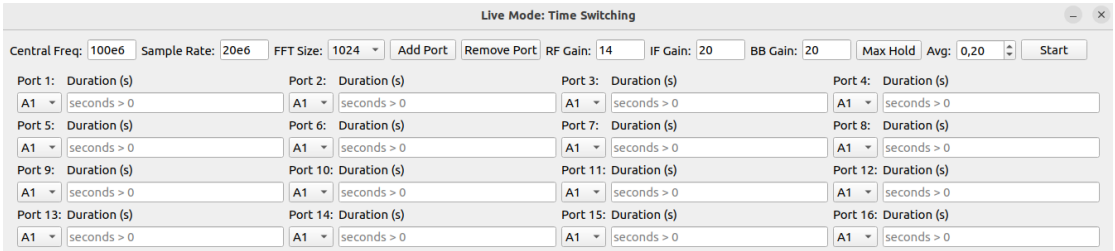

Figure 30. Time switching configuration panel with multiple editable port-duration rows

Additional Implementation Highlights:

- The loop was designed to wrap around, restarting from the beginning once the end of the sequence was reached.
- Edge cases, such as invalid durations, or empty configurations, were caught and prevented from running.
- Switching intervals were consistent and non-blocking, ensuring uninterrupted visualization.

5.9 Delay Mode

After validating switching logic and real-time control through earlier modes, the Delay switching was developed to experimentally estimate the hardware-level delay between antenna input switching and signal arrival. Unlike previous attempts (such as the GRC file playback or Python timestamping), this mode provided visual and interactive feedback for more precise delay observations.

The main goal was to observe the time lag between switching antenna ports and see the corresponding signal appear in the spectrogram. To achieve this, the interface was simplified and tuned for short-duration signal bursts and rapid switching:

- Only the QtGUI Waterfall Sink was displayed (no frequency spectrum).
- The waterfall was configured to show only ~100 milliseconds of scrolling history.
- Manual switching of antenna ports was used to isolate delays.

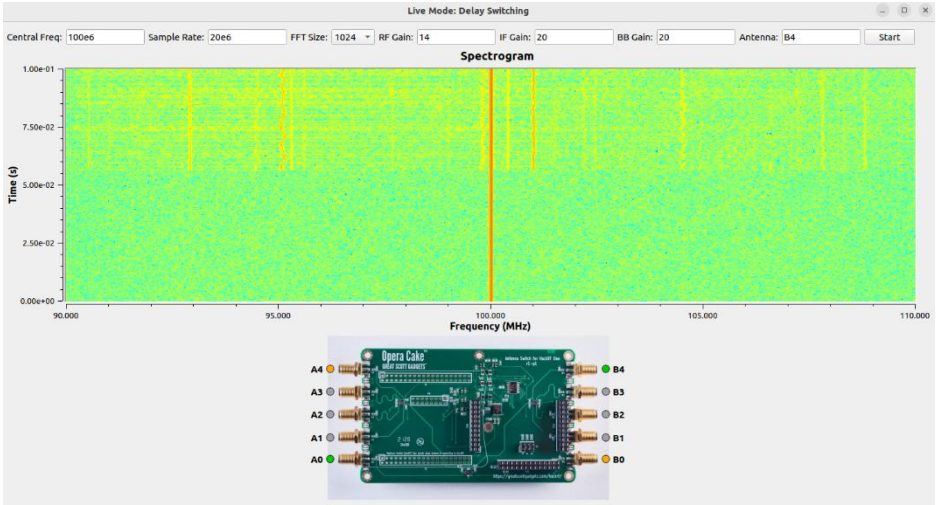

Figure 31. Delay switching UI, transition captured switching antenna ports

This is what Delay switching looks like and as you can see on the time axis it displays only 100ms of scrolling history. This frame was obtained while running on port A4 with an antenna connected then switched to port B4 with no antenna such that we can easily see the difference between signals.

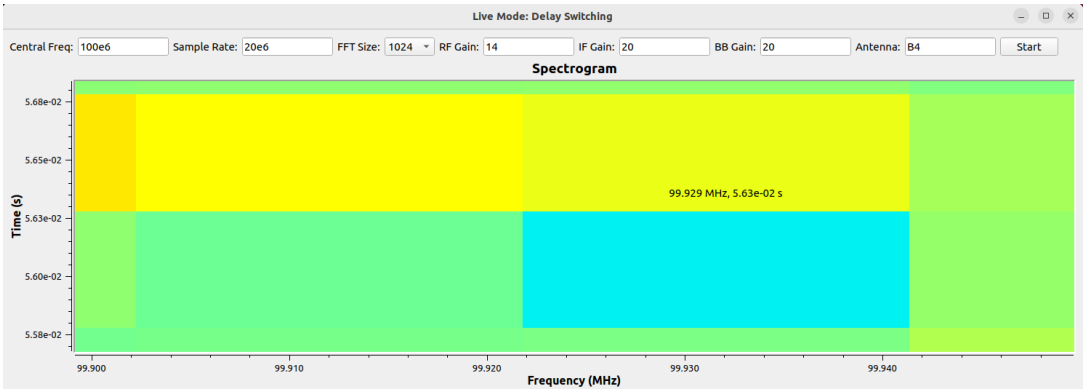

Figure 32. Delay switching zoomed in spectogram

This zoomed-in spectogram makes it possible to visually detect transitions that occur during port switching, showing the moment of transition from an active antenna (the highlighted region) to an inactive antenna port offering a much clearer picture of hardware-induced latency.

This mode does not produce numeric delay values directly, it serves as a powerful visual aid for estimating delays, verifying switching behavior, and validating SDR responsiveness in real time.

However, due to the limitations of display resolution and FFT update rate, the absolute accuracy of delay estimation is still constrained. Therefore, this mode is best interpreted as a qualitative tool that demonstrates the near-instantaneous nature of hardware switching, rather than a precise measurement method.

5.10 Wide Spectrum Mode

With this mode I try to recreate the hackrf_sweep functionality to be able to visualize bandwidths wider than the instantaneous 20 MHz of the HackRF by stitching multiple FFT windowds across a user-defined span. For a range [f_{start}, f_{end}], the center frequency is stepped in increments of the sample rate. After a short settle, one spectrum is captured per step and concatenated into a single wideband trace.

```
self.center_freqs = [sweep_start + (i + 0.5) * step for i in range(num_steps)]
self.total_steps  = len(self.center_freqs)
self.total_bins   = self.total_steps * self.fft_size


bin_freqs = np.linspace(-step / 2.0, step / 2.0, self.fft_size, endpoint=False)
self.freq_axis = np.concatenate([bin_freqs + cf for cf in self.center_freqs])
```

Figure 33. Build center frequency per sample-rate chunk and create the frequency axis by concatenating the basebands

The DSP chain uses a Blackman-Harris window, FFT magnitude scaling, a small DC notch and light edge tapering to reduce stitching artifacts.

```
raw = self.probe.level()
mags = np.asarray(raw, dtype=np.float32)
mags *= (1.0 / float(self.fft_size))

c = self.fft_size // 2
k = 2
if self.fft_size >= 16:
    loL, loR = max(0, c - (k + 3)), max(0, c - (k + 1))
    hiL, hiR = min(self.fft_size, c + (k + 1)), min(self.fft_size, c + (k + 3))
    neigh = []
    if loL < loR: neigh.extend(mags[loL:loR])
    if hiL < hiR: neigh.extend(mags[hiL:hiR])
    if len(neigh) >= 2:
        mags[max(0, c - k):min(self.fft_size, c + k + 1)] = float(np.mean(neigh))

s = self.sweep_ptr * self.fft_size
e = s + self.fft_size
self.sweep_buffer[s:e] = mags
```

Figure 34. Read and scale each FFT's magnitude per frame, DC notch using neighbor mean then write the window into the stitched buffer

While the on-screen trace applies optional exponential averaging and peak-hold for readability, downsampling is performed only for plotting efficiency, the internal accumulation retains the full spectral resolution of each step.

```
mask  = (self.freq_axis >= start) & (self.freq_axis <= end)
idx   = np.flatnonzero(mask)
stride = int(np.ceil(idx.size / max_bins))
sel = idx[::stride]
y_lin = self.sweep_buffer[sel]

a = float(np.clip(getattr(self, "sweep_avg_alpha", 0.20), 0.0, 1.0))
prev = getattr(self, "sweep_prev_linear", None)
sm_lin = (1.0 - a) * prev + a * y_lin
self.sweep_prev_linear = sm_lin

db = 20.0 * np.log10(np.clip(sm_lin, 1e-12, 1e9)) + float(getattr(self, "sweep_db_cal", 0.0))
db = np.maximum(db, -140.0)

self.pg_curve.setData(x_hz / 1e6, db)
self.sweep_ptr = (self.sweep_ptr + 1) % len(self.center_freqs)
```

Figure 35. Downsample display vectors, exponential moving average for plotting efficiency, conversion to dB and floor limitation, then plot and advance to the next step

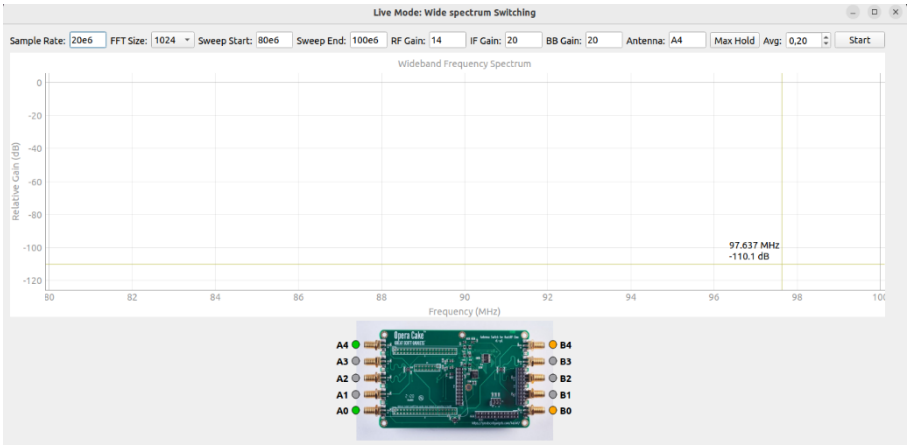

Figure 36. Wide Spectrum Mode Window

As you can see from this window, the central frequency field isn't present any more since here we input the frequency starting point in Sweep Start field and the ending point in Sweep End field.

5.11 Wide Spectrum Frequency Mode

Going from the initial Wide Spectrum Mode, this port sweep is just an extension of it, with ranges bound to the OperaCake switch, enabling hardware-level path selection during scanning. Each port (A4…B1) is assigned a [f_{start}, f_{end}] interval, selecting a port immediately rebuilds the sweep plan for that interval.

```
sel = str(port).upper()
lo_mhz, hi_mhz = map(float, self.port_inputs[sel].text().strip().replace(" ", "").split(":"))
assert hi_mhz > lo_mhz, "upper bound must be > lower bound"

self.selected_port = sel
self.antenna_select = sel
self.oc_panel.set_active(sel, fixed_input="A0", connected=self.is_hackrf_connected())
try:
    self.src.set_antenna(sel, 0)
    self.antenna_changed.emit(sel)
except Exception:
    pass

if self.validate_sweep_inputs(quiet=quiet) and start_immediately and self.toggle_btn.isChecked():
    try: self.sweep_timer.stop()
    except Exception: pass
    self.sweep_ptr    = 0
    self.sweep_active = True
    Qt.QTimer.singleShot(int(getattr(self, "_sweep_settle_ms", 70)), self.begin_sweep_timer)
```

Figure 37. Read and validate the selected port's range, update Operacake panel and switch RF path then recomputes sweep plan for the specific port and start the timer

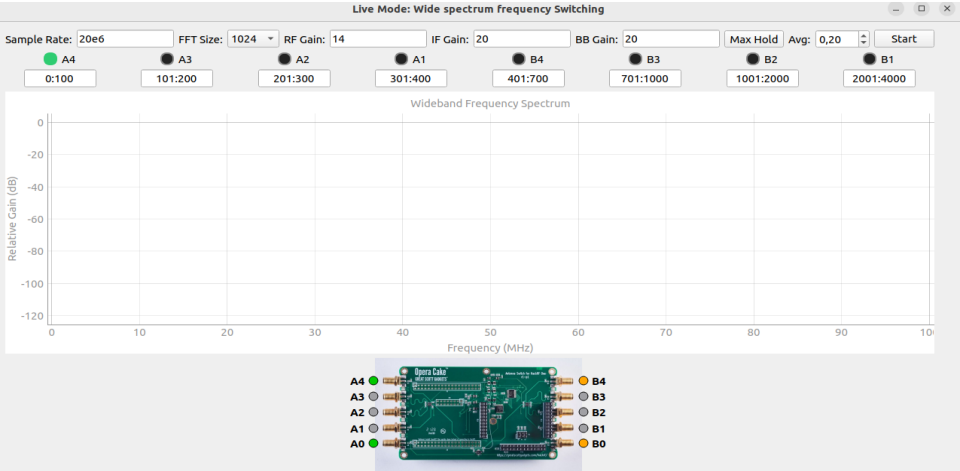

Figure 38. Wide Spectrum Frequency Mode window

Compared to Fig.4 this mode removed the Antenna; Sweep Start and Sweep End fields and brought back the port ranges fields from the original Frequency Mode.

5.12 Event Detection and Switching Mode

With this implementation we follow to detect interference by comparing the current spectrum against a baseline exponential moving average (EMA), after masking DC and edge bins. A detection event requires a combination of metrics:

- ΔdB occupancy above baseline over a sufficient fraction of bins
- spectral flatness to characterize wideband/OFDM-like energy

$$SFM_{dB} = 10\log_{10}(\frac{gmean}{amean})$$

- median/band-mean power lifts with hysteresis to prevent flapping

```python
def spectral_flatness_db(power_lin, eps=1e-12):
    x = np.clip(power_lin, eps, None)
    gmean = np.exp(np.mean(np.log(x)))
    amean = np.mean(x)
    sfm = gmean / (amean + eps)
    return 10.0 * np.log10(sfm + eps)
```

Figure 39. Flatness helper used by the detector

When the smooth score exceeds its switching threshold and holds for a short dwell, the system rotates to the next antenna port, displays a visual alert, and freezes the baseline briefly before resuming EMA updates (with a cooldown to avowing ping-pong). Thresholds for ΔdB occupancy, and SFM are user-tunable, allowing the detector to be adapted to noise conditions and operational risk tolerances.

For Wi-Fi, we found the following UI settings effective:

- Balanced: ΔdB = 6.5, Occ = 0.22, SFM≥ = −5.0 dB.
- Conservative: ΔdB = 8.0, Occ = 0.28, SFM≥ = −4.0 dB.
- Sensitive: ΔdB = 5.0, Occ = 0.16, SFM≥ = −7.0 dB.

These presets trade sensitivity vs. false positives in bursty Wi-Fi traffic without changing the algorithm.

Figure 40. Event detection and switching mode UI

5.13 Real-Time Enhancements and Error Handling

After the core functionality for each mode was implemented and tested, several improvements were introduced to enhance application robustness, user experience, and system stability. These refinements were necessary to ensure that the system could handle invalid user input, prevent crashes during rapid interaction, and allow real-time configuration changes without interrupting the signal processing flow.

The most notable enhancements included:

- Dynamic SDR Reconfiguration, apply_sdr_config() method was adapted to support real-time updates of parameters such as sample rate, central frequency, gains, FFT size, and antenna port. This enabled users to change configuration settings while the flowgraph was running.
- Port Switch Debouncing, to avoid redundant or unstable switching operations, a simple debouncing mechanism was introduced. If the same antenna was already active, the system would skip the switching operation.
- Mode Cleanup Logic Each mode was given its own cleanup function, such as:

```
if self.mode == "time" and hasattr(self, "timer"):
    print("[CLEANUP] Stopping time-switching timer on window close")
    self.timer.stop()
    self.timer_start_time = None
```

Figure 41. Logic to ensure clean stop/disconnect/reset of UI elements

These ensured timers and dynamic UI elements were properly stopped, disconnected, or reset when a user exited the mode. This prevented residual callbacks from affecting the next session.

- Parameter Validation and Sanitization, values entered for central frequency, port fields, gains and sample rates were no longer blindly accepted. Validation logic was added to:
    - Clamp frequencies within supported HackRF range
    - Check for empty or malformed port entries
    - Validate time durations in the Time Switching Mode
    - Detect invalid frequency formats before conversion
- Inline GUI Error Messaging Instead of printing all errors to the terminal, the GUI now featured a dedicated warning label (ex. show_temporary_label("freq_range_label") which showed validation messages directly to the user, color-coded and styled for visibility:

```
if freq <= 0:
    freq = 1.0
    self.freq_edit.setText("1")
    self.show_temporary_label("freq_range_label", "0 < fc <= 4e9. Set to to 1 Hz.")
elif freq > 4e9:
    freq = 4e9
    self.freq_edit.setText("4e9")
    self.show_temporary_label("freq_range_label", "0 < fc <= 4e9. Set to to 4e9 Hz.")
```

Figure 42. GUI error condition and display warning label

And two more examples would be in time switching mode if you remove all ports and try to start you get the warning and the flow won't start.

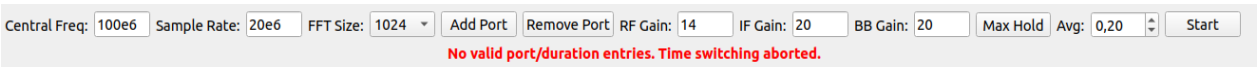

Figure 43. GUI error display warning for time switching mode

Or in frequency switching mode if 2 ranges overlap the program won't enable the start with this error.

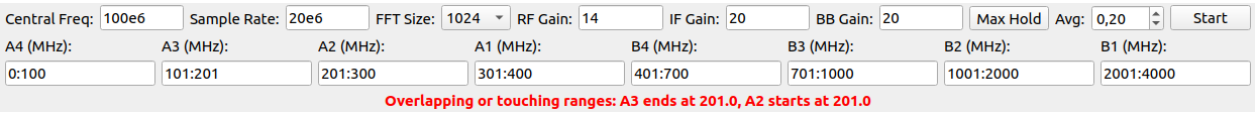

Figure 44. GUI error display warning for frequency switching mode

5.14 Overall Application Architecture

The final application was structured into two major Python modules:

- GUI.py is the main graphical launcher that allows users to select the desired mode. To improve usability and professional appearance, I redesigned this launcher to adopt a clean and modern aesthetic look including custom button icons, color theme, and responsive layout handling + a nice look to the actual operacake board with its ports.
- live_spectogram.py – the backend logic that contains the LiveSpectrogramWindow class responsible for SDR configuration, antenna switching behavior, real-time GUI updates, and mode-specific logic.

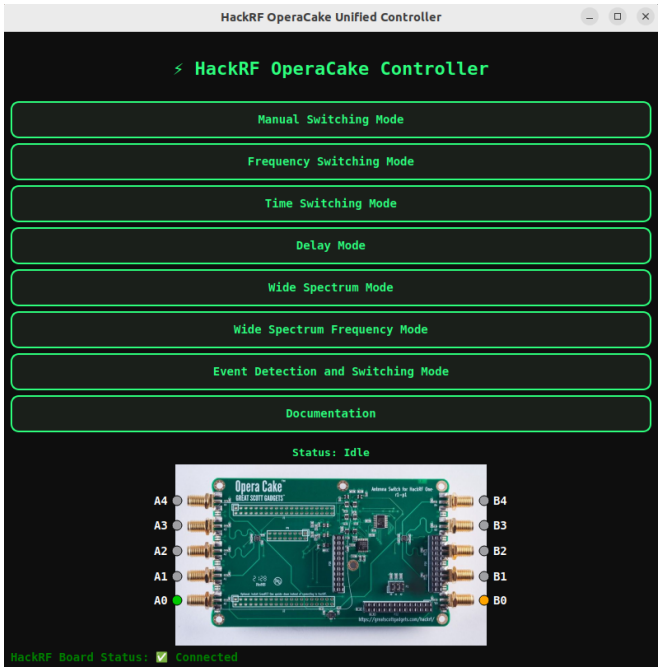

Figure 45. Final Graphical User Interface

Fig. 8 presents the final GUI launcher for the HackRF OperaCake Controller, showing mode selection buttons for all capabilities of the program.

Most modes reused the unified SDR parameter control panel introduced earlier, reducing duplication and improving consistency. Resource cleanup and graceful shutdowns were handled inside the closeEvent() method.

This iterative and layered approach enabled the transformation of a single. grc test file into a complete, multi-mode, cross-platform SDR application with robust real-time interaction suitable for both research experiments and academic demonstration.

6. Experimental Results

This chapter presents a structured walkthrough of the testing procedures and observations conducted with the final application developed for this thesis. The goal is to demonstrate the functionality, stability, and real-time performance of the system across all implemented modes. Tests were performed using the HackRF One and OperaCake switch, running in a controlled environment within a virtual machine on Ubuntu 22.04.

6.1 Pre-GUI Experiments and Validation

6.1.1 Basic Spectrum Visualization

The first experiment was designed to validate HackRF One's reception capability using a simple GNU Radio Companion flowgraph. A single antenna was connected directly to HackRF, and the flowgraph included a QT GUI Frequency Sink and QT GUI Waterfall Sink to observe RF activity in real time.

The initial parameters for all testing which are to come are:

- Central frequency = 100MHz
- Sample rate = 20 MS/s
- FFT size = 1024
- RF Gain = 14, IF Gain = 20, BB Gain = 20

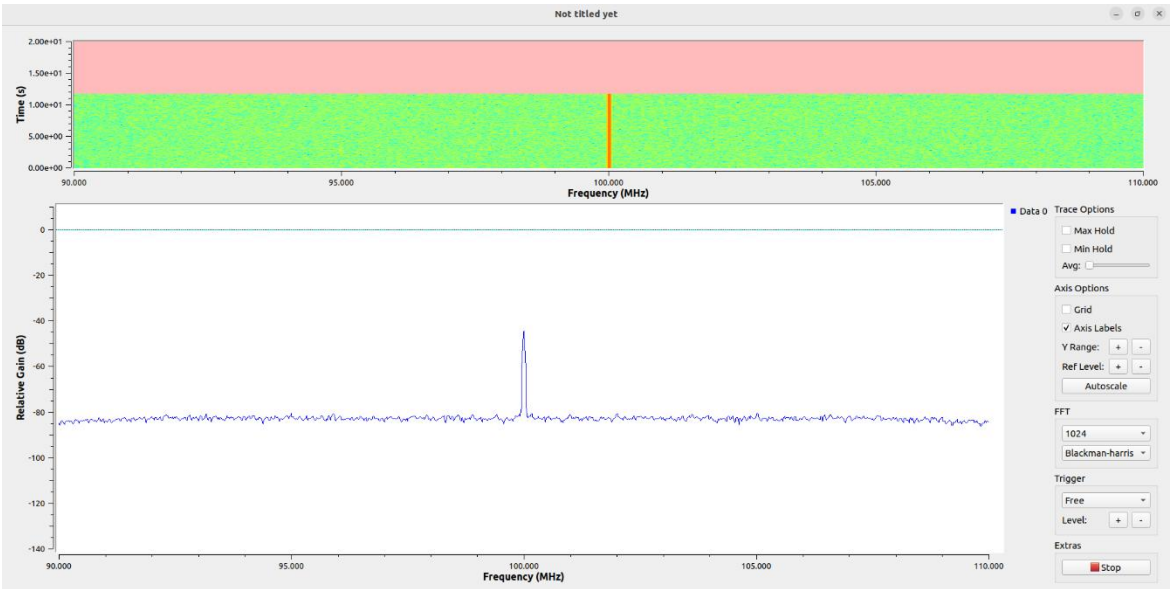

Figure 46. HackRF no antenna connection to board's ANTENNA Port

Even though no antenna is connected, a prominent spike is visible at the central frequency. This is a known artifact of direct-conversion SDR's, such as HackRF One. It results from the oscillator leakage and DC offset, which cause a false signal to appear, but this can be ignored. Figure 47 shows that when connecting an antenna, multiple narrowband carriers are visible.

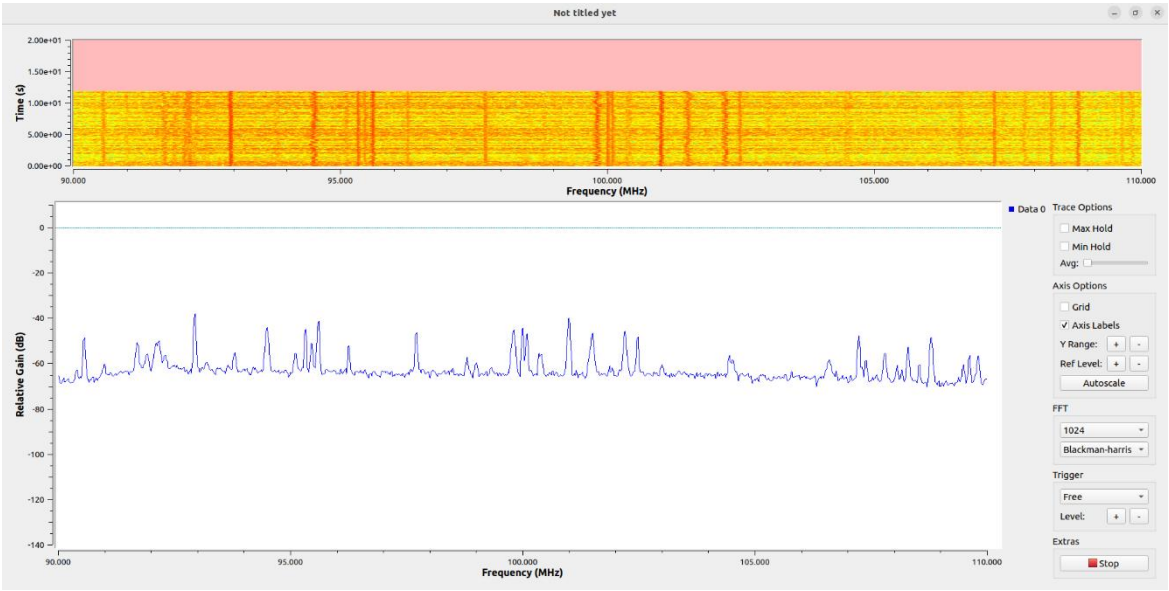

Figure 47. HackRF antenna connection to board's ANTENNA Port

For both tests the central frequency was set to 100MHz (FM radio band), with a sample rate of 20MS/s and minimum averages. Strong peaks were observed from nearby FM broadcast stations, confirming correct signal acquisition and real-time visualization.

Table 1. Received power comparison between antenna connection and no antenna connection

Frequency [MHz]	**Received Power [dB] / Relative Gain (relative to full scale=0)**	
	No antenna	**Antenna**
92.95	-81.8dB	-37.7dB
101	-83.27dB	-40.45dB
107.25	-82.58dB	-47.4dB

6.1.2 Terminal-Based OperaCake Integration

OperaCake RF switch connected to HackRF One. Manual switching between all eight antenna ports using the 'hackrf_operacake' utility in terminal.

In this test, the same antenna was physically moved from port A1 to port B1 of the OperaCake RF switch, and the received signals were recorded under identical conditions. The goal was to evaluate whether port location (A-side vs. B-side) affects reception quality for the same RF input.

From the spectrograms and frequency spectra, we observe minimal differences between A1 and B1 in terms of signal intensity and spectral shape. However, slight fluctuations in signal level or noise floor may be attributed to internal routing differences or minor impedance mismatches within the switch.

The captures were taken back-to-back within seconds of each other so the time difference can be assumed to be negligible. Though both ports are part of the same switch matrix, the signal routing paths inside OperaCake are not perfectly symmetrical, which may introduce minor variation in insertion loss.

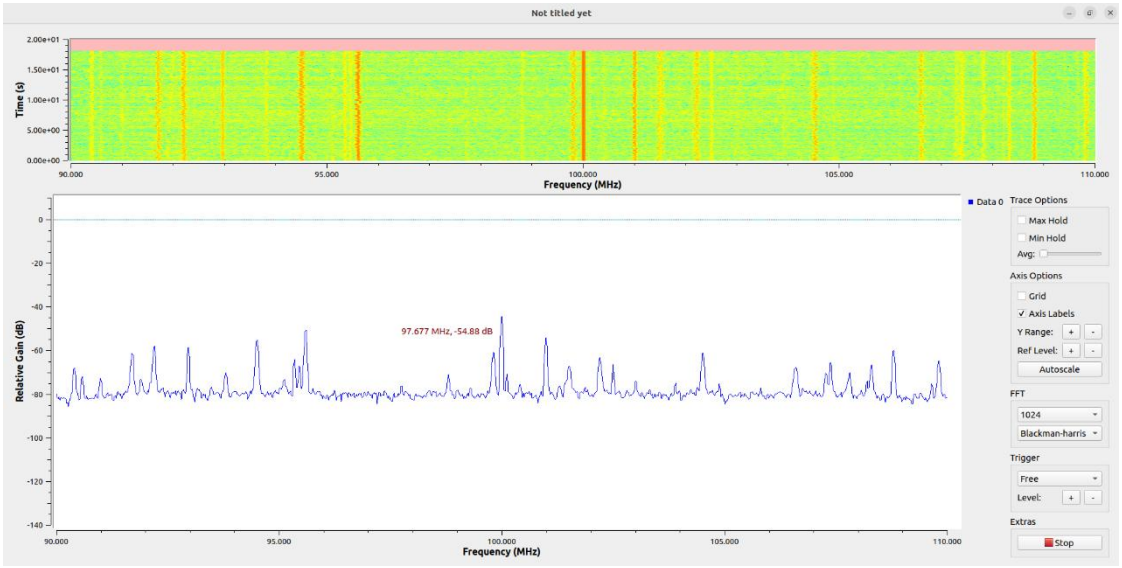

Figure 48. OperaCake antenna connection on port A1

Table 2. Received power comparison between no antenna connection and antenna connection across OperaCake's ports

Frequency [MHz]	**Relative Gain [dB]**								
	No antenna	**Antenna**							
		A1	A2	A3	A4	B1	B2	B3	B4
92.95	-81.81	-59.67	-59.59	-60.23	-59.95	-59.49	-56.52	-56.7	-56
101	-83.27	-53.91	-53.5	-53.65	-54.95	-50.96	-48.55	-51.59	-50.7
107.25	-82.58	-70.18	-71.39	-70.42	-71.2	-63.6	-61.42	-59.27	-59.73

There are no significant differences between ports A1 and B1, with only minor variations visible above the 100 MHz threshold in both the spectrogram and frequency spectrum, as reflected in the table. However, when compared to the earlier case where the antenna was connected directly to the HackRF SMA port, the differences become more pronounced. While FM peaks remain detectable through the OperaCake, their amplitude is reduced. In the direct connection setup, peak levels reach approximately ~38.5 dB, whereas with the switch in place, they fall between ~-50←→-70 dB, depending on the port. This indicates a typical insertion loss of 10-30 dB, attributed to OperaCake's internal switching path and external cabling.

6.1.3 Delay Estimation Attempts

To better understand the switching latency introduced by the OperaCake RF switch and its software interface, two different approaches were tested prior to implementing real-time spectrogram visualization in the GUI:

- Timestamp logging using Python scripts
- Visual analysis using GNU Radio flowgraphs (record/playback)

1) Python-Based Switching and Timestamp Logging

Two scripts were developed, one for Linux (latency_calc.py) and one for Windows (latency_calc_wind.py) to measure the time it takes for the 'hackrf_operacake' command to switch ports. Both scripts looped over 50 port switch cycles (A4-B4) and recorded precise timestamps before and after each command.

I. Linux Results (latency_calc.py):

- Stable and consistent values
- Average latency: 14.32 ms
- Max: 32.32 ms, Min: 9.98 ms

Results showed relatively low switching latency under Linux with minimal system overhead.

```
03:35:22.193957] >>> Switching to A4
03:35:22.232067] <<< Switch to A4 complete
03:35:23.735111] >>> Switching to B4
03:35:23.746701] <<< Switch to B4 complete
03:35:25.248638] >>> Switching to A4
03:35:25.261074] <<< Switch to A4 complete
03:35:26.763000] >>> Switching to B4
03:35:26.776386] <<< Switch to B4 complete
03:35:28.278127] >>> Switching to A4
03:35:28.290161] <<< Switch to A4 complete
03:35:29.792898] >>> Switching to B4
03:35:29.805010] <<< Switch to B4 complete
03:35:31.307318] >>> Switching to A4
03:35:31.318548] <<< Switch to A4 complete
03:35:32.820279] >>> Switching to B4
03:35:32.840259] <<< Switch to B4 complete
03:35:34.351777] >>> Switching to A4
03:35:34.363355] <<< Switch to A4 complete
```

Figure 49. Terminal view of port-switching latency logs (timestamps HH:MM:SS:ss)

```
Max latency: 32.32 ms
Min latency: 9.98 ms
Average latency: 14.31 ms
```

Figure 50. Terminal view of port-switching latency statistics

II. Windows Results (latency_calc_wind.py):

These results are printed by using a python program created by me solely for this purpose.

```python
def switch_port(port):
    start_time = datetime.now()
    print(f"[{start_time}] >>> Switching to {port}")
    proc = subprocess.run(["hackrf_operacake.exe", "-o", "0", "-a", port], capture_output=True)
    end_time = datetime.now()
    print(f"[{end_time}] <<< Switch to {port} complete")
    latency_ms = (end_time - start_time).total_seconds() * 1000
    return latency_ms
```

Figure 51. Port-switching code

- Highly variable, higher latency
- Average latency: 44.18 ms
- Max: 49.09 ms, Min: 41.30 ms
- Execution time was roughly 3x slower than on Linux

Likely causes: USB driver delays, subprocess handling, and lack of efficient I/O timing in PowerShell.

```
[2025-09-01 01:54:42.586668] >>> Switching to A4
[2025-09-01 01:54:42.633191] <<< Switch to A4 complete
[2025-09-01 01:54:44.134574] >>> Switching to B4
[2025-09-01 01:54:44.179173] <<< Switch to B4 complete
[2025-09-01 01:54:45.680225] >>> Switching to A4
[2025-09-01 01:54:45.726666] <<< Switch to A4 complete
[2025-09-01 01:54:47.227508] >>> Switching to B4
[2025-09-01 01:54:47.271398] <<< Switch to B4 complete
[2025-09-01 01:54:48.772132] >>> Switching to A4
[2025-09-01 01:54:48.815659] <<< Switch to A4 complete
[2025-09-01 01:54:50.316602] >>> Switching to B4
[2025-09-01 01:54:50.360638] <<< Switch to B4 complete
[2025-09-01 01:54:51.861412] >>> Switching to A4
[2025-09-01 01:54:51.906666] <<< Switch to A4 complete
[2025-09-01 01:54:53.407080] >>> Switching to B4
[2025-09-01 01:54:53.452695] <<< Switch to B4 complete
[2025-09-01 01:54:54.953332] >>> Switching to A4
[2025-09-01 01:54:54.996989] <<< Switch to A4 complete
```

Figure 52. CMD view of port-switching latency logs (timestamps YY-MM-DD HH:MM:SS:ss)

```
Max latency: 49.09 ms
Min latency: 41.30 ms
Average latency: 44.18 ms
```

Figure 53. CMD view of port-switching latency statistics

Table 3. Software level port-switching latency results comparison between Linux and Windows

Platform	Min Latency[ms]	Max Latency [ms]	Average Latency [ms]
Linux (VM)	9.98	32.32	14.31
Windows	41.30	49.09	44.18

2) GRC Flowgraph Method: Record & Playback

To approach the problem from a signal-based perspective, two GNU Radio Companion flowgraphs were built:

I. Capture Mode:

An osmocom source connected to HackRF recorded raw I/Q data to a file (result.iq) while time port switching was done during capture. Antenna was moved from signal-present to signal-absent ports to induce a visible change.

II. Playback Mode:

The recorded file was played using File Source connected to QT GUI Waterfall Sink, a throttle block being used to simulate real-time playback speed, preserving timing characteristics visually. When replayed, the transition between signal presence and absence became visible as intensity changes in the spectrogram.

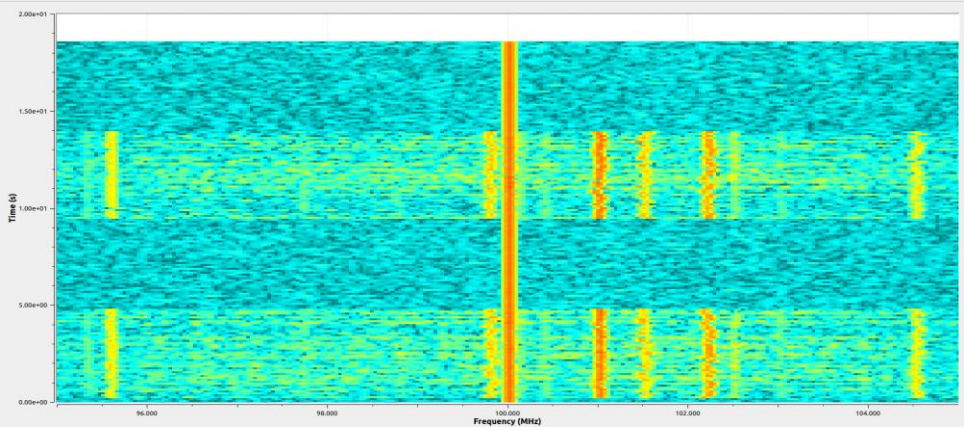

Figure 54. Waterfall output replaying captured I/Q file

The spectrogram showed clear visual transitions at the expected moments of switching. However, because GRC does not natively support timestamped event overlay or direct marking of command execution times, the exact delay between switch command and signal appearance could not be determined. Additionally, waterfall rendering delay and buffer latency introduced some uncertainty in aligning perceived signal changes to precise switching times.

6.1.4 Frequency command analysis

While testing frequency-based switching using the 'hackrf_operacake' utility, an interesting behavior was observed regarding how OperaCake handles frequencies outside the explicitly defined ranges.

When the HackRF central frequency is tuned to a value not covered by any defined range, OperaCake defaults to the last port defined in the command, in this case, B4. This behavior was consistent during multiple tests. Even if more frequency-port mappings are added, always the last one acted as the default catch-all.

hackrf_operacake -m frequency -f B3:250:300 -f B4:500:1000 -f A4:2000:3000

This configuration command assigns port B3 to frequencies between 250MHz and 300MHz, port B4 to frequencies between 500MHz and 1000MHz and port A4 to frequencies between 2000MHz and 3000MHz.

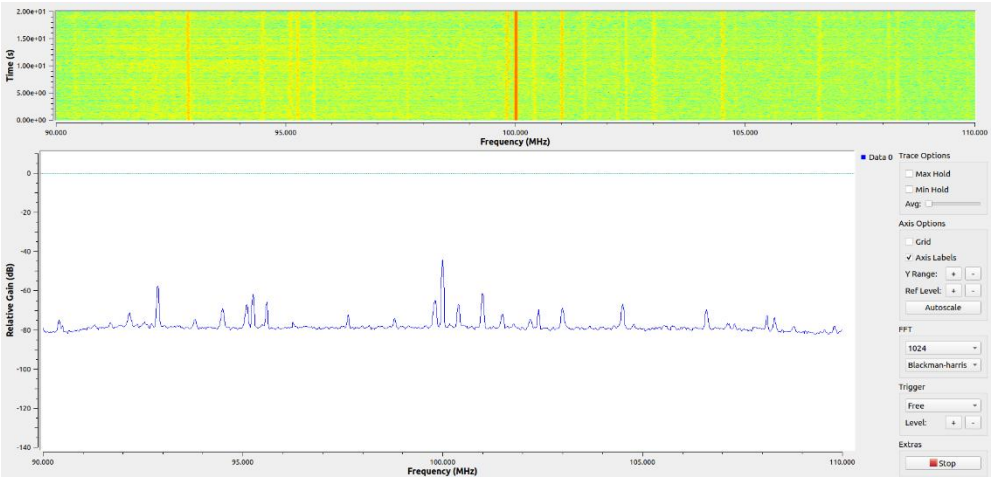

Figure 55. Frequency switching UI at 100MHz in GNU Radio

Figure 56. Frequency switching default behavior (tuning to a frequency outside of the defined ranges is routed to the last mapping(A4))

In the figures above we can observe this default behavior in action. Although the central frequency was set outside the defined ranges, OperaCake still routed the signal through port A4, which was the last mapping command. The illuminated status LED on A4 in fig. x confirms that the port was selected.

6.2 Manual Switching

This mode was the first to be implemented in the custom GUI and served as the foundation for validating live antenna port switching within a running SDR stream.

No additional spectrogram or frequency plots are included in this section, as the results observed during manual switching closely replicate those already documented in the earlier validation phase (Section 3.1.2). The same setup and signal behavior were reused here to validate the GUI-driven switching mechanism. Since there were no significant differences in signal characteristics or switching behavior, redundancy was avoided for clarity and brevity.

For reference, the only change lies in the control method where the antenna switching was performed using the GUI port selection button (see Fig. x), rather than terminal commands. This demonstrates that the backend logic and real-time SDR updates in the GUI reflect the expected

functionality, confirming that manual control performs equivalently to the previously tested terminal-based approach.

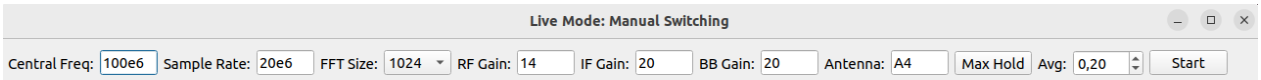

Figure 57. Control toolbar for manual switching

6.3 Frequency Switching

The frequency switching mode was designed to automatically change the active antenna port based on the current central frequency of the HackRF. This mode allows the OperaCake switch to route different frequency bands to specific antennas, enabling seamless spectrum scanning across multiple optimized inputs.

```
[INFO] Central frequency set to 100000000.00 Hz
[DEBUG] Checking port A4 for range 0.0-100.0 MHz
[SKIP] A4 already selected.
```

Figure 58. Terminal log for frequency switching

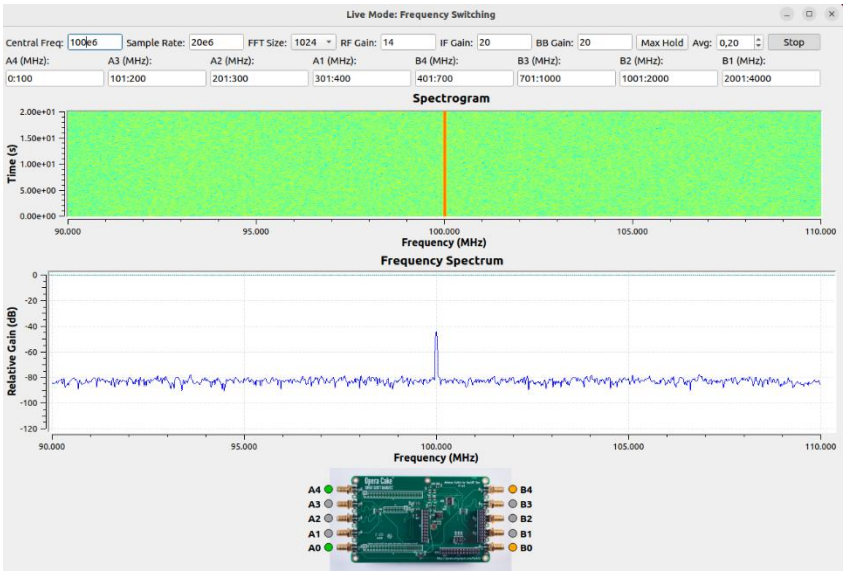

Figure 59. Frequency switching UI initial state

This figure shows the frequency switching interface with a complete mapping of antenna ports to frequency ranges. In the initial configuration, the central frequency is set to 100 MHz, when the central frequency is set to 100 MHz, the system detects that it falls under port A4, and no switch is needed because it is already selected at the startup, so no switch is triggered.

```
[INFO] Central frequency set to 955000000.00 Hz
[DEBUG] Checking port A4 for range 0.0-100.0 MHz
[DEBUG] Checking port A3 for range 101.0-200.0 MHz
[DEBUG] Checking port A2 for range 201.0-300.0 MHz
[DEBUG] Checking port A1 for range 301.0-400.0 MHz
[DEBUG] Checking port B4 for range 401.0-700.0 MHz
[DEBUG] Checking port B3 for range 901.0-1000.0 MHz
[AUTO-SWITCH] Switching to B3 for 955.00 MHz
[INFO] Switched antenna to: B3
```

Figure 60. Terminal log for frequency switching

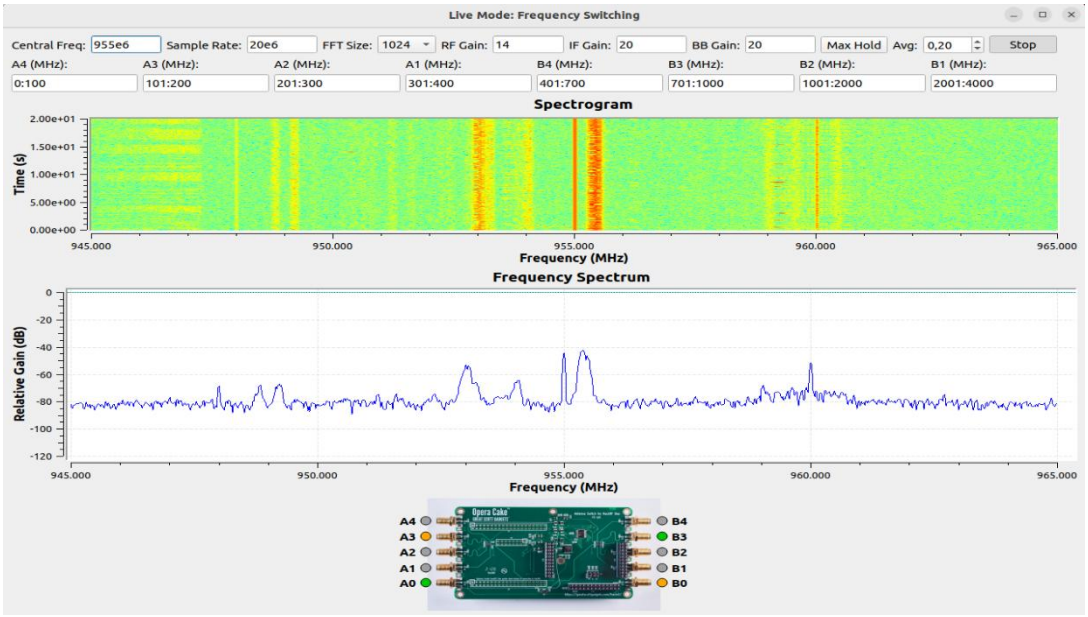

Figure 61. Frequency switching UI after tuning frequency

When the central frequency is manually changed to 955 MHz from the central frequency text input field the debug console displays the internal check for all defined port ranges in their order. Once B3 is identified as the correct match, the system triggers an automatic switch to B3, and this is immediately reflected in the spectrogram as a shift in signal pattern and in the frequency spectrum.

6.4 Time Switching

The Time Switching mode enables OperaCake to automatically cycle through a list of antenna ports, each active for a user-defined duration. This mode simulates round-robin scanning and is ideal for comparing signal conditions across multiple antennas over time.

The switching configuration was entered through the GUI as port-duration pairs. Example: A1 – 3 s, B2 – 4 s, B3 – 4 s.

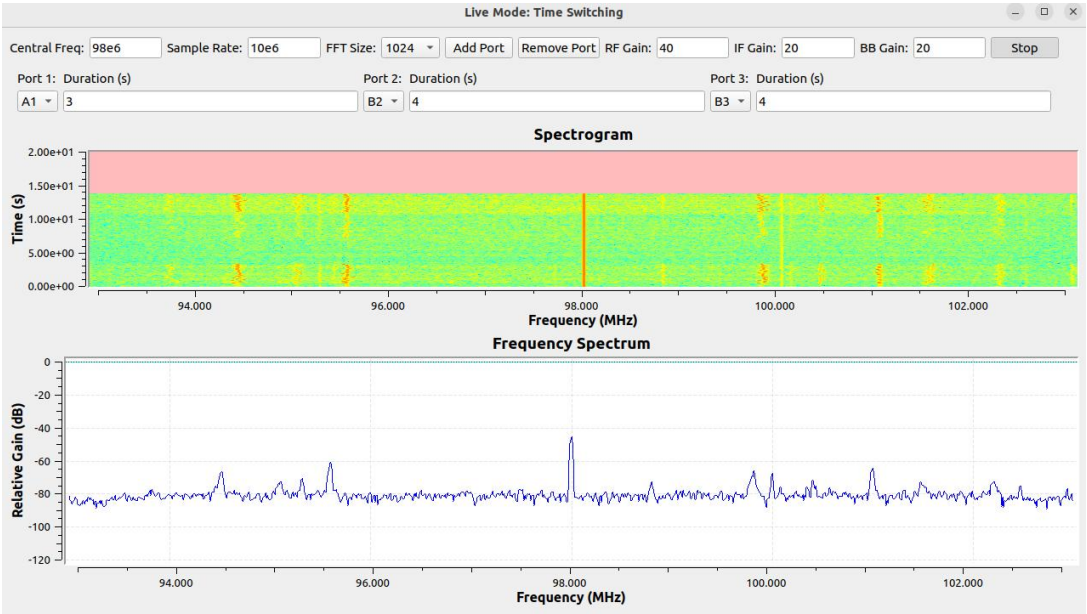

Figure 62. Time switching UI after going through the set port and restarting the queue

As seen in the figure above:

- Each port was activated for a specified duration before switching.
- The switching loop ran continuously and restarted from the beginning once the end of the list was reached.
- On the spectrogram, vertical gaps or sudden intensity changes were visible when the active port changed to one without an antenna.

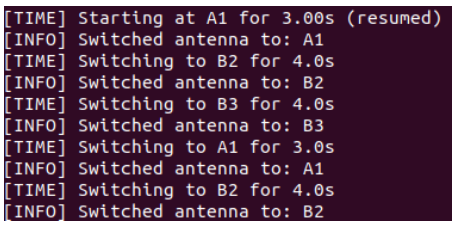

Figure 63. Terminal log for time switching

The terminal output in Fig. Y confirms the correct execution of the time-switching logic. The system begins by activating port A1 for 3.00 seconds, then proceeds to port B2 for 4.00 seconds, followed by port B3 for another 4.00 seconds. Once the defined list is completed, the loop restarts from the beginning, demonstrating a continuous cycle: returning to A1 and continuing through the same sequence.

This output complements the visual spectrogram representation and confirms that the switching logic operates exactly as configured in the GUI, with seamless port transitions and accurate timing.

6.5 Delay

The Delay mode was developed to provide a visual estimation of the response time between an antenna port switch and the moment a signal becomes visible (or disappears) in the spectrogram. This was necessary because earlier approaches, such as timestamped logs or offline I/Q playback lacked synchronization between switching commands and RF-level feedback.

When Delay mode is clicked in the main controller, a Delay Tools window now opens.

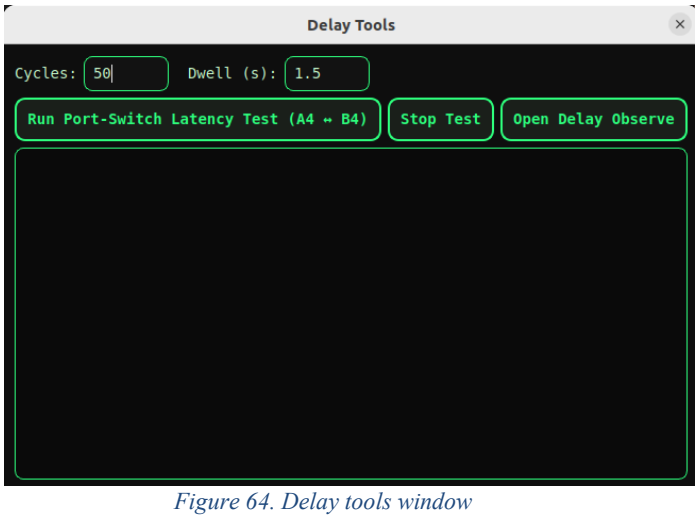

Figure 64. Delay tools window

This launcher provides two complementary tools for evaluating switching delays, Delay switching and Port-Switch Latency Test between A4 and B4 to which we've seen the results at 7.1.3 but what's different is that now we can specify the number of cycles wanted for the test and the time between switching.

The GUI when opening Delay Observe, which is visually optimized for transition detection contains the following:

- Only QT GUI Waterfall Sink is displayed
- The time axis was zoomed to display only 100 ms of scrolling history

To maximize visibility of the signal transition, only port A4 was connected to an antenna. During the test, the active port was manually switched from A4 to B4, which had no antenna attached. This ensured a clear and immediate drop in signal, making the difference in reception easily distinguishable in the spectrogram.

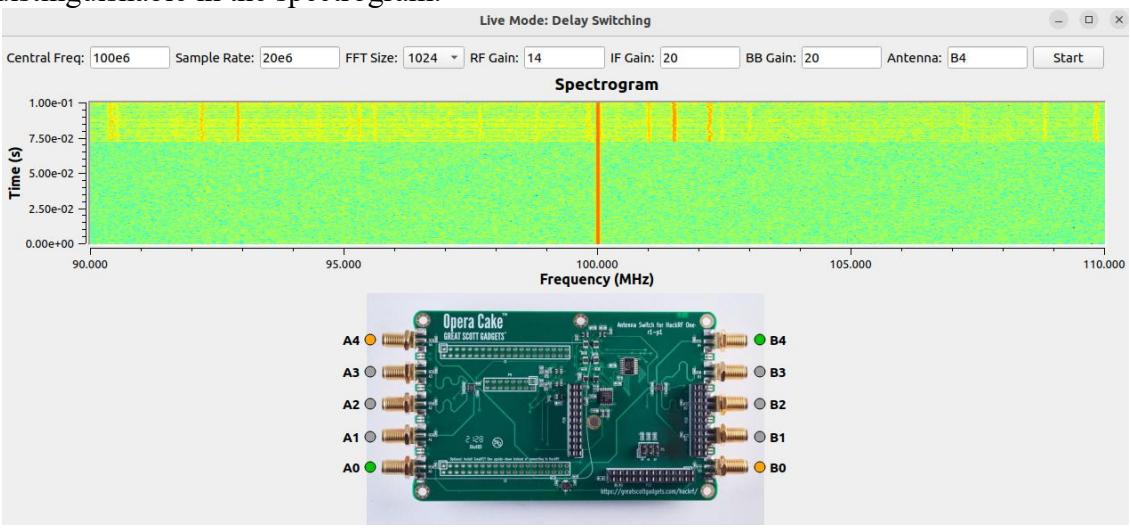

Figure 65. Delay switching UI after changing from port A4(antenna) to port B4(no antenna)

As shown in the spectrogram sequence, the signal is initially visible while receiving from port A4, which had the antenna connected. When switching manually to B4, which had no antenna, the signal drops sharply.

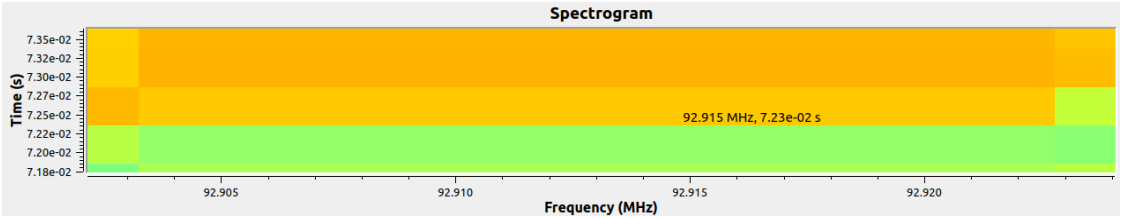

Figure 66. Delay switching spectogram hardware latency observe

By zooming into the waterfall at the transition point, it becomes clear that each vertical line represents approximately 0.5 ms.

Despite this fine time resolution, the switching event occurs faster than a single spectrogram line, meaning the signal loss is reflected in less than 0.5 ms. Although the exact switching time cannot be precisely measured visually, this confirms that the delay is negligible for practical SDR applications and well below 1 ms.

6.6 Wide Spectrum

This mode extends HackRF's instantaneous 20 MHz bandwidth to a wideband view by sweeping the central frequency across a user-defined span and stitching the per-step spectra into a single trace.

So, as you can see from Figure 67, the bandwidth is more than 20MHz, spanning from 500MHz to 600MHz.

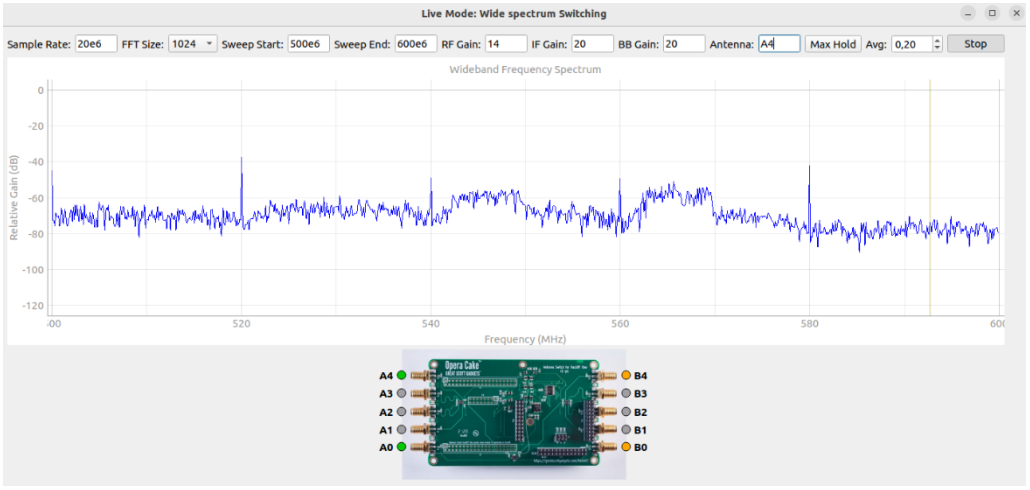

Figure 67. Wide frequecy spectrum UI

6.7 Wide Spectrum Frequency

We're still using the wide spectrum base mode fused with the frequency switching mode with port fields where you can modify their range. An addition is the presence of the radio buttons, such that now each OperaCake port is assigned a tunable span [f_{min}, f_{max}] (MHz)

In Figure 68 you can see as a first example that we visualize a wideband from 87.5MHz to 108Mhz, covering the FM Regional Frequency assignment [21].

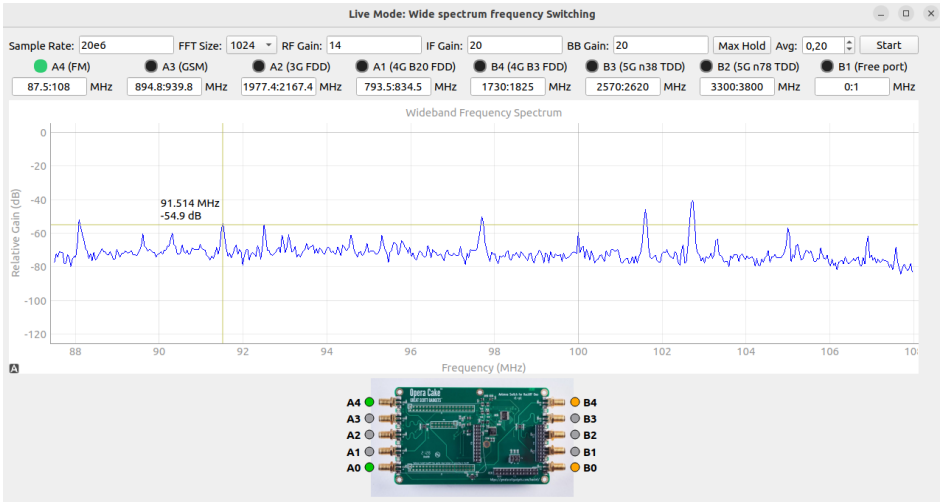

Figure 68. Wide frequency spectrum switching mode UI, FM port

As we can see we detect multiple narrowband carriers, FM channels, and one would be this one on 91.5MHz which corresponds to Kiss FM Zalau [22]. To display a wider cellular allocation, we switch to port A1(4G B20 FDD).

Cell 81	
Cell Identifier	33345105
System Subtype	LTE
PCI	411 (137/0)
Bandwidth	5 MHz
EARFCN	6175
Maximum Signal (RSRP)	-85 dBm
Direction	NE (52°)
First Seen	2/14/2023
Last Seen	6/25/2025
Actions	• Go to Cell
Uplink Frequency	834.5 MHz
Downlink Frequency	793.5 MHz
Frequency Band	EU Digital Dividend (B20 FDD)

Figure 69. Antenna tower cell information

From the live network readout in Figure 69 [23], by inputting the EARFCN into sqimway.com [24] we got the following bandwidth check.

20	800 DD	FDD	791 6150	806 6300	821 6449	30	832 24150	847 24300	862 24449

Figure 70. 4G B20 frequency band catalogue

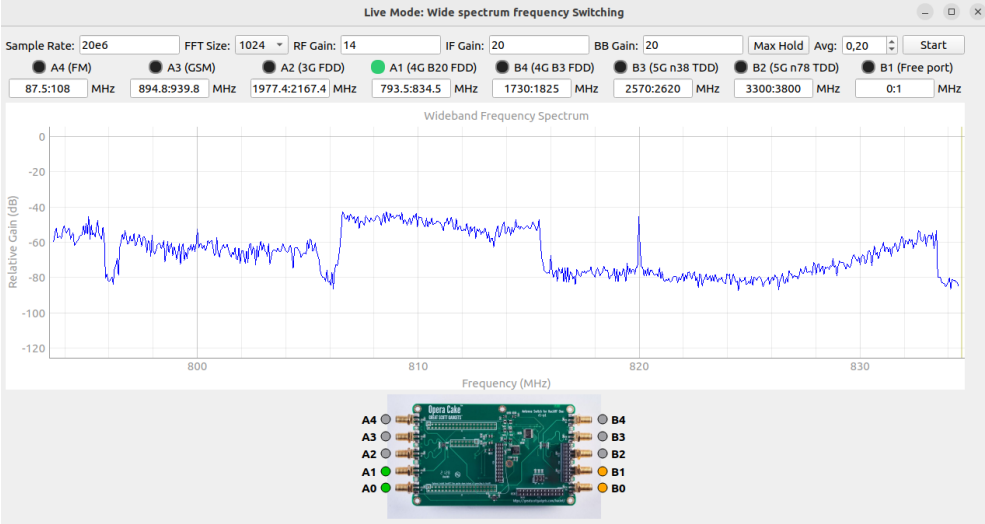

Figure 71. Wide frequency spectrum switching mode UI, 4G port

The catalogue entry in Figure 70 confirms that Band 20 provides a 30MHz downlink block (791-821MHz) paired with a 30MHz uplink block (832-862MHz). In our capture in Figure 71 summing the widths of the visible LTE carriers remain ≤ 30MHz.

6.8 Event Detection and Switching

To test this functionality, the receiver was tuned to 2.442GHz using the balanced preset (ΔdB = 6.5, Occ = 0.22, SFM≥ = −5.0 dB) to record a baseline of normal Wi-Fi activity, then start a speedtest.net measurement [25] (download phase) and let the test to transition to the upload phase. As can be seen in Figure 74 the spectrogram shows a mostly green/yellow band, the instantaneous spectrum has a low, relatively flat floor with a few narrow carries and short bursts. The ΔdB occupancy stays below the Occ gate and SFM remains more negative than the SFM threshold, so no detection is issued.

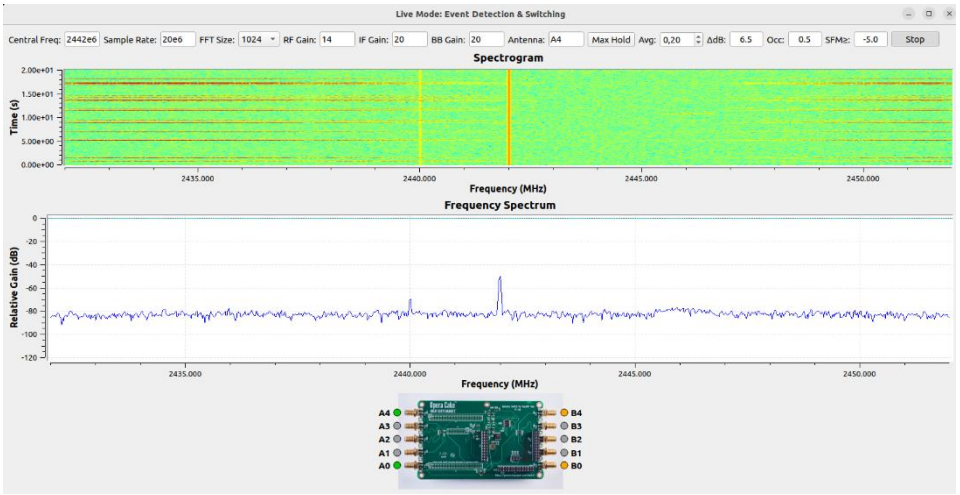

Figure 72. Event detection and switching mode UI baseline activity at 2.442GHz

After starting the download phase, the channel utilization increases, the spectrogram intensifies near the tuned channel, the median mean and band power rise by a few dB, but the event remains borderline.

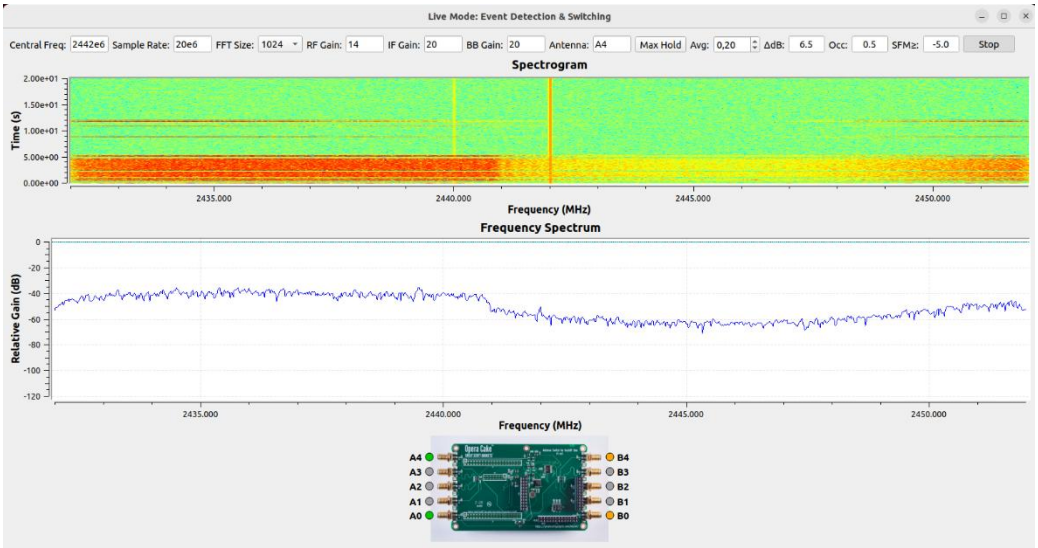

Figure 73. Event detection and switching mode UI download phase. Increased downlink utilization, more noise-like SFM. Hysteresis and the hold timer avoid a premature decision

In the upload phase, the uplink saturates the channel, now the ΔdB occupancy exceeds the configured gate on the masked bins, the band-mean and median lifts across their on-thresholds and SFM clears the gate, after the hold time elapses, the detection is confirmed and the UI shows a red banner around the spectrum panel and the OperaCake rotates to next port as it can be visible in the toolbar panel it switched from A4 to A3.

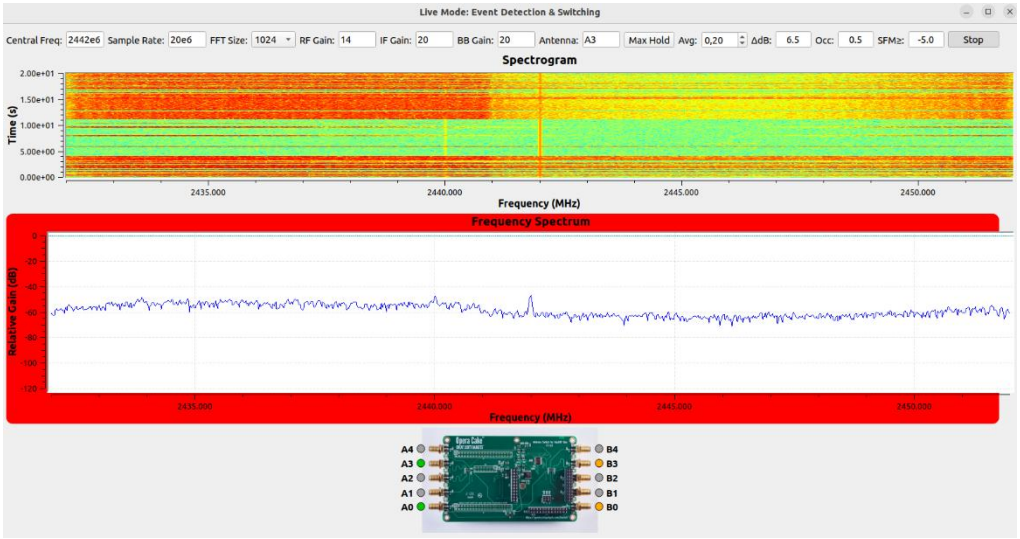

Figure 74. Event detection and switching mode UI upload phase and confirmed event. Channel heavily occupied across most bins, ΔdB occupancy, mean lifts and SFM all meet their criteria. System rotates the activate OperaCake path (A4 -> A3)

7. Conclusions

This thesis sets out to design, implement, and validate a software-defined radio (SDR) platform for spectrum analysis and RF switching, using low-cost hardware (HackRF One and OperaCake RF switch) and a modular software environment (GNU Radio integrated with a custom PyQt5 graphical interface). The work addressed both theoretical signal-processing aspects and practical implementation challenges.

From a theoretical perspective, the project consolidated the main principles underpinning SDR operation: complex baseband representation, sampling and aliasing, FFT-based spectral estimation, windowing and resolution bandwidth, averaging strategies, gain staging, and wideband scanning by step-tuned stitching. These concepts were not treated abstractly; rather, they were carefully aligned with the constraints and capabilities of HackRF and OperaCake. This ensured that the formulas, design choices, and performance metrics were consistent with the actual signal path implemented in code.

On the implementation side, the project produced a functional application capable of operating in multiple modes: manual port selection, frequency-based switching, time-based sequencing, stitched wideband spectrum visualization, latency testing, and anomaly/jamming detection based on spectral metrics such as ΔdB occupancy, spectral flatness, and median/mean lifts. The interface exposed practical controls for FFT size, averaging, gain levels, and anomaly thresholds, while enforcing validation to avoid unsafe configurations. Together, these features resulted in a versatile, extensible tool that can adapt to diverse radio environments.

The experimental results confirmed that the system can capture and visualize signals across different frequency ranges, switch OperaCake ports automatically under frequency or jamming conditions, and provide latency feedback at both command-line and GUI levels. Although the absolute power calibration remains relative (due to hardware and time constraints), the system is robust enough to identify spectral activity and interference patterns with high confidence. Performance limitations were mainly linked to HackRF's single-channel, half-duplex nature and to the refresh rate trade-offs when stitching wide spans. Nevertheless, the work demonstrated that meaningful spectrum analysis can be achieved within these constraints.

Looking forward, several extensions are possible. Absolute calibration and antenna characterization would enable quantitative measurements. Multi-device synchronization could expand the system into a cooperative monitoring network. Integrating machine learning for anomaly classification could improve detection robustness in crowded bands. Finally, optimizing the graphical interface for responsiveness at very wide spans would push the system closer to professional spectrum analyzers.

In conclusion, the objectives of the thesis were successfully met. The developed system demonstrates that with a well-founded theoretical background and careful software engineering, low-cost SDR hardware can be transformed into a powerful spectrum analysis and anomaly detection platform. This outcome validates the broader vision of SDR: flexibility, accessibility, and adaptability in modern radio environments.

8. References

[1] D. DiPuccio, T. W. Rondeau, J. Reed, and M. Lichtman, Software Defined Radio for Engineers. Norwood, MA, USA: Artech House, 2018.
[2] S. W. Ellingson, Radio Systems Engineering. Cambridge, UK: Cambridge University Press, 2016.
[3] B. Scheers, "Survey of SDR tools for education and research," International Journal on Advances in Networks and Services, vol. 11, nos. 1–2, pp. 98–105, 2018.
[4] GNU Radio Foundation, "GNU Radio official documentation," 2023. [Online]. Available: https://wiki.gnuradio.org/
[5] HackRF Docs, "OperaCake switching modes and commands," Great Scott Gadgets, 2022. [Online]. Available: https://hackrf.readthedocs.io/en/latest/opera_cake.html
[6] Great Scott Gadgets, "OperaCake RF switch for HackRF One," 2022. [Online]. Available: https://greatscottgadgets.com/hackrf/operacake/
[7] Authors, "The principle behind complex sampling for SDR transceivers," in Proceedings of ASEE NCS Conference, 2019.
[8] A. Alshammari, "Time and frequency domain signal capture in low cost SDR systems," Wireless Personal Communications, vol. 114, pp. 1185–1201, 2020.
[9] M. Lichtman, R. Rao, J. Reed, and T. Rondeau, "A communications theoretic view of software defined radio latency," IEEE Communications Magazine, vol. 57, no. 2, pp. 18–24, Feb. 2019.
[10] Author(s), "Understanding FFT algorithms and applications," Fast Fourier Transform: Algorithms and Applications, 2016.
[11] R. Zhao, D. Li, and H. Wang, "Visual radio signal analysis in SDR frameworks," Sensors, vol. 21, no. 3, pp. 842–858, Jan. 2021.
[12] J. A. Tropp, J. N. Laska, M. F. Duarte, J. K. Romberg, and R. G. Baraniuk, "Beyond Nyquist: Efficient sampling of sparse wideband analog signals," IEEE Trans. Signal Process., vol. 57, no. 7, pp. 2942–2954, July 2009.
[13] L. Liu, K. Lee, and Y. Bai, "Design and implementation of a multi antenna SDR testbed using HackRF and GNU Radio," IEEE Access, vol. 10, pp. 121345–121356, 2022.
[14] GNU Radio Companion Help, GNU Radio v3.10, 2023.
[15] X. Mikos, "QSpectrumAnalyzer – GNU Radio frequency sweep GUI," GitHub, 2023. [Online]. Available: https://github.com/xmikos/qspectrumanalyzer
[16] GQRX Developers, "GQRX SDR receiver and spectrum analyzer," 2023. [Online]. Available: https://gqrx.dk/
[17] HackRF Tools Help, "hackrf_operacake command line utility," HackRF ReadTheDocs, 2022.
[18] Ubuntu Documentation, "Ubuntu 22.04 LTS release notes," Ubuntu, 2022. [Online]. Available: https://help.ubuntu.com/
[19] Oracle Corp., "Oracle VM VirtualBox User Manual," Oracle, 2023. [Online]. Available: https://docs.oracle.com/en/virtualization/virtualbox/
[20] https://github.com/osmocom/gr-osmosdr/commit/c3ec9cc6761cd3ce2684a4ad9bd7f4853254265c
[21] https://www.itu.int/en/ITU-R/terrestrial/broadcast/pages/fmtv.aspx
[22] https://www.radio-romania.com/kiss-fm
[23] https://www.cellmapper.net/map?MCC=226&MNC=5&type=LTE&latitude=47.19955833278607&longitude=23.048305744589012&zoom=16.074110472646414&showTowers=true&showIcons=true&showTowerLabels=true&clusterEnabled=true&tilesEnabled=true&showOrphans=false&showNoFrequencyOnly=false&showFrequencyOnly=false&showBandwidthOnly=false&DateFilterType=Last&showHex=false&showVerifiedOnly=false&showUnverifiedOnly=false&showLTECAOnly=false&showENDCOnly=false&showBand=0&showSectorColours=true&mapType=roadmap&darkMode=false&imperialUnits=false

[24] https://www.sqimway.com/lte_band.php
[25] https://www.speedtest.net/

9. Appendix

```
#  Imports
from PyQt5 import Qt
from PyQt5.QtCore import pyqtSignal, QTimer, QEvent
from PyQt5.QtGui import QDoubleValidator, QIntValidator
from gnuradio import gr, qtgui, blocks
from gnuradio.fft import fft_vcc, window
import osmosdr, sip, numpy as np, pyqtgraph as pg, re, time, subprocess
from pyqtgraph import PlotWidget, mkPen
from time import monotonic
from LED import OperaCakePanel
from pathlib import Path

# Asset paths
APP_ROOT  = Path(__file__).resolve().parent
ASSETS_DIR = APP_ROOT / "assets"
def asset(*parts) -> Path:
    return ASSETS_DIR.joinpath(*parts)

# Regex helpers
_FREQ_REGEX  = re.compile(r"^\s*([+-]?(?:\d+(?:\.\d+)?|\.\d+)(?:e[+-]?\d+)?)\s*(hz|khz|mhz|ghz)?\s*$", re.I)
_FLOAT_REGEX = re.compile(r"^\s*[+-]?(?:\d+(?:\.\d+)?|\.\d+)(?:e[+-]?\d+)?\s*$")
_INT_REGEX   = re.compile(r"^\s*\d+\s*$")

# Input parsers
def parse_freq_input(text: str) -> float:
    if text is None:
        raise ValueError("Empty frequency")
    m = _FREQ_REGEX.match(text)
    if not m:
        raise ValueError(f"Invalid frequency format: '{text}'")
    val = float(m.group(1))
    unit = (m.group(2) or "").lower()
    mult = {"":1.0, "hz":1.0, "khz":1e3, "mhz":1e6, "ghz":1e9}[unit]
    return val * mult

def parse_float_strict(text: str, vmin=None, vmax=None, name="value") -> float:
    """Strict float (allows sci notation). No extra chars allowed."""
    if text is None or not _FLOAT_REGEX.match(text):
        raise ValueError(f"Invalid {name}: '{text}'")
    val = float(text)
    if vmin is not None and val < vmin:
        raise ValueError(f"{name} must be ≥ {vmin}")
    if vmax is not None and val > vmax:
        raise ValueError(f"{name} must be ≤ {vmax}")
    return val

def parse_int_strict(text: str, vmin=None, vmax=None, name="value") -> int:
    """Strict int. No extra chars."""
    if text is None or not _INT_REGEX.match(text):
        raise ValueError(f"Invalid {name}: '{text}'")
    val = int(text)
    if vmin is not None and val < vmin:
        raise ValueError(f"{name} must be ≥ {vmin}")
    if vmax is not None and val > vmax:
        raise ValueError(f"{name} must be ≤ {vmax}")
    return val

# Disable mouse interactions when needed
class _MouseBlocker(Qt.QObject):
    def __init__(self, owner):
        super().__init__(owner)
        self._owner = owner

    def eventFilter(self, obj, ev):
        if not getattr(self._owner, "_interactions_enabled", False):
            if ev.type() in (QEvent.Wheel, QEvent.MouseButtonPress, QEvent.MouseButtonRelease, QEvent.MouseMove):
                return True
        return False

# DSP utilities
def process_fft_data(mags):
    mags = np.copy(mags)
    n = len(mags)
    center = n // 2

    # Suppress DC and surrounding bins
    if center >= 3:
        mags[center-2:center+3] = np.mean([
            mags[center-4], mags[center-3],
            mags[center+3], mags[center+4]
        ])

    # Taper edges to avoid chunk artifacts
    fade = 10
```

```
    mags[:fade] *= np.linspace(0.3, 1.0, fade)
    mags[-fade:] *= np.linspace(1.0, 0.3, fade)
    return mags

def spectral_flatness_db(power_lin, eps=1e-12):
    x = np.clip(power_lin, eps, None)
    gmean = np.exp(np.mean(np.log(x)))
    amean = np.mean(x)
    return 10.0 * np.log10((gmean / (amean + eps)) + eps)

def _find_peaks(db, n=5, min_sep_bins=8):
    peaks = []
    used = np.zeros_like(db, dtype=bool)
    for _ in range(n):
        masked = np.where(~used, db, -1e9)
        i = int(np.argmax(masked))
        if masked[i] <= -1e8:
            break
        peaks.append(i)
        used[max(0, i - min_sep_bins): i + min_sep_bins + 1] = True
    return sorted(peaks)

def clamp(v, lo, hi):
    return lo if v < lo else hi if v > hi else v

# Main application window
class LiveSpectrogramWindow(gr.top_block, Qt.QWidget):
    closed = pyqtSignal()
    antenna_changed = pyqtSignal(str)

    def __init__(self, mode):
        gr.top_block.__init__(self, "Live Spectrogram + Frequency Spectrum", catch_exceptions=True)
        Qt.QWidget.__init__(self)
        if mode == "jamming":
            self.setWindowTitle("Live Mode: Event Detection & Switching")
        else:
            self.setWindowTitle(f"Live Mode: {mode.capitalize()} Switching")
        qtgui.util.check_set_qss()

        # Default SDR config
        self.mode = mode
        self.center_freq = 100e6
        self.samp_rate = 20e6
        self.fft_size = 1024
        self.antenna_select = 'A4'
        self.rf_gain = 14
        self.if_gain = 20
        self.bb_gain = 20
        self.last_time_index = 0  # For time switching resume

        # Internal state
        self.sweep_ptr = 0
        self.total_steps = 0
        self.freq_axis = np.array([])
        self.sweep_buffer = np.array([])
        self.sweep_active = False
        self.sweep_busy = False

        self.port_tags = {
            "A4": "FM",
            "A3": "GSM",
            "A2": "3G FDD",
            "A1": "4G B20 FDD",
            "B4": "4G B3 FDD",
            "B3": "5G n38 TDD",
            "B2": "5G n78 TDD",
            "B1": "Free port",
        }

        # Layout and UI
        self.top_layout = Qt.QGridLayout(self)

        # UI BUILD ZONE
        form = Qt.QHBoxLayout()
        self.toolbar = form

        # jamming banner
        if mode == "jamming":
            self.jam_alert_label = Qt.QLabel("⚠️ Event Detected — Switching Antenna")
            self.jam_alert_label.setAlignment(Qt.Qt.AlignCenter)
            self.jam_alert_label.setStyleSheet("""
                QLabel {
                    background-color: #ff0000;
                    color: white;
                    font-weight: bold;
                    font-size: 20px;
                    border-radius: 10px;
                    padding: 10px;
                }
```

```
        """)
        self.jam_alert_label.hide()
        self.top_layout.addWidget(self.jam_alert_label, 4, 0)

    if self.mode not in ("wide spectrum", "wide spectrum frequency"):
        self.freq_edit = Qt.QLineEdit("100e6")
        form.addWidget(Qt.QLabel("Central Freq:"))
        form.addWidget(self.freq_edit)
        self.freq_edit.returnPressed.connect(self.update_center_freq)

        self.freq_range_label = Qt.QLabel("")
        self.freq_range_label.setStyleSheet("color: red; font-weight:bold")
        self.freq_range_label.hide()
        form.addWidget(self.freq_range_label)

    # Sample rate
    self.samp_rate_edit = Qt.QLineEdit("20e6")
    form.addWidget(Qt.QLabel("Sample Rate:"))
    form.addWidget(self.samp_rate_edit)
    self.samp_rate_edit.returnPressed.connect(self.update_sample_rate)

    self.samp_rate_range_label = Qt.QLabel("")
    self.samp_rate_range_label.setStyleSheet("color: red; font-weight: bold")
    self.samp_rate_range_label.hide()
    form.addWidget(self.samp_rate_range_label)

    # FFT size
    self.fft_combo = Qt.QComboBox()
    self.fft_combo.addItems([str(v) for v in [32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384]])
    self.fft_combo.setCurrentText(str(self.fft_size))
    self.fft_combo.currentTextChanged.connect(self.update_fft_size)
    form.addWidget(Qt.QLabel("FFT Size:"))
    form.addWidget(self.fft_combo)

    # Mode-specific inputs
    if self.mode == "time":
        self._build_time_mode_ui()
    elif self.mode == "frequency":
        self._build_frequency_mode_ui()
    elif self.mode == "wide spectrum":
        self._build_sweep_mode_ui(form)
    elif self.mode == "wide spectrum frequency":
        self._build_port_sweep_ui()
    # Gains
    self.rf_gain_min, self.rf_gain_max = 0, 14
    self.if_gain_min, self.if_gain_max = 0, 40
    self.bb_gain_min, self.bb_gain_max = 0, 62

    self.rf_gain_edit = Qt.QLineEdit(str(self.rf_gain))
    self.if_gain_edit = Qt.QLineEdit(str(self.if_gain))
    self.bb_gain_edit = Qt.QLineEdit(str(self.bb_gain))

    # Put fields on the toolbar/form
    form.addWidget(Qt.QLabel("RF Gain:")); form.addWidget(self.rf_gain_edit)
    self.rf_gain_error_label = Qt.QLabel("")
    self.rf_gain_error_label.setStyleSheet("color: red; font-weight: bold")
    self.rf_gain_error_label.hide()
    form.addWidget(self.rf_gain_error_label)

    form.addWidget(Qt.QLabel("IF Gain:")); form.addWidget(self.if_gain_edit)
    self.if_gain_error_label = Qt.QLabel("")
    self.if_gain_error_label.setStyleSheet("color: red; font-weight: bold")
    self.if_gain_error_label.hide()
    form.addWidget(self.if_gain_error_label)

    form.addWidget(Qt.QLabel("BB Gain:")); form.addWidget(self.bb_gain_edit)
    self.bb_gain_error_label = Qt.QLabel("")
    self.bb_gain_error_label.setStyleSheet("color: red; font-weight: bold")
    self.bb_gain_error_label.hide()
    form.addWidget(self.bb_gain_error_label)

    # Keep a "dirty" flag so clicking/focusing a field doesn't re-apply.
    self._gains_dirty = False
    self._last_applied_gains = (self.rf_gain, self.if_gain, self.bb_gain)

    def _on_gain_edited(_=None):
        self._gains_dirty = True

    for edit in (self.rf_gain_edit, self.if_gain_edit, self.bb_gain_edit):
        edit.setValidator(QIntValidator(-9999, 9999, self))
        edit.textEdited.connect(_on_gain_edited)
        edit.editingFinished.connect(self.update_gains)
        edit.returnPressed.connect(self.update_gains)  # Enter triggers apply

    # Antenna (ONLY for modes that need manual entry)
    if self.mode not in ("time", "frequency", "wide spectrum frequency"):
        self.antenna_edit = Qt.QLineEdit(self.antenna_select)
        self.antenna_edit.returnPressed.connect(self.update_antenna)
        self.antenna_error_label = Qt.QLabel("")
```

```
    self.antenna_error_label.setStyleSheet("color: red; font-weight: bold")
    self.antenna_error_label.setAlignment(Qt.Qt.AlignCenter)
    self.antenna_error_label.hide()

    form.addWidget(Qt.QLabel("Antenna:"))
    form.addWidget(self.antenna_edit)
    form.addWidget(self.antenna_error_label)
else:
    # Keep a placeholder attribute to avoid hasattr checks all over
    self.antenna_edit = None
    self.antenna_error_label = None
if self.mode not in ("wide spectrum", "wide spectrum frequency", "delay"):
    ctrl_wrap = Qt.QWidget()
    ctrl = Qt.QHBoxLayout(ctrl_wrap)
    ctrl.setContentsMargins(0, 0, 0, 0)
    ctrl.setSpacing(6)

    # Max Hold toggle (mapped to freq_sink.enable_max_hold)
    self.btn_fs_max = Qt.QToolButton()
    self.btn_fs_max.setText("Max Hold")
    self.btn_fs_max.setCheckable(True)
    self.btn_fs_max.setChecked(False)
    self.btn_fs_max.setToolTip("Hold the maximum value per-bin")
    ctrl.addWidget(self.btn_fs_max)

    # Avg control (0..1, mapped to set_fft_average)
    ctrl.addWidget(Qt.QLabel("Avg:"))
    self.spin_fs_avg = Qt.QDoubleSpinBox()
    self.spin_fs_avg.setRange(0.0, 1.0)
    self.spin_fs_avg.setSingleStep(0.05)
    self.spin_fs_avg.setDecimals(2)
    self.spin_fs_avg.setValue(0.20)
    self.spin_fs_avg.setToolTip("Exponential averaging factor (0 = off, 1 = full)")
    self.spin_fs_avg.setFixedWidth(72)
    ctrl.addWidget(self.spin_fs_avg)

    # Insert into the toolbar
    idx = self.toolbar.indexOf(self.toggle_btn) if hasattr(self, "toggle_btn") else -1
    if idx >= 0:
        self.toolbar.insertWidget(idx, ctrl_wrap)
    else:
        self.toolbar.addWidget(ctrl_wrap)
else:
    self.btn_fs_max = None
    self.spin_fs_avg = None

# Start/Stop
self.toggle_btn = Qt.QPushButton("Start")
self.toggle_btn.setCheckable(True)
self.toggle_btn.clicked.connect(self.toggle_stream)
form.addWidget(self.toggle_btn)

self.top_layout.addLayout(self.toolbar, 0, 0)

# SDR CONFIG + PIPES
self.src = osmosdr.source("numchan=1 hackrf")
self.apply_sdr_config()

self.sweep_timer = QTimer(self)
self.jam_timer = QTimer(self)
self.jam_timer.timeout.connect(self.check_jamming)

# Base plots
if self.mode == "delay":
    self.init_waterfall()
elif self.mode not in ("wide spectrum", "wide spectrum frequency"):
    self.init_waterfall()
    self.init_freq_spectrum()

# Sweep pipeline
if self.mode in ("wide spectrum", "wide spectrum frequency"):
    self._build_sweep_pipeline()
    self._build_sweep_plot()
    self.validate_sweep_inputs(quiet=True)

# Jamming pipeline
if self.mode == "jamming":
    self._build_jamming_pipeline(form)

BOARD_IMG = str(asset("operacake.jpeg"))

self.oc_panel = OperaCakePanel(BOARD_IMG, self)

self.top_layout.addWidget(self.oc_panel, 5, 0, 1, 5)
self.oc_panel.setSizePolicy(Qt.QSizePolicy.Expanding, Qt.QSizePolicy.Preferred)
self.oc_panel.setMinimumHeight(160)
self.oc_panel.setMaximumHeight(260)
self.oc_panel.set_active(self.antenna_select, fixed_input="A0", connected=self.is_hackrf_connected())
```

```
# UI BUILD — subparts
def _build_time_mode_ui(self):
    self.time_config = []
    self.time_inputs = []
    self.time_labels = []
    self.timer = QTimer(self)
    self.timer.timeout.connect(self.perform_time_switch)

    self.timer_start_time = None
    self.elapsed_before_stop = 0
    self.per_port_elapsed = {}

    # Buttons time mode
    self.add_row_btn = Qt.QPushButton("Add Port")
    self.add_row_btn.setToolTip("Add another port-duration pair")
    self.add_row_btn.clicked.connect(self.add_time_row)

    self.remove_row_btn = Qt.QPushButton("Remove Port")
    self.remove_row_btn.setToolTip("Remove the last port row")
    self.remove_row_btn.clicked.connect(self.remove_time_row)

    idx = self.toolbar.indexOf(self.fft_combo)
    if idx != -1:
        self.toolbar.insertWidget(idx + 1, self.add_row_btn)
        self.toolbar.insertWidget(idx + 2, self.remove_row_btn)
    else:
        self.toolbar.addWidget(self.add_row_btn)
        self.toolbar.addWidget(self.remove_row_btn)

    # Error label under toolbar
    self.time_error_label = Qt.QLabel("")
    self.time_error_label.setStyleSheet("color: red; font-weight: bold")
    self.time_error_label.setAlignment(Qt.Qt.AlignCenter)
    self.time_error_label.hide()
    self.top_layout.addWidget(self.time_error_label, 1, 0, 1, 5)

    # Container for rows
    self.time_input_layout = Qt.QGridLayout()
    self.time_input_rows = []
    self.time_input_container = Qt.QWidget()
    self.time_input_container.setLayout(self.time_input_layout)
    self.top_layout.addWidget(self.time_input_container, 2, 0, 1, 5)

    self.add_time_row()

def _build_frequency_mode_ui(self):
    self.port_inputs = {}

    ports = ['A4', 'A3', 'A2', 'A1', 'B4', 'B3', 'B2', 'B1']
    default_ranges = ['0:100', '101:200', '201:300', '301:400',
                      '401:700', '701:1000', '1001:2000', '2001:4000']

    port_layout = Qt.QGridLayout()
    for i, port in enumerate(ports):
        port_label = Qt.QLabel(f"{port} (MHz):")
        port_input = Qt.QLineEdit(default_ranges[i])
        port_layout.addWidget(port_label, 0, i)
        port_layout.addWidget(port_input, 1, i)
        self.port_inputs[port] = port_input
        port_input.editingFinished.connect(self.refresh_freq_config)

    self.port_error_label = Qt.QLabel("")
    self.port_error_label.setStyleSheet("color: red; font-weight: bold")
    self.port_error_label.setAlignment(Qt.Qt.AlignCenter)
    self.port_error_label.hide()
    port_layout.addWidget(self.port_error_label, 2, 0, 1, len(ports))
    self.top_layout.addLayout(port_layout, 1, 0)

def _build_sweep_mode_ui(self, form):
    self.sweep_start_edit = Qt.QLineEdit("80e6")
    self.sweep_end_edit = Qt.QLineEdit("100e6")
    form.addWidget(Qt.QLabel("Sweep Start:"))
    form.addWidget(self.sweep_start_edit)
    form.addWidget(Qt.QLabel("Sweep End:"))
    form.addWidget(self.sweep_end_edit)

    self.sweep_error_label = Qt.QLabel("")
    self.sweep_error_label.setStyleSheet("color: red; font-weight: bold")
    self.sweep_error_label.setAlignment(Qt.Qt.AlignCenter)
    self.sweep_error_label.hide()
    self.top_layout.addWidget(self.sweep_error_label, 1, 0, 1, 5)

    self.sweep_start_edit.returnPressed.connect(self.validate_sweep_inputs)
    self.sweep_end_edit.returnPressed.connect(self.validate_sweep_inputs)

    self.last_valid_sweep_start = parse_freq_input(self.sweep_start_edit.text())
    self.last_valid_sweep_end = parse_freq_input(self.sweep_end_edit.text())

    self._rebuilding = False
```

```
def _port_title(self, p: str) -> str:
        tag = self.port_tags.get(p, "")
        return f"{p} ({tag})" if tag else p

def _build_port_sweep_ui(self):
    # Hidden edits so validate_sweep_inputs() keeps working as-is
    self.sweep_start_edit = Qt.QLineEdit("80e6")
    self.sweep_end_edit   = Qt.QLineEdit("100e6")
    self.sweep_error_label = Qt.QLabel("")
    self.sweep_error_label.setStyleSheet("color: red; font-weight: bold")
    self.sweep_error_label.setAlignment(Qt.Qt.AlignCenter)
    self.sweep_error_label.hide()

    # Port grid (like frequency mode, but with radios)
    self.port_inputs  = {}
    self.port_buttons = {}
    self.selected_port = None

    ports = ['A4','A3','A2','A1','B4','B3','B2','B1']
    default_ranges = ['87.5:108', '894.8:939.8', '1977.4:2167.4', '793.5:834.5',
                      '1730:1825', '2570:2620', '3300:3800', '0:1']

    grid = Qt.QGridLayout()
    grid.setHorizontalSpacing(10)
    grid.setVerticalSpacing(6)

    # Radio group for exclusivity
    self._port_group = Qt.QButtonGroup(self)
    self._port_group.setExclusive(True)

    # Cosmetic: green when selected
    rb_css = """
    QRadioButton::indicator {
        width: 14px; height: 14px;
        border: 2px solid #888; border-radius: 8px; background: #222;
    }
    QRadioButton::indicator:checked {
        background: #2ecc71;
        border: 2px solid #2ecc71;
    }
    """

    for i, port in enumerate(ports):
        col = i
        rb = Qt.QRadioButton(self._port_title(port))
        rb.setStyleSheet(rb_css)
        edit = Qt.QLineEdit(default_ranges[i])
        edit.setAlignment(Qt.Qt.AlignCenter)
        edit.setFixedWidth(102)
        edit.setToolTip("MHz range, like 80:120")

        wrap = Qt.QWidget()
        hl = Qt.QHBoxLayout(wrap)
        hl.setContentsMargins(0, 0, 0, 0)
        hl.setSpacing(4)
        hl.addWidget(edit)
        unit = Qt.QLabel("MHz")
        unit.setStyleSheet("color:#black;")
        hl.addWidget(unit)

        self.port_inputs[port]  = edit
        self.port_buttons[port] = rb
        self._port_group.addButton(rb)

        # Row 0: radios, Row 1: edits
        grid.addWidget(rb,   0, col, alignment=Qt.Qt.AlignHCenter)
        grid.addWidget(wrap, 1, col, alignment=Qt.Qt.AlignHCenter)

        # When user edits ranges while selected, re-apply sweep limits on the fly
        def _on_edit_finished(p=port):
            if self.selected_port == p and self.toggle_btn.isChecked():
                self._select_port_and_sweep(p)
        edit.editingFinished.connect(_on_edit_finished)

        def _on_toggled(on, p=port):
            if on:
                self._select_port_and_sweep(p)
        rb.toggled.connect(_on_toggled)

    # Place grid + (hidden) error label
    self.top_layout.addLayout(grid, 1, 0)
    self.top_layout.addWidget(self.sweep_error_label, 2, 0, 1, 5)

    # Auto-select a sensible default (current antenna if matches, else A4)
    default_port = self.antenna_select if self.antenna_select in ports else 'A4'

    # Prevent _on_toggled from firing during initial check mark
    self._port_group.blockSignals(True)
```

```
        self.port_buttons[default_port].setChecked(True)
        self._port_group.blockSignals(False)

        # Prime sweep ranges (no hardware touch thanks to guarded update_antenna)
        self._select_port_and_sweep(default_port, start_immediately=False)

    # SDR sub-pipelines
    def _build_sweep_pipeline(self):
        self.stream_to_vector = blocks.stream_to_vector(gr.sizeof_gr_complex, self.fft_size)
        self.fft = fft_vcc(self.fft_size, True, window.blackmanharris(self.fft_size), True)
        self.c2mag = blocks.complex_to_mag(self.fft_size)
        self.probe = blocks.probe_signal_vf(self.fft_size)

        self.connect((self.src), (self.stream_to_vector, 0))
        self.connect((self.stream_to_vector, 0), (self.fft, 0))
        self.connect((self.fft, 0), (self.c2mag, 0))
        self.connect((self.c2mag, 0), (self.probe, 0))

    def _build_sweep_plot(self):
        self.pg_plot = PlotWidget()
        self.pg_plot.setTitle("Wideband Frequency Spectrum")
        self.pg_plot.setLabel("bottom", "Frequency", units="MHz")
        self.pg_plot.setLabel("left", "Relative Gain", units="dB")
        self.pg_plot.showGrid(x=True, y=True)
        self.pg_plot.setYRange(-120, 0)
        self.pg_plot.setBackground('w')
        self.pg_curve = self.pg_plot.plot(pen=mkPen(color='b', width=1))

        # Extra traces for peak-hold (hidden by default)
        self.pg_peak = self.pg_plot.plot(pen=mkPen(width=1))
        self.pg_peak.setVisible(False)

        # Lock autorange BEFORE start to avoid jumpy view
        self.pg_plot.enableAutoRange(x=False, y=False)
        self._update_sweep_xrange()

        # Crosshair (start hidden)
        self._vline = pg.InfiniteLine(angle=90, movable=False); self._vline.setVisible(False)
        self._hline = pg.InfiniteLine(angle=0,  movable=False); self._hline.setVisible(False)
        self.pg_plot.addItem(self._vline, ignoreBounds=True)
        self.pg_plot.addItem(self._hline, ignoreBounds=True)

        self._mouse_text = pg.TextItem("", anchor=(1,1), color="k")
        self._mouse_text.setVisible(False)
        self.pg_plot.addItem(self._mouse_text)

        vb = self.pg_plot.getViewBox()

        def _on_move(pos):
            inside = self.pg_plot.sceneBoundingRect().contains(pos)
            if not inside:
                # hide crosshair + label when cursor leaves the plot
                self._vline.setVisible(False)
                self._hline.setVisible(False)
                self._mouse_text.setVisible(False)
                return

            mousePoint = vb.mapSceneToView(pos)
            fx = mousePoint.x(); fy = mousePoint.y()
            self._vline.setPos(fx); self._hline.setPos(fy)
            self._vline.setVisible(True); self._hline.setVisible(True)

            self._mouse_text.setText(f"{fx:.3f} MHz\n{fy:.1f} dB")
            self._mouse_text.setPos(fx, fy)
            self._mouse_text.setVisible(True)

        self.pg_plot.scene().sigMouseMoved.connect(_on_move)

        # Right-click anywhere on the plot to reset peak envelopes
        def _on_click(ev):
            if ev.button() == Qt.Qt.RightButton:
                self.peak_env = None
                if hasattr(self, "_peak_scatter"):
                    self._peak_scatter.clear()
                ev.accept()
        self.pg_plot.scene().sigMouseClicked.connect(_on_click)

        self.top_layout.addWidget(self.pg_plot, 2, 0)

        # Peak toggle buttons on the toolbar
        self.peak_enabled = False

        btn_wrap = Qt.QWidget()
        btns = Qt.QHBoxLayout(btn_wrap)
        btns.setContentsMargins(0, 0, 0, 0)
        btns.setSpacing(6)

        self.btn_peak_hold = Qt.QToolButton()
        self.btn_peak_hold.setText("Max Hold")
```

```
    self.btn_peak_hold.setCheckable(True)
    self.btn_peak_hold.setChecked(False)

    btns.addWidget(self.btn_peak_hold)

    btns.addWidget(Qt.QLabel("Avg:"))
    self.sweep_avg_alpha = 0.20
    self.sweep_avg_spin = Qt.QDoubleSpinBox()
    self.sweep_avg_spin.setRange(0.0, 1.0)
    self.sweep_avg_spin.setSingleStep(0.05)
    self.sweep_avg_spin.setDecimals(2)
    self.sweep_avg_spin.setValue(0.20)  # default like the Qt sink
    self.sweep_avg_spin.setToolTip("Exponential averaging factor (0 = off, 1 = full)")
    self.sweep_avg_spin.setFixedWidth(72)
    btns.addWidget(self.sweep_avg_spin)

    # Sweep EMA factor
    def _on_sweep_avg_changed(val):
        try:
            self.sweep_avg_alpha = float(val)
            if self.sweep_avg_alpha <= 0.0:
                # make "0" truly = no averaging
                self.sweep_prev_linear = None
        except Exception:
            pass

    self.sweep_avg_spin.valueChanged.connect(_on_sweep_avg_changed)

    # Place buttons just before Start/Stop if present
    idx = self.toolbar.indexOf(self.toggle_btn) if hasattr(self, "toggle_btn") else -1
    if idx >= 0:
        self.toolbar.insertWidget(idx, btn_wrap)
    else:
        self.toolbar.addWidget(btn_wrap)

    def _on_peak_toggle(on: bool):
        self.peak_enabled = on
        self.pg_peak.setVisible(on)
        if not on:
            self.peak_env = None  # clear stored envelope when disabled

    self.sweep_db_cal = 0.0
    self.btn_peak_hold.toggled.connect(_on_peak_toggle)
    self._ensure_mouse_blocker()

def _build_jamming_pipeline(self, form):
    self.stream_to_vector = blocks.stream_to_vector(gr.sizeof_gr_complex, self.fft_size)
    self.fft = fft_vcc(self.fft_size, True, window.blackmanharris(self.fft_size), True)
    self.c2mag = blocks.complex_to_mag(self.fft_size)
    self.probe = blocks.probe_signal_vf(self.fft_size)

    self.connect((self.src), (self.stream_to_vector, 0))
    self.connect((self.stream_to_vector, 0), (self.fft, 0))
    self.connect((self.fft, 0), (self.c2mag, 0))
    self.connect((self.c2mag, 0), (self.probe, 0))

    # Detector params/state
    self.jam_alpha          = 0.05   # baseline EMA
    self.jam_thresh_db      = 6.5    # per-bin ΔdB over baseline
    self.jam_occ_min        = 0.5    # base occupancy requirement
    self.jam_occ_multipeak  = 0.15   # occupancy for multipeak branch
    self.jam_span_min       = 0.008  # contiguous run fraction (~0.8% of masked bins)
    self.jam_sfm_min_db     = -5.0   # allow OFDM (not super flat) to trigger
    self._sfm_offset        = 0.8    # SFM gate = ref + 0.8 dB
    self.jam_median_db      = 2.0    # median-lift fallback (dB)
    self.jam_band_db        = 1.5    # band-power (mean) lift (dB)
    self.jam_hold_s         = 0.4    # hold time to confirm
    self.jam_cooldown_s     = 4.0    # cooldown after switching
    self.jam_ready_at       = time.time() + 1.5
    self.jam_baseline_db    = None
    self.jam_started        = None
    self.jam_cooldown_until = 0.0
    self.sfm_ref            = None
    self.ports = ['A4','A3','A2','A1','B4','B3','B2','B1']
    self.current_port_index = self.ports.index(self.antenna_select) if self.antenna_select in self.ports else 0

    self._rebuild_jam_mask()

    # Debug
    self.jam_debug = True
    self._jam_last_dbg = 0.0

    # Knobs
    knob_bar = Qt.QHBoxLayout()
    knob_bar.setContentsMargins(0, 0, 0, 0)
    knob_bar.setSpacing(6)

    h = self.freq_edit.sizeHint().height() if hasattr(self, "freq_edit") else 24
```

```
    def _mk_edit(txt, tip, vmin, vmax, prec, width=56):
        e = Qt.QLineEdit(txt)
        e.setAlignment(Qt.Qt.AlignCenter)
        e.setFixedWidth(width)
        e.setFixedHeight(h)
        dv = QDoubleValidator(vmin, vmax, prec, self)
        dv.setNotation(QDoubleValidator.ScientificNotation)
        e.setValidator(dv)
        e.setToolTip(tip)

        def on_change(_):
            if e.text() == "":
                e.setStyleSheet("")
            elif e.hasAcceptableInput():
                e.setStyleSheet("")
            else:
                e.setStyleSheet("border: 2px solid #d00;")
        e.textChanged.connect(on_change)
        return e

    knob_bar.addWidget(Qt.QLabel("ΔdB:"))
    self.jam_thresh_edit = _mk_edit(
        str(self.jam_thresh_db), "Per-bin threshold above baseline (dB)", -100.0, 100.0, 1
    )
    knob_bar.addWidget(self.jam_thresh_edit)

    knob_bar.addWidget(Qt.QLabel("Occ:"))
    self.jam_occ_edit = _mk_edit(
        str(self.jam_occ_min), "Minimum fraction of bins above ΔdB (0–1)", 0.0, 1.0, 2
    )
    knob_bar.addWidget(self.jam_occ_edit)

    knob_bar.addWidget(Qt.QLabel("SFM≥:"))
    self.jam_sfm_edit = _mk_edit(
        str(self.jam_sfm_min_db), "Minimum spectral flatness (dB)", -60.0, 10.0, 1
    )
    knob_bar.addWidget(self.jam_sfm_edit)

    knob_wrap = Qt.QWidget()
    knob_wrap.setLayout(knob_bar)
    start_idx = form.indexOf(self.toggle_btn)
    if start_idx >= 0:
        form.insertWidget(start_idx, knob_wrap)
    else:
        form.addWidget(knob_wrap)

    # Wire inputs to strict apply
    self.jam_thresh_edit.returnPressed.connect(self._apply_jam_knobs)
    self.jam_occ_edit.returnPressed.connect(self._apply_jam_knobs)
    self.jam_sfm_edit.returnPressed.connect(self._apply_jam_knobs)
    # React on focus loss
    self.jam_thresh_edit.editingFinished.connect(self._apply_jam_knobs)
    self.jam_occ_edit.editingFinished.connect(self._apply_jam_knobs)
    self.jam_sfm_edit.editingFinished.connect(self._apply_jam_knobs)

def _apply_jam_knobs(self):
    bad = False
    try:
        new_thresh = parse_float_strict(self.jam_thresh_edit.text(), -100.0, 100.0, name="ΔdB threshold")
        self.jam_thresh_edit.setStyleSheet("")
    except ValueError as e:
        bad = True; self.jam_thresh_edit.setStyleSheet("border: 2px solid #d00;"); print(f"[ERROR] {e}")

    try:
        new_occ = parse_float_strict(self.jam_occ_edit.text(), 0.0, 1.0, name="occupancy")
        self.jam_occ_edit.setStyleSheet("")
    except ValueError as e:
        bad = True; self.jam_occ_edit.setStyleSheet("border: 2px solid #d00;"); print(f"[ERROR] {e}")

    try:
        new_sfm = parse_float_strict(self.jam_sfm_edit.text(), -60.0, 10.0, name="SFM minimum")
        self.jam_sfm_edit.setStyleSheet("")
    except ValueError as e:
        bad = True; self.jam_sfm_edit.setStyleSheet("border: 2px solid #d00;"); print(f"[ERROR] {e}")

    if bad:
        self._stop_stream_safely()
        return

    self.jam_thresh_db  = new_thresh
    self.jam_occ_min    = new_occ
    self.jam_sfm_min_db = new_sfm
    self.jam_ignore_until = time.time() + 0.5

    if getattr(self, "jam_debug", False):
        print(f"[JDBG] knobs applied: ΔdB={self.jam_thresh_db:.2f}, "
            f"Occ={self.jam_occ_min:.2f}, SFM≥{self.jam_sfm_min_db:.2f} dB")

#  Close / Cleanup
```

```
def closeEvent(self, event):
    try:
        self.stop()
        self.wait()
    except Exception as e:
        print(f"[WARN] Exception on stop: {e}")

    if self.mode == "time" and hasattr(self, "timer"):
        self.timer.stop()
        self.timer_start_time = None

    if self.mode in ("wide spectrum", "wide spectrum frequency") and hasattr(self, "sweep_timer"):
        self.sweep_timer.stop()

    if self.mode == "jamming" and hasattr(self, "jam_timer"):
        self.jam_timer.stop()

    if hasattr(self.parent(), "oc_panel") and hasattr(self, "antenna_select"):
        try:
            parent_connected = self.parent().is_hackrf_connected() if hasattr(self.parent(), "is_hackrf_connected") else self.is_hackrf_connected()
            self.parent().oc_panel.set_active(self.antenna_select, fixed_input="A0", connected=parent_connected)
        except Exception:
            pass

    self.closed.emit()
    event.accept()

#  Plots / Sinks
def init_waterfall(self):
    if self.mode == "delay":
        self.waterfall = qtgui.waterfall_sink_c(
            self.fft_size, window.WIN_BLACKMAN_hARRIS, self.center_freq, self.samp_rate, "Spectrogram", 1, None
        )
        self.waterfall.set_update_time(0.0005)
    else:
        self.waterfall = qtgui.waterfall_sink_c(
            self.fft_size, window.WIN_BLACKMAN_hARRIS, self.center_freq, self.samp_rate, "Spectrogram", 1, None
        )
        self.waterfall.set_update_time(0.10)

    self.waterfall.enable_grid(False)
    self.waterfall.enable_axis_labels(True)
    self.waterfall.set_intensity_range(-140, 10)

    self._waterfall_win = sip.wrapinstance(self.waterfall.qwidget(), Qt.QWidget)
    self._waterfall_win.setContentsMargins(0, 0, 0, 0)
    self._waterfall_win.setSizePolicy(Qt.QSizePolicy.Expanding, Qt.QSizePolicy.Expanding)

    self.top_layout.addWidget(self._waterfall_win, 3, 0, 1, 5)
    if self.mode != "wide spectrum":
        self.connect((self.src, 0), (self.waterfall, 0))

def init_freq_spectrum(self):
    self.freq_sink = qtgui.freq_sink_c(
        self.fft_size, window.WIN_BLACKMAN_hARRIS, self.center_freq, self.samp_rate, "Frequency Spectrum", 1
    )
    self.freq_sink.set_update_time(0.10)
    self.freq_sink.set_y_axis(-120, 0)
    self.freq_sink.set_y_label("Relative Gain", "dB")
    self.freq_sink.enable_grid(True)
    self.freq_sink.enable_axis_labels(True)
    self.freq_sink.disable_legend()
    self.freq_sink.set_line_label(0, "Spectrum")

    try:
        self.freq_sink.enable_control_panel(False)
    except Exception:
        pass

    self.freq_sink.set_fft_average(0.20)
    try:
        self.freq_sink.enable_max_hold(False)   # start with Max Hold off
        self.freq_sink.enable_min_hold(False)   # force Min Hold off
    except Exception:
        pass

    self._freq_win = sip.wrapinstance(self.freq_sink.qwidget(), Qt.QWidget)
    self._freq_win.setContentsMargins(0, 0, 0, 0)
    self._freq_win.setSizePolicy(Qt.QSizePolicy.Expanding, Qt.QSizePolicy.Expanding)
    self.top_layout.addWidget(self._freq_win, 4, 0, 1, 5)

    if self.mode != "wide spectrum":
        self.connect((self.src, 0), (self.freq_sink, 0))

    if (getattr(self, "btn_fs_max", None) is not None) and (getattr(self, "spin_fs_avg", None) is not None):
        def _on_max_toggled(on):
            try:
                # never allow Min Hold
                self.freq_sink.enable_min_hold(False)
```

```
            self.freq_sink.enable_max_hold(bool(on))
        except Exception:
            pass

    def _on_avg_changed(val):
        try:
            self.freq_sink.set_fft_average(float(val))
        except Exception:
            pass

    # connect once
    try:
        self.btn_fs_max.toggled.disconnect()
    except Exception:
        pass
    try:
        self.spin_fs_avg.valueChanged.disconnect()
    except Exception:
        pass

    self.btn_fs_max.toggled.connect(_on_max_toggled)
    self.spin_fs_avg.valueChanged.connect(_on_avg_changed)

    _on_max_toggled(self.btn_fs_max.isChecked())
    _on_avg_changed(self.spin_fs_avg.value())

#  Update Handlers (freq/rate/antenna/gains/fft)
def update_center_freq(self):
    try:
        freq = parse_freq_input(self.freq_edit.text())

        # Clamp
        if freq <= 0:
            freq = 1.0
            self.freq_edit.setText("1")
            self.show_temporary_label("freq_range_label", "0 < fc <= 4e9. Set to to 1 Hz.")
        elif freq > 4e9:
            freq = 4e9
            self.freq_edit.setText("4e9")
            self.show_temporary_label("freq_range_label", "0 < fc <= 4e9. Set to to 4e9 Hz.")
        else:
            # only hide when no clamp
            self._hide_label("freq_range_label")

        if abs(freq - getattr(self, "center_freq", 0)) < 1e-6:
            return

        temp_freq = freq

        if self.mode == "frequency":
            self.center_freq = temp_freq
            if not self.validate_port_ranges():
                print("[ABORT] Central frequency not updated due to invalid port ranges.")
                self.freq_edit.setText(f"{self.center_freq / 1e6:.2f} MHz")
                if self.toggle_btn.isChecked():
                    self.toggle_btn.setChecked(False)
                    self.toggle_stream()
                return

        self.center_freq = temp_freq
        self.src.set_center_freq(temp_freq, 0)

        if hasattr(self, "waterfall"):
            self.waterfall.set_frequency_range(temp_freq, self.samp_rate)
        if hasattr(self, "freq_sink"):
            self.freq_sink.set_frequency_range(temp_freq, self.samp_rate)

        print(f"[INFO] Central frequency set to {temp_freq:.2f} Hz")

        if self.mode == "frequency":
            self.auto_switch_port_if_needed()

        if self.mode == "jamming":
            self.jam_baseline_db = None
            self.jam_ready_at = time.time() + 1.5

    except ValueError as e:
        self._input_error("freq_range_label", str(e), field=self.freq_edit)
        return

def update_sample_rate(self):
    try:
        sr = parse_freq_input(self.samp_rate_edit.text())

        if sr <= 0:
            sr = 1.0
            self.samp_rate_edit.setText("1")
            self.show_temporary_label("samp_rate_range_label", "0 < sr <= 20e6. Set to 1 S/s.")
        elif sr > 20e6:
```

```
            sr = 20e6
            self.samp_rate_edit.setText("20e6")
            self.show_temporary_label("samp_rate_range_label", "0 < sr <= 20e6. Set to 20 MS/s.")
        else:
            self._hide_label("samp_rate_range_label")

        if self.mode in ("wide spectrum", "wide spectrum frequency"):
            if self.sweep_active:
                print("[INFO] Sweep active — applying sample rate live.")
                self.lock()
                try:
                    self.src.set_sample_rate(sr)
                    self.samp_rate = sr
                    self.validate_sweep_inputs()
                finally:
                    self.unlock()
                self.sweep_ptr = 0
                self.sweep_timer.stop()
                self.sweep_active = True
                QTimer.singleShot(100, self.begin_sweep_timer)
            else:
                self.src.set_sample_rate(sr)
                self.samp_rate = sr
                self.validate_sweep_inputs()
        else:
            self.src.set_sample_rate(sr)
            self.samp_rate = sr

            if self.mode == "jamming":
                self.jam_baseline_db = None
                self.jam_ready_at = time.time() + 1.5

            if hasattr(self, "waterfall"):
                self.waterfall.set_frequency_range(self.center_freq, sr)
            if hasattr(self, "freq_sink"):
                self.freq_sink.set_frequency_range(self.center_freq, sr)

        print(f"[INFO] Sample rate set to {sr:.2f} S/s")

    except ValueError as e:
        self._input_error("samp_rate_range_label", str(e), field=self.samp_rate_edit)
        return

def update_antenna(self, val: str = None):
    if val is None:
        if self.antenna_edit is None:
            val = self.antenna_select
        else:
            val = str(self.antenna_edit.text()).strip().upper()
    else:
        val = str(val).strip().upper()

    valid = ['A1','A2','A3','A4','B1','B2','B3','B4']
    if val in valid:
        self.antenna_select = val

        # SDR build guard
        if hasattr(self, "src") and self.src is not None:
            try:
                self.src.set_antenna(val, 0)
            except Exception as e:
                print(f"[WARN] set_antenna({val}) failed: {e}")

        print(f"[INFO] Switched antenna to: {val}")

        # Panel build guard
        if hasattr(self, "oc_panel") and self.oc_panel is not None:
            self.oc_panel.set_active(val, fixed_input="A0", connected=self.is_hackrf_connected())

        self.antenna_changed.emit(val)

        if self.mode == "jamming":
            self.jam_baseline_db = None
            self.jam_ready_at = time.time() + 1.5
            if hasattr(self, "ports") and self.antenna_select in self.ports:
                self.current_port_index = self.ports.index(self.antenna_select)
            self.jam_last_switch = time.time()
    else:
        print(f"[WARN] Invalid antenna: {val}")
        if self.antenna_error_label is not None:
            self.show_temporary_label(
                "antenna_error_label",
                f"Antenna '{val}' doesn't exist.\nAvailable: {', '.join(valid)}"
            )
        if self.antenna_edit is not None:
            self.antenna_edit.setText(self.antenna_select)

def update_gains(self):
    if not getattr(self, "_gains_dirty", False):
```

```
            return

        fields = [
            ("RF Gain", self.rf_gain_edit, "rf_gain_error_label", self.rf_gain_min, self.rf_gain_max),
            ("IF Gain", self.if_gain_edit, "if_gain_error_label", self.if_gain_min, self.if_gain_max),
            ("BB Gain", self.bb_gain_edit, "bb_gain_error_label", self.bb_gain_min, self.bb_gain_max),
        ]

        # Empty/invalid erro message
        for name, edit, lbl_attr, vmin, vmax in fields:
            if edit.text().strip() == "" or not edit.hasAcceptableInput():
                edit.setStyleSheet("border: 2px solid #d00;")
                QTimer.singleShot(1200, lambda e=edit: e.setStyleSheet(""))
                self.show_temporary_label(lbl_attr, f"{name} must be an integer between {vmin} and {vmax}")
                self._stop_stream_safely()
                return

        raw_vals, clamped_vals, clamped_flags = [], [], []
        for name, edit, lbl_attr, vmin, vmax in fields:
            raw = int(edit.text())
            val = clamp(raw, vmin, vmax)
            raw_vals.append(raw)
            clamped_vals.append(val)
            clamped_flags.append(val != raw)

        rf, ifg, bb = clamped_vals

        # Reflect values
        for (_, edit, _, _, _), val in zip(fields, clamped_vals):
            edit.setText(str(val))

        # Per-field inline error when clamped
        for (name, edit, lbl_attr, vmin, vmax), was_clamped in zip(fields, clamped_flags):
            if was_clamped:
                self.show_temporary_label(lbl_attr, f"{name} must be an integer between {vmin} and {vmax}")
                edit.setStyleSheet("border: 2px solid #d00;")
                QTimer.singleShot(1200, lambda e=edit: e.setStyleSheet(""))

        if (rf, ifg, bb) == getattr(self, "_last_applied_gains", (None, None, None)) and not any(clamped_flags):
            self._gains_dirty = False
            return

        # Apply gains
        self.rf_gain, self.if_gain, self.bb_gain = rf, ifg, bb
        try:
            self.src.set_gain(rf, 0)
            self.src.set_if_gain(ifg, 0)
            self.src.set_bb_gain(bb, 0)
            print(f"[INFO] Gains set: RF={rf}, IF={ifg}, BB={bb}")
        except Exception as e:
            print(f"[WARN] Applying gains failed: {e}")

        self._last_applied_gains = (rf, ifg, bb)
        self._gains_dirty = False

        if self.mode == "jamming":
            self.jam_baseline_db = None
            self.jam_ready_at = time.time() + 1.5

    def update_fft_size(self):
        try:
            size = int(self.fft_combo.currentText())
        except Exception:
            print("[ERROR] FFT size update: invalid combo value")
            return
        if size == self.fft_size:
            return

        print(f"[INFO] FFT size changed to {size}, rebuilding sinks.")
        was_running = self.toggle_btn.isChecked()

        # stop timers that could fire mid-rebuild
        if self.mode in ("wide spectrum", "wide spectrum frequency") and hasattr(self, "sweep_timer"):
            try:
                self.sweep_timer.stop()
                self.sweep_timer.timeout.disconnect(self.update_sweep_freq)
            except Exception:
                pass
        if self.mode == "jamming" and hasattr(self, "jam_timer"):
            try:
                self.jam_timer.stop()
            except Exception:
                pass

        self._rebuilding = True

        # Stop FG if it was running
        if was_running:
            try:
```

```
            self.stop(); self.wait()
        except Exception:
            pass

    # (optional) hide footer image briefly to avoid flicker
    footer_was_vis = hasattr(self, "oc_panel") and self.oc_panel.isVisible()
    if footer_was_vis:
        try: self.oc_panel.hide()
        except Exception: pass

    # disconnect everything & invalidate old DSP refs
    try:
        self.disconnect_all()
    except Exception:
        pass
    self.stream_to_vector = None
    self.fft = None
    self.c2mag = None
    self.probe = None

    # apply new size
    self.fft_size = size

    try:
        if self.mode in ("wide spectrum", "wide spectrum frequency"):
            # Fresh sweep chain for new size
            self.stream_to_vector = blocks.stream_to_vector(gr.sizeof_gr_complex, self.fft_size)
            self.fft = fft_vcc(self.fft_size, True, window.blackmanharris(self.fft_size), True)
            self.c2mag = blocks.complex_to_mag(self.fft_size)
            self.probe = blocks.probe_signal_vf(self.fft_size)

            # reset size-dependent state
            self.sweep_prev_linear = None
            self.peak_env = None
            self.last_fft = None
            self._cap_retries = 0
            self._step_freq_hz = None
            self.sweep_busy = False

            # recompute sweep plan & buffer
            ok = self.validate_sweep_inputs()
            if not ok:
                print("[BLOCK] Sweep config invalid after FFT resize.")

            # connect pipeline
            self.connect((self.src), (self.stream_to_vector, 0))
            self.connect((self.stream_to_vector, 0), (self.fft, 0))
            self.connect((self.fft, 0), (self.c2mag, 0))
            self.connect((self.c2mag, 0), (self.probe, 0))

        elif self.mode == "jamming":
            # Rebuild QtGUI sinks (waterfall/freq)
            if hasattr(self, "_waterfall_win"):
                try:
                    self.top_layout.removeWidget(self._waterfall_win)
                    self._waterfall_win.deleteLater()
                except Exception:
                    pass
            if hasattr(self, "_freq_win"):
                try:
                    self.top_layout.removeWidget(self._freq_win)
                    self._freq_win.deleteLater()
                except Exception:
                    pass
            self.init_waterfall()
            self.init_freq_spectrum()

            # Rebuild jamming DSP chain
            self._reinit_jamming_dsp_chain()

            # Re-apply toolbar state to the new freq_sink
            if hasattr(self, "btn_fs_max") and self.btn_fs_max is not None:
                try:
                    self.freq_sink.enable_min_hold(False)
                    self.freq_sink.enable_max_hold(self.btn_fs_max.isChecked())
                except Exception:
                    pass
            if hasattr(self, "spin_fs_avg") and self.spin_fs_avg is not None:
                try:
                    self.freq_sink.set_fft_average(float(self.spin_fs_avg.value()))
                except Exception:
                    pass

        else:
            # Rebuild QtGUI for non sweep modes
            if hasattr(self, "_waterfall_win"):
                try:
                    self.top_layout.removeWidget(self._waterfall_win)
                    self._waterfall_win.deleteLater()
```

```
                except Exception:
                    pass
            if hasattr(self, "_freq_win"):
                try:
                    self.top_layout.removeWidget(self._freq_win)
                    self._freq_win.deleteLater()
                except Exception:
                    pass
            self.init_waterfall()
            self.init_freq_spectrum()

        except Exception as e:
            print(f"[ERROR] FFT size rebuild failed: {e}")

        try:
            if was_running:
                self.start()
                if self.mode in ("wide spectrum", "wide spectrum frequency"):
                    self.sweep_ptr = 0
                    self.sweep_active = True
                    try:
                        self.begin_sweep_timer()
                    except Exception as e:
                        print(f"[WARN] begin_sweep_timer failed: {e}")
                elif self.mode == "jamming" and hasattr(self, "jam_timer"):
                    try:
                        self.jam_timer.start(100)
                    except Exception:
                        pass
            else:
                if self.mode in ("wide spectrum", "wide spectrum frequency"):
                    self.sweep_active = False
        finally:
            if footer_was_vis:
                try:
                    self.oc_panel.show()
                    self.oc_panel.repaint()
                except Exception:
                    pass
            self._rebuilding = False

    #  Frequency-mode helpers
    def auto_switch_port_if_needed(self):
        if not self.validate_port_ranges():
            print("[ABORT] Port ranges are invalid — please fix them first.")
            return

        current_mhz = self.center_freq / 1e6
        selected_port = None
        for port, line in self.port_inputs.items():
            text = line.text().strip()
            if not text:
                continue
            try:
                parts = text.replace(" ", "").split(":")
                low = float(parts[0])
                high = float(parts[1])
                print(f"[DEBUG] Checking port {port} for range {low}–{high} MHz")
                if low <= current_mhz <= high:
                    selected_port = port
                    break
            except Exception as e:
                print(f"[WARN] Bad format in {port}: '{text}' — {e}")

        if selected_port:
            if selected_port != self.antenna_select:
                print(f"[AUTO-SWITCH] Switching to {selected_port} for {current_mhz:.2f} MHz")
                self.update_antenna(selected_port)
            else:
                print(f"[SKIP] {selected_port} already selected.")
        else:
            print(f"[INFO] No matching port for {current_mhz:.2f} MHz")

    def validate_port_ranges(self):
        ranges = []
        for port, line in self.port_inputs.items():
            text = line.text().strip()
            try:
                parts = text.replace(" ", "").split(":")
                if len(parts) != 2:
                    raise ValueError("Invalid format")
                low = float(parts[0])
                high = float(parts[1])
                if low >= high:
                    raise ValueError("Lower bound must be < upper bound")
                if high > 4000:
                    raise ValueError("Upper limit exceeds 4000 MHz")
                ranges.append((low, high, port))
            except Exception as e:
```

```
            msg = f"Invalid range in {port}: '{text}' — {e}"
            print(msg)
            self.show_temporary_label("port_error_label", msg)
            return False

    ranges.sort()
    for i in range(len(ranges) - 1):
        current_high = ranges[i][1]
        next_low = ranges[i + 1][0]
        if current_high >= next_low:
            msg = (f"Overlapping or touching ranges: {ranges[i][2]} ends at {current_high}, "
                   f"{ranges[i+1][2]} starts at {next_low}")
            print(msg)
            self.show_temporary_label("port_error_label", msg)
            return False
    return True

# Time mode
def add_time_row(self):
    index = len(self.time_inputs)
    row = index // 4
    col = (index % 4) * 2

    port_label = Qt.QLabel(f"Port {index + 1}:")
    port_input = Qt.QComboBox()
    valid_ports = ['A1', 'A2', 'A3', 'A4', 'B1', 'B2', 'B3', 'B4']
    port_input.addItems(valid_ports)
    dur_label = Qt.QLabel("Duration (s)")
    dur_input = Qt.QLineEdit()

    dur_input.setPlaceholderText("seconds > 0")

    # Accept positive floats (optionally allow scientific notation)
    v = QDoubleValidator(0.000001, 1e9, 6, self)   # min, max, decimals
    v.setNotation(QDoubleValidator.ScientificNotation)  # allow "1e-3" etc
    dur_input.setValidator(v)

    def _on_dur_edited():
        txt = dur_input.text().strip()
        if not txt:
            dur_input.setStyleSheet("")
            dur_input.setToolTip("")
            return
        # Acceptable? clear; otherwise paint red
        if dur_input.hasAcceptableInput():
            dur_input.setStyleSheet("")
            dur_input.setToolTip("")
        else:
            dur_input.setStyleSheet("border: 2px solid #d00;")
            dur_input.setToolTip("Enter a positive number of seconds (e.g., 1, 0.5, 2e-3)")

    dur_input.textChanged.connect(_on_dur_edited)

    self.time_input_layout.addWidget(port_label, row * 2, col)
    self.time_input_layout.addWidget(port_input, row * 2 + 1, col)
    self.time_input_layout.addWidget(dur_label, row * 2, col + 1)
    self.time_input_layout.addWidget(dur_input, row * 2 + 1, col + 1)

    self.time_inputs.append((port_input, dur_input))
    self.time_labels.append((port_label, dur_label))

    def safe_refresh():
        if hasattr(self, "toggle_btn") and self.toggle_btn.isChecked():
            self.refresh_time_config()

    dur_input.editingFinished.connect(safe_refresh)
    port_input.currentIndexChanged.connect(safe_refresh)

def remove_time_row(self):
    if self.time_inputs:
        port_input, dur_input = self.time_inputs.pop()
        port_label, dur_label = self.time_labels.pop()
        for w in [port_input, dur_input, port_label, dur_label]:
            self.time_input_layout.removeWidget(w)
            w.deleteLater()
        if hasattr(self, "toggle_btn") and self.toggle_btn.isChecked():
            valid = self.refresh_time_config()
            if not valid:
                print("[INFO] All ports removed. Stream halted.")
    else:
        self.show_temporary_label("time_error_label", "No more ports to remove.")

# Start/Stop + Timers
def toggle_stream(self):
    if self.toggle_btn.isChecked():
        # START branch
        if not self._validate_all_inputs():
            print("[BLOCK] Cannot start — fix the highlighted fields.")
            self.toggle_btn.setChecked(False)
```

```
    return

if self.mode == "wide spectrum" and not self.validate_sweep_inputs():
    print("[BLOCK] Cannot start sweep due to invalid input.")
    self.toggle_btn.setChecked(False)
    return

if self.mode == "frequency":
    if not self.validate_port_ranges():
        print("[ABORT] Invalid port ranges — blocking Start.")
        self.toggle_btn.setChecked(False)
        return
    try:
        center_freq_mhz = parse_freq_input(self.freq_edit.text()) / 1e6
    except ValueError:
        print("[ABORT] Central frequency text is invalid.")
        self.toggle_btn.setChecked(False)
        return
    matched = False
    for line in self.port_inputs.values():
        text = line.text().strip()
        if ":" in text:
            try:
                low, high = map(float, text.replace(" ", "").split(":"))
                if low <= center_freq_mhz <= high:
                    matched = True
                    break
            except:
                continue
    if not matched:
        print(f"[ABORT] Central frequency {center_freq_mhz:.2f} MHz does not match any port range.")
        self.toggle_btn.setChecked(False)
        return

if self.mode == "time":

    if hasattr(self, "timer"):
        self.timer.stop()
    new_cfg = []
    for port_input, dur_input in self.time_inputs:
        try:
            port = port_input.currentText()
            dur = float(dur_input.text().strip())
            if port and dur > 0:
                new_cfg.append((port, dur))
        except:
            print("[WARN] Invalid entry skipped")
    if not new_cfg:
        self.show_temporary_label("time_error_label",
                                  "No valid port/duration entries. Time switching aborted.")
        self.toggle_btn.setChecked(False)
        self.toggle_btn.setText("Start")
        return

    old_ports = [p for p, _ in getattr(self, "time_config", [])]
    new_ports = [p for p, _ in new_cfg]
    if old_ports != new_ports:

        self.per_port_elapsed.clear()
        self.last_time_index = 0

    self.time_config = new_cfg

if self.mode != "wide spectrum" and hasattr(self, "freq_edit"):
    self.update_center_freq()

try:
    self.stop()
    self.wait()
except Exception:
    pass

try:
    self.start()
except RuntimeError as e:
    print(f"[ERROR] Failed to start flowgraph: {e}")
    self.toggle_btn.setChecked(False)
    self.toggle_btn.setText("Start")
    return

self.toggle_btn.setText("Stop")

# Enable interactions everywhere while running
self._set_interactions(True)

# Peak buttons are usable while running
if hasattr(self, "btn_peak_hold"): self.btn_peak_hold.setEnabled(True)

# Make trace visibility follow current toggle state (don’t reset envelopes)
```

```
        if hasattr(self, "pg_peak"): self.pg_peak.setVisible(getattr(self, "btn_peak_hold", None) and self.btn_peak_hold.isChecked())

        if self.mode in ("wide spectrum", "wide spectrum frequency"):
            self.sweep_timer.stop()
            self.sweep_active = True
            self.sweep_ptr = 0
            QTimer.singleShot(100, self.begin_sweep_timer)

        if self.mode == "jamming":
            self.jam_baseline_db = None
            self.jam_started = None
            self.jam_cooldown_until = 0.0
            self.jam_ready_at = time.time() + 1.5
            self.jam_last_switch = time.time()
            self.jam_last_score_at_switch = 0.0
            self._score_ema = 0.0
            if hasattr(self, "jam_timer"):
                self.jam_timer.stop()
                self.jam_timer.start(100)

        if self.mode == "time":
            num = len(self.time_config)
            # Find current or next port with remaining time
            for _ in range(num):
                idx = self.last_time_index
                port, dur = self.time_config[idx]
                elapsed = self.per_port_elapsed.get(idx, 0.0)
                remaining = max(0.0, dur - elapsed)
                if remaining > 0.0:
                    break
                # finished - reset and advance
                self.per_port_elapsed[idx] = 0.0
                self.last_time_index = (self.last_time_index + 1) % num
            else:
                # all ports finished - start fresh cycle from current index
                self.per_port_elapsed = {i: 0.0 for i in range(num)}
                idx = self.last_time_index
                port, dur = self.time_config[idx]
                remaining = dur

            self.timer_start_time = time.time()
            print(f"[TIME] Starting at {port} for {remaining:.2f}s (resumed)")
            self.update_antenna(port)
            self.timer.setSingleShot(True)
            self.timer.start(int(remaining * 1000))

    else:
        # STOP branch
        if self.mode == "time" and self.timer_start_time is not None:
            elapsed = time.time() - self.timer_start_time
            self.per_port_elapsed[self.last_time_index] = self.per_port_elapsed.get(self.last_time_index, 0) + elapsed
            print(f"[PAUSE] Elapsed on port {self.last_time_index}: {elapsed:.2f}s")

        try:
            self.stop()
            self.wait()
        except Exception as e:
            print(f"[WARN] Error stopping flowgraph: {e}")

        self.toggle_btn.setText("Start")

        # Disable SDR-driven interactions (QtGUI sinks), BUT allow zoom/pan on the sweep plot
        self._set_interactions(False)
        if hasattr(self, "pg_plot") and self.pg_plot is not None:
            try:
                vb = self.pg_plot.getViewBox()
                vb.setMouseEnabled(True, True)  # let user zoom/pan on the frozen plot
            except Exception:
                pass

        # Peak buttons enabled
        if hasattr(self, "btn_peak_hold"): self.btn_peak_hold.setEnabled(True)

        if self.mode == "time":
            self.timer.stop()
            self.timer_start_time = None

        if self.mode in ("wide spectrum", "wide spectrum frequency"):
            self.sweep_timer.stop()
            self.sweep_active = False

        if self.mode == "jamming" and hasattr(self, "jam_timer"):
            self.jam_timer.stop()

def perform_time_switch(self):
    if not self.time_config:
        print("[WARN] perform_time_switch(): Empty config, stopping timer and flowgraph.")
        self.timer.stop()
        try:
```

```
            self.stop(); self.wait()
        except Exception as e:
            print(f"[WARN] Failed to stop flowgraph: {e}")
        if hasattr(self, "toggle_btn"):
            self.toggle_btn.setChecked(False)
            self.toggle_btn.setText("Start")
        return

    now = time.time()

    if self.timer_start_time is not None:
        elapsed_seg = now - self.timer_start_time
        self.per_port_elapsed[self.last_time_index] = (
            self.per_port_elapsed.get(self.last_time_index, 0.0) + elapsed_seg
        )

        port, dur = self.time_config[self.last_time_index]
        if self.per_port_elapsed.get(self.last_time_index, 0.0) >= (dur - 1e-3):
            self.per_port_elapsed[self.last_time_index] = 0.0

    num = len(self.time_config)
    self.last_time_index = (self.last_time_index + 1) % num

    for _ in range(num):
        idx = self.last_time_index
        port, dur = self.time_config[idx]
        elapsed = self.per_port_elapsed.get(idx, 0.0)
        remaining = max(0.0, dur - elapsed)
        if remaining > 0.0:
            break
        self.per_port_elapsed[idx] = 0.0
        self.last_time_index = (self.last_time_index + 1) % num
    else:
        self.per_port_elapsed = {i: 0.0 for i in range(num)}
        idx = self.last_time_index
        port, dur = self.time_config[idx]
        remaining = dur

    self.timer_start_time = time.time()
    print(f"[TIME] Switching to {port} for {remaining:.3f}s")
    self.update_antenna(port)
    self.timer.setSingleShot(True)
    self.timer.start(int(remaining * 1000))

#  Sweep logic
def begin_sweep_timer(self):
    if self.mode not in ("wide spectrum", "wide spectrum frequency"):
        return
    try:
        self.sweep_timer.timeout.disconnect(self.update_sweep_freq)
    except Exception:
        pass
    self._sweep_settle_ms = 85
    self.sweep_timer.timeout.connect(self.update_sweep_freq)
    self.sweep_timer.start(30)

def _update_sweep_xrange(self):
    if not hasattr(self, "pg_plot") or self.pg_plot is None:
        return

    s = getattr(self, "last_valid_sweep_start", 80e6)
    e = getattr(self, "last_valid_sweep_end", 100e6)

    # ~0.2% of span, min 100 kHz padding on each side
    span = max(1.0, e - s)
    eps  = max(1e5, 0.005 * span)
    self._sweep_eps = eps

    lo = (s - eps) / 1e6
    hi = (e + eps) / 1e6

    self.pg_plot.setXRange(lo, hi, padding=0)

    vb = self.pg_plot.getViewBox()
    vb.setLimits(
        xMin=lo, xMax=hi,
        minXRange=(e - s) * 0.05 / 1e6,      # don't zoom in past 5% of span
        maxXRange=(e - s + 2*eps) / 1e6      # don't zoom out past padded span
    )

def update_sweep_freq(self):
    if self.mode not in ("wide spectrum", "wide spectrum frequency") or not self.sweep_active or self.sweep_busy or getattr(self, "_rebuilding", False):
        return
    cf_list = getattr(self, "center_freqs", None)
    if not cf_list:
        return

    self.sweep_busy = True
    try:
```

```
        if self.sweep_ptr >= len(cf_list):
            self.sweep_ptr = 0

        cf_hz = float(cf_list[self.sweep_ptr])
        self._step_freq_hz = cf_hz
        self._cap_retries = 0

        try:
            tuned = float(self.src.get_center_freq(0))
        except Exception:
            tuned = None
        if tuned is None or abs(tuned - cf_hz) > 1.0:
            try:
                self.src.set_center_freq(cf_hz)
            except Exception as e:
                print(f"[WARN] set_center_freq({cf_hz/1e6:.3f} MHz) failed: {e}")
                QTimer.singleShot(int(getattr(self, "_sweep_settle_ms", 85)), self.capture_fft)
                return

        idx   = self.sweep_ptr + 1
        total = len(cf_list)
        if self.mode == "wide spectrum frequency":
            port = getattr(self, "selected_port", self.antenna_select)
            print(f"[STEP] ({port}) {idx}/{total} @ {cf_hz/1e6:.3f} MHz")
        else:
            print(f"[STEP] {idx}/{total} @ {cf_hz/1e6:.3f} MHz")

        QTimer.singleShot(int(getattr(self, "_sweep_settle_ms", 85)), self.capture_fft)
    finally:
        pass

def capture_fft(self):
    # Guards
    if self.mode not in ("wide spectrum", "wide spectrum frequency") or not self.sweep_active or getattr(self, "_rebuilding", False):
        self.sweep_busy = False
        return
    if getattr(self, "probe", None) is None or not hasattr(self, "center_freqs") or not self.center_freqs:
        self.sweep_busy = False
        return

    try:
        # Make sure update_sweep_freq() set the step
        step = getattr(self, "_step_freq_hz", None)
        if step is None:
            self._cap_retries = 0
            self.sweep_busy = False
            return

        # Verify tuner already at that step (allow a few retries)
        try:
            tuned = float(self.src.get_center_freq(0))
        except Exception:
            tuned = None
        if (tuned is not None) and (abs(tuned - float(step)) > 1.0):
            if self._cap_retries < 3:
                self._cap_retries += 1
                Qt.QTimer.singleShot(8, self.capture_fft)
                return
            self.sweep_ptr = (self.sweep_ptr + 1) % len(self.center_freqs)
            self._cap_retries = 0
            return

        # Read FFT magnitudes from the probe
        try:
            raw = self.probe.level()
        except Exception:
            raw = None
        mags = np.asarray(raw, dtype=np.float32) if raw is not None else None
        if mags is None or mags.ndim != 1 or mags.size != self.fft_size or not np.isfinite(mags).all():
            if self._cap_retries < 3:
                self._cap_retries += 1
                Qt.QTimer.singleShot(8, self.capture_fft)
                return
            self.sweep_ptr = (self.sweep_ptr + 1) % len(self.center_freqs)
            self._cap_retries = 0
            return

        # Re-check tuner AFTER capture; discard if LO moved late
        try:
            tuned_after = float(self.src.get_center_freq(0))
        except Exception:
            tuned_after = None
        if (tuned_after is None) or (abs(tuned_after - float(step)) > 1.0):
            if self._cap_retries < 3:
                self._cap_retries += 1
                Qt.QTimer.singleShot(8, self.capture_fft)
                return
            self.sweep_ptr = (self.sweep_ptr + 1) % len(self.center_freqs)
            self._cap_retries = 0
```

```
    return

# Stale-vector guard: identical to previous capture → retry
if hasattr(self, "last_fft") and (self.last_fft is not None):
    if np.allclose(mags, self.last_fft, rtol=0.0, atol=1e-8):
        if self._cap_retries < 3:
            self._cap_retries += 1
            Qt.QTimer.singleShot(8, self.capture_fft)
            return
        # If it keeps repeating, skip this step once
        self.sweep_ptr = (self.sweep_ptr + 1) % len(self.center_freqs)
        self._cap_retries = 0
        return
self.last_fft = mags.copy()

# Scale like QtGUI sink
mags *= (1.0 / float(self.fft_size))

# DC notch (center ±2 bins, filled with neighbor mean)
c = self.fft_size // 2
k = 2
if self.fft_size >= 16:
    loL, loR = max(0, c - (k + 3)), max(0, c - (k + 1))
    hiL, hiR = min(self.fft_size, c + (k + 1)), min(self.fft_size, c + (k + 3))
    neigh = []
    if loL < loR: neigh.extend(mags[loL:loR])
    if hiL < hiR: neigh.extend(mags[hiL:hiR])
    if len(neigh) >= 2:
        mags[max(0, c - k):min(self.fft_size, c + k + 1)] = float(np.mean(neigh))

# Ensure sweep buffer
expected = len(self.center_freqs) * self.fft_size
if self.sweep_buffer.size != expected:
    self.sweep_buffer = np.zeros(expected, dtype=np.float32)

# Write this step's segment
s = self.sweep_ptr * self.fft_size
e = s + self.fft_size
self.sweep_buffer[s:e] = mags

# Build the real X/Y that lie inside [start, end] (no linspace)
start = self.last_valid_sweep_start
end   = self.last_valid_sweep_end
mask  = (self.freq_axis >= start) & (self.freq_axis <= end)
idx   = np.flatnonzero(mask)
if idx.size == 0:
    self.sweep_ptr = (self.sweep_ptr + 1) % len(self.center_freqs)
    self._cap_retries = 0
    self.sweep_busy = False
    return

# Downsample to at most 1024 points for plotting
max_bins = 1024
stride = int(np.ceil(idx.size / max_bins))
sel = idx[::stride]

y_lin = self.sweep_buffer[sel]
x_hz  = self.freq_axis[sel]

# Averaging
a = float(np.clip(getattr(self, "sweep_avg_alpha", 0.20), 0.0, 1.0))
prev = getattr(self, "sweep_prev_linear", None)
if (prev is None) or (getattr(prev, "size", 0) != y_lin.size):
    prev = y_lin.copy()
sm_lin = (1.0 - a) * prev + a * y_lin
self.sweep_prev_linear = sm_lin

# To dB
db = 20.0 * np.log10(np.clip(sm_lin, 1e-12, 1e9)) + float(getattr(self, "sweep_db_cal", 0.0))
db = np.maximum(db, -140.0)

# Plot at true frequencies
self.pg_curve.setData(x_hz / 1e6, db)

# Peak-hold overlay (same X)
if getattr(self, "peak_enabled", False):
    if not hasattr(self, "peak_env") or self.peak_env is None or self.peak_env.size != db.size:
        self.peak_env = db.copy()
    else:
        self.peak_env = np.maximum(self.peak_env, db)
    self.pg_peak.setData(x_hz / 1e6, self.peak_env)
    self.pg_peak.setVisible(True)
else:
    self.pg_peak.setVisible(False)

self._update_sweep_xrange()

# Advance to next center
self.sweep_ptr = (self.sweep_ptr + 1) % len(self.center_freqs)
```

```
            self._cap_retries = 0

        except Exception as e:
            print(f"[ERROR] capture_fft failed: {e}")
        finally:
            self.sweep_busy = False

    def validate_sweep_inputs(self, *, quiet=False):
        try:
            sweep_start = parse_freq_input(self.sweep_start_edit.text())
            sweep_end   = parse_freq_input(self.sweep_end_edit.text())

            # Clamp hard limits (1 Hz .. 4 GHz) with inline hints
            if sweep_start <= 0:
                sweep_start = 1.0
                self.sweep_start_edit.setText("1")
                self.show_temporary_label("sweep_error_label", "Sweep Start set to 1 Hz")
            elif sweep_start > 4e9:
                sweep_start = 4e9
                self.sweep_start_edit.setText("4e9")
                self.show_temporary_label("sweep_error_label", "Sweep Start set to 4 GHz")

            if sweep_end <= 0:
                sweep_end = 1.0
                self.sweep_end_edit.setText("1")
                self.show_temporary_label("sweep_error_label", "Sweep End set to 1 Hz")
            elif sweep_end > 4e9:
                sweep_end = 4e9
                self.sweep_end_edit.setText("4e9")
                self.show_temporary_label("sweep_error_label", "Sweep End set to 4 GHz")

            if sweep_end <= sweep_start:
                msg = (f"Sweep End ({sweep_end/1e6:.2f} MHz) must be greater than "
                       f"Sweep Start ({sweep_start/1e6:.2f} MHz)")
                self.show_temporary_label("sweep_error_label", msg)

                # Safe fallback so downstream plot code doesn't crash
                self.total_bins = self.fft_size
                self.freq_axis = np.linspace(0, self.samp_rate, self.fft_size)
                self.sweep_buffer = np.zeros(self.total_bins, dtype=np.float32)

                # If we were running, stop the sweep safely
                if hasattr(self, "toggle_btn") and self.toggle_btn.isChecked():
                    try:
                        self.stop(); self.wait()
                        if hasattr(self, "sweep_timer"):
                            self.sweep_timer.stop()
                    except Exception as e:
                        print(f"[WARN] Error stopping flowgraph: {e}")
                    self.toggle_btn.setChecked(False)
                    self.toggle_btn.setText("Start")
                return False

            # Save last valid range for the plot's x-range helper
            self.last_valid_sweep_start = sweep_start
            self.last_valid_sweep_end   = sweep_end

            # Plan steps: one FFT window per sample-rate chunk
            step = float(self.samp_rate)
            if step <= 0.0:
                raise ValueError("Sample rate must be > 0 for sweep planning.")
            ratio = (sweep_end - sweep_start) / step
            num_steps = int(np.ceil(ratio))
            if num_steps < 1:
                raise ValueError("Sweep range too narrow for current sample rate.")

            # Round the ends a bit to tame text edits that don't change the plan materially.
            sig = (round(sweep_start, 1), round(sweep_end, 1), int(round(step)), int(self.fft_size), int(num_steps))
            same_as_last = (getattr(self, "_last_sweep_sig", None) == sig)
            self._last_sweep_sig = sig

            # Build the center frequencies for each step
            self.center_freqs = [sweep_start + (i + 0.5) * step for i in range(num_steps)]
            self.total_steps  = len(self.center_freqs)
            self.total_bins   = self.total_steps * self.fft_size

            # Frequency axis for the stitched spectrum (use baseband bins around each CF)
            bin_freqs = np.linspace(-step / 2.0, step / 2.0, self.fft_size, endpoint=False)
            self.freq_axis = np.concatenate([bin_freqs + cf for cf in self.center_freqs])

            # (Re)allocate sweep buffer + reset state tied to size/plan
            if self.sweep_buffer.size != self.total_bins:
                self.sweep_buffer = np.zeros(self.total_bins, dtype=np.float32)
            else:
                self.sweep_buffer.fill(0.0)

            self.sweep_prev_linear = None
            self.sweep_ptr = 0
            self.peak_env = None
```

```
        if hasattr(self, "_peak_scatter"):
            try: self._peak_scatter.clear()
            except Exception: pass

        # Update X limits of the pyqtgraph plot to the new span
        self._update_sweep_xrange()

        if not quiet and not same_as_last:
            print(f"[SWEEP] Configured: {self.total_steps} steps × {self.fft_size} bins = {self.total_bins} total bins")

        return True

    except Exception as e:
        # Parsing/error fallbacks
        print(f"[ERROR] Failed to parse sweep inputs: {e}")
        self.show_temporary_label("sweep_error_label", "[ERROR] Invalid sweep inputs")

        self.total_bins = self.fft_size
        self.freq_axis = np.linspace(0, self.samp_rate, self.fft_size)
        self.sweep_buffer = np.zeros(self.total_bins, dtype=np.float32)

        if hasattr(self, "toggle_btn") and self.toggle_btn.isChecked():
            try:
                self.stop(); self.wait()
            except Exception as ee:
                print(f"[WARN] Error stopping flowgraph: {ee}")
            self.toggle_btn.setChecked(False)
            self.toggle_btn.setText("Start")
        return False

#  Jamming detection
def check_jamming(self):
    if getattr(self, "_rebuilding", False):
        return
    if self.mode != "jamming" or not self.toggle_btn.isChecked():
        return
    try:
        try:
            raw = self.probe.level()
        except Exception:
            return

        mags = np.asarray(raw, dtype=np.float32) if raw is not None else None
        if mags is None or mags.size != self.fft_size or not np.isfinite(mags).all():
            return

        mags *= (1.0 / float(self.fft_size))
        power_db = 20.0 * np.log10(np.clip(mags, 1e-12, 1e9))

        now = time.time()
        if hasattr(self, "jam_ignore_until") and now < self.jam_ignore_until:
            return

        # baseline update with FREEZE window
        freeze_until = getattr(self, "_baseline_freeze_until", 0.0)
        if self.jam_baseline_db is None:
            self.jam_baseline_db = power_db.copy()
            return
        if now >= freeze_until:
            self.jam_baseline_db = (1.0 - self.jam_alpha) * self.jam_baseline_db + self.jam_alpha * power_db
        if now < self.jam_ready_at:
            return

        m = self.jam_mask
        if m is None or m.size == 0 or not np.any(m):
            return

        masked_curr = power_db[m]
        masked_base = self.jam_baseline_db[m]
        Nmask = int(masked_curr.size)

        base_med = float(np.median(masked_base))
        curr_med = float(np.median(masked_curr))
        med_lift = curr_med - base_med

        # protect against huge AGC/step changes
        if med_lift > 8.0:
            self.jam_baseline_db = power_db.copy()
            self.jam_ignore_until = now + 0.8
            return

        # spectral flatness
        p_lin = 10.0 ** (masked_curr / 10.0)
        sfm_db = spectral_flatness_db(p_lin)
        if self.sfm_ref is None:
            self.sfm_ref = sfm_db
        else:
            self.sfm_ref = 0.95 * self.sfm_ref + 0.05 * sfm_db
```

```
sfm_gate = max(self.jam_sfm_min_db, self.sfm_ref + getattr(self, "_sfm_offset", 0.8))

# occupancy vs baseline
over = (masked_curr - masked_base) >= self.jam_thresh_db
occ  = float(np.sum(over)) / max(1, Nmask)

# scale occ requirement + mild SFM relief
scale   = (256.0 / max(64.0, float(Nmask))) ** 0.5
occ_req = max(0.10, min(0.45, self.jam_occ_min * scale))
if   sfm_db >= -3.5: occ_req = max(0.10, 0.85 * occ_req)
elif sfm_db >= -4.0: occ_req = max(0.12, 0.92 * occ_req)

# smooth the printed/used occ requirement so it doesn't flicker
if not hasattr(self, "_occ_req_ema") or self._occ_req_ema is None:
    self._occ_req_ema = occ_req
else:
    self._occ_req_ema = 0.6 * self._occ_req_ema + 0.4 * occ_req
occ_req_eff = float(np.clip(self._occ_req_ema, 0.10, 0.45))

# band/mean power lift
eps = 1e-12
band_lift = 10.0 * np.log10(
    (np.mean(10.0**(masked_curr/10.0)) + eps) /
    (np.mean(10.0**(masked_base/10.0)) + eps)
)

# longest contiguous run
over_i = over.astype(np.int8)
edges  = np.diff(np.pad(over_i, (1, 1)))
starts = np.where(edges == +1)[0]
ends   = np.where(edges == -1)[0]
run_lengths = (ends - starts) if (starts.size and ends.size) else np.array([], dtype=int)
max_run     = int(run_lengths.max()) if run_lengths.size else 0
span_frac   = max_run / float(Nmask)

# ANTI-FLAP: hysteresis + min bin count + baseline freeze
band_on = getattr(self, "jam_band_db", 1.5)
med_on  = getattr(self, "jam_median_db", 2.0)
band_off = max(0.5, band_on - 0.8)
med_off  = max(0.5, med_on  - 0.8)

band_state = getattr(self, "_jam_band_state", False)
med_state  = getattr(self, "_jam_med_state",  False)
band_ok = (band_lift >= (band_on if not band_state else band_off))
med_ok  = (med_lift  >= (med_on  if not med_state  else med_off))
self._jam_band_state = band_ok
self._jam_med_state  = med_ok

# minimum absolute number of "over" bins
min_bins = max(6, int(0.006 * Nmask))
enough_bins = (int(np.sum(over)) >= min_bins)

# pre-condition to freeze baseline briefly (prevents EMA chasing)
pre = (occ >= 0.8 * occ_req_eff) and (band_lift >= 0.8 * band_on or med_lift >= 0.8 * med_on or span_frac >= 0.8 * self.jam_span_min)
if pre and now >= freeze_until:
    self._baseline_freeze_until = now + 0.6  # ~6 ticks if jam_timer=100 ms

# decision with anti-flap guards
cond_wideband  = enough_bins and (occ >= occ_req_eff) and ((sfm_db >= sfm_gate) or band_ok)
cond_multipeak = enough_bins and (occ >= self.jam_occ_multipeak) and (span_frac >= self.jam_span_min)
cond_median    = med_ok
cond = cond_wideband or cond_multipeak or cond_median

# optional score + stickiness to avoid ping-pong
score = 0.6 * (occ / max(occ_req_eff, 1e-6)) + 0.25 * max(0.0, band_lift / max(band_on, 1e-6)) + 0.15 * max(0.0, med_lift / max(med_on, 1e-6))
if not hasattr(self, "_score_ema"):
    self._score_ema = score
else:
    self._score_ema = 0.8 * self._score_ema + 0.2 * score

age   = now - getattr(self, "jam_last_switch", 0.0)
decay = 0.5 ** max(0.0, age / 8.0)
last  = getattr(self, "_last_score_at_switch", 0.0) * decay

need  = max(1.10, last + 0.10)
improves = (self._score_ema >= need)

if getattr(self, "jam_debug", False) and (now - getattr(self, "_jam_last_dbg", 0.0)) > 0.5:
    print(f"[JDBG] ready={now >= self.jam_ready_at} occ={occ:.2f}/{occ_req_eff:.2f} "
        f"sfm={sfm_db:.1f}/{sfm_gate:.1f}dB span={span_frac:.3f} "
        f"band+{band_lift:.1f}dB med+{med_lift:.1f}dB bins={np.sum(over)}/{Nmask} "
        f"bandHys={band_ok} medHys={med_ok}")
    self._jam_last_dbg = now

# hold / cooldown / dwell gating + switch (with 'improves' stickiness)
if cond and improves:
    if self.jam_started is None:
        self.jam_started = now
```

```
            held       = (now - self.jam_started) >= self.jam_hold_s
            dwell_ok   = (now - getattr(self, "jam_last_switch", 0.0)) >= 0.6
            cooldown_ok = now >= self.jam_cooldown_until

            if held and dwell_ok and cooldown_ok:
                # visual alert
                self.jam_alert_label.show()
                QTimer.singleShot(3000, self.jam_alert_label.hide)

                # rotate port
                self.current_port_index = (self.current_port_index + 1) % len(self.ports)
                new_port = self.ports[self.current_port_index]
                if self.antenna_edit is not None:
                    self.antenna_edit.setText(new_port)
                self.update_antenna()

                # reset / re-arm
                self.jam_last_switch     = now
                self.jam_cooldown_until  = now + self.jam_cooldown_s
                self.jam_started         = None
                self.jam_ready_at        = now + 0.8
                self._baseline_freeze_until = now + 0.6  # keep baseline steady right after switch
                self.jam_baseline_db     = power_db.copy()
                self.sfm_ref             = sfm_db
                self._last_score_at_switch = self._score_ema
                self._jam_last_dbg       = now
            else:
                self.jam_started = None

        except Exception as e:
            print(f"[ERROR] check_jamming: {e}")

    #  UI helpers / misc
    def _hide_label(self, label_attr: str):
        label = getattr(self, label_attr, None)
        if isinstance(label, Qt.QLabel):
            label.hide()
            label.setText("")
            if hasattr(self, "_label_timers") and label in self._label_timers:
                try:
                    self._label_timers[label].stop()
                    del self._label_timers[label]
                except Exception:
                    pass

    def show_temporary_label(self, label_attr, message=""):
        if not hasattr(self, "_label_timers"):
            self._label_timers = {}

        label = getattr(self, label_attr, None)
        if not isinstance(label, Qt.QLabel):
            return

        if message:
            label.setText(message)
        label.show()

        if label in self._label_timers:
            self._label_timers[label].stop()

        timer = QTimer(self)
        timer.setSingleShot(True)
        timer.timeout.connect(label.hide)
        timer.start(5000)
        self._label_timers[label] = timer

    def _reinit_jamming_dsp_chain(self):
        # build fresh blocks
        self.stream_to_vector = blocks.stream_to_vector(gr.sizeof_gr_complex, self.fft_size)
        self.fft = fft_vcc(self.fft_size, True, window.blackmanharris(self.fft_size), True)
        self.c2mag = blocks.complex_to_mag(self.fft_size)
        self.probe = blocks.probe_signal_vf(self.fft_size)

        # wire them up
        self.connect((self.src, 0), (self.stream_to_vector, 0))
        self.connect((self.stream_to_vector, 0), (self.fft, 0))
        self.connect((self.fft, 0), (self.c2mag, 0))
        self.connect((self.c2mag, 0), (self.probe, 0))

        # refresh mask + detection state
        self._rebuild_jam_mask()
        self.jam_baseline_db = None
        self.jam_started = None
        self.jam_cooldown_until = 0.0
        self.jam_ready_at = time.time() + 1.5
        self._jam_last_dbg = 0.0

    def refresh_time_config(self):
        self.time_config = []
```

```
    for port_input, dur_input in self.time_inputs:
        try:
            port = port_input.currentText()
            dur = float(dur_input.text().strip())
            if port and dur > 0:
                self.time_config.append((port, dur))
        except:
            print("[WARN] Invalid entry skipped")

    if not self.time_config:
        print("[ERROR] No valid port/duration entries. Time switching halted.")
        self.timer.stop()
        try:
            self.stop()
            self.wait()
        except Exception as e:
            print(f"[WARN] Flowgraph stop failed: {e}")
        if hasattr(self, "toggle_btn"):
            self.toggle_btn.setChecked(False)
            self.toggle_btn.setText("Start")
        return False

    if self.last_time_index >= len(self.time_config):
        self.last_time_index = 0
    return True

def refresh_freq_config(self):
    valid = True
    error_text = ""
    try:
        ranges = []
        for port, line in self.port_inputs.items():
            text = line.text().strip()
            parts = text.replace(" ", "").split(":")
            if len(parts) != 2:
                raise ValueError(f"Invalid format in {port}: '{text}'")
            low = float(parts[0])
            high = float(parts[1])
            if low >= high:
                raise ValueError(f"Invalid range in {port}: '{text}' — Lower must be < Upper")
            if high > 4000:
                raise ValueError(f"Invalid range in {port}: '{text}' — Upper limit exceeds 4000 MHz")
            ranges.append((low, high, port))

        ranges.sort()
        for i in range(len(ranges) - 1):
            if ranges[i][1] >= ranges[i+1][0]:
                raise ValueError(f"Overlapping or touching ranges: {ranges[i][2]} ends at {ranges[i][1]}, "
                                 f"{ranges[i+1][2]} starts at {ranges[i+1][0]}")
    except ValueError as e:
        error_text = str(e)
        valid = False

    if not valid:
        print(error_text)
        self.show_temporary_label("port_error_label", error_text)
        if self.toggle_btn.isChecked():
            print("[STOP] Flowgraph halted due to invalid frequency range.")
            try:
                self.stop()
                self.wait()
            except Exception as e:
                print(f"[WARN] Error stopping flowgraph: {e}")
            self.toggle_btn.setChecked(False)
            self.toggle_btn.setText("Start")
    else:
        self.port_error_label.hide()

def _rebuild_jam_mask(self):
    self.jam_center_k = max(2, self.fft_size // 256)   # scales with N
    c = self.fft_size // 2
    m = np.ones(self.fft_size, dtype=bool)
    m[max(0, c - self.jam_center_k): min(self.fft_size, c + self.jam_center_k + 1)] = False
    m[:2] = False
    m[-2:] = False
    self.jam_mask = m
    if getattr(self, "jam_debug", False):
        print(f"[JDBG] jam mask: center ±{self.jam_center_k} bins, edges 2 bins")

def _stop_stream_safely(self):
    if hasattr(self, "toggle_btn") and self.toggle_btn.isChecked():
        try:
            self.stop()
            self.wait()
        except Exception as e:
            print(f"[WARN] Error stopping flowgraph: {e}")
        # stop any timers tied to modes
        if self.mode == "time"   and hasattr(self, "timer"):        self.timer.stop()
        if self.mode == "wide spectrum"  and hasattr(self, "sweep_timer"):  self.sweep_timer.stop(); self.sweep_active = False
```

```
        if self.mode == "jamming" and hasattr(self, "jam_timer"):   self.jam_timer.stop()
        self.toggle_btn.setChecked(False)
        self.toggle_btn.setText("Start")

    def _input_error(self, label_attr: str, message: str, field: Qt.QLineEdit = None):
        print(f"[ERROR] {message}")
        self.show_temporary_label(label_attr, message)
        if field is not None:
            field.setStyleSheet("border: 2px solid #d00;")
            QTimer.singleShot(1800, lambda: field.setStyleSheet(""))
        self._stop_stream_safely()

    def _select_port_and_sweep(self, port: str, *, start_immediately: bool = True, quiet: bool = True):
        try:
            sel = str(port).upper()
            if sel not in self.port_inputs:
                raise KeyError(f"Unknown port '{sel}'")

            txt = self.port_inputs[sel].text().strip().replace(" ", "")
            if ":" not in txt:
                raise ValueError("expected format 'low:high' in MHz")
            lo_s, hi_s = txt.split(":", 1)
            lo_mhz = float(lo_s)
            hi_mhz = float(hi_s)
            if not np.isfinite(lo_mhz) or not np.isfinite(hi_mhz):
                raise ValueError("bounds must be finite numbers")
            if hi_mhz <= lo_mhz:
                raise ValueError("upper bound must be > lower bound")

            # Remember selection
            self.selected_port = sel

            # Update UI footer (OperaCake panel), regardless of SDR build state
            if hasattr(self, "oc_panel"):
                try:
                    self.oc_panel.set_active(sel, fixed_input="A0", connected=self.is_hackrf_connected())
                except Exception:
                    pass

            # Switch hardware path ONLY if SDR already exists; stay silent otherwise
            if hasattr(self, "src") and self.src is not None:
                try:
                    self.antenna_select = sel
                    self.src.set_antenna(sel, 0)
                    self.antenna_changed.emit(sel)
                except Exception:
                    pass
            else:
                self.antenna_select = sel

            # Feed the hidden sweep inputs so validate_sweep_inputs() works unchanged
            self.sweep_start_edit.setText(f"{lo_mhz}e6")
            self.sweep_end_edit.setText(f"{hi_mhz}e6")

            # Recompute sweep plan (quiet to avoid duplicate
            ok = self.validate_sweep_inputs(quiet=bool(quiet))
            if not ok:
                return

            # If already running, (re)start the sweep timer after a small settle
            if start_immediately and hasattr(self, "toggle_btn") and self.toggle_btn.isChecked():
                try:
                    self.sweep_timer.stop()
                except Exception:
                    pass
                self.sweep_ptr = 0
                self.sweep_active = True
                # Let RF switch + tuner settle a bit
                QTimer.singleShot(int(getattr(self, "_sweep_settle_ms",85)), self.begin_sweep_timer)

        except Exception as e:
            self.show_temporary_label(
                "sweep_error_label",
                f"Invalid range for {port}: '{self.port_inputs.get(port, Qt.QLineEdit()).text()}' — {e}"
            )
            return

    def _validate_all_inputs(self):
        bad = False

        if hasattr(self, "freq_edit"):
            try:
                _ = parse_freq_input(self.freq_edit.text())
                self.freq_edit.setStyleSheet("")
            except ValueError as e:
                bad = True
                self.freq_edit.setStyleSheet("border: 2px solid #d00;")
                print(f"[ERROR] {e}")
```

```
        try:
            _ = parse_freq_input(self.samp_rate_edit.text())
            self.samp_rate_edit.setStyleSheet("")
        except ValueError as e:
            bad = True
            self.samp_rate_edit.setStyleSheet("border: 2px solid #d00;")
            print(f"[ERROR] {e}")

        for edit in [self.rf_gain_edit, self.if_gain_edit, self.bb_gain_edit]:
            if edit.text() == "" or not edit.hasAcceptableInput():
                bad = True
                edit.setStyleSheet("border: 2px solid #d00;")

        if self.mode == "frequency" and not self.validate_port_ranges():
            bad = True

        if self.mode in ("wide spectrum","wide spectrum frequency") and not self.validate_sweep_inputs():
            bad = True

        return not bad

    def is_hackrf_connected(self) -> bool:
        try:
            r = subprocess.run(["hackrf_info"], stdout=subprocess.PIPE, stderr=subprocess.PIPE)
            return b"Found HackRF" in r.stdout
        except Exception:
            return False

    def _ensure_mouse_blocker(self):
        # Create once
        if not hasattr(self, "_mouse_blocker"):
            self._interactions_enabled = False
            self._mouse_blocker = _MouseBlocker(self)
        # (Re)install on currently built widgets
        for wname in ("_freq_win", "_waterfall_win"):
            w = getattr(self, wname, None)
            if isinstance(w, Qt.QWidget):
                w.installEventFilter(self._mouse_blocker)
        if hasattr(self, "pg_plot") and self.pg_plot is not None:
            self.pg_plot.installEventFilter(self._mouse_blocker)
            # For PyQtGraph also hard-disable zoom/pan until enabled
            vb = self.pg_plot.getViewBox()
            vb.setMouseEnabled(False, False)
            vb.setMenuEnabled(False)

    def _set_interactions(self, enabled: bool):
        self._interactions_enabled = bool(enabled)
        # Re-toggle pyqtgraph ViewBox mouse
        if hasattr(self, "pg_plot") and self.pg_plot is not None:
            vb = self.pg_plot.getViewBox()
            vb.setMouseEnabled(enabled, enabled)

    def apply_sdr_config(self):
        self.src.set_sample_rate(self.samp_rate)
        self.src.set_center_freq(self.center_freq)
        self.src.set_gain(self.rf_gain, 0)
        self.src.set_if_gain(self.if_gain, 0)
        self.src.set_bb_gain(self.bb_gain, 0)
        self.src.set_antenna(self.antenna_select, 0)
```

10. CV Europass

europass

Fabian Ungurusan

Date of birth: 14/11/2002 | **Place of birth:** Zalau, Romania | **Nationality:** Romanian | **Gender:** Male | **Phone number:** (+40) 0741379606 (Mobile) | **Email address:** fabianadrian82@yahoo.com | **LinkedIn:** https://www.linkedin.com/in/fabian-ungurusan-b12a94250/ | **Address:** Victor Deleu Zalau, 450137, Zalau, Romania (Home)

ABOUT ME

A motivated engineering graduate with a basic understanding of software and hardware concepts. I'm looking forward to gaining hands-on experience, improving my technical skills, and contributing to meaningful tech solutions through teamwork and continuous learning.

WORK EXPERIENCE

TENARIS SILCOTUB – ZALAU, ROMANIA

INTERNSHIP – 01/11/2024 – 31/05/2025

Acquisition of specific GPS hardware for tracking platforms and pallets inside the mill.
Mobile/web application where requests for internal maneuvers can be inserted and followed by our drivers.

ELECTROGRUP – ZALAU, ROMANIA

UNQUALIFIED WORKER(ELECTRICAL DOMAIN) – 10/07/2023 – 01/09/2024

Learned precisely what operation I had to perform in accordance with the tasks outlined by the team leader.

SKILLS

C/C++ | Java | JavaScript | SQL | Python | Power Apps

LANGUAGE SKILLS

Mother tongue(s): **ROMANIAN**

Other language(s):

	UNDERSTANDING		SPEAKING		WRITING
	Listening	Reading	Spoken production	Spoken interaction	
ENGLISH	C1	C1	C1	C1	C1

Levels: A1 and A2: Basic user; B1 and B2: Independent user; C1 and C2: Proficient user

HOBBIES AND INTERESTS

Gym

I like to stay active, going 3-4 times a week

Gaming

Swimming

ROUMANIE

ROMÂNIA

ROMANIA

CARTE D'IDENTITE

CARTE DE IDENTITATE

IDENTITY CARD

CNP 5021114314013

SERIA SX NR 605388

07R9W

Nume/Nom/Last name
UNGURUŞAN

Prenume/Prenom/First name
FABIAN-ADRIAN

Cetățenie/Nationalite/Nationality
Română / ROU

Sex/Sexe/Sex
M

Loc naștere/Lieu de naissance/Place of birth
Jud.SJ Mun.Zalău

Domiciliu/Adresse/Address
Jud.SJ Mun.Zalău
Str.Victor Deleu nr.19 bl.G10 ap.13/5

631 SJ

Emisă de/Delivree par/Issued by
SPCLEP Zalău

Valabilitate/Validite/Validity
07.02.25-03.08.2031

IDROUUNGURUSAN<<FABIAN<ADRIAN<<<<<<<

SX605388<9ROU0211141M3108033531401 31

conform cu originalul