

PhantomSDR-Plus

Complete Documentation

16 documents · generated 2026-10-03

Contents

1. README	8
Key Features	8
Supported Hardware	10
Quick Start	10
Performance Benchmarks	11
Usage	12
Configuration	12
Waterfall Floor — waterfall.sh	13
Customization	14
Documentation	15
Online WebSDR List	16
Troubleshooting	16
Contributing	16
License	17
Authors & Credits	17
Links	17
Support	17
Performance Tips	17
2. Installation Guide	18
Table of Contents	18
System Requirements	18
Pre-Installation Preparation	19
Installing Dependencies	20
Installing Node.js and npm	21
Installing OpenCL (Optional but Recommended)	22
Installing PhantomSDR-Plus	24
Installing Opus Audio Codec	29
Autorun Spot Reporter (FT8/FT4/WSPR)	30
KiwiSDR Client Emulation (optional)	32
Configuration	32
SDR Device-Specific Setup	35
Testing and Verification	38
Setting Up Autostart	39
CPU Thermal Protection	41
Troubleshooting	43
Performance Optimization	47
Updating PhantomSDR-Plus	47
Backup and Restore	50
Security Considerations	51
Getting Help	51
Appendix A: Complete Dependency List	52
Appendix B: Configuration Examples	52
3. Admin Panel Setup	54
Overview	54
What the admin panel gives you	54

Requirements	55
Step 1 — Files	55
Step 2 — Run the setup script	55
Step 3 — Start / Stop / Restart	56
Step 4 — Open firewall port	58
Step 5 — Kick Users feature	58
Configuration — admin_config.json	59
The sdr_host setting	60
proxy.py — what it is and when you need it	60
Changing ports after setup	61
Useful manual commands	61
Log files	62
Access summary	64
Chat message deletion	64
Waterfall broadcast messages	64
System Graphs	65
Thermal Guard	66
Spot Reporting (Autorun FT8/FT4/J58/WSPR)	69
Troubleshooting	74
4. Thermal Guard	77
Contents	77
1. What it does	77
2. Quick start	78
3. How it decides	78
4. The four modes	79
5. Choosing a mode	81
6. Configuration reference	82
7. Running it without the admin panel	82
8. Enabling the throttle stage without root	83
9. Unattended operation	84
10. Testing it before you trust it	85
11. Reading the log	85
12. Lockout and the rate limit	86
13. Troubleshooting	87
14. Turning it off or removing it	87
See also	87
5. Connection Limits	88
Contents	88
1. What you are actually defending against	88
2. The three layers	89
3. Per-IP limits	89
4. Idle connections	90
5. The kernel guard	91
6. Configuration reference	92
7. What a refused listener sees	93
8. Reading the logs and counters	94
9. Choosing your numbers	94
10. Traps worth knowing about	95
11. Turning it all off	95

6. Several Receivers	96
Contents	96
1. How it works	96
2. Two ways to publish the receivers	97
3. Adding a second receiver	97
4. receivers.toml	99
5. Starting and stopping	99
6. What listeners see	100
7. The S-meter above 30 MHz	100
8. Security	101
9. How many receivers fit	101
10. Traps worth knowing about	101
7. Statistics Server	103
Overview	103
Prerequisites	103
Installation Methods	103
Configure Your PhantomSDR Application	106
Testing the Installation	108
Service Management	109
Manual Installation (Option 2)	109
Changing the Port After Installation	111
Troubleshooting	111
What You'll See	114
Temperature Detection	114
Features Summary	115
Security Notes	115
Support & Resources	115
8. Decoders	118
Table of Contents	118
1. How to Start a Decoder	118
2. FT8	121
3. FT4-FT2	122
4. JS8	123
5. CW — Morse Code	124
6. QRSS Grabber	125
7. WSPR	128
8. HF FAX / WEFAX	129
9. NAVTEX	130
10. FSK / RTTY — including PSK31 and Olivia	131
11. SSTV	135
12. General Tips	137
9. RADE README	138
Installing it — the short way	138
Table of Contents	138
How It Works	139
Architecture Overview	140
RADEL vs RADEU	140
Prerequisites	141

Installing by hand	141
The Patched Files	141
Sidecar control	142
Port 8074	143
Using RADE in the Browser	143
Verification and Debugging	144
Environment Variables	145
Files Changed	146
Signal Flow Summary	146
Troubleshooting	147
Measuring Concurrency on Your Own Hardware	149
Updating RADE v1	151
10. RADE General Install (Linux)	154
Step 1 — System packages	154
Step 2 — Python packages	155
Step 3 — Clone and build the radae repository	156
Step 4 — Verify the decode pipeline	156
Step 5 — Build the PhantomSDR-Plus frontend	158
Step 6 — Sidecar control	158
Step 7 — Open port 8074	158
Step 8 — Start the server	159
Step 9 — Verify RADE	159
Step 10 — Using RADE in the browser	160
CPU note	160
Updating RADE in future	161
11. Editing Variants	162
1. What the Variants Are	162
2. Which File Does What	163
3. Making a Change That Affects Only Some Variants	165
4. Where the Variant Switches Are	165
5. Other Knobs Worth Knowing	166
6. After You Edit — Rebuilding	166
7. How to Check Your Edit Before Trusting It	167
12. User Guide	169
Table of Contents	169
Introduction	169
Getting Started	170
Interface Overview	171
Basic Operations	172
Advanced Features	175
Demodulation Modes	178
Digital Decoders	181
Keyboard Shortcuts	184
Bookmarks	185
Mobile Usage	186
Tips and Best Practices	187
Troubleshooting	188
Frequently Asked Questions	189

Resources	190
Appendix: Common Frequencies	191
Glossary	192
13. KiwiSDR Client Emulation	193
Table of Contents	193
1. What a Kiwi client gets	193
2. Installing the bridge	194
3. Switching it on	194
4. Configuration reference	195
5. Connecting a client	195
6. Audio level	195
7. The S-meter	196
8. Waterfall and spectrum	197
9. If something looks wrong	198
10. How far this has been verified	198
14. Receive Diversity	200
What it does, and what it does not	200
Quick start	201
Using a WebSDR	202
Reading the panel	203
Remote SNR trim	204
Choosing a second receiver	205
Troubleshooting	205
Limitations	206
15. Rig Control	207
What you need	207
Supported receivers	207
What it does, and what it does not	208
Quick start	208
Choosing how to reach the rig	208
Serial settings	209
Sync	210
Modes	211
Filter width	211
Mute while transmitting	212
Frequency offset	212
The Rig menu	212
Linux	213
Windows	213
TCI-CAT on the receiver page	213
For receiver operators	221
For developers: the page interface	222
Troubleshooting	223
Known limitations	223
16. Project Structure	224
Table of Contents	224
Directory Tree	225
Root Directory	238

Source Code (src/)	241
Frontend (frontend/)	242
Frequency Lists (frequencylist/)	246
Configuration Files	246
Build System	248
File Dependencies	248
Data Flow	249
File Modification Guide	249
Important Paths	250
Common File Operations	250
Version Control	251

CHAPTER 1

README

README.md

PhantomSDR-Plus

An enhanced open-source WebSDR server with advanced features and futuristic design

PhantomSDR-Plus is a fork of PhantomSDR that provides a high-performance Software Defined Radio (SDR) web server capable of handling hundreds of simultaneous users. It features an enhanced user interface, multiple decoder support, band plan visualization, and compatibility with various SDR hardware platforms.

Key Features

Performance & Scalability

- **Multi-user support:** Handle hundreds of concurrent users depending on hardware
- **High sample rate:** Support for SDRs up to 70 MSPS (real) / 35 MSPS (IQ)
- **Hardware acceleration:** OpenCL and CUDA support for GPU-accelerated processing
- **Optimized streaming:** Low-latency FLAC and Opus audio compression. The browser decodes each FLAC packet the moment it arrives, and its playback buffer adjusts itself to the listener's connection, keeping only as much audio in hand as needed — typically 100–150 ms from receiver to loudspeaker on a phone over Wi-Fi

User Interface

- **Futuristic design:** Modern, responsive web interface
- **Mobile-optimized:** Enhanced mobile GUI for on-the-go listening
- **Band plan visualization:** Interactive waterfall with frequency band overlays
- **Customizable colormaps:** Multiple waterfall color schemes
- **Dual S-meter:** Choice between analog or digital signal meter display
- **One-touch decoder buttons:** A Decoders row sits on the main panel directly under the Modes selector (FT8, FT4, FT2, CW, WSPR, FAX, SSTV, NAVTEX, RTTY), with RADEL and RADEU beside the Modes selector itself and repeated in the Modes and Bands pop-ups. One press starts the decoder and opens its window, a second press stops it. The old Bandwidth row has been removed.
- **Channel scanner:** sweeps the band, or exactly what the waterfall is showing, and stops on the first channel rising a chosen number of dB above the band's own noise floor, then carries on by itself once that channel goes quiet. The step follows the mode and stops land on the channel grid (1 kHz SSB, 100 Hz CW, 5 kHz shortwave AM, 9/10 kHz medium wave, 9 kHz long wave); a *Skip empty* mode reads the waterfall spectrum and jumps straight to the signals; a lockout button drops an always-busy channel out of the sweep. Everything is local to the listener's browser, so scanning never moves the receiver for anyone else (see [User Guide](#))
- **Receive diversity:** pairs the receiver with a second one somewhere else and plays whichever of the two currently has the better signal, so a fade at one site is covered by the other. The partner can be another PhantomSDR-Plus, a KiwiSDR, an UberSDR or a WebSDR; the first three need nothing beyond the listener's browser, and only a WebSDR needs a small relay on your own server. Alignment takes about fifteen seconds, and a remote SNR trim balances the two sites against each other. Each

listener keeps a named list of second receivers in their own browser, in whatever order they choose, and can export it to a file and import it back (see [Receive Diversity](#))

- **Several receivers, one page:** one computer can run two or more receivers at once — for example an RX-888 for HF and an RTL-SDR for 2 m — and listeners switch between them with buttons in the page header. Each receiver keeps its own waterfall, chat, markers, listener list and station details, and the S-meter reads VHF S-units ($S9 = -93$ dBm) above 30 MHz (see [Several Receivers](#))

Signal Processing

- **Multiple demodulation modes:** AM, FM, USB, LSB, CW, and more. A version 1 RADE decoder has also been implemented.
- **Synchronous AM detection:** Improved AM reception quality
- **Noise reduction:** Advanced NR, NC (Noise Cancel), and NB (Noise Blanker)
- **AGC options:** Multiple Automatic Gain Control modes
- **Auto squelch:** Automatic noise-based squelch threshold
- **CTCSS tone squelch:** FM audio passes only while the chosen subtone — or any standard one — is received; the tone is picked from a small window beside the CTCSS button, which then shows it
- **AI noise reduction:** RNNNoise neural network in the listener's browser, removes band noise from speech (voice modes only; see [User Guide](#))

Advanced Features

- **Digital decoders:** FT8, FT4, FT2, JS8, CW, QRSS Grabber, WSPR, HF FAX, SSTV, NAVTEX, FSK/RTTY, PSK31, Olivia and FreeDV RADE, each running on its own background thread so decoding never interrupts the audio (see [Decoders](#))
- **Decoder ID:** names the digital mode in the passband and offers the matching decoder in one click, from measured bandwidth, tone spacing, symbol rate and burst timing, with the frequency taken into account as supporting evidence — which is also what separates **FT8 from JS8**, two modes whose signals are identical. It only suggests, never switches by itself, and stays quiet rather than guessing when the signal is too weak. Fading does not silence it: when QSB breaks a two-minute WSPR transmission into fragments it still names the mode, reading the cadence from the whole listening history, and lowers the confidence it shows to say that the evidence is weaker. Between about 400 Hz and 2700 Hz the position in the passband makes no difference; outside that an amber **Recentre & retry** button appears, which moves the dial once and starts a fresh measurement. When you do accept a suggestion, the receiver is set up to work that mode: the tuned frequency moves to the middle of the waterfall, the view becomes about 100 kHz wide around it, and the sideband and passband become the ones the mode is worked with — the 3 kHz sub-band for the FT8 family, 1350–1650 Hz for WSPR, ± 250 Hz and CW for Morse, and so on. Off by default, and about 0.5% of one core while running (see [Decoders](#))
- **Digital mode presets:** pressing FT8, FT4, FT2, JS8 or WSPR sets both the sideband and the passband
- **JS8 conversation panel:** multi-frame messages are reassembled into whole sentences
- **Statistics dashboard:** Real-time server and user statistics
- **Connected users list:** everyone listening, with a tune-to button on each row. Your own session is marked **you** — both in the pop-up window and when `users.html` is opened as a page of its own
- **Automatic spot reporting:** an autorun daemon decodes FT8, FT4 and WSPR on the server and uploads spots to PSK Reporter and WSPRnet, with two counters in the admin panel — per-decoder tiles for the current run, and an all-time total beside each band/mode checkbox
- **Server graphs:** an admin **Graphs** page plotting CPU frequency, CPU load, CPU temperature and users online over the last 15 minutes to 24 hours (see [Admin Panel](#))

- **Thermal protection:** a guard in the admin panel stops the server if the CPU overheats and restarts it once cool, with thresholds derived from your own CPU's critical trip point and no assumptions about how you start or stop the server; it ships in log-only mode, so it does nothing until you enable it (see the [Thermal Guard manual](#))
- **KiwiSDR client emulation:** an optional bridge that answers the KiwiSDR protocol on the same host and port, so Kiwi software such as **AetherSDR** and `kiwiclient` connects to the receiver directly — with real retuning, waterfall and an S-meter on the same scale as the web one. Off until `[kiwi_emulation] enabled = true` is added to your config (see [KiwiSDR Client Emulation](#))
- **Transceiver control (CAT):** keeps your own rig and your receiver window on the same frequency, mode and filter, in either direction or both — through the Rig menu of [Desktop PhantomSDR+](#) (Icom, Yaesu, Kenwood, Elecraft, FlexRadio and QRP Labs directly, every other rig through Hamlib, or flrig) or the CATsync Tool for WebSDRs in a browser. Pages expose `catsync_*` functions for frequency, mode, filter width and mute. It works with PhantomSDR-Plus, KiwiSDR, PA3FWM WebSDR and UberSDR receivers (see [Rig Control](#))
- **Bookmark system:** Import/export frequency bookmarks
- **Keyboard shortcuts:** Efficient navigation and control
- **Mouse wheel support:** Intuitive frequency tuning
- **WebSDR directory:** Integration with <https://sdr-list.xyz>

Supported Hardware

PhantomSDR-Plus supports a wide range of SDR receivers:

Device	Sample Rate	Format	Interface
RX888 MK2	Up to 64 MHz	16-bit	Native
RTL-SDR	Up to 3.2 MHz	8-bit	<code>rtl_sdr</code>
HackRF One	Up to 20 MHz	8-bit	<code>hackrf_transfer</code>
Airspy HF+ / Discovery	Up to 912 kHz	16-bit	SoapySDR (SoapyAirspyHF) + <code>rx_sdr</code>
SDRplay RSP1A	Up to 10 MHz	16-bit	SoapySDR (libmirisdr-5 + SoapyMiri, or the SDRplay API) + <code>rx_sdr</code>
RigExpert Fobos SDR	50 MHz (HF direct sampling) / up to 20 MHz (RF)	16-bit / 32-bit float	SoapySDR (SoapyFobosSDR) + <code>rx_sdr</code>
Other devices	Variable	Various	SoapySDR/ <code>rx_tools</code>

Quick Start

System Requirements

Minimum:

- Ubuntu 22.04 LTS (recommended) or Fedora
- 2-core CPU
- 4 GB RAM
- 10 GB disk space

Recommended:

- Ubuntu 24.04 LTS
- 4+ core CPU (Ryzen 5 2600 or Intel i5-6500T or better)
- 8 GB RAM
- GPU with OpenCL support (optional but highly recommended)
- SSD storage

Installation

```
# Clone the repository
git clone --recursive https://github.com/sv1btl/PhantomSDR-Plus
cd PhantomSDR-Plus

# Make scripts executable
chmod +x *.sh

# Run the installer
./install.sh
```

Note: After running `install.sh`, restart your terminal before proceeding.

Already running PhantomSDR-Plus? Do not install it again — update it. Fetch the updater once and run it; your configuration, markers, admin password, frequency list and chat history are never touched, and anything you edited yourself is put to you rather than overwritten:

```
bash cd ~/PhantomSDR-Plus curl -fLO https://raw.githubusercontent.com/sv1btl/PhantomSDR-Plus/main/update.sh
chmod +x update.sh ./update.sh
```

That last line only *reports* what would change and writes nothing; `./update.sh --apply` does it. The *Updating PhantomSDR-Plus* chapter has the details. Re-running the installer over a working site is only needed when a rebuild fails because system packages are missing.

Use `install_fedora.sh` on Fedora, `install_arch.sh` on Arch or `install_opensuse.sh` on openSUSE Tumbleweed — they do the same work with a different package manager. Every Debian and Ubuntu release is covered by `install.sh` itself.

Nothing has to be installed by hand first. The script installs the build dependencies and Node.js, builds the backend, builds the driver for the receiver you choose (and the RX888 udev rules with it), opens `site_information.json` for you to fill in, builds the frontend, installs OpenCL when the hardware supports it, then sets up the **admin panel**, the **FreeDV RADE decoder** and the **statistics server** — all three by default, answer `n` to skip any — and finishes with a full `recompile.sh`. It is interactive throughout and can take from twenty minutes to over an hour.

For the step-by-step description, see [INSTALLATION.md](#) → [What the installer does](#).

Performance Benchmarks

Hardware	Sample Rate	CPU Usage	User Capacity
Ryzen 5 2600 (all cores)	64 MHz (32 MHz IQ)	38-40%	100+ users
AMD RX 580 (GPU)	64 MHz (32 MHz IQ)	28-35%	100+ users
Intel i5-6500T (with OpenCL)	60 MHz (30 MHz IQ)	10-12%	100+ users

Note: Per-user CPU overhead is minimal (<1% per user) when hardware acceleration is enabled.

Usage

Basic Operation

1. **Configure your SDR:** Edit the appropriate config file (e.g., `config-rtl.toml`)
2. **Update site information:** Edit `frontend/site_information.json`
3. **Start the server:** Run the appropriate start script: `bash ./start-rtl.sh` # For RTL-SDR `./start-rsp1a.sh` # For SDRplay RSP1A `./start-airspyhf.sh` # For Airspy HF+ `./start-fobos-hf.sh` # For RigExpert Fobos SDR, HF1/HF2 (0-25 MHz) `./start-fobos.sh` # For RigExpert Fobos SDR, RF (25-6000 MHz) `./start-hackrf.sh` # For HackRF One `./start-rx888mk2.sh` # For RX888 MK2 The RSP1A, Airspy HF+ and Fobos run through SoapySDR and `rx_sdr`; the installer sets up their driver when you pick them (options 3, 6 and 5), or run `./setup-rsp1a.sh`, `./setup-airspyhf.sh` or `./setup-fobos.sh` on an installed station. See [INSTALLATION.md](#). The HackRF One (option 7) needs only the distribution's `hackrf` package: `./setup-hackrf.sh`. Each start script is **self-contained**: it stops any running instance, launches the receiver + `spectrumserver`, **detaches into the background** (survives closing the terminal), and runs a **watchdog** that auto-restarts the chain if it dies. Progress is logged to `logwebsdr.txt` — watch it with `tail -f logwebsdr.txt`. Re-running a start script is a clean restart, and a `flock` lock ensures only one receiver runs at a time. Optional: `SPECTRUM_CORES=0-3` to pin CPUs, `RX_ARGS=""` to override receiver args without editing the file (`RX888_ARGS` for RX888), `RX_DRIVER=miri|sdrplay` to choose the RSP1A driver.
4. **Access the interface:** Open your browser to `http://localhost:PORT` (default varies by config)

Stopping the Server

One shared stop script works for **every** receiver — it stops the watchdog first (so it can't auto-restart), then the receiver and `spectrumserver`:

```
./stop-websdr.sh
```

Configuration

Essential Configuration Files

1. `config-[device].toml` - Server and SDR configuration
2. Server settings (port, threads, HTML root)
3. Input settings (sample rate, FFT size, frequency)
4. WebSDR registration options
5. Audio compression settings
6. After modifying it, restart the server.
7. `frontend/site_information.json` - Public site information
8. Operator details (callsign, email, location)
9. Hardware and antenna information
10. Region and bandwidth settings
11. Chat enable/disable
12. After modifying it, run `recompile.sh` in a terminal.

13. `markers.json` - Frequency markers and band plan
14. `frontend/src/bands-config.js` - Band plan configuration This file is located in `frontend/src/bands-config.js` and defines the bands the SysOp creates.
15. You will see something like the following, and you are free to edit it as you like: ``
16. `const bands = .....`
17. `{ ITU: 1, name: '40m', min: -30, max: 110, initFreq: '7120000', publishBand: '1', startFreq: 7000000, endFreq: 7200000, stepi: 1000, color: 'rgba(50, 168, 72, 0.6)', modes: [{ mode: MODES.CW, startFreq: 7000000, endFreq: 7040000 }, { mode: MODES.LSB, startFreq: 7040000, endFreq: 7200000 }] }, { ITU: 2, name: '40m', min: -30, max: 110, initFreq: '', publishBand: '1', startFreq: 7000000, endFreq: 7300000, stepi: 1000, color: 'rgba(50, 168, 72, 0.6)', modes: [{ mode: MODES.CW, startFreq: 7000000, endFreq: 7050000 }, { mode: MODES.LSB, startFreq: 7050000, endFreq: 7300000 }] }, { ITU: 3, name: '40m', min: -30, max: 110, initFreq: '', publishBand: '1', startFreq: 7000000, endFreq: 7200000, stepi: 1000, color: 'rgba(50, 168, 72, 0.6)', modes: [{ mode: MODES.CW, startFreq: 7000000, endFreq: 7040000 }, { mode: MODES.LSB, startFreq: 7040000, endFreq: 7200000 }] } etc `` Where:`
18. **ITU** is the region in which the server is located,
19. **min & max** are the waterfall brightness limits for the given band,
20. **initFreq** is the initial frequency to tune to when the band is selected,
21. **publishBand** sets whether the band is shown: 1 for amateur bands, 2 for broadcast bands,
22. **startFreq & endFreq** define the band limits,
23. **stepi** is the default mouse-wheel step for the band, and **modes** is the preferred default mode for the band.
24. After making changes, **run `recompile.sh` in a terminal** from the PhantomSDR-Plus folder.

For configuration examples, see the included sample files:

- `config-rtl.toml` - RTL-SDR configuration
- `config-rspla.toml` - SDRplay RSP1A configuration
- `config-airspyhf.toml` - Airspy HF+ configuration
- `config-fobos-hf.toml` / `config-fobos.toml` - RigExpert Fobos SDR configuration (HF / RF)
- `config-hackrf.toml` - HackRF One
- `config-rx888mk2.toml` - RX888 MK2 configuration

Waterfall Floor — `waterfall.sh`

`waterfall.sh` (in the repository root) changes the **default minimum waterfall level** in dB — the floor the waterfall starts from on a fresh visit.

That value lives in two source files at five different places (`frontend/src/waterfall.js` → `this.minWaterfall`, and `frontend/src/App.svelte` → the initial `let min_waterfall`, the two "min" reset cases and the bookmark default). Editing them by hand is easy to get wrong, so the script does it for you. It never uses hard-coded line numbers — every spot is located by pattern, so it keeps working after the sources are updated.

- A **higher** value (e.g. `-15`) gives a **darker** waterfall.
- A **lower** value (e.g. `-45`) gives a **brighter** waterfall.

```
./waterfall.sh          # interactive – shows the current values and asks for the new one
./waterfall.sh -v -15   # set the value without being asked
./waterfall.sh -v -15 -y # ... and skip every confirmation
./waterfall.sh -s       # show the current values only, change nothing
```

Before editing, it timestamps a backup of both files (`*.bak-YYYYmmdd-HHMMSS`) and prints the restore command. Only literals equal to the value currently in use are touched, so unrelated numbers in those files can never be rewritten by accident. If the new value does not show up afterwards, the script restores the backups by itself and exits with an error. It also warns you when the two files are out of sync and sets both.

- After the change the frontend must be rebuilt — the script offers to run `recompile.sh` for you (with `-y` it skips the rebuild, so run `./recompile.sh` yourself when you are ready).

Customization

Changing the Background Image

Replace `frontend/src/assets/background.jpg` with your preferred image (keep the same filename).

- After modifying it, run `recompile.sh` in a terminal.

Audio Codec Selection

Choose between FLAC and Opus in your `.toml` configuration:

```
[input]
audio_compression="opus" # or "flac"
```

- After modifying it, restart the server.

S-meter calibration

The two meters are deliberately on **separate chains**, so they are tuned with two different knobs.

The two knobs

Both live in the `[input]` section of the config the server is started with (`config.toml`, `config-rx888mk2.toml`, etc.):

```
[input]
smeter_offset=5      # digital bar only
analog_smeter_offset=5 # analog needle only
```

They are read in `src/websocket.cpp` and sent to the browser in the first `basic_info` frame, so **a server restart is enough** — no frontend rebuild is required.

Analog chain

```
displayed dBm = -130 + (rawDb + analog_smeter_offset + 130) x 1.1
```

- The visual gain of 1.1 is applied *after* the offset, pivoting at -130 dBm. So one unit of `analog_smeter_offset` moves the reading by 1.1 dB. To shift the display by X dB, use `analog_smeter_offset ~= 0.91 x X`. The TOML value is parsed as an integer, so only whole steps are available.

- The needle law (`powerFromDbm` in `frontend/src/lib/SMeterAnalog.svelte`) puts **S9 at -73 dBm**, at 60 of 100 needle units, with a `pow(...,0.6)` curve from -130 to -73 below S9 and a `pow(...,0.8)` curve from -73 to -13 (S9+60) above it.
- It is **not** a linear 6 dB per S-unit scale, so S9 is the meaningful calibration anchor — the marks below S9 will not track a signal generator in 6 dB steps.

Digital chain

```
value      = (rawDb / 150) x 100 + smeter_offset  -> clamped to [-100, 0]
segments = round((value + 100) x 35 / 100) + DIGITAL_BAR_TRIM
```

- Note the `/150 x 100`: the digital scale is **compressed to 2/3**, and it never sees `analog_smeter_offset` or the 1.1 visual gain. That is why the bar and the needle never agree exactly — it is intentional.
- 35 segments over 100 units means one segment is 2.86 offset units, roughly 4.3 raw dB. So `smeter_offset=3` is about one segment.
- `DIGITAL_BAR_TRIM` in `frontend/src/lib/SMeterDigital.svelte` (currently 0) shifts the bar by whole segments. Changing it is a source edit and **does** require a frontend rebuild.

The LED windows are analog-driven in both meters

The dBm / dBμV / SNR / NF windows in **both** meters read the analog-calibrated value and add a fixed `VISUAL_DBM_OFFSET` of 5, hardcoded in `SMeterAnalog.svelte` and `SMeterDigital.svelte`. Therefore:

- `smeter_offset` does **not** change any number in the windows, only the bar length.
- `dBμV = dBm + 107` (50 ohm), and `NF = dBm - SNR` by construction, so NF follows automatically.
- To correct the **numbers** without touching the needle position, `VISUAL_DBM_OFFSET` is the knob — but it is a source constant, needs a rebuild, and must be changed in **both** files to stay consistent.

Procedure

1. Feed a known level into the antenna port (signal generator at -73 dBm = S9), in SSB/CW with a stable passband. Readings depend on bandwidth and AGC, so fix the mode and bandwidth first.
2. Note what the analog dBm window shows, then set `analog_smeter_offset` `≈ 0.91 x (-73 - shown)`. Restart the server and recheck. One iteration should be enough; the needle should land on S9.
3. Switch to the digital meter and set `smeter_offset` so the bar length places S9 where you want it — roughly 3 units per segment. This step is cosmetic matching only and has no effect on the readouts.
4. Without a signal generator, a quiet band with a known-level reference beacon is a reasonable substitute, or simply anchor the noise floor to a plausible value (e.g. -120 dBm on 20 m with a decent antenna) — accepting that this calibrates the whole chain including antenna gain, not the receiver alone.

GUI Variants

All four GUI variants (analog/digital S-meter x v1/v2 layout) live in the one build. Visitors switch between them with the ☿ menu at the top right of the page — nothing reloads, so audio, the waterfall and any running decoder carry on — and each visitor's choice is remembered in their own browser.

- Run `recompile.sh` in a terminal to set the *starting* variant, the one a first-time visitor sees.
- **EDITING_VARIANTS.md** - How to edit the frontend variants (S-meter and layout) and rebuild them

Documentation

- **INSTALLATION.md** - Complete installation guide for system operators

- **ADMIN_PANEL_SETUP.md** - Installation guide for admin system operators
 - **USER_GUIDE.md** - End-user guide for operating the WebSDR
 - **THERMAL_GUARD.md** - Sysop manual for the CPU over-temperature guard: the four modes and what to do with each
 - **CONNECTION_LIMITS.md** - Sysop manual for connection limits: protecting a public receiver from connection floods
 - **MULTI_RECEIVER.md** - Sysop manual for running several receivers on one computer: instances, `receivers.toml`, the receiver picker and `start-all.sh`
 - **PROJECT_STRUCTURE.md** - Directory structure and code organization
 - **EDITING_VARIANTS.md** - How to edit the frontend variants (S-meter and layout) and rebuild them
-

Online WebSDR List

Register your WebSDR on the official directory: <https://sdr-list.xyz>

Set `register_online=true` in your `.toml` configuration file to automatically register.

Troubleshooting

Common Issues

OpenCL Not Working

- Ensure drivers are properly installed
- Check with `clinfo` command
- See `INSTALLATION.md` for detailed OpenCL setup

Audio Latency Issues

- Keep FLAC as the codec: it is the lower-latency one, while Opus holds back up to 20 ms per packet
- Open the page with `?diag=1` added to its address (for example `/mobile?diag=1`) to see the delay split into packet, buffer and output; the buffer shrinks by itself within a minute or two of a network hiccup
- If the audio breaks up, move the buffer control one step higher (see the User Guide, *Audio Buffer and Delay*)
- Ensure adequate CPU/GPU resources

Build Failures

- Verify all dependencies are installed
 - Try cleaning build directory: `rm -rf build && meson setup build`
 - Check for conflicting library versions
-

Contributing

Contributions are welcome! This is an independent fork with additional features. Please:

1. Fork the repository
2. Create a feature branch
3. Commit your changes

4. Push to the branch
 5. Create a Pull Request
-

License

This project is licensed under the GNU General Public License v3.0 - see the [LICENSE](#) file for details.

Authors & Credits

- **SV1BTL & SV2AMK** - PhantomSDR-Plus development and enhancements
 - Based on the original PhantomSDR project
-

Links

- **Live Demo:** <http://phantomsdr.no-ip.org:8900/>
 - **WebSDR Directory:** <https://sdr-list.xyz>
 - **GitHub Repository:** <https://github.com/sv1btl/PhantomSDR-Plus>
-

Support



- **Issues:** Report bugs via [GitHub Issues](#)
 - **Email:** contact sv1btl@otenet.gr
-

Performance Tips

1. **Enable OpenCL/CUDA** for GPU acceleration - dramatically reduces CPU usage
 2. **Use SSD storage** for better I/O performance
 3. **Allocate sufficient RAM** - at least 8 GB recommended
 4. **Optimize FFT size** - larger FFT = better resolution but more CPU usage
 5. **Consider dedicated GPU** - AMD or NVIDIA with OpenCL support
-

73 de SV1BTL & SV2AMK

*For detailed setup instructions, see [INSTALLATION.md](#) For end-user operation guide, see [USER_GUIDE.md](#)
For other translations in various languages, please refer to the folders in this section*

CHAPTER 2

Installation Guide

INSTALLATION.md

PhantomSDR-Plus Installation Guide for System Operators

This comprehensive guide will walk you through installing and configuring PhantomSDR-Plus on your server.

Table of Contents

1. System Requirements
2. Pre-Installation Preparation
3. Installing PhantomSDR-Plus — the installer, and [what it does](#)
4. Autorun Spot Reporter (FT8/FT4/WSPR)
5. KiwiSDR Client Emulation (optional)
6. Configuration
7. SDR Device-Specific Setup
8. Testing and Verification
9. Setting Up Autostart
10. CPU Thermal Protection
11. Troubleshooting

Reference only — the installer already does all of this for you. Read these sections when you are on a distribution none of the installers covers, or when you need to repair one piece by hand:

- [Installing Dependencies](#)
 - [Installing Node.js and npm](#)
 - [Installing OpenCL](#)
 - [Building by hand](#)
 - [Installing Opus Audio Codec](#)
-

System Requirements

Supported Operating Systems

Primary (Recommended):

- Ubuntu 22.04 LTS (Jammy), 24.04 LTS (Noble) and 26.04 LTS (Resolute) — 24.04 recommended
- Debian 12 (Bookworm) and Debian 13 (Trixie)

Alternative:

- Fedora (latest stable release)
- Arch Linux (rolling)

- openSUSE Tumbleweed (not Leap — see the note further down)

Hardware Requirements

Minimum Configuration:

- CPU: Dual-core processor (2+ GHz)
- RAM: 4 GB
- Storage: 10 GB free space
- Network: 100 Mbps connection

Recommended Configuration:

- CPU: Quad-core or better (Ryzen 5 2600, Intel i5-6500T or better)
- RAM: 8 GB or more
- Storage: 20 GB+ SSD
- GPU: AMD/NVIDIA with OpenCL support (highly recommended)
- Network: 1 Gbps connection

High-Performance Configuration:

- CPU: 6+ cores (Ryzen 7, Intel i7 or better)
- RAM: 16 GB or more
- Storage: NVMe SSD
- GPU: Dedicated GPU with OpenCL/CUDA support
- Network: 1 Gbps or better

Pre-Installation Preparation

1. Update Your System

```
# Ubuntu/Debian
sudo apt update
sudo apt upgrade -y
sudo reboot

# Fedora
sudo dnf update -y
sudo reboot
```

2. Verify Ubuntu Version

```
lsb_release -a
```

Expected output should show: Ubuntu 24.04 LTS

3. Check Available Disk Space

```
df -h
```

Ensure you have at least 10 GB free in your home directory.

4. Check CPU Information

```
lscpu
```

Note the number of cores/threads for configuration optimization.

Installing Dependencies

IMPORTANT

You do not need this section for a normal installation. `./install.sh` — or the installer for your distribution — does all of it for you; see [What the installer does](#). What follows is a reference for a manual setup, for a distribution none of the installers covers, or for repairing one piece by hand.

Ubuntu 24.04 LTS

```
sudo apt install -y \  
  build-essential \  
  cmake \  
  pkg-config \  
  meson \  
  libfftw3-dev \  
  libwebsocketpp-dev \  
  libflac++-dev \  
  zlib1g-dev \  
  libzstd-dev \  
  libboost-all-dev \  
  libopus-dev \  
  libliquid-dev \  
  git \  
  psmisc \  
  wget \  
  curl
```

Fedora

```
sudo dnf install -y \  
  g++ \  
  meson \  
  cmake \  
  fftw3-devel \  
  websocketpp-devel \  
  flac-devel \  
  zlib-devel \  
  boost-devel \  
  libzstd-devel \  
  opus-devel \  
  liquid-dsp-devel \  
  git \  
  psmisc \  
  wget \  
  curl
```

Verify Installation

```
# Check if key dependencies are installed
pkg-config --modversion fftw3
cmake --version
meson --version
```

Installing Node.js and npm

IMPORTANT

You do not need this section for a normal installation. `./install.sh` — or the installer for your distribution — does all of it for you; see [What the installer does](#). What follows is a reference for a manual setup, for a distribution none of the installers covers, or for repairing one piece by hand.

PhantomSDR-Plus requires Node.js for building the frontend. We'll use NVM (Node Version Manager) for installation.

1. Install NVM

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.4/install.sh | bash
```

2. Load NVM

IMPORTANT: Close and reopen your terminal, or run:

```
export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh"
```

3. Verify NVM Installation

```
nvm --version
```

Expected output: 0.40.4 or similar

4. Install Node.js

```
# Install Node.js 22 - the version every install script requires
nvm install 22

# Make it the default, so systemd units and cron see it too
nvm alias default 22

# Verify installation
node --version
npm --version
```

5. Optional: Install Additional Node Versions

```
# Install latest version
nvm install node

# Install specific version (if needed)
nvm install 22.22.3

# List installed versions
nvm list

# Use specific version
nvm use 22
```

Installing OpenCL (Optional but Recommended)

IMPORTANT

You do not need this section for a normal installation. `./install.sh` — or the installer for your distribution — does all of it for you; see [What the installer does](#). What follows is a reference for a manual setup, for a distribution none of the installers covers, or for repairing one piece by hand.

OpenCL dramatically improves performance by offloading FFT calculations to the GPU. This section covers Intel integrated graphics. For AMD/NVIDIA GPUs, refer to manufacturer documentation.

Intel CPU with Integrated Graphics

1. Install Base OpenCL Components

```
sudo apt install -y \
  build-essential \
  cmake \
  pkg-config \
  meson \
  libfftw3-dev \
  libwebsocketpp-dev \
  libflac++-dev \
  zlib1g-dev \
  libzstd-dev \
  libboost-all-dev \
  libopus-dev \
  libliquid-dev \
  libclfft-dev \
  ocl-icd-opensl-dev \
  clinfo
```

2. Download Intel Compute Runtime

```
mkdir -p ~/neo
cd ~/neo

wget https://github.com/intel/compute-runtime/releases/download/19.07.12410/intel-
gmmlib_18.4.1_amd64.deb
wget https://github.com/intel/compute-runtime/releases/download/19.07.12410/intel-igc-
core_18.50.1270_amd64.deb
wget https://github.com/intel/compute-runtime/releases/download/19.07.12410/intel-igc-
opencl_18.50.1270_amd64.deb
wget https://github.com/intel/compute-runtime/releases/download/19.07.12410/intel-
opencl_19.07.12410_amd64.deb
wget https://github.com/intel/compute-runtime/releases/download/19.07.12410/intel-
ocloc_19.07.12410_amd64.deb
```

3. Install Intel OpenCL Runtime

```
sudo dpkg -i *.deb
```

If dependency errors occur:

```
sudo apt --fix-broken install
```

4. Install OpenCL ICD Loader

```
sudo apt update
sudo apt install intel-opencl-icd
```

If errors occur:

```
sudo apt --fix-broken install
sudo apt install intel-opencl-icd
```

5. Verify OpenCL Installation

```
sudo clinfo
```

You should see information about your OpenCL platform and devices. Look for:

- Number of platforms: 1 (or more)
- Platform name: Intel(R) OpenCL (or similar)
- Device type: GPU or CPU

6. Reboot

```
sudo reboot
```

AMD GPU OpenCL

For AMD GPUs, install ROCm:

```
# Add ROCm repository
wget -q -O - https://repo.radeon.com/rocm/rocm.gpg.key | sudo apt-key add -
echo 'deb [arch=amd64] https://repo.radeon.com/rocm/apt/debian/ ubuntu main' | sudo tee /etc/apt/
sources.list.d/rocm.list

# Install ROCm
sudo apt update
sudo apt install rocm-opencl rocm-clinfo

# Add user to video group
sudo usermod -a -G video $USER

# Reboot
sudo reboot

# Verify
clinfo
```

NVIDIA GPU OpenCL

For NVIDIA GPUs, install CUDA:

```
# Install NVIDIA drivers
sudo apt install nvidia-driver-525

# Install CUDA toolkit
wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2204/x86_64/cuda-
keyring_1.0-1_all.deb
sudo dpkg -i cuda-keyring_1.0-1_all.deb
sudo apt update
sudo apt install cuda

# Reboot
sudo reboot

# Verify
nvidia-smi
clinfo
```

Installing PhantomSDR-Plus

Clone the Repository, Make Scripts Executable, Run the Installer

```
cd ~
git clone --recursive https://github.com/svlbtl/PhantomSDR-Plus
cd PhantomSDR-Plus
chmod +x *.sh
./install.sh
```

⚠ **IMPORTANT:** After the installer completes, **restart your terminal** before continuing — the newly installed Node.js and Rust are not on your PATH until you do.

Already running PhantomSDR-Plus? Do not install it again — update it. Fetch the updater once and run it; your configuration, markers, admin password, frequency list and chat history are never touched, and anything you edited yourself is put to you rather than overwritten:

```
bash cd ~/PhantomSDR-Plus curl -fLO https://raw.githubusercontent.com/sv1btl/PhantomSDR-Plus/main/update.sh chmod +x update.sh ./update.sh
```

That last line only *reports* what would change and writes nothing; `./update.sh --apply` does it. The *Updating PhantomSDR-Plus* chapter has the details. Re-running the installer over a working site is only needed when a rebuild fails because system packages are missing.

Use the script that matches your system. They do the same work and ask the same questions; only the package manager differs:

System	Script
Ubuntu 22.04 / 24.04 / 26.04, Debian 12 / 13	<code>./install.sh</code>
Fedora	<code>./install_fedora.sh</code>
Arch	<code>./install_arch.sh</code>
openSUSE Tumbleweed	<code>./install_opensuse.sh</code>

One installer for every Debian and Ubuntu release. `install.sh` reads `/etc/os-release` and the installed Boost version and adapts itself; the old `install_ubuntu22.sh`, `install-Deb12.sh` and `install_ubuntu26.sh` are gone. It sets `DEBIAN_FRONTEND=noninteractive`, so the `tzdata` that arrives with `python3-matplotlib` cannot stop the run to ask for your time zone and then swallow the answer meant for the next question. On Jammy it calls `install_rade_ubuntu22.sh` directly, because Jammy's `python3-websockets` is 10.1 and RADE needs 11.0 or newer. It knows where `intel-ocl-icd` lives on each release: in the repositories on 22.04 and 26.04, in `non-free` on Debian 12, and in Intel's own graphics repository on 24.04 and Debian 13. And where Boost is 1.87 or newer, the `websocketpp` header patch stops being optional — the installer verifies it landed and refuses to build without it. No newer compiler is ever needed: the stock GCC accepts `-std=c++23` on all five releases, so do not install `gcc-12` for this.

The installation runs in 17 clearly numbered steps, and every point where it waits for you is fenced by a  **YOUR INPUT IS NEEDED** banner, so a question cannot be mistaken for progress output scrolling past. The seven questions are listed up front, before anything is installed. Setting `PHANTOM_NONINTERACTIVE=1` answers all of them with their defaults; see the header of `install.sh` for the `PHANTOM_*` overrides.

Every run writes `install.txt`. When the installer finishes — or dies halfway — it writes a report to `install.txt` in the PhantomSDR-Plus directory: the result, each of the 17 steps as OK / SKIPPED / PARTIAL / FAILED, what it detected (distribution, Boost, compiler, Node.js), which components were installed, and every warning that came up. A run that fails leaves a report ending at the step that failed, with the reason and a note that the installer is safe to re-run. It is the first file to read when something did not work, and the first to attach to a bug report. Each run overwrites it, so keep a copy if you want to compare two installations.

Ubuntu 26.04, Arch and openSUSE Tumbleweed build, but have not been run on air. All three ship a Boost newer than 1.87, which removed the `io_service` API that the vendored `websocketpp` 0.8.2 was written against. The patched headers the installer copies in bridge that gap (`io_context`, `executor_work_guard`, `boost::asio::post`, the modern resolver), and where Boost is 1.87 or newer the installer treats that patch as mandatory rather than optional — it verifies the copies landed and refuses to build without them. A complete install, receiver driver and all optional components included, has been verified end to end in containers on Boost 1.90 (Ubuntu 26.04), 1.91 (openSUSE Tumbleweed) and 1.92 (Arch). That proves the server compiles and starts — not that it serves a receiver for hours. Treat all three as untested in production until someone reports back.

openSUSE means Tumbleweed. That is where `install_opensuse.sh` is verified. Leap 15.6 does not work: its repositories carry no `liquid-dsp-devel` at all, which the backend needs, and Boost is only available under versioned package names. Supporting Leap would mean adding third-party OBS repositories, so it is out of scope for now.

What the installer does

Nothing has to be prepared by hand first — no dependency list to paste, no Node.js to fetch, no OpenCL packages to hunt down. It runs as 19 numbered steps and stops to ask you up to ten questions, each one fenced by a "YOUR INPUT IS NEEDED" banner — so either stay at the keyboard, or set `PHANTOM_NONINTERACTIVE=1` and let it answer everything with its defaults (see below), and allow anywhere from about twenty minutes to well over an hour depending on the machine and on how many extras you keep.

#	Step	What you are asked
1	Lists the PhantomSDR-Plus services that are running right now — admin panel, reverse proxy, statistics server, receiver — and offers to stop them before anything is touched. Only a running start script counts as the receiver; one that is merely open in an editor is left alone.	confirm, default yes
2–6	Detects the distribution and installs every build dependency (compiler, meson/ninja, FFTW, Boost, FLAC, Opus, liquid-dsp, zlib/zstd, libcurl ...), installing Node.js 22 through nvm if the system Node is missing or too old	nothing
7	Builds the backend with meson	nothing
8	Builds the driver for your receiver — RX888 MkII / RX888, RTL-SDR (Blog V4 asked separately), SDRplay RSP1A, RigExpert Fobos SDR, Airspy HF+, HackRF One, or none. Picking the RX888 also installs the udev rules , so the server never needs <code>sudo</code> for the device. The RSP1A, Fobos and Airspy HF+ run through SoapySDR: their choice runs <code>setup-rsp1a.sh</code> , <code>setup-fobos.sh</code> or <code>setup-airspyhf.sh</code> , which builds the driver and <code>rx_sdr</code> and installs a udev rule as well (see Receivers on SoapySDR) The HackRF One needs no SoapySDR: its choice runs <code>setup-hackrf.sh</code> , which installs the distribution's <code>hackrf</code> package and a udev rule.	which SDR you have
9	Opens <code>frontend/site_information.json</code> in your editor	your callsign, locator, hardware, antenna — do not skip this
10–11	Installs the frontend dependencies and builds the desktop and <code>/mobile</code> pages	nothing
12	Installs OpenCL, choosing the provider from the hardware it finds (Intel / AMD / NVIDIA GPU, or the x86 CPU runtime). If there is no OpenCL-capable device it says so and moves on	confirm, default yes
13–15	Installs the admin panel , the FreeDV RADE V1 decoder and the statistics server — all three by default	confirm each, default yes; each has its own questions
16	Re-applies the five patched websocketpp headers over the meson subproject and verifies they landed. Three of them are the Boost ≥ 1.87 compatibility work, without which the backend cannot compile on Boost 1.90; the other two are the project's own changes, one being the fix websdr.org registration needs	nothing
17	Installs the KiwiSDR client emulation by running <code>kiwi_install.sh</code> , so Kiwi clients such as AetherSDR can connect to this receiver. Patches the sources and adds <code>[kiwi_emulation]</code> to the config files in the repository root — see KiwiSDR Client Emulation	confirm, default yes
18	Runs <code>recompile.sh</code> , so everything is built from the patched sources	<code>[3]</code> Both backend and frontend → starting variant → <code>[1]</code> <code>build-all.sh</code>
19	Prints the summary: every step with its verdict, and every component with what was installed	nothing

Installing unattended

Every question has an environment-variable override, and the installer also switches to defaults on its own when stdin is not a terminal (a pipe, a container, a CI job). This works the same way in all four installers — `install.sh`, `install_arch.sh`, `install_fedora.sh` and `install_opensuse.sh`. Set `PHANTOM_NONINTERACTIVE=1` and the whole run completes without asking anything:

Variable	Effect
<code>PHANTOM_NONINTERACTIVE=1</code>	answer every question with its default
<code>PHANTOM_SDR=1...7</code>	RX888 · RTL-SDR · SDRplay RSP1A · skip · Fobos · Airspy HF+ · HackRF (default 4)
<code>PHANTOM_RTLSDR_V4=y\ n</code>	RTL-SDR Blog V4 driver (default n)
<code>PHANTOM_SITE_EDIT=y\ n</code>	open <code>site_information.json</code> in an editor
<code>PHANTOM_OPENCL=y\ n</code>	install OpenCL (default y)
<code>PHANTOM_OPENCL_PROVIDER=1\ 2\ 3</code>	Intel · Mesa/Rusticl · POCL (default: from the detected hardware)
<code>PHANTOM_ADMIN=y\ n</code>	admin panel (y interactive, n unattended)
<code>PHANTOM_RADE=y\ n</code>	RADE / FreeDV (y interactive, n unattended)
<code>PHANTOM_STATS=y\ n</code>	statistics server (y interactive, n unattended)
<code>PHANTOM_KIWI=y\ n</code>	KiwiSDR client emulation (default y)
<code>PHANTOM_RECOMPILE=y\ n</code>	final rebuild (default y)
<code>PHANTOM_CURLPP=y\ n</code>	continue without curlpp — Arch and openSUSE only (default y)
<code>PHANTOM_FIX_CLOCK_SKEW=y\ n</code>	reset source file timestamps dated in the future, so meson can build (default y)

The three sub-installers marked *n unattended* are interactive scripts of their own, so an unattended run skips them by default rather than hang on their prompts. Name them explicitly to include them:

```
# a complete unattended install for an RX888 MkII
PHANTOM_NONINTERACTIVE=1 PHANTOM_SDR=1 ./install.sh

# ... and with the admin panel, RADE and the statistics server too
PHANTOM_NONINTERACTIVE=1 PHANTOM_SDR=1 \
  PHANTOM_ADMIN=y PHANTOM_RADE=y PHANTOM_STATS=y ./install.sh
```

After a reboot the RX888 udev rules and any OpenCL driver are fully in effect; the installer tells you when a reboot or a re-login is needed.

When it sets up the admin panel, the installer offers to install the two systemd units as well, and **strongly recommends** accepting: the CPU over-temperature guard runs inside the panel, so without them a reboot or a crash leaves the machine unprotected. The offer is skipped automatically where systemd is not running (containers, WSL1, OpenRC) — see the [Admin Panel guide](#).

Each extra is a normal script you can also run on its own at any time — `./setup_admin.sh` ([manual](#)), `./install_rade.sh` ([manual](#)), `./install-stats-server.sh` ([manual](#)), `./setup-rx888-udev.sh` and the receiver scripts `./setup-rsp1a.sh`, `./setup-fobos.sh`, `./setup-airspyhf.sh` and `./setup-hackrf.sh` — they are exactly what the installer calls.

Building by hand (reference)

IMPORTANT

You do not need this section for a normal installation. `./install.sh` — or the installer for your distribution — does all of it for you; see [What the installer does](#). What follows is a reference for a manual setup, for a distribution none of the installers covers, or for repairing one piece by hand.

Should you ever have to build without the installer — a distribution none of the scripts covers, or repairing a half-finished build:

Build the Backend

```
# Clean any previous builds, Configure build with optimization, Compile
rm -rf build
meson setup build --buildtype=release --optimization=3
meson compile -C build

# Verify binary was created
ls -lh build/spectrumserver
```

Build the Frontend

```
cd frontend

# Install dependencies, Fix any vulnerabilities, Build the frontend
npm install
npm audit fix
npm run build

# Return to project root
cd ..
```

Verify Installation

```
# Check if binary exists
ls -lh build/spectrumserver

# Check if frontend was built
ls -lh frontend/dist/

# List important files
ls -lh *.sh *.toml
```

Installing Opus Audio Codec

IMPORTANT

You do not need this section for a normal installation. `./install.sh` — or the installer for your distribution — does all of it for you; see [What the installer does](#). What follows is a reference for a manual setup, for a distribution none of the installers covers, or for repairing one piece by hand.

Opus provides better audio quality and lower latency compared to FLAC.

1. Install libopus System Library

```
sudo apt-get update
sudo apt-get install -y libopus0 libopus-dev
```

2. Install Opus Decoder for Frontend

```
cd PhantomSDR-Plus/frontend

# Install npm dependencies if not already done, Install Opus WASM decoder, Fix any vulnerabilities,
# Rebuild frontend
npm install
npm install @wasm-audio-decoders/opus-ml
npm audit fix
npm run build

# Return to project root
cd ..
```

3. Verify Opus Installation

```
# Check if Opus system library is installed
pkg-config --modversion opus

# Check if Opus npm package is installed
cd frontend
npm list @wasm-audio-decoders/opus-ml
cd ..
```

Autorun Spot Reporter (FT8/FT4/WSPR)

PhantomSDR-Plus ships an optional **autorun spot reporter** (in the `autorun/` directory). It decodes FT8/FT4/WSPR server-side straight off the receiver and uploads spots to the reporting networks:

- **FT8 / FT4 → PSK Reporter** (IPFIX/UDP)
- **WSPR → wsprnet.org** (HTTP)

It is controlled entirely from the **admin panel** → **"Spot Reporting"** tab, and **reporting is OFF by default** — nothing is transmitted or uploaded until you enable it there. The decoder runs as a separate Node.js daemon and taps the receiver's audio locally, so it adds no extra RF hardware.

Requirements

Requirement	Notes
Node.js 22+	Same runtime the frontend build already needs — installed in the Node.js step .
ws + cbor-x npm packages	Resolved through <code>autorun/node_modules</code> , a symlink to the frontend's <code>node_modules</code> (both packages are declared in <code>frontend/package.json</code>).
util-linux (taskset)	The admin "Start" button pins the daemon to the E-cores with <code>taskset</code> . Present on virtually every distro; the installers add it explicitly.
Callsign + grid	Read from <code>frontend/site_information.json</code> (<code>siteSysop</code> / <code>siteGridSquare</code>) unless overridden in the admin tab. Spots are uploaded under this callsign.

⚠️ **Report only what you actually receive.** Spots are uploaded to public networks under your callsign — only enable bands/modes your receiver genuinely hears, and use your correct grid square.

Automatic installation

`install.sh` (and the per-distro variants: `install_arch.sh`, `install_fedora.sh`, `install_opensuse.sh`) handle everything for you: they install Node.js 22 and `util-linux`, run the frontend `npm install`, and then create the `autorun/node_modules` symlink automatically. No extra steps are required — the feature is ready as soon as `install.sh` finishes.

Manual installation

NOTE

The installer already does this for you: it creates the symlink and installs `util-linux`. Use the steps below only to repair an installation by hand.

If you installed manually (Option B), create the symlink yourself after the frontend `npm install` so the daemon can resolve its dependencies:

```
cd ~/PhantomSDR-Plus

# 1. Frontend deps must be installed first (ws + cbor-x live there)
cd frontend && npm install && cd ..

# 2. Link the autorun daemon to the frontend's node_modules
ln -sfn ../frontend/node_modules autorun/node_modules

# 3. Ensure taskset is available (Debian/Ubuntu shown; use your package manager)
sudo apt install -y util-linux
```

Note: `autorun/node_modules` is git-ignored, so a fresh `git clone` never contains it — the symlink must be (re)created after every clean checkout. The installers do this for you; the command above is only for manual setups.

Verifying

```
# The symlink should point at ../frontend/node_modules
ls -l autorun/node_modules

# ws + cbor-x must resolve
ls -ld frontend/node_modules/ws frontend/node_modules/cbor-x

# taskset must be on PATH
command -v taskset
```

Then open the admin panel, go to the **Spot Reporting** tab, set your identity, tick the bands/modes to decode, enable the destination(s), and press **Start**. A "REPORTING" badge appears on the main waterfall whenever reporting is active. (PSK Reporter uploads batch every 5 minutes and `wspnrt` every 2 minutes, so a freshly started daemon shows "0 sent" for the first few minutes — that is normal.)

Two different counters are then shown, and it is easy to mistake one for the other. The **SPOTS UPLOADED PER DECODER** tiles count only the current run, so Stop/Start resets them to zero; the number beside each band/mode checkbox is that slot's **all-time** total, kept in `autorun-totals.json` so it

survives restarts. See the [Admin Panel guide](#) for both, and for the **Graphs** page that plots CPU frequency, load, temperature and users online over the last 15 minutes to 24 hours.

If your machine runs hot, the panel also carries a [Thermal Guard](#) that stops the server when the CPU reaches a dangerous temperature and restarts it once cool. It derives its thresholds from your own CPU's critical trip point, so there is nothing to calculate, and it works whatever start/stop method you use. It is installed with the panel but starts in log-only mode: it records what it *would* have done and changes nothing until you enable it in Settings. Continuous decoding keeps a CPU busy around the clock, so it is worth reading the CRASH tab after a week of autorun to see how close your machine actually gets.

Server port is auto-detected. The daemon taps spectrumserver's own `[server]` port directly (loopback + token, bypassing the proxy). It reads that port from the config file the running spectrumserver was launched with, so it works on any port with no configuration — the `[autorun]` tap `backend: ...` line in `autorun.log` shows what it resolved. If your server wasn't running when the daemon started, or you use an unusual setup, pin it in `autorun.json`: `json "server": { "host": "127.0.0.1", "port": 9002 }` A wrong port shows up as `504 / tap closed 1006` in `autorun.log` with `decodes` stuck at 0.

KiwiSDR Client Emulation (optional)

Since v4.1.0 PhantomSDR-Plus can also answer the **KiwiSDR protocol**, so software written for a KiwiSDR — **AetherSDR**, `kiwiclient` and the rest — connects to your receiver directly, on the same host and port you already publish. It is off until `[kiwi_emulation] enabled = true` is added to the config your receiver runs with.

The installer offers it as step 17, or `./kiwi_install.sh` applies it to a tree that is already installed.

Full documentation: [KiwiSDR Client Emulation](#) — what the bridge does, how to install it, every `[kiwi_emulation]` key, connecting a client, audio level, the S-meter, the waterfall rate, and a symptom table.

Configuration

1. Choose Your Configuration File

Select the appropriate config file for your SDR:

- `config-rtl.toml` - RTL-SDR dongles
- `config-rspla.toml` - SDRplay RSP1A
- `config-airspyhf.toml` - Airspy HF+ Discovery
- `config-fobos-hf.toml` - RigExpert Fobos SDR, HF1/HF2 direct sampling (0-25 MHz)
- `config-fobos.toml` - RigExpert Fobos SDR, RF path (25-6000 MHz)
- `config-rx888mk2.toml` - RX888 MK2
- `config-hackrf.toml` - HackRF One
- `config.example.hackrf.toml` - HackRF One

For this example, we'll use RTL-SDR.

2. Edit Configuration File

```
nano config-rtl.toml
```

Key Settings to Configure

```
[server]
port = 9002                # Web interface port (or everything else)
html_root = "frontend/dist/" # Frontend location
otherusers = 1             # Show other users (1=yes, 0=no)
threads = 2                # Number of server threads

[websdr]
register_online = true      # Register on sdr-list.xyz (true/false)
name = "Your WebSDR Name"  # Display name
antenna = "Your Antenna Type" # e.g., "Vertical", "Loop", "Dipole"
grid_locator = "AB12cd"    # Your Maidenhead grid square
hostname = "your.domain.com" # Your domain or IP address

[input]
sps = 2048000              # Sample rate (adjust for your coverage)
fft_size = 131072          # FFT size (higher = better resolution)
brightness_offset = -10    # Waterfall brightness adjustment
frequency = 145000000       # Base frequency in Hz (145 MHz for 2m)
signal = "iq"              # "iq" for complex, "real" for real sampling
fft_threads = 2            # FFT processing threads
accelerator = "opencl"     # "none", "cuda", or "opencl"
audio_sps = 12000          # Audio sample rate (keep at 12000)
audio_compression = "opus" # "flac" or "opus"
smeter_offset = -2         # S-meter calibration
waterfall_size = 1024      # Waterfall FFT size
waterfall_compression = "zstd" # Waterfall compression

[input.driver]
name = "stdin"             # Input driver
format = "u8"              # Sample format for RTL-SDR

[input.defaults]
frequency = 145500000       # Default tuning frequency
modulation = "FM"          # Default modulation mode
```

Sample Rate Guidelines

Coverage	Sample Rate	FFT Size
2 MHz	2048000	131072
3.2 MHz	3200000	131072
10 MHz	10000000	1048576
30 MHz	30000000	2097152
60 MHz	60000000	4194304

3. Configure Site Information

```
nano frontend/site_information.json
```

Edit the following fields:

```
{
  "siteSysop": "YourCallsign",
  "siteSysopEmailAddress": "your@email.com",
  "siteGridSquare": "AB12cd",
  "siteCity": "Your City, Country",
  "siteInformation": "https://github.com/sv1btl/PhantomSDR-Plus",
  "siteHardware": "Computer specifications",
  "siteSoftware": "PhantomSDR-Plus v5.0.0",
  "siteReceiver": "Your SDR model",
  "siteAntenna": "Antenna description",
  "siteNote": "Additional information",
  "siteIP": "http://your.domain.com:9002",
  "siteSDRBaseFrequency": 0,
  "siteSDRBandwidth": 2048000,
  "siteRegion": 1,
  "siteChatEnabled": true
}
```

IARU Regions:

- **1:** Europe, Africa, Middle East, Northern Asia
- **2:** Americas (North, Central, South), Caribbean
- **3:** Asia-Pacific, Oceania

4. Customize Frequency Markers (Optional)

```
nano markers.json
```

Add your favorite frequencies, repeaters, and broadcast stations.

5. Edit Start Script

```
nano start-rtl.sh
```

The start script is a self-contained launcher + watchdog. Edit only the **RECEIVER CONFIGURATION** block near the top so the receiver arguments match your setup:

```
# == RECEIVER CONFIGURATION (the only receiver-specific part) ==
RX_LABEL="RTL-SDR"
RX_COMM="rtl_sdr"                # process name to monitor/kill
CONFIG="$PHANTOMDIR/config-rtl.toml" # your .toml
FIFO="$PHANTOMDIR/rtl.fifo"
RX_ARGS="{RX_ARGS:-f 145000000 -s 2048000 -}" # receiver args (edit these)
```

Parameters inside `RX_ARGS`:

- `-f 145000000`: Center frequency (145 MHz)
- `-s 2048000`: Sample rate (2.048 MSPS)

`CONFIG` points at your `.toml`. You do **not** need to edit anything else in the script — the start/restart/watchdog/logging logic is generic. You can also override the receiver args at launch without editing the file: `RX_ARGS="-f 145000000 -s 2048000 -" ./start-rtl.sh` (`RX888_ARGS` for `start-rx888mk2.sh`).

SDR Device-Specific Setup

RTL-SDR

Install RTL-SDR Tools

```
sudo apt install -y rtl-sdr
```

Test RTL-SDR

```
rtl_test
```

Press Ctrl+C to stop. You should see sample rate information.

Edit Configuration

```
nano config-rtl.toml
```

Common settings for RTL-SDR:

```
sps = 2048000
frequency = 145000000
signal = "iq"
format = "u8"
```

Receivers on SoapySDR (RSP1A, Fobos, Airspy HF+)

The SDRplay RSP1A, the RigExpert Fobos SDR and the Airspy HF+ all reach the server the same way: a SoapySDR driver for the device, and `rx_sdr` (from `rx_tools`) to stream its samples into `spectrumserver`. One script per receiver installs the whole chain:

Receiver	Script	Installer option	Launcher	Config
SDRplay RSP1A	<code>./setup-rspla.sh</code>	3	<code>start-rspla.sh</code>	<code>config-rspla.toml</code>
RigExpert Fobos SDR	<code>./setup-fobos.sh</code>	5	<code>start-fobos-hf.sh</code> (HF) / <code>start-fobos.sh</code> (RF)	<code>config-fobos-hf.toml</code> / <code>config-fobos.toml</code>
Airspy HF+	<code>./setup-airspyhf.sh</code>	6	<code>start-airspyhf.sh</code>	<code>config-airspyhf.toml</code>

Each script:

- installs `git`, `cmake`, a compiler, `libusb` and **SoapySDR** from your distribution's own packages — it recognises `apt`, `dnf`, `pacman` and `zypper`, so it works on Debian/Ubuntu, Fedora, Arch and openSUSE alike;
- builds the receiver's driver from source into `sdr_drivers/` (not tracked by `git`);
- builds **rx_tools** for `rx_sdr`, unless `rx_sdr` is already installed — one copy serves every receiver;
- installs a **udev rule** and adds you to the `plugdev` group, so the start script never needs `sudo` for the device. **Log out and back in once** for the group to take effect;
- finishes by checking that SoapySDR really lists the driver, and stops with an error if it does not.

Run them as your normal user — they call `sudo` themselves. They are idempotent, so re-running one updates the driver sources with `git pull` and rebuilds. The installer calls exactly these scripts; on a

station that is already installed, run the one for your receiver on its own. The shared parts live in `setup-sdr-common.sh`, which is not meant to be run by itself. Two optional overrides apply to all three: `SDR_USER=<name>` puts another account in `plugdev`, and `SDR_SKIP_DEPS=1` skips the package installation.

Check that the receiver is seen, with it plugged in:

```
SoapySDRUtil --find="driver=soapyMiri"    # RSP1A
SoapySDRUtil --find="driver=fobos"       # Fobos
SoapySDRUtil --find="driver=airspyhf"    # Airspy HF+
```

SDRplay RSP1A

`setup-rsp1a.sh` installs the **open-source driver**: `libmirisdr-5`, the reverse-engineered driver for the Mirics MSI2500/MSi001 chips inside the RSP1 and RSP1A, with its SoapySDR module `SoapyMiri` (`driver=soapyMiri`). It needs no background service and no root. The script also blacklists the kernel's `msi2500` and `msi001` drivers in `/etc/modprobe.d/blacklist-msi2500.conf`, because they claim the RSP1A as a V4L2 radio before `rx_sdr` can open it. If the device was plugged in while they were loaded, replug it once (or reboot).

SDRplay's own closed API (`driver=sdrplay`, with `SoapySDRPlay` and the `sdrplay` service) still works — the included file `instructions-for-rsp1a` describes that installation. `start-rsp1a.sh` chooses between the two by itself: if SoapySDR has the `sdrplay` driver it uses the API and restarts the `sdrplay` service first, otherwise it uses `libmirisdr-5`. A station set up on the API therefore keeps working unchanged. To force one, start it with `RX_DRIVER=miri` or `RX_DRIVER=sdrplay`. The chosen driver is written to `logwebsdr.txt` as `RSP1A driver: miri` or `RSP1A driver: sdrplay`.

RigExpert Fobos SDR

The Fobos streams **one path at a time**, so it has two launchers that share the watchdog lock like every `start-*.sh` — starting one stops the other:

- **start-fobos-hf.sh** — the HF1/HF2 inputs in direct sampling, 0-25 MHz, on port 9003. There is no local oscillator in this mode: the ADC digitises 0-25 MHz all the time (`-f 0` is intentional, and `rx_sdr`'s *Failed to set center freq* warning is expected). `rx_sdr` delivers CF32 at 50 Msps; the small converter `cf32_to_real` (built by `setup-fobos.sh` from `cf32_to_real.c`) keeps the I channel and scales it to `s16`, so `config-fobos-hf.toml` uses `signal="real"` and `format="s16"`.
- **start-fobos.sh** — the RF path, 25-6000 MHz, IQ, on port 9002. Choose the window before the first start: `-f` (centre) and `-s` (sample rate) in the `RX_ARGS` line must match `frequency=` and `sps=` in `config-fobos.toml`. The shipped values (97 MHz, 20 Msps) are only a starting point to check that it runs.

IMPORTANT

`config-fobos-hf.toml` ships with `accelerator="opencl"`: a 50 Msps stream through a 1M-bin FFT is as heavy as the RX888's 60 Msps, and on the CPU it drops samples and heats the machine. On a machine with no OpenCL device `spectrumserver` will not start with that setting — set `accelerator="none"` there.

`SoapyFobosSDR` needs the headers of both `libfobos` and `libfobos-sdr-agile` to build, so `setup-fobos.sh` builds both, even for the stock firmware. The Fobos support comes from a script another station wrote after bringing a Fobos up on PhantomSDR-Plus; the HF values in `config-fobos-hf.toml` are the ones that worked there.

Airspy HF+

`setup-airspyhf.sh` installs `libairspyhf` from the distribution where it is packaged (Debian and Ubuntu, Fedora, openSUSE) and builds it from source where it is not (Arch has it only in the AUR), then builds `SoapyAirspyHF` (`driver=airspyhf`). `start-airspyhf.sh` streams 912 ksps IQ in CS16, which `config-`

airspyhf.toml reads as `format="s16"`. The included file `instructions-for-airspy` describes the manual route.

RX888 MK2

Install RX888 Tools

```
# Install rx_tools
git clone https://github.com/rxseger/rx_tools.git
cd rx_tools
mkdir build && cd build
cmake ..
make -j4
sudo make install
sudo ldconfig

# Install RX888 firmware and support
# Follow manufacturer instructions
```

Run rx888_stream Without sudo (udev rules)

By default the RX-888's Cypress FX3 USB device is only accessible to root, so `rx888_stream` would need `sudo`. Run the bundled helper once to install udev rules and add yourself to the `plugdev` group:

```
./setup-rx888-udev.sh          # re-runs itself with sudo automatically
```

It writes `/etc/udev/rules.d/99-rx888.rules` for all three Cypress FX3 product ids (`04b4:00f1/00f3` bootloader and `04b4:8613` firmware-loaded), reloads udev, and creates a stable `/dev/rx888` symlink. **Log out and back in** (for the group change) and **replug the device** once, then `rx888_stream` — and the `start-rx888mk2.sh` launcher — run without `sudo`. It is idempotent, so it's safe to re-run. To set up a different account: `RX888_USER=<name> ./setup-rx888-udev.sh`.

Edit Configuration

```
nano config-rx888mk2.toml
```

Common settings for RX888:

```
sps = 60000000
frequency = 0
signal = "real"
format = "s16"
fft_size = 4194304
```

HackRF One

The HackRF needs nothing built: `hackrf_transfer`, from your distribution's `hackrf` package, streams the samples straight into `spectrumserver`. `./setup-hackrf.sh` (installer option 7) installs that package with `apt`, `dnf`, `pacman` or `zypper` and adds a udev rule plus the `plugdev` group, so `start-hackrf.sh` runs without `sudo` — log out and back in once afterwards.

```
./setup-hackrf.sh    # installs the hackrf package and the udev rule
hackrf_info          # with the HackRF plugged in: must list the board
```

`start-hackrf.sh` streams one window at a time. Choose it before the first start: `-f` (centre) and `-s` (sample rate) in its `RX_ARGS` line must match `frequency=` and `sps=` in `config-hackrf.toml`. `-l` and `-g` are

the LNA (0-40 dB, 8 dB steps) and VGA (0-62 dB, 2 dB steps) gains, and `-a 1` switches the RF amplifier on. The shipped values (98 MHz, 20 Msps) are only a starting point to check that it runs.

`hackrf_transfer` writes **signed** 8-bit IQ, so the config needs `format="s8"`. `config.example.hackrf.toml` said `u8` until September 2026; with `u8` every sample is off by 128 and the waterfall shows only noise — check an older copy of it.

Several receivers on one computer

One computer can run more than one of the receivers above at the same time — for example an RX-888 for HF and an RTL-SDR for 2 m — each as its own server, with a receiver picker on the page. Each extra receiver is started with `INSTANCE=<name> ./start-<radio>.sh` and set up in `instances/<name>/receivers.toml` lists them for the proxy and for `start-all.sh`. Everything is in [Several Receivers](#). The quickest way to add a receiver is `./add-receiver.sh`, which the installers offer at the end: it installs the driver (`setup-rtlsdr.sh` for an RTL-SDR, the matching `setup-*.sh` for the others) and writes the instance folder and `receivers.toml` for you.

Testing and Verification

1. Test Run

```
# For RTL-SDR
./start-rtl.sh
```

This starts the server **in the background** (it detaches and returns immediately) and begins logging to `logwebsdr.txt`.

2. Check for Errors

The script logs its progress to `logwebsdr.txt` — watch it live:

```
tail -f logwebsdr.txt
```

Look for:

- `Starting the initialization script of the PhantomSDR Server`
- `<receiver> running (PID ...)`
- `Server started normally`
- `Server ... FAILED ...` or `ERROR: receiver binary '...' not found` (fix the receiver args/ `.toml`, or install the receiver tool, then re-run the start script)

3. Access Web Interface

Open your browser to:

```
http://localhost:9002
```

(Replace port number with your configured port)

4. Verify Functionality

- Waterfall should be displaying
- Audio should play when clicking on signals

- S-meter should respond to signals
- User count should show "1"

5. Test from Another Device

From another computer on your network:

```
http://YOUR_SERVER_IP:9002
```

6. Check Resource Usage

```
# Monitor CPU usage
htop

# Monitor GPU usage (if OpenCL/CUDA enabled)
nvidia-smi # For NVIDIA
radeontop  # For AMD

# Monitor network usage
iftop
```

7. Stop the Server

The server runs in the background under a watchdog, so **Ctrl+C won't stop it** (and the watchdog would just restart it). Use the shared stop script, which works for any receiver — it stops the watchdog first, then the receiver and spectrumserver:

```
./stop-websdr.sh
```

Setting Up Autostart

Using systemd (Recommended)

1. Create Service File

```
sudo nano /etc/systemd/system/phantomsdr.service
```

Add the following content (adjust paths and user):

```
[Unit]
Description=PhantomSDR-Plus WebSDR Server
After=network.target

[Service]
Type=simple
User=youruser
WorkingDirectory=/home/youruser/PhantomSDR-Plus
# Run the watchdog in the FOREGROUND (note the --watchdog flag) so systemd can
# track it. Do NOT use the plain "./start-rtl.sh" here – that form detaches into
# the background and exits, which systemd would treat as the service stopping.
ExecStart=/home/youruser/PhantomSDR-Plus/start-rtl.sh --watchdog
ExecStop=/home/youruser/PhantomSDR-Plus/stop-websdr.sh
Restart=always
RestartSec=10

[Install]
WantedBy=multi-user.target
```

The `--watchdog` process already auto-restarts the receiver/spectrumserver on its own; `Restart=always` is just a backstop for the rare case the watchdog itself exits. Because systemd supervises the server here, you can use `systemctl start/stop/restart` and `journalctl -u phantomsdr -f` in place of running the scripts by hand.

2. Enable and Start Service

```
# Reload systemd
sudo systemctl daemon-reload

# Enable service to start on boot
sudo systemctl enable phantomsdr.service

# Start service now
sudo systemctl start phantomsdr.service

# Check status
sudo systemctl status phantomsdr.service
```

3. Manage Service

```
# Start
sudo systemctl start phantomsdr

# Stop
sudo systemctl stop phantomsdr

# Restart
sudo systemctl restart phantomsdr

# View logs
sudo journalctl -u phantomsdr -f
```

Using Screen (Alternative)

Usually unnecessary: `./start-rtl.sh` already detaches into the background (via `setsid`) and keeps running after you log out, with its own watchdog. Screen is only handy if you specifically want an interactive session to run the foreground `./start-rtl.sh --watchdog` form.

1. Install Screen

```
sudo apt install -y screen
```

2. Start in Screen Session

```
screen -S phantomsdr
./start-rtl.sh
```

Press Ctrl+A, then D to detach.

3. Reattach to Session

```
screen -r phantomsdr
```

CPU Thermal Protection

Full manual: [THERMAL_GUARD.md](#) — the four modes and what a sysop must do for each, the standalone systemd service, enabling the throttle stage without root, testing and troubleshooting.

PhantomSDR-Plus ships a guard that stops the server if the CPU reaches a dangerous temperature and starts it again once it has cooled. It derives its thresholds from the critical trip point your own CPU publishes, so there is nothing to calculate, and it works whatever start/stop method you use.

If you run the admin panel, you already have it — it runs inside the panel and is configured on its Settings page. See the [Admin Panel guide](#). The rest of this section is for installations **without** the panel.

1. Check what the guard sees on your machine

```
cd ~/PhantomSDR-Plus
python3 thermal_guard.py --once
```

```
sensor      : hwmon:coretemp
temperature: 58.0 C
crit        : 100.0 C
mode        : log
thresholds  : warn 88.0  throttle 92.0  stop 95.0  resume 75.0 (C)
cpufreq     : not writable – throttle stage disabled
```

Only real CPU-die sensors are used (`coretemp`, `k10temp`, `zenpower`, `cpu_thermal`); `acpitz` and unlabelled thermal zones are ignored on purpose, because they often report a case or board temperature tens of degrees below the CPU. If this prints `sensor : NONE`, that machine cannot be protected — common on a VPS or inside a container — and the guard will stay inactive rather than pretend.

Nothing needs to be installed for this: `thermal_guard.py` uses only the Python standard library.

2. Configure it

The guard reads `admin_config.json` next to `thermal_guard.py`. Create it if you do not have one — with no panel installed you will not. The minimum useful file:

```
{
  "sdr_process_name": "spectrumserver",
  "stop_script": "stop-websdr.sh",
  "start_script": "start-rx88mk2.sh",
  "thermal_mode": "stop+restart"
}
```

- `thermal_mode` — `log` (record only, the default), `throttle`, `stop`, or `stop+restart`. **This is the one you must set**, because the guard ships inert: left alone it only writes what it *would* have done. You can also pass it on the command line as `--mode stop+restart`, which overrides the file — so if you are happy with every other default you need no config file at all.
- `stop_script` / `start_script` — how the guard stops and starts your server. Without a stop script it falls back to `SIGTERM`, then `SIGKILL` after 10 seconds, on `sdr_process_name`. Without a start script it stops but never restarts.
- Thresholds and timings can be set here too, with the same key names the panel uses — the full table is in the [Admin Panel guide](#).

If systemd supervises your SDR server, point `stop_script` at a small wrapper that runs `systemctl stop your-unit` rather than letting the guard signal the process directly. The guard will win either way — while the CPU is too hot it re-issues the stop every 2 seconds, so anything that revives the server is undone — but a clean stop beats a fight every two seconds.

3. Run it as a service

The repository includes a ready-made unit, `thermal-guard.service`. Edit `User=` and the two paths in it, then:

```
sudo cp thermal-guard.service /etc/systemd/system/
sudo systemctl daemon-reload
sudo systemctl enable --now thermal-guard
systemctl status thermal-guard
```

`User=` must be an account allowed to run your start/stop scripts — normally the same user that runs the SDR server. **A guard running as a user that cannot stop the server leaves you unprotected while looking protected.**

To arm the guard from the unit instead of the config file, add `--mode` to `ExecStart`:

```
ExecStart=/usr/bin/python3 -u /home/sysop/PhantomSDR-Plus/thermal_guard.py --mode stop+restart
```

To run it without systemd:

```
cd ~/PhantomSDR-Plus
nohup python3 -u thermal_guard.py --mode stop+restart >> thermal.log 2>&1 &
```

`python3 thermal_guard.py --help` lists every option, including `--config` for a config file somewhere other than beside the script.

4. Watch it, then trust it

Everything the guard does is written to `crash.log` in the PhantomSDR-Plus directory, one event per line:

```
[THERMAL] warn: 89.0C (warn=88.0 crit=100.0) sustained 31s
[THERMAL] STOPPED server: 96.0C sustained 62s (stop=95.0 crit=100.0) – Started stop-websdr.sh
[THERMAL] lockout: process reappeared at 94.0C – re-stopped [3 time(s) so far]
[THERMAL] recovered: 74.0C held below 75.0 – lockout cleared
```

```
grep THERMAL crash.log
```

Leave `thermal_mode` at `"log"` for a week first and read that file after your heavy moments — a full rebuild, a hot afternoon. If nothing appears, your machine never came close. Then set the mode to `stop`. Before relying on it, prove the whole path once by adding a fake reading above your stop threshold:

```
"thermal_test_temp": 97
```

The guard treats it as real, so `warn` → `stop` → `lockout` → `recover` → `restart` all run on demand. Remove the line (or set it to `null`) to return to the real sensor. With the mode at `stop` this really does stop the server and disconnect your listeners, so do it when nobody is on — or run the test at `log`, where it shows what *would* have happened without touching anything.

Troubleshooting

Build Errors

Dependency Issues

```
# Ubuntu: Install missing dependencies
sudo apt install --fix-broken

# Fedora: Install missing dependencies
sudo dnf install <package-name>
```

Meson Configuration Fails

```
# Clean and reconfigure
rm -rf build
meson setup build --buildtype=release
```

meson setup dies with `ModuleNotFoundError: No module named 'mesonbuild'`

```
Traceback (most recent call last):
  File "/home/<user>/local/bin/meson", line 3, in <module>
    from mesonbuild.mesonmain import main
ModuleNotFoundError: No module named 'mesonbuild'
```

Nothing is wrong with the source tree. A leftover `pip install --user meson` left a launcher script in `~/.local/bin`, which comes before `/usr/bin` on `PATH` and therefore hides the working copy the package manager installed. A distribution upgrade (Ubuntu 24.04 → 26.04, for example) moves Python to a new version, the old `site-packages` that held `mesonbuild` is no longer on the import path, and the script dies before it does any work. The same thing can happen to `ninja`.

The installer detects this and works around it for the duration of the run, warning you in the process, but repair the system:

```
rm -f ~/.local/bin/meson
hash -r
meson --version          # must print a version
```

Keep a pip-installed meson instead if you prefer, by reinstalling it for the Python this system has now:

```
python3 -m pip install --user --force-reinstall --break-system-packages meson
```

meson setup stops with Clock skew detected

```
ERROR: Clock skew detected. File /home/pi/PhantomSDR-Plus/build/../meson.build has a time stamp
10392.2144s in the future.
```

Nothing is wrong with the source tree: the system clock is behind the files. meson and ninja refuse to build when an input is newer than the current time, because they cannot tell which outputs are out of date. It happens on a Raspberry Pi with no cell in its RTC holder — every boot starts from the last time it knew, so a build launched before NTP catches up sees the whole tree dated in the future — and on any tree unpacked or copied from a machine whose clock is ahead.

Correct the clock first:

```
timedatectl                      # is the time right? is NTP synchronised?
sudo timedatectl set-ntp true
```

Wait a few seconds for it to settle, then run the installer again. It checks for this before calling meson and offers to reset the offending timestamps (`PHANTOM_FIX_CLOCK_SKEW=y|n`). By hand, from the source tree, that is:

```
find . -path ./build -prune -o -path ./git -prune -o -newermt now -print0 | xargs -0r touch
```

The backend built but there is no web page

The symptom is an installation that looks like it mostly worked: `build/spectrumserver` exists, the server starts, and the browser gets nothing — because `frontend/dist/` was never produced. Look back through the installer output for:

```
[vite-plugin-top-level-await] missing field `type`
  at Compiler.printSync (node_modules/@swc/core/index.js:257:29)
error during build:
  Failed: Default
```

`vite-plugin-top-level-await` asks for `@swc/core ^1.12.14`, so a fresh `npm install` resolves 1.16.0, whose `printSync()` rejects the syntax tree the plugin gives it. It has nothing to do with your distribution — every installer now pins the working version, and so does `frontend/package.json`. A machine installed before that pin was added keeps working until its `node_modules` is deleted, which is why this appears out of nowhere on a reinstall.

If you are repairing an older checkout by hand:

```
cd frontend
npm pkg set 'overrides.@swc/core=1.15.33'
npm install
./build-all.sh
```

Runtime Errors

Port Already in Use

```
# Find process using port
sudo lsof -i :9002

# Kill process
sudo kill -9 <PID>
```

Permission Denied for SDR

```
# Add user to plugdev group
sudo usermod -a -G plugdev $USER

# Reload groups (or log out and back in)
newgrp plugdev
```

For an **RX-888 MkII** this isn't enough on its own — the Cypress FX3 device also needs udev rules. Run the bundled helper (installs the rules *and* adds you to `plugdev`), then log out/in and replug the device:

```
./setup-rx888-udev.sh
```

Audio Issues

```
# Try different codec
# Edit config file and change:
audio_compression = "flac" # or "opus"

# Rebuild
cd frontend && npm run build && cd ..
```

OpenCL Issues

clinfo Shows No Devices

```
# Verify drivers are loaded
sudo clinfo

# Check OpenCL ICD files
ls /etc/OpenCL/vendors/

# Reinstall drivers (Intel example)
sudo apt remove --purge intel-opencl-icd
sudo apt install intel-opencl-icd
```

Performance Not Improved

```
# Verify OpenCL is enabled in config
accelerator = "opencl"

# Try different device
# Add to config file:
[input.opencl]
device_id = 0 # Try 0, 1, 2, etc.
```

Network Issues

Can't Access from Other Devices

```
# Check firewall
sudo ufw status
sudo ufw allow 9002/tcp

# Or disable firewall temporarily for testing
sudo ufw disable
```

High Latency

```
# Reduce waterfall size in config
waterfall_size = 512

# Increase buffer size
buffer_size = 16384

# Use Opus instead of FLAC
audio_compression = "opus"
```

SDR Device Issues

RTL-SDR Not Found

```
# Check if device is recognized
lsusb | grep Realtek

# Test with rtl_test
rtl_test

# Try blacklisting DVB-T drivers
echo "blacklist dvb_usb_rtl28xxu" | sudo tee /etc/modprobe.d/blacklist-rtl.conf
sudo rmmod dvb_usb_rtl28xxu
```

SDRplay Not Found

`grep "RSP1A driver" logwebsdr.txt` shows which driver `start-rsp1a.sh` picked.

With the open driver (libmirisdr-5, RSP1A driver: miri):

```
# The kernel's msi2500 driver must not hold the device
lsmod | grep msi2500 && sudo modprobe -r msi2500 msi001

# Test with SoapySDR
SoapySDRUtil --find="driver=soapyMiri"
```

With SDRplay's API (RSP1A driver: sdrplay):

```
# Check API installation
systemctl status sdrplay

# Restart API
sudo systemctl restart sdrplay

# Test with SoapySDR
SoapySDRUtil --find="driver=sdrplay"
```

Fobos or Airspy HF+ Not Found

```
SoapySDRUtil --info | grep -i factories    # must list fobos / airspyhf
SoapySDRUtil --find="driver=fobos"        # or driver=airspyhf
id | grep plugdev                         # log out and back in if missing
```

If the driver is missing from the list, re-run `./setup-fobos.sh` or `./setup-airspyhf.sh`: it stops with the reason if the build or the SoapySDR check fails.

Performance Optimization

CPU Optimization

```
# Adjust thread counts based on CPU cores
[server]
threads = 4 # Set to number of cores

[input]
fft_threads = 4 # Set to number of cores
```

Memory Optimization

```
# Reduce memory usage
[input]
waterfall_size = 512 # Lower value = less memory
fft_size = 65536     # Lower value = less memory
```

Network Optimization

```
# Optimize for limited bandwidth
[input]
audio_compression = "opus" # More efficient than FLAC
waterfall_compression = "zstd" # Efficient compression
```

Updating PhantomSDR-Plus

Since version 4.1.0 the repository ships **update.sh**, an updater that brings an installed receiver up to date with the published tree **without touching the files that make it your site**. It replaces the hand-written `git pull` script that earlier editions of this guide asked you to create, and it does not need git at all: the published tree is downloaded as a tarball and compared with yours file by file, so it works the same whether you cloned the repository, unpacked an `update.zip`, or copied the tree off a USB stick.

If your installation does not have update.sh yet

An older tree will not contain the script. Fetch it once — it is the only step of this whole procedure you ever do by hand:

```
cd ~/PhantomSDR-Plus
curl -fLO https://raw.githubusercontent.com/sv1btl/PhantomSDR-Plus/main/update.sh
chmod +x update.sh
```

From then on everything — sources, frontend, documentation, installers, and `update.sh` itself — is brought in by the tool.

Step 1 — see what would change (this writes nothing)

```
cd ~/PhantomSDR-Plus
./update.sh
```

It downloads the published tree, compares it with yours and prints a report. It writes nothing at all, so it is safe to run at any time, including on a receiver that is on the air. The exit status is `0` when you are already up to date and `10` when an update is waiting, so a cron job can tell you when there is something to do.

Step 2 — apply it

```
./update.sh --apply
```

Three kinds of file are treated differently, and that difference is the whole point:

Files	What happens
<code>config.toml</code> , <code>config-<receiver>.toml</code> , <code>markers.json</code> , <code>admin_config.json</code> , <code>autorun.json</code> , <code>frequencylist/</code> , <code>chat_history.txt</code> , <code>frontend/variant.json</code> , <code>frontend/site_information.json</code> , the logs, <code>build/</code> , <code>frontend/dist/</code>	Never touched , and never even shown in a prompt. These are what make the machine <i>your</i> receiver.
<code>start-*.sh</code> , <code>stop-websdr.sh</code> , the <code>*.service</code> units, <code>install*.sh</code> , <code>recompile.sh</code> , <code>setup_admin.sh</code> , <code>smeter_theme.sh</code> , <code>proxy.py</code> , <code>admin_server.py</code> , <code>thermal_guard.py</code> , <code>websdr_relay.py</code> , <code>frontend/src/bands-config.js</code> , the sample configs <code>config.example.*.toml</code>	Always asked about , because these are the files a sysop has a reason to have edited.
Everything else	Updated, after a copy of the old file is saved in the backup archive (see <i>Undoing an update</i>).

For each file in the middle group you are shown the differences and given three choices:

```
start-rx888mk2.sh  [K]eep mine / [u]pstream / [b]oth  (ENTER = Keep mine)
```

- **Keep mine** — your file is left exactly as it is.
- **upstream** — the new version is installed, and your file is backed up first.
- **both** — the new version is written beside yours as `start-rx888mk2.sh.new`, so you can merge your own changes into it in your own time.

What a first run feels like. The first time you run it there is no record of which version your files came from, so every file in the middle group above is put to you: about ten questions. Answer them like this:

Your situation	Answer
You never edited that file	<code>u</code> — take the new version. This is the usual case.
You edited it (your own <code>RX888_ARGS</code> , CPU pinning, a tweaked unit)	<code>b</code> — yours is kept, and the new one lands beside it as <code><file>.new</code> to merge later.
You are not sure	<code>ENTER</code> — yours is kept, nothing is lost, and you can compare afterwards.

Your configuration is never part of this: the questions are only ever about scripts and service units.

`update.sh` records the version of every file it installs in `.update-state/`. From the second run on it can therefore tell a file **you** edited from a file that is merely old, and it only stops to ask about the ones you actually changed.

Before writing anything it stops the receiver, the admin panel, the reverse proxy and the WebSDR diversity relay **of the installation it is updating** — a component serving another directory is listed and left running, so a second clone can be updated while the first one stays on the air — and when it has finished it starts back exactly what it stopped. If source or frontend files changed, it offers to run `recompile.sh` for you. Nothing is ever deleted: files that have gone from the repository are reported, and removed only if you ask with `--prune`.

Undoing an update

```
./update.sh --restore LAST
```

Every file an update overwrites is saved first, in one dated archive per run in the plain, visible folder `update-backups/`:

```
update-backups/phantomsdr-backup-20260923-164530.zip
```

The archive holds only the files that run replaced, each at the path it has in your tree, so it can be copied to another machine as a single file, opened anywhere, or unpacked straight back over the installation. Where `zip` is not installed it is written as `.tar.gz` instead. The three most recent are kept. Either of these puts back what an update replaced:

```
./update.sh --restore LAST
./update.sh --restore phantomsdr-backup-20260923-164530.zip
```

Each archive also carries its own `restore.sh`, for the case where `update.sh` itself is the problem: unpack the archive and give it your installation directory, for example `./restore.sh /home/pi/PhantomSDR-Plus`. While restoring, `update.sh` runs from a copy outside the tree, because it is usually one of the files being put back.

Backups made before 23 September 2026 sit in the hidden folder `.update-backups/<timestamp>/`; they can still be restored the same way and count towards the three that are kept.

Other options

```
./update.sh --apply --yes      # never asks; every file you edited is KEPT
./update.sh --ref v5.0.0      # a tag, branch or commit instead of the current tree
./update.sh --list-excludes    # print the never-touch rules as they resolve here
./update.sh --verbose          # list every file, not only the first 40
```

You can add your own never-touch rules by putting one glob per line in `update-exclude.txt` in the root folder of the installation.

If the rebuild fails on a very old installation

`update.sh` updates files, not system packages. If your tree is old enough that the build now needs libraries you do not have, `recompile.sh` will stop with a compiler or meson error. That is not a broken update — you need the dependencies:

```
./install.sh
```

The installer is itself brought up to date by the same run, and your configuration survives it too.

Updating by hand

If you would rather apply a file drop yourself:

```
cd ~
unzip -o update.zip
cd PhantomSDR-Plus
./stop-websdr.sh
cp -a ~/update/. .
chmod +x *.sh
./recompile.sh
./start-rx888mk2.sh      # or the start script for your receiver
```

Back up your configuration first — `config-*.toml`, `frontend/site_information.json`, `admin_config.json` and `markers.json` — because a file drop cannot tell your edits from the release's. That is exactly the problem `update.sh` exists to solve.

Backup and Restore

Files to Backup

- Configuration files: `*.toml`
- Site information: `frontend/site_information.json`
- Markers: `markers.json`
- Custom scripts: `start-*.sh`, `stop-*.sh`
- Chat history: `chat_history.txt`
- Background image: `frontend/src/assets/background.jpg`

Backup Command

```
cd ~/PhantomSDR-Plus
tar -czf phantomsdr-backup-$(date +%Y%m%d).tar.gz \
  *.toml \
  *.sh \
  markers.json \
  chat_history.txt \
  frontend/site_information.json \
  frontend/src/assets/background.jpg
```

Restore Command

```
tar -xzf phantomsdr-backup-YYYYMMDD.tar.gz
```

Security Considerations

Firewall Configuration

```
# Allow only necessary ports
sudo ufw allow 9002/tcp
sudo ufw enable
```

Reverse Proxy (Optional)

Consider using nginx or Apache as a reverse proxy for:

- SSL/TLS encryption
- Domain name mapping
- Load balancing
- Access control

User Limits

Edit `config.toml`:

```
[server]
max_users = 100 # Limit concurrent users
```

Getting Help

Resources

- **Documentation:** This guide, README.md, USER_GUIDE.md
- **GitHub Issues:** <https://github.com/sv1btl/PhantomSDR-Plus/issues>
- **Live Demo:** <http://phantomsdr.no-ip.org:8900/>

Reporting Issues

When reporting issues, include:

1. Operating system and version
2. SDR device model
3. Configuration file contents
4. Error messages
5. System resource usage (CPU, RAM, GPU)

Community Support

- Check existing GitHub issues before creating new ones
 - Provide detailed information about your setup
 - Include logs and error messages
 - Be patient and respectful
-

Appendix A: Complete Dependency List

Ubuntu 24.04 Package List

```
build-essential
cmake
pkg-config
meson
libfftw3-dev
libwebsocketpp-dev
libflac++-dev
zlibg-dev
libzstd-dev
libboost-all-dev
libopus-dev
libliquid-dev
git
util-linux (taskset – for the autorun spot reporter)
psmisc
wget
curl
rtl-sdr (for RTL-SDR)
airspy (for Airspy)
hackrf (for HackRF)
libclfft-dev (for OpenCL)
ocl-icd-opencl-dev (for OpenCL)
clinfo (for OpenCL)
```

Appendix B: Configuration Examples

Example 1: RTL-SDR for VHF/UHF

```
[input]
sps = 2048000
frequency = 145000000
signal = "iq"

[input.driver]
format = "u8"

[input.defaults]
frequency = 145000000
modulation = "FM"
```

Example 2: RX-888 mk2 for HF (0-30 MHz)

The receiver most sysops run. This is a complete `[input]` section rather than a fragment, with the values from the `config-rx888mk2.toml` that ships with the repository.

```
[input]
sps = 60000000          # 0-30 MHz by direct sampling
fft_size = 4194304      # see the note below
fft_threads = 8
brightness_offset = -9  # more negative if the waterfall shows black patches
frequency = 0           # baseband: the RX-888 samples from DC
signal = "real"         # not "iq" - direct sampling gives a real stream
accelerator = "opencl"  # "none" if there is no OpenCL runtime
audio_sps = 12000
audio_compression = "flac"
waterfall_size = 1024
waterfall_compression = "zstd"
smeter_offset = 5
analog_smeter_offset = 5

[input.driver]
name = "stdin"
format = "s16"

[input.defaults]
frequency = 7120000
modulation = "LSB"
```

Feed it from `rx888_stream`, which `start-rx888mk2.sh` does for you:

```
rx888_stream -f /path/to/SDDC_FX3.img -s 60000000 -g 50 -a 0 -m low --pga -d -r -o - \
| build/spectrumserver --config config-rx888mk2.toml
```

On `fft_size`: 4194304 is the size to use at 60 MSPS. 8388608 doubles the waterfall resolution and doubles the memory and CPU cost of every transform with it, which on most machines buys sharper bins at the price of dropped frames. Start at 4194304 and only go higher if the server is comfortably idle.

Example 3: HackRF for Wideband FM

```
[input]
sps = 10000000
frequency = 100900000
signal = "iq"

[input.driver]
format = "s8"

[input.defaults]
frequency = 100900000
modulation = "WBFM"
```

Installation complete! You should now have a fully functional PhantomSDR-Plus server.

73 de SV1BTL & SV2AMK

CHAPTER 3

Admin Panel Setup

ADMIN_PANEL_SETUP.md

PhantomSDR-Plus Admin Panel — Setup Guide

⚠ **Security notice:** Keep this admin panel on your home network or behind a VPN. Public exposure of the admin port is not recommended without proper authentication hardening.

Overview

The admin panel consists of two Python services:

Service	File	Role
Admin panel	<code>admin_server.py</code>	Flask web app, binds to 127.0.0.1 (internal only)
Reverse proxy	<code>proxy.py</code>	Exposes both the SDR and the admin panel on a single public port

Both services read their configuration from `admin_config.json`, which is written by `setup_admin.sh`. **Do not edit port constants directly in the Python files** — all ports and the SDR host IP are stored in `admin_config.json`.

What the admin panel gives you

- **Dashboard** — live server status, CPU/RAM, top processes, recent log output, terminal
- **Config Editor** — view, edit and save any `.toml`, `.sh`, `.json`, `.h`, `.cpp` file in your installation
- **Log Viewer** — tail any log file in real time (SDR server, admin panel, RADE, crash, autorun, proxy)
- **Markers** — view and edit frequency markers
- **Chat History** — view the WebSDR chat log, clear it entirely, or delete individual messages without restarting the server
- **Waterfall Message** — broadcast a persistent banner message to all connected users, visible on the waterfall display in real time
- **Users** — live list of connected listeners with their tuned frequency, mode, duration and a `Kick` button
- **Chat message deletion** — Delete button that removes that single message from the log immediately
- **Waterfall broadcast messages** — lets you push a persistent text banner to every connected user's waterfall
- **Spot Reporting** — start/stop the autorun FT8/FT4/JS8/WSPR decoder, pick bands/modes, and report spots to PSK Reporter / wsprnet (OFF by default), with two counters: per-decoder tiles for the current run and an all-time total beside each band/mode checkbox
- **Graphs** — CPU frequency, CPU load, CPU temperature and connected users plotted over the last 15 minutes / 1 hour / 4 hours / 12 hours / 24 hours

- **Thermal Guard** — stops the server if the CPU overheats and restarts it once cool, with thresholds derived from your own CPU; works with any start/stop method, and ships in log-only mode so it does nothing until you enable it
- **Settings** — change admin password, SDR base directory, process name, public port

Requirements

- PhantomSDR-Plus already installed and running
- Python 3.8+

NOTE

The installer normally does this for you. `./install.sh` — and each of the four distro installers, both RADE installers and the stats-server installer — sets the admin panel up by default, installing `pip` first with the system's own package manager if it is missing. This page is what happens during that step, and what to do if you skipped it or want to change something afterwards.

The panel's Python libraries (`flask`, `psutil`, `aiohttp`, `tomli-w`) are installed by `setup_admin.sh`; it prints the exact command to run by hand if that ever fails.

Step 1 — Files

All four files ship with the repository, so there is nothing to copy:

```
admin_server.py
manage_admin.sh
setup_admin.sh
proxy.py
```

`chmod +x *.sh` after cloning (which the installation instructions already tell you to do) makes the scripts executable.

Step 2 — Run the setup script

```
./setup_admin.sh
```

The script will:

1. Check that Python 3 is installed
2. Verify `admin_server.py` and `manage_admin.sh` are present
3. Record `127.0.0.1` as `sdr_host` in config (see [The sdr_host setting](#))
4. Ask for three port numbers. Each offers a default in brackets that a bare Enter accepts, so a normal setup is three keystrokes:
5. **Spectrumserver port** — the port your SDR server listens on (default `8900`, which is what every `config-*.toml` in the repository ships with)
6. **Admin panel internal port** — where `admin_server.py` binds locally (default `3000`)
7. **Proxy public port** — the single external port that combines SDR + admin (default `8902`)

Invalid answers are rejected and re-asked, but only five times — after that the default is used. A run whose input is not a terminal (piped, cron, unattended) takes the defaults straight away instead of waiting for input that never comes.

1. Install `flask`, `psutil`, `aiohttp` and `tomli-w` via `pip`
2. Grant `ss` the `cap_net_admin` capability (fallback path of Kick Users, used only when the panel runs without the proxy)
3. Ask which script starts and which stops the receiver — the panel drives the server through these, and so does the thermal guard
4. Ask how far the CPU over-temperature guard may act on its own — `log`, `throttle`, `stop` or `stop+restart` — and offer to run `setup-cpufreq-perms.sh` so the throttle stage works without root (see [THERMAL_GUARD.md](#))
5. Write `admin_config.json` with all settings
6. Offer to install two systemd units (panel + proxy) — auto-start on boot and restart after a crash.
Recommended, and the default (a bare Enter accepts it): the CPU over-temperature guard only runs while the panel runs, so without the units a reboot leaves the machine unprotected. The step is skipped automatically where systemd is not PID 1 (containers, WSL1, OpenRC), or where you have no `sudo`.

After setup, open your browser via the proxy:

```
http://YOUR_SERVER_IP:<proxy_port>/admin
```

Default password: **admin**

⚠ **Change the password immediately** — go to Settings on first login. A first-run wizard will guide you through setting the SDR directory, process name, public port and a new password.

Step 3 — Start / Stop / Restart

There are two ways to run the panel and the proxy. **Pick one and stay with it** — the warning at the end of this section explains what happens if you mix them.

Method B (systemd) is strongly recommended wherever systemd is available. The CPU over-temperature guard runs inside the panel, so a panel that dies at 03:00 or a machine that reboots leaves the CPU unprotected until someone starts it again by hand. `setup_admin.sh` now offers the units by default. Use Method A when there is no systemd — containers, WSL1, OpenRC/sysvinit — or when you deliberately want to supervise things yourself.

Method A — `manage_admin.sh` (no root, nothing installed)

```
./manage_admin.sh start      # start admin panel and proxy
./manage_admin.sh stop       # stop both
./manage_admin.sh status     # show running status and PIDs
```

There is deliberately **no restart**. It used to kill the panel and immediately start its own copy, which collides with systemd on any machine using Method B. To restart under Method A, run `stop` then `start` — see [Restarting the panel](#) below.

One script controls **both** processes. It checks that `aiohttp` is installed before starting the proxy and prints a clear error with the fix command if it is missing.

Nothing starts by itself: after a reboot, or if the panel crashes, you start it again by hand. That is fine for a receiver you watch, but note that the CPU over-temperature guard runs inside the panel — while the panel is down, nothing is protecting the machine (see [THERMAL_GUARD.md](#)).

Method B — systemd units (starts at boot, restarts after a crash)

The repository ships two ready units, `phantomsdr-admin.service` and `phantomsdr-proxy.service`. Edit `User=` and the paths in both, then:

```
sudo cp phantomsdr-admin.service phantomsdr-proxy.service /etc/systemd/system/
sudo systemctl daemon-reload
./manage_admin.sh stop          # free the ports first
sudo systemctl enable --now phantomsdr-admin
sudo systemctl enable --now phantomsdr-proxy
```

Day-to-day commands then become:

```
sudo systemctl start    phantomsdr-admin
sudo systemctl stop     phantomsdr-admin
sudo systemctl restart  phantomsdr-admin
systemctl status        phantomsdr-admin # no sudo needed
sudo journalctl -u phantomsdr-admin -f   # live logs
```

The same four with `phantomsdr-proxy`, or both at once: `sudo systemctl restart phantomsdr-admin phantomsdr-proxy`.

`admin.log` and `proxy.log` keep working exactly as before — the units append to those same files.

One thing to do once: install the `tmpfiles` rule at `tmpfiles/phantomsdr-logs.conf` (edit its paths and user first). `systemd` creates those two logs as `root` before it drops to `User=`, and the panel's **Clear Logs** button then fails on them with `Permission denied` — see “[Clear Logs](#)” says [Permission denied](#) below. `setup_admin.sh` installs the rule for you.

The admin unit carries `SupplementaryGroups=cpufreq`, which is what lets the thermal guard's throttle stage lower the CPU clock without the panel running as `root`. Create that group first with `./setup-cpufreq-perms.sh`, or `systemd` will refuse to start the unit; delete the line if you do not use the throttle stage.

To go back to Method A:

```
sudo systemctl disable --now phantomsdr-admin
sudo systemctl disable --now phantomsdr-proxy
```

⚠**Do not mix the two.** With the units enabled, `./manage_admin.sh stop` is undone by `systemd` five seconds later, and `./manage_admin.sh start` gives you a second, unmanaged copy fighting the `systemd` one over the same port. (`restart` was removed from the script for exactly this reason; it now prints an error telling you to use `systemctl`.) `./manage_admin.sh status` only reads, so it stays useful either way.

Restarting the panel

The rest of this document says “restart the panel” in a number of places. It means whichever of these matches your method:

```
# Method B – systemd units (the normal setup)
sudo systemctl restart phantomsdr-admin phantomsdr-proxy

# Method A – no units installed
./manage_admin.sh stop && ./manage_admin.sh start
```

The unit ships with `KillMode=process`, and that line is load-bearing. If you start the SDR from the panel, the receiver is a child of the admin unit and inherits its cgroup. Under systemd's default `KillMode=control-group`, restarting the panel would take spectrumserver, its watchdog and the autorun daemon down with it, and stall for the full 90-second stop timeout first. With `KillMode=process` systemd signals only the panel, so `sudo systemctl restart phantomsdr-admin` leaves a busy receiver on air.

Upgrading a unit installed before v4.1.0. Older unit files have no such line, and restarting the panel will not add one: `systemctl restart` re-runs the program, it does not change the program's configuration. Edit the *installed* copy — the one in the repository is only a template systemd never reads:

```
sudo nano /etc/systemd/system/phantomsdr-admin.service
```

Add `KillMode=process` in the `[Service]` section, next to `Restart=always`. Then make systemd re-read the file and confirm:

```
sudo systemctl daemon-reload
systemctl show phantomsdr-admin -p KillMode
```

The second command must print `KillMode=process`. If it still prints `control-group`, systemd is running its cached copy — `systemctl show phantomsdr-admin -p NeedDaemonReload` prints `yes` while a reload is pending. Leading whitespace in unit files is ignored, so indentation does not matter.

Step 4 — Open firewall port

Open only the **proxy public port** in your firewall. There is no need to expose the admin panel internal port externally:

```
sudo ufw allow <proxy_port>
```

Step 5 — Kick Users feature

The Users page shows every currently connected listener. Each row displays the user's IP, tuned frequency, mode and how long they have been connected. To disconnect one user, click the **Kick** button on their row — only that TCP connection is dropped. All other listeners stay connected and hear nothing.

The kick is instant and surgical: the server does not restart, no audio is interrupted for anyone else, and the kicked user can reconnect immediately (the feature is for removing misbehaving or stuck connections, not for banning).

Under the hood the kick uses `ss -K dst <IP> dport <port>` to tear down the specific TCP socket. This requires a Linux capability that the setup script grants automatically. If you skipped setup or the button reports an error, apply it manually:

```
sudo setcap cap_net_admin+ep $(which ss)
# Verify:
getcap $(which ss)    # should show: cap_net_admin=ep
```

Configuration — admin_config.json

All runtime configuration lives in `admin_config.json` in your PhantomSDR-Plus directory. `setup_admin.sh` writes the initial file; subsequent changes can be made via the Settings page or by editing the file directly.

Key fields written by `setup_admin.sh`:

Key	Description
<code>port</code>	Admin panel internal port (<code>admin_server.py</code> binds here)
<code>public_port</code>	Spectrumserver port (used by the Users page to query <code>/users</code>)
<code>proxy_port</code>	Public port the proxy listens on
<code>sdr_host</code>	Host the proxy uses to reach spectrumserver — <code>127.0.0.1</code> (see below)
<code>password_hash</code>	SHA-256 hash of the admin password
<code>sdr_base_dir</code>	Path to your PhantomSDR-Plus installation
<code>sdr_process_name</code>	Process name to monitor (default: <code>spectrumserver</code>)
<code>start_script</code> / <code>stop_script</code>	Scripts the panel runs to start and stop the SDR server — also what the thermal guard acts through

Several receivers on one computer? Set `start_script` to `start-all.sh`. The stop script already stops every receiver, and Restart and the thermal guard use the start script, so with `start-all.sh` they bring back every receiver in `receivers.toml` instead of only the main one. The Users page, the user count and the kick then cover every receiver as well. See [Several Receivers](#).

Thermal guard keys (all optional — the panel writes them when you save the Thermal Guard settings, and any that are missing fall back to these defaults):

Key	Default	Description
thermal_enabled	true	Master switch for the guard
thermal_mode	"log"	log (record only, never act) · throttle · stop · stop+restart
thermal_warn	null	Warn temperature °C; null = derive from your CPU's critical trip point
thermal_throttle	null	Temperature at which the CPU clock ceiling is lowered; null = auto
thermal_stop	null	Temperature at which the server is stopped; null = auto
thermal_resume	null	Temperature the CPU must fall below to recover; null = auto
thermal_sustain_s	60	Seconds the stop temperature must be held before acting
thermal_warn_sustain_s	30	Seconds the warn/throttle temperatures must be held
thermal_resume_s	300	Seconds below the resume temperature before the lockout is released
thermal_max_stops_hour	2	Thermal stops per hour before the automatic restart is disabled
thermal_test_temp	null	Inject a fake temperature to test the guard; null = use the real sensor

Changes to these take effect on the next sample — within a couple of seconds, with no restart.

To change ports after initial setup, edit `admin_config.json` and restart:

```
sudo systemctl restart phantomsdr-admin phantomsdr-proxy # Method B
./manage_admin.sh stop && ./manage_admin.sh start # Method A
```

The `sdr_host` setting

`proxy.py` connects to `spectrumserver` over loopback — `sdr_host` is `127.0.0.1` in `admin_config.json`, written there by `setup_admin.sh`. You should not need to change it.

Set it to a real address only if `spectrumserver` runs on a **different machine** than the proxy. Then [restart the panel](#).

Changed: earlier versions required `sdr_host` to be the machine's LAN IP, because `spectrumserver` closed WebSockets arriving from the loopback address. That filter has been removed, so local connections work normally. Two consequences: browsing `http://localhost:<port>` on the server machine itself now shows the full GUI (it used to load the page but stay blank), and the proxy no longer breaks when DHCP reassigns the machine's IP. If you are upgrading and your `admin_config.json` still holds a LAN IP, change it to `127.0.0.1` or re-run `setup_admin.sh`.

proxy.py — what it is and when you need it

`proxy.py` puts both the SDR server and the admin panel on a **single public port**, routing by path prefix:

Path	Routed to
<code>/admin*</code>	Admin panel (<code>localhost:<port></code>)
Everything else	Spectrumserver (<code><sdr_host>:<public_port></code>)

Without the proxy you must expose two ports separately. With the proxy you expose only one.

The proxy also removes the 4 MB WebSocket message-size cap (important for large FFT frames from the RX-888 at 60 MSPS), forwards real client IPs via X-Forwarded-* headers, and keeps long-lived connections alive through NAT with a 30-second heartbeat.

Changing ports after setup

Edit `admin_config.json` directly:

```
{
  "port":      3000,
  "public_port": 9001,
  "proxy_port": 9002,
  "sdr_host":   "127.0.0.1"
}
```

Then restart both services:

```
sudo systemctl restart phantomsdr-admin phantomsdr-proxy # Method B
./manage_admin.sh stop && ./manage_admin.sh start        # Method A
```

If you also need `admin_server.py` to bind on a different port at launch (e.g. the systemd service), you can override with the environment variable:

```
ADMIN_PORT=3000 python3 admin_server.py
```

By default `admin_server.py` binds to `127.0.0.1` only. To run it standalone without the proxy (development/testing), bind to all interfaces:

```
ADMIN_BIND=0.0.0.0 python3 admin_server.py
```

Useful manual commands

```
# Start admin panel manually (foreground)
python3 ~/PhantomSDR-Plus/admin_server.py

# Kill the admin panel
pkill -f admin_server.py

# Kill the proxy
pkill -f proxy.py

# Free a port that is stuck in use
sudo fuser -k 3000/tcp

# Check what is listening on a port
ss -tlnp | grep 3000
```

Log files

File	Contents
<code>admin.log</code>	Admin panel output
<code>proxy.log</code>	Proxy startup banner + one access-log line per proxied request
<code>logwebsdr.txt</code>	Launcher and watchdog log — start/stop, receiver driver, websdr.org registration
<code>spectrumserver.log</code>	The receiver process's own output — FFT and OpenCL setup, per-client connections, errors. The biggest of these files; it is rotated, so <code>spectrumserver.log.1</code> may sit beside it
<code>rade.log</code>	RADE/FreeDV sidecar log
<code>autorun.log</code>	Decoder daemon log — spots, SNR, drift
<code>crash.log</code>	Crash reports and thermal events: [CRASH] / [TERMINATE] backtraces, [EXIT] notes and [THERMAL] lines from the guard. Normally empty — anything in it is worth reading

All of these are readable from the panel's **Log Viewer** tabs (LOGWEBSDR, SPECTRUMSERVER, ADMIN, RADE, CRASH, AUTORUN, PROXY). The viewer shows the last 150 lines of whichever tab is selected and refreshes every 3 seconds when auto-refresh is on; it reads only the tail, so the size of the file does not matter.

Rotating proxy.log and admin.log

`proxy.log` records every proxied request and `admin.log` every request that reaches the panel itself. Neither is rotated by default. The dashboard's polling used to dominate both — `/admin/api/status` fires every ~3 seconds for as long as a tab is open — but those polls are now filtered out of both files (see below), so rotation is there to bound slow growth rather than a flood.

The repo ships a logrotate config at `logrotate/phantomsdr` (daily, or sooner if a log passes 10 MB; keeps 7 gzipped archives in `logproxy/`, so they do not clutter the project root). **It is site-specific — open it and change the three log paths, the `olddir` path and the `su` user to match your machine before installing**, then:

```
sudo cp logrotate/phantomsdr /etc/logrotate.d/phantomsdr
sudo logrotate -d /etc/logrotate.d/phantomsdr # dry run, should list all three logs
```

`logrotate/phantomsdr` in the repo and `/etc/logrotate.d/phantomsdr` are two **independent copies** — `cp` does not link them. Editing the repo file changes nothing on a running system until you re-run the `sudo cp` above. Only the `/etc` copy is what logrotate actually reads.

The config uses `copytruncate`, which is required: `manage_admin.sh` starts both processes with `>> <log>` and neither reopens its stdout, so a rename-based rotation would leave them writing into the rotated file while the new log stays empty forever.

Archives land in `logproxy/` (created automatically by `createolddir`) and are named `proxy.log.1.gz` ... `proxy.log.7.gz`, newest first. Read one with `zcat logproxy/proxy.log.1.gz | less`, or search all of them at once with `zgrep "pattern" logproxy/*.gz`.

The panel's **Clear Logs** button truncates **every** file in the table above, `autorun.log` and `proxy.log` included (`CLEAR_EXCLUDE` in `admin_server.py` is empty). Note what that costs on `autorun.log`: its decode history is the only record of what was heard and when, and nothing can reconstruct it — what survives a clear is whatever logrotate has already rotated into `logproxy/`, so a clear discards everything decoded since the last rotation. Clearing it under a running decoder is safe: the daemon holds it open `O_APPEND` and goes on appending to the same inode. The toast names what was cleared, and if a file could not be written — a `proxy.log` owned by a different user, say, on a split-user install — it names that file and the

error instead of reporting success, and leaves that log's pane on screen rather than blanking it. The files are truncated in place, never deleted: both `manage_admin.sh` and the systemd units open them `O_APPEND`, so an unlinked file would go on filling an invisible inode until the service restarts.

Successful polls are filtered out of **both** access logs, by the `QuietPollFilter` in `proxy.py` and the one in `admin_server.py`. The dashboard hits `/admin/api/status` every 3 seconds, and `/admin/api/logs`, `/admin/api/autorun/status`, `/admin/api/users`, `/admin/api/graph-stats` and `/admin/api/chat` every 5 seconds; unfiltered they are ~95% of both files. Only plain 200s on those paths are dropped — any other status, path or method is logged as before, so a failing poll stays visible and state-changing actions (`kick`, `autorun/start`, ...) are separate paths and always recorded. The one deliberate exception is `/admin/api/logs/clear`, filtered in both files: its own access line is written *after* the truncate, so without the filter every successful clear left one fresh line behind in the log it had just emptied, and the button looked broken.

Because `proxy.log` is an access log, it contains visitor IP addresses and user-agent strings — keep that in mind before sharing it when asking for help.

"Clear Logs" says Permission denied

On a machine where the systemd units were installed on a clean tree, the button can come back with:

```
x Could not clear admin.log ([Errno 13] Permission denied: /home/you/PhantomSDR-Plus/admin.log),
proxy.log ([Errno 13] Permission denied: /home/you/PhantomSDR-Plus/proxy.log)
```

It is always those two files and never the other five, because they are the only ones systemd creates itself. `StandardOutput=append:` is opened by PID 1 **before** it drops to `User=`, so a log that does not exist yet is created `root:root 0644`. The panel runs as your user and cannot truncate it. The other five logs are created by the launcher scripts, which already run as your user, so they clear normally. Where the two logs predate the units — `manage_admin.sh` created them with `>>` — they are already owned correctly and the button works, which is why the problem only shows up on a fresh systemd-first install.

Confirm with `ls -l admin.log proxy.log`: the failing ones say `root root`. Repair them:

```
sudo chown "$(id -un):$(id -gn)" admin.log proxy.log
sudo chmod 664 admin.log proxy.log
```

No restart is needed — systemd keeps writing through the file descriptor it already holds, and appending does not change ownership. Press **Clear Logs** again and both are emptied.

Do not delete the files instead. Both are held open `O_APPEND`; unlinking one leaves systemd filling an invisible inode until the service restarts, and the file that reappears is root-owned again.

To stop it coming back, install the `tmpfiles` rule the repo ships at `tmpfiles/phantomsdr-logs.conf`. It re-creates both logs with the right owner at every boot, before the units start, so a removed log — manual cleanup, a reinstall, a moved installation directory — never returns root-owned. **It is site-specific: open it and change the two paths and the user:group to match your machine first**, then:

```
sudo cp tmpfiles/phantomsdr-logs.conf /etc/tmpfiles.d/99-phantomsdr-logs.conf
sudo systemd-tmpfiles --create /etc/tmpfiles.d/99-phantomsdr-logs.conf
```

As with `logrotate/phantomsdr`, the repo file and the `/etc` copy are independent — editing the repo copy changes nothing until you re-run the `cp`. On systemd 254 and newer the rule also repairs the ownership and mode of a log that already exists, so it fixes the current failure as well as future ones; on older versions the `chown` above is what unblocks the button.

`setup_admin.sh` does all of this for you when it installs the units, so a setup made with the script never hits this. The steps above are for a hand-installed pair, or for an installation that predates the script's log-ownership step.

Logrotate does not reintroduce the problem: the shipped config uses `copytruncate`, which keeps the same inode and therefore the same owner.

Access summary

Setup	SDR	Admin panel
Without proxy	<code>http://YOUR_IP:<public_port></code>	<code>http://YOUR_IP:<port>/admin</code>
With proxy	<code>http://YOUR_IP:<proxy_port></code>	<code>http://YOUR_IP:<proxy_port>/admin</code>

Chat message deletion

The Chat History page lists every message in the current chat log. Each entry has a **Delete** button that removes that single message from the log immediately — no server restart required.

How it works:

- The admin panel rewrites the chat log file in place, stripping only the selected message line.
- The change takes effect for any user who reloads the chat; already-loaded chat history in open browser tabs is not retroactively updated.
- Clearing the entire log (the **Clear All** button) truncates the file, which also takes effect without a restart.
- A **chat reply** is stored as an ordinary message that begins with a marker naming the message it answers, for example `[RE:2026-09-25 12:00:01|SV1BTL]`, so in this list it shows with that marker in front. Deleting a reply removes only that reply. Deleting the message it answers leaves its replies in place, showing only the name and time they answer.

The delete button is rendered consistently across all five Svelte App variants. If a deletion appears not to take effect, confirm the admin panel has write permission to the chat log file:

```
ls -l ~/PhantomSDR-Plus/chat.jsonl # path depends on your config
```

Waterfall broadcast messages

The **Waterfall Message** panel lets you push a persistent text banner to every connected user's waterfall display without touching the server process.

Sending a message

1. Open the admin panel and go to **Waterfall Message**.
2. Type your message text and choose a colour (hex, e.g. `#ffdd00`).
3. Click **Send** — the message appears on all active waterfall views immediately.

Clearing the message

Click **Clear** to remove the banner from all waterfalls. The message state is held in memory by the admin panel; it is cleared automatically if the admin panel process is restarted.

How it works

The admin panel exposes two internal endpoints that the proxy forwards:

Endpoint	Method	Purpose
/admin/api/waterfall-message	POST	Set or clear the current banner text and colour
/admin/api/waterfall-message	GET	Return the current message state as JSON

The frontend polls the message state and renders it as an overlay on the waterfall canvas. No WebSocket reconnection or page reload is needed on the client side.

Typical uses

- Announce scheduled maintenance: "Server restart in 10 minutes"
- Flag band conditions: "Solar flux 180 – 10m wide open"
- Welcome message: "Welcome to SV1BTL WebSDR – Athens, KM17"

System Graphs

The **Graphs** page plots four measures against one shared time axis:

- **CPU frequency** — average clock across all cores, in GHz
- **CPU load** — total utilisation, in percent
- **CPU temperature** — in °C, with dashed guides at 70 °C (warm) and 80 °C (critical)
- **Users online** — listeners connected to the WebSDR at that moment

Pick a range with **15 MIN / 1 HOUR / 4 HOURS / 12 HOURS / 24 HOURS**. Hovering the plot draws a crosshair and reads out all four values at that instant. The four cards above the plot always show the newest sample.

How sampling works

A background thread in `admin_server.py` takes one sample every **2 seconds**. Samples are held in two tiers: the last **1 hour** at full 2-second resolution (1800 points), and **24 hours** of 30-second averages (2880 points) for the longer ranges. The page picks the tier that covers the range you chose, and says **30s averages** in the header when you are looking at the aggregated one.

Storing a full day at 2-second resolution would mean ~43000 points — tens of MB of memory, a multi-megabyte first load, and about 40 points per pixel of canvas, which no screen can render. Averaging to 30 seconds keeps a day in 2880 points.

Sampling starts together with the admin panel, not when you first open the page, so the page opens onto history that already exists.

The buffer is **memory-only**: nothing is written to disk, and the history is lost when the admin panel restarts. That is deliberate — it keeps the feature out of log rotation and off your disk.

2 seconds gives a near-live view and is short enough not to alias CPU frequency, which on a modern CPU swings between its idle and turbo clock from one sample to the next. A sample costs about a millisecond, so the sampler is negligible next to the SDR server. The page polls at the tier's own cadence

— every 2 seconds on the live ranges, every 30 seconds on the aggregated ones, because asking more often than points are produced only returns empty replies.

To change the cadence or the retention, edit these constants near the top of the sampler block in `admin_server.py` and restart:

```
GRAPH_INTERVAL_S    = 2          # seconds between fine samples
GRAPH_FINE_S        = 3600       # how far back the fine tier reaches (1h)
GRAPH_COARSE_EVERY  = 15         # fine samples per coarse point (15 x 2s = 30s)
GRAPH_COARSE_S      = 24 * 3600 # how far back the coarse tier reaches (24h)
```

Where the numbers come from

Measure	Source
CPU frequency	<code>psutil.cpu_freq()</code> , falling back to <code>/sys/devices/system/cpu/cpu*/cpufreq/</code> . This is the same kernel data <code>cpufreq-info</code> formats, so cpufrequtils is not required .
CPU load	<code>psutil.cpu_percent()</code>
CPU temperature	<code>psutil.sensors_temperatures()</code> , falling back to <code>/sys/class/thermal</code> , then the <code>sensors</code> command
Users online	spectrumserver's own <code>/users</code> endpoint — the authoritative session list. Counting sockets with <code>ss</code> cannot see real client IPs behind proxy.py.

If a measure is unavailable on your machine, that panel says so rather than drawing a flat zero line. A reading that fails mid-run breaks the line, so a gap in the data never looks like a measured value.

Why four panels and not one graph

GHz, percent, degrees and a head-count share no common scale. Drawing them on one plot needs several y-axes, and the crossings that produces are artefacts of the scaling rather than facts about the server. Four stacked panels on one time axis keep the comparison honest. Every panel is anchored at zero, so curve height stays proportional to the value.

Colour is reserved for temperature, where amber and red mark the thresholds above. The other panels share one colour because each holds a single series that its own title already names.

Endpoint

`GET /admin/api/graph-stats?since=<epoch>` (login required) returns the samples newer than `since`, so the page polls incrementally instead of re-downloading the whole buffer every tick.

Thermal Guard

Full manual: [THERMAL_GUARD.md](#) — the four modes and what a sysop must do for each, running it without the panel, enabling the throttle stage without root, testing and troubleshooting. This section covers the panel side.

Stops the SDR server when the CPU gets dangerously hot, and starts it again once it has cooled. It is part of the panel — nothing extra to install — and appears as a **THERMAL GUARD** card on the Dashboard with its settings under **Settings**.

It ships **inert**: the mode starts at `log`, so out of the box it only records what it *would* have done. Nothing touches your server until you change the mode.

It works with any start/stop method

The guard never tries to work out what supervises your server. It acts through whatever scripts you set as **Default start script** and **Default stop script** in Settings, and while the CPU is over-temp it re-issues the stop on **every check** (every 2 s). Anything that brings the server back — the watchdog inside `start-*.sh`, a systemd unit, a cron job, a `tmux` session — is undone within a couple of seconds until the machine has cooled. That is the *lockout*.

If no stop script is configured, it falls back to `SIGTERM` to the process named in `sdr_process_name`, then `SIGKILL` after a 10-second grace period.

Thresholds come from your own CPU

Rather than a fixed number that is wrong on most hardware, the guard reads the critical trip point the kernel publishes for your CPU and works backwards from it:

Your CPU reports	warn	throttle	stop	resume
Intel, limit 100 °C	88	92	95	75
AMD Ryzen, limit 95 °C	83	87	90	70
Raspberry Pi, limit 85 °C	73	77	80	60

This matters: a fixed 95 °C would be near-normal on an Intel, but on a Ryzen — whose Tctl sits at 95 °C by design under boost — it would stop the server on a healthy machine, and on a Pi, which throttles at 80 °C, it would never fire at all. Any threshold can still be set by hand in Settings; leave a field blank for *auto*.

To see what your machine looks like to the guard:

```
python3 thermal_guard.py --once
```

```
sensor      : hwmon:coretemp
temperature: 58.0 C
crit        : 100.0 C
mode        : log
thresholds  : warn 88.0  throttle 92.0  stop 95.0  resume 75.0 (C)
cpufreq     : not writable – throttle stage disabled
```

The four stages

Stopping the server is the last resort, not the only tool:

1. **warn** — logged, nothing else.
2. **throttle** — lowers the CPU clock ceiling by 20 %. Needs a `cpufreq` driver *and* write access to `scaling_max_freq`, which normally means running as root; where that is unavailable (the usual case for an unprivileged panel, and on most VPS) the stage **disables itself** and the guard falls back to `warn` → `stop`. `./setup-cpufreq-perms.sh` grants that write access to a `cpufreq` group, so the stage works without running the panel as root — see [THERMAL_GUARD.md](#).
3. **stop** — runs your stop script and holds the server down (see *lockout* above).
4. **resume** — after the CPU has stayed below the resume temperature for `thermal_resume_s` (5 minutes by default), the clock ceiling is restored and, in `stop+restart` mode, your start script runs again.

Each stage must be **held** for its sustain time — 30 s for `warn` and `throttle`, 60 s for `stop` — so a brief spike during a compile never trips it.

Repeated trips are capped by `thermal_max_stops_hour` (2 by default). Once that is reached the *automatic restart* is disabled while the protection keeps working, so an overheating machine cannot flap between running and stopped.

Which sensors are used

Only real CPU-die sensors: `coretemp` (Intel), `k10temp` / `zenpower` (AMD), `cpu_thermal` / `soc_thermal` (ARM, Raspberry Pi). The hottest reading on the chip is used, not the average.

`acpitz` and unlabelled thermal zones are **deliberately ignored** — they often report a case or board temperature tens of degrees below the actual CPU, so a guard trusting them would never fire when it mattered. If your machine has no usable sensor (common on a VPS or inside a container) the card says `no trusted CPU sensor` and the guard stays inactive rather than pretending to protect you.

What it writes

Everything goes to `crash.log`, i.e. the **CRASH** tab of the Log Viewer, one event per line:

```
[THERMAL] warn: 89.0C (warn=88.0 crit=100.0) sustained 31s
[THERMAL] throttle: 93.0C sustained 30s - cpufreq max lowered 20%
[THERMAL] STOPPED server: 96.0C sustained 62s (stop=95.0 crit=100.0) - Started stop-websdr.sh
[THERMAL] lockout: process reappeared at 94.0C - re-stopped [3 time(s) so far]
[THERMAL] recovered: 74.0C held below 75.0 - lockout cleared
[THERMAL] auto-restart: Started start-rx88mk2.sh
```

The lockout line is rate-limited to one per minute, so a supervisor that keeps fighting the guard cannot flood the log.

Testing it before you trust it

Set **TEST TEMPERATURE** in Settings to a value above your stop threshold. The guard treats it as a real reading, so the whole path — warn, throttle, stop, lockout, recover, restart — runs on demand. Clear the field to return to the real sensor.

Running this test with the mode set to `stop` really does stop the server and disconnect your listeners. Do it when nobody is on, or leave the mode at `log`, where the test shows what *would* have happened without touching anything.

Suggested way to adopt it

1. Leave the mode at `log` for a week and carry on as normal.
2. Read the CRASH tab after your heavy moments — a full rebuild, a hot afternoon. If nothing appears, your machine never came close.
3. If the logged temperatures look right, set the mode to `stop` (or `stop+restart` to have it come back by itself).
4. Prove it once with the test-temperature field, then leave it alone.

Recovering from a lockout

CLEAR LOCKOUT on the Dashboard card clears the lockout and the rate limit. It does not disarm the guard: if the CPU is still too hot, the next check simply stops the server again. Use it after you have fixed the cooling.

Endpoint

`GET /admin/api/thermal` (login required) returns the guard's full state — sensor, thresholds, stage, lockout, last event. `POST` with `{"action": "reset"}` does the same as the CLEAR LOCKOUT button.

Without the admin panel

`thermal_guard.py` is stdlib-only and runs on its own, reading the same `admin_config.json`:

```
python3 thermal_guard.py --once           # print sensor, trip point and thresholds
python3 thermal_guard.py                 # run as a service (log-only until armed)
python3 thermal_guard.py --mode stop+restart # arm it without writing any config file
python3 thermal_guard.py --config /path/to/other.json
python3 thermal_guard.py --help
```

`--mode` overrides `thermal_mode` from the config file, so a machine with no panel can be armed straight from its systemd unit with no JSON at all.

Spot Reporting (Aurun FT8/FT4/JS8/WSPR)

The **Spot Reporting** tab controls the autorun daemon (`autorun/index.js`) — a Node.js process that decodes FT8/FT4/JS8/WSPR server-side straight off the receiver and uploads spots to the reporting networks:

- **FT8 / FT4 / JS8 → PSK Reporter** (IPFIX/UDP)
- **WSPR → wsprnet.org** (HTTP)

JS8 is spotted at Normal speed only — the 15 s calling cycle, where heartbeats and CQs are. Spots come from heartbeats, compound frames and directed messages; group destinations such as `@ALLCALL` and callsigns the decoder could not resolve (`<...>`) are never reported. JS8 shares the PSK Reporter queue with FT8 and FT4 and is counted separately, exactly as they are.

Reporting is OFF by default. Nothing is uploaded until you enable a destination and press **Start**. Decoding and uploading are independent: the daemon can decode and log with reporting off, so you can verify activity before anything goes public.

⚠ Spots are uploaded to public networks under **your callsign**. Only enable bands/modes your receiver genuinely hears, and set the correct grid square.

Prerequisites

The autorun daemon needs Node.js 22+, the `ws` + `cbor-x` npm packages (resolved via the `autorun/node_modules` symlink → `frontend/node_modules`), and `util-linux` (`taskset`). The installers set all of this up automatically — see the [Aurun Spot Reporter section of INSTALLATION.md](#). If **Start** fails, that symlink or `taskset` is the usual cause (see Troubleshooting below).

Using the tab

1. **Identity** — callsign and grid. Pre-filled from `frontend/site_information.json` (`siteSysop` / `siteGridSquare`, truncated to 6 chars); override here if needed.
2. **Band × mode matrix** — tick the combinations to decode. Rows are bands, columns are FT8 / FT4 / JS8 / WSPR; unsupported cells are disabled. WSPR additionally covers the LF/MF bands (2200m, 630m) and an extra 80m EU channel.
3. **Destinations** — enable **PSK Reporter** and/or **wsprnet** (both off by default).
4. **Max slots** — a safety cap on how many band/mode slots can run at once.
5. **Start / Stop** — spawns/kills the daemon (pinned to the top CPU cores with `taskset`, chosen automatically for the machine — see *Resource requirements & limitations* below). **Save / Reload** persist and re-read the config; **Free All** clears every slot.

The status card polls every 5 s and shows running state, decode/upload counts and the last upload time. A **REPORTING** badge appears on the main waterfall while reporting is active.

"0 sent" for the first few minutes is normal. PSK Reporter uploads batch every **5 minutes** and wsprnet every **2 minutes** — spots queue until the next flush. The status card shows the pending count and cadence.

Changing bands or modes while it is running

Changes do not take effect until the daemon is restarted. `autorun/index.js` reads `autorun.json` once, at startup. There is no file watcher and no reload signal — the only signals it handles are SIGINT and SIGTERM, both meaning shut down. A running daemon therefore keeps decoding the slots it was launched with, whatever you save afterwards.

To apply a change:

1. Tick / untick the bands and modes you want
2. **SAVE CONFIG**
3. **STOP**
4. **START**

Or simply **STOP** → change the boxes → **START**, since START saves the config before launching.

⚠ **Pressing START without stopping first does not apply the change.** START saves the config, then the start request answers `already running` and nothing relaunches. You get a "saved" toast beside an "already running" error while the daemon carries on with the **old** bands — easy to read as success. Always STOP first.

What a restart does to the counters:

- **The all-time counts beside each checkbox survive** — they live in `autorun-totals.json`, which is not deleted on shutdown.
- **The per-decoder tiles reset to zero**, being per-run by design.
- **Nothing queued is lost:** shutdown flushes pending spots before exiting.
- A **newly enabled** band starts at 0; a band you had used **before** resumes its previous total.
- A band you **deselect keeps its number** next to the now-unchecked box; the history is not deleted.

Decoding stops only for the few seconds between STOP and START.

Spot counters

Two different counts are shown, and they answer different questions.

SPOTS UPLOADED PER DECODER — a tile per decoder (FT8 / FT4 / JS8 / WSPR) with the spots uploaded **since the daemon last started**, under it the decode count and how many are still queued. Stop/Start resets these to zero. A decoder whose destination is switched off shows its decodes and `reporting off` rather than a bare 0, because zero uploads is a setting there, not a fault.

The number beside each checkbox in BANDS & MODES is that band+mode's **all-time** uploaded spots. These are kept in `autorun-totals.json` and survive restarts. **Amber with a dot in front** (· 123) means that slot has decoded but not uploaded yet. That is normal between flushes — PSK Reporter uploads every 5 minutes and wsprnet every 2 — so FT8/FT4 counts sit amber for the first few minutes after a start. It also stays amber if that decoder's destination is switched off. A plain 0 with no dot means that slot is enabled but has decoded nothing yet — WSPR sits at zero for several minutes after a start, because it runs on a 2-minute cycle while FT8 is already counting in the hundreds. A slot the daemon has

never run shows nothing at all. Hover any number for the uploads, the decodes and the time of the last upload.

FT8 and FT4 share one PSK Reporter upload queue, so the daemon counts each spot against its own mode and band as it goes out; the split is not estimated from the totals afterwards.

Clearing the counters. FREE ALL SLOTS unticks every band/mode, turns both destinations off, stops the daemon and **erases the all-time counters** — after it, no number is shown beside any checkbox. This is the only way to reset them, and it cannot be undone. The button stops the daemon and waits for it to exit before deleting `autorun-totals.json`, because the daemon rewrites that file on shutdown; deleting it under a live daemon would simply bring the numbers back.

Resource requirements & limitations

The daemon is deliberately lightweight and runs on low-resource hardware (a 4-core i5 is fine), but there are real limits to be aware of.

Autorun CPU pinning is config-aware and automatic. The **autorun daemon** is always launched by the admin panel (`admin_server.py`), so its pin is set on every install without any configuration: it derives a `taskset` core range from the CPU count, reserving a few of the top cores for decoding, or runs unpinned on ≤ 4 cores.

spectrumserver pinning is up to your own start method. How you launch spectrumserver varies between installs (SDR type, tooling, personal scripts), so this document does not assume any particular launcher. The admin panel does not start or pin spectrumserver — that is entirely down to whatever command or service you use to run it. If you want spectrumserver kept off the cores the autorun daemon uses, pin it yourself with `taskset` in your own launch command (see the override section below). If you don't, it simply runs unpinned and the OS scheduler balances it — correct and safe, you just lose the deliberate core separation.

For reference, the autorun daemon reserves these top cores (so if you pin spectrumserver, keep it to the lower cores to avoid overlap):

Logical CPUs	autorun daemon uses	leave for spectrumserver
≤ 4	<i>unpinned</i>	<i>unpinned</i>
6	5	0-4
8	6-7	0-5
12 (e.g. 8P+4E hybrid)	8-11	0-7
16	12-15	0-11

On a **4-core (or fewer)** machine there is nothing to segregate, so the autorun daemon also runs *unpinned* and shares all cores — correct and safe, but the SDR FFT and the decode bursts compete for the same cores.

Known limitations:

1. **The real bottleneck is spectrumserver + SDR bandwidth, not the daemon.** An RX888 @ 30 MHz needs OpenCL/GPU; a low-core CPU without a capable GPU cannot sustain that FFT. Pair low-resource hardware with a narrower SDR (RSP1A $\approx$ 10 MHz, RTL-SDR $\approx$ 2.4 MHz).
2. **WSPR is the CPU long pole.** Its Fano decoder is a JS port ($\sim$ 30 s per band, single-threaded). The 4-worker pool runs 4 decodes in parallel; enabling **more than $\sim$ 4 WSPR bands** at once can queue past the 120 s slot, especially while the SDR is competing for cores. FT8/FT4 decodes are cheap by comparison.

3. **Slot-count scaling.** The ~2–2.5 cores / 600–700 MB figure is for the full ~28 slots on an 8-core box. On 4 shared cores, keep it to a handful of bands (guideline: ≤ 6 FT8/FT4 + ≤ 3 WSPR) and watch the pool's `queued` stat.
4. **Pinning assumes Intel hybrid topology** (lower cores = faster P-cores). On AMD or CPUs with interleaved SMT numbering the auto split is still valid (no overlap, no crash) but "lower cores are faster" may not literally hold, and on a hyperthreaded 4C/8T part the top cores are SMT siblings — confined, not fully isolated. Where the auto range isn't ideal, use the **manual override** below.
5. **Band coverage is limited by the receiver.** The RX888 config (`sps=600000000` → 30 MHz Nyquist) cannot reach 6 m and above; the band table is 160 m–10 m. Other SDRs cover whatever their tuned window allows.
6. **Reporting cadence.** PSK Reporter flushes every 5 min, wsprnet every 2 min — "0 sent" in the first few minutes is normal, not a fault.

Manual CPU-pin override (advanced)

The auto-derived core split is right for most machines, but you can force a specific pinning where it isn't (AMD CCX/CCD, ARM big.LITTLE, interleaved SMT). Two independent overrides, each accepting a `taskset -c` core list (2-3, 0,2,4, 0-2,5) or `none/off/unpinned` to disable pinning. An invalid value is ignored and the auto-derive runs — a typo can never stop a launch.

Autorun daemon — `AUTORUN_CORES` env var, or a "cores" field in `autorun.json` (env wins). The "cores" field is preserved across admin-panel **Save**, so a hand-edit sticks. Two ways to set it:

Option A — `autorun.json` (persistent, recommended). Add a "cores" line to the config at the repo root, then **Stop** → **Start** the daemon on the Spot Reporting tab:

```
// autorun.json
{ "identity": { "callsign": "SV1BTL", "grid": "KM17VX" },
  "reporting": { "pskreporter": true, "wsprnet": false },
  "slots": [ /* ... */ ],
  "cores": "2-3" }           // or "none" to run unpinned
```

Option B — *environment variable (one-off)*. The daemon is spawned by the admin panel, so the var must be in the **admin panel's** environment — set it and restart the panel (env wins over the `autorun.json` value):

```
# Method A — the var must be set on the process that starts:
./manage_admin.sh stop && AUTORUN_CORES=2-3 ./manage_admin.sh start

# Method B — systemd ignores a var set on the systemctl command line,
# so put it in the unit instead:
sudo systemctl set-environment AUTORUN_CORES=2-3
sudo systemctl restart phantomsdr-admin
```

spectrumserver — pin it yourself by prepending `taskset -c <cores>` to whatever command you use to start the server. This works with any start method (a direct launch, an SDR pipeline, a service unit, etc.):

```
# pin the server to cores 0-3, direct launch:
taskset -c 0-3 ./build/spectrumserver --config <your-config>.toml && <your-input>

# or in an SDR pipeline:
<your-sdr-source> | taskset -c 0-3 ./build/spectrumserver --config <your-config>.toml
```

⚠ **Make it survive restarts.** An inline `taskset` only applies to that one launch. If something auto-restarts the server (a watchdog, a systemd service, a `cron/@reboot` entry, ...), add the `taskset` prefix inside the script or unit that actually starts it — otherwise the restart drops the pin.

Verify the pin took — print the running processes' actual CPU affinity:

```
taskset -cp "$(pgrep -f 'autorun/index.js')" # autorun daemon
taskset -cp "$(pgrep -x spectrumserver)"    # spectrumserver
```

Accepted values (both overrides): a `taskset -c` list (2-3, 0,2,4, 0-2,5), or none/off/unpinned for no pinning. Anything malformed is ignored and the auto-derive runs, so a typo can never block startup.

On a Raspberry Pi / any ≤ 4 -core box you normally need neither — the auto path already runs both processes unpinned, which is the correct choice on a small, homogeneous CPU. The override is for larger non-Intel-hybrid machines.

Files and endpoints

Item	Purpose
<code>autorun.json</code>	Saved config (identity, slots, destinations, max slots, optional <code>cores</code> pin override). Written by the tab; git-ignored.
<code>autorun-status.json</code>	Live status the status card reads (pid, counts, last upload). Written every 15 s; removed on stop.
<code>frontend/dist/autorun-active.json</code>	Public badge data served at <code>/autorun-active.json</code> . Empty when reporting is off.
<code>autorun.log</code>	Daemon stdout/stderr.
GET/POST <code>/admin/api/autorun/config</code>	Read / write <code>autorun.json</code> (validates callsign, grid, band/mode combos, slot cap).
GET <code>/admin/api/autorun/status</code>	Running state (pgrep) + <code>autorun-status.json</code> .
POST <code>/admin/api/autorun/start / stop</code>	Spawn (<code>taskset -c <auto> node autorun/index.js</code>) / SIGTERM. The core range is derived from the CPU (see below).

Note: after upgrading `admin_server.py` you must [restart the panel](#) to load new autorun routes or an updated band/mode matrix. The daemon runs as a separate process, so it survives admin-panel restarts.

Troubleshooting

Problem	Fix
Admin panel not starting	Check <code>admin.log</code> — usually a missing Python package
Proxy not starting	Run <code>python3 -c "import aiohttp"</code> — if it fails: <code>pip3 install aiohttp --break-system-packages</code>
<code>proxy.log</code> is empty	First check the proxy is actually running (<code>./manage_admin.sh status</code>). If it is, you are on an old version: the launcher must use <code>python3 -u</code> (unbuffered — otherwise the startup banner never leaves the 8 KB buffer), and <code>proxy.py</code> 's <code>main()</code> must call <code>logging.basicConfig()</code> (otherwise <code>aiohttp</code> 's access logger has no handler and every request line is discarded, because <code>web.AppRunner</code> — unlike <code>web.run_app</code> — does not configure logging).
Proxy starts but waterfall is blank	Check <code>sdr_host</code> in <code>admin_config.json</code> — normally <code>127.0.0.1</code> . If it holds an old LAN IP from a previous version, change it and restart the panel
Kick button says "no connections"	Run <code>getcap \$(which ss)</code> — if no <code>cap_net_admin</code> , run <code>sudo setcap cap_net_admin+ep \$(which ss)</code>
Status always shows OFFLINE	Go to Settings → SDR Process Name — set it to the exact name shown by <code>ps -eo comm,args \ grep -v grep</code>
Users page shows no data	Verify the endpoint works: <code>curl http://127.0.0.1:<public_port>/users</code>
External browser gets NetworkError	Check proxy is running: <code>./manage_admin.sh status</code>
Proxy broke after a network change	Only relevant if <code>sdr_host</code> still holds a LAN IP — set it to <code>127.0.0.1</code> and restart the panel
Chat delete button has no effect	Check write permission on the chat log file: <code>ls -l ~/PhantomSDR-Plus/chat.jsonl</code>
Waterfall message not appearing	Confirm the proxy is running (<code>./manage_admin.sh status</code>) and the frontend is on a recent build that includes the overlay renderer
Waterfall message lost after restart	Expected — message state is in-memory only; re-send it after restarting the admin panel
Spot Reporting "Start" fails / daemon exits immediately	Check <code>autorun.log</code> . Usually the missing <code>autorun/node_modules symlink (ln -sfn ../frontend/node_modules autorun/node_modules)</code> or <code>taskset</code> not installed (<code>sudo apt install -y util-linux</code>).
Spot Reporting: daemon runs but decodes stays 0 and <code>autorun.log</code> shows 504 / tap closed 1006	The audio tap can't reach spectrumserver. The daemon auto-detects the port from the running server's config; the <code>[autorun]</code> tap backend: <code>HOST:PORT</code> log line must match your <code>[server]</code> port. If it's wrong (or the server wasn't running at start), pin it in <code>autorun.json</code> : <code>"server": { "host": "127.0.0.1", "port": 9002 }</code> , then Stop→Start.
Spot Reporting shows "0 sent"	Normal for the first few minutes — PSK Reporter flushes every 5 min, <code>wspnrt</code> every 2 min. Confirm a destination is enabled and the daemon is running.
New bands/modes not showing in the matrix	Restart the panel — the matrix is loaded at admin start.
<code>autorun.log</code> shows <code>MODULE_TYPELESS_PACKAGE_JSON /</code>	Cosmetic warning (decoding still works). <code>frontend/src/modules/package.json</code> is missing <code>"type": "module"</code> ; add that line near the

Problem	Fix
"Reparsing as ES module ... performance overhead"	top, then Stop→Start. No frontend rebuild needed — the daemon imports the source file directly.
Spot Reporting "Start" fails with <code>taskset: ... Invalid argument</code> on an old/small PC	Should not happen with the config-aware launcher — the core range is derived from the CPU count and goes unpinned on ≤ 4 cores. If seen, <code>admin_server.py</code> predates that change; update it (<code>_autorun_taskset_prefix</code>) and restart the panel .
REPORTING badge not on the waterfall	Reporting must be ON with at least one slot; the badge reads <code>/autorun-active.json</code> . Rebuild the frontend if <code>dist/index.html</code> predates the badge.

CHAPTER 4

Thermal Guard

THERMAL_GUARD.md

Thermal Guard — Sysop Manual

CPU over-temperature protection for PhantomSDR-Plus.

A WebSDR is one of the few things a computer does that keeps every core busy, all day, forever. If a fan clogs, a pump stalls or the room gets hot in August, nothing tells you — the server just keeps decoding until the CPU is cooking itself. The thermal guard is the piece that notices and pulls the plug for you.

It ships **inert**: out of the box it only writes to a log. Nothing is protected until you choose a mode. This manual is about making that choice and living with it.

In a hurry? Run `python3 thermal_guard.py --once`, read the thresholds it prints, then set `thermal_mode` to `stop+restart` in the admin panel. That is the whole job. Everything below is the *why*.

Contents

1. What it does
2. Quick start
3. How it decides
4. The four modes — what each one does and what you must do
5. Choosing a mode
6. Configuration reference
7. Running it without the admin panel
8. Enabling the throttle stage without root
9. Unattended operation — when nobody is watching
10. Testing it before you trust it
11. Reading the log
12. Lockout and the rate limit
13. Troubleshooting
14. Turning it off or removing it

1. What it does

Every 2 seconds the guard reads your CPU's die temperature and compares it against four thresholds. When the temperature stays above one of them for long enough, it moves up a ladder:

```

normal → warn → throttle → stop → (cool down) → restart
        log a line  lower the  run your  hold the  run your
                   CPU clock  stop script  server down  start script

```

Two design decisions make it work on *any* sysop's machine, which is worth understanding before you rely on it:

It never asks how you run your server. systemd, a watchdog loop, cron, tmux, a bare shell — the guard does not know and does not care. Instead, while the machine is over-temperature it holds a **lockout**: it re-issues the stop on every tick. If your watchdog restarts the server, the guard stops it again within two seconds. Whatever is trying to keep the server alive loses, until the CPU has cooled. This is the part that makes it universal, and it is tested against a real watchdog.

It never hard-codes a temperature. "Stop at 95 °C" is a bad rule: on an Intel with Tjmax 100 that is normal-ish load, on a Ryzen it is *below* where Tctl sits under boost by design, and on a Raspberry Pi it is unreachable because the SoC throttles itself at 80. So the guard reads the critical trip point your own kernel publishes (`tempN_crit`) and works backwards from it. Same code, sensible numbers everywhere.

2. Quick start

1. See what your machine looks like to the guard. Do this first, always:

```
cd ~/PhantomSDR-Plus
python3 thermal_guard.py --once
```

```
sensor      : hwmon:coretemp
temperature: 65.0 C
crit        : 100.0 C
mode        : log
thresholds  : warn 88.0  throttle 92.0  stop 95.0  resume 75.0 (C)
cpufreq     : not writable – throttle stage disabled
```

If `sensor` says `NONE`, stop here and read [Troubleshooting](#) — the guard cannot protect a machine it cannot measure, and it will tell you so rather than pretend.

2. Leave it in log mode for a few days. It is already running inside the admin panel. Let it watch a normal week — summer afternoons, a big build, a busy Saturday contest — and see whether it would ever have fired.

3. Arm it. Admin panel → **Settings** → **THERMAL GUARD** → **MODE** → stop+restart → **SAVE THERMAL SETTINGS**. It takes effect within about 2 seconds; no restart needed.

3. How it decides

The sensor

The guard only trusts real CPU-die sensors:

Source	Typical hardware
<code>coretemp</code>	Intel
<code>k10temp</code> , <code>zenpower</code> , <code>zenpower3</code>	AMD
<code>cpu_thermal</code> , <code>soc_thermal</code>	Raspberry Pi, ARM SBCs
<code>x86_pkg_temp</code> thermal zone	fallback on x86

`acpitz` and unlabelled thermal zones are **deliberately excluded**. On a typical desktop `acpitz` reads about 28 °C while the CPU package is at 59 °C — a guard trusting it would simply never fire. Reporting "disabled, no trusted sensor" is more honest than pretending to protect.

`psutil` is used if it happens to be installed, but is never required. The guard is pure standard library.

The thresholds

Each one is an offset below the critical trip point the machine itself publishes:

Threshold	Offset below crit	What it means
warn	−12 °C	Something is wrong, log it
throttle	−8 °C	Try to cool it down by slowing the CPU
stop	−5 °C	Too close to the edge — stop the server
resume	−25 °C	Genuinely cool again, safe to come back

Which lands like this in practice:

Your CPU reports	warn	throttle	stop	resume
Intel, crit 100 °C	88	92	95	75
AMD Ryzen, crit 95 °C	83	87	90	70
Raspberry Pi, crit 85 °C	73	77	80	60

If the machine publishes no critical point at all, the guard falls back to a conservative fixed set and says so. You can override any threshold with an absolute number (see [Configuration reference](#)), but you should not need to.

Sustain times — why a spike never acts

Nothing happens on a single hot reading. A threshold must be held **continuously**:

- **warn** and **throttle**: 30 s (`thermal_warn_sustain_s`)
- **stop**: 60 s (`thermal_sustain_s`)
- **resume**: 300 s below the resume threshold before a restart (`thermal_resume_s`)

A compile spike, a burst of decoding, a fan ramping up — all far too short to matter. One dip below the threshold resets the clock. On top of that the guard requires 3 consecutive valid readings before it will act at all after starting, and it discards any reading outside 20–125 °C as a sensor glitch rather than a temperature.

4. The four modes

`thermal_mode` is the single switch that decides how far up the ladder the guard is allowed to go. Each mode **includes everything the modes before it do**.

Mode	Warns	Throttles	Stops	Restarts	Server can go down?
log	✓	—	—	—	No — touches nothing
throttle	✓	✓	—	—	No
stop	✓	✓	✓	—	Yes, stays down until you act
stop+restart	✓	✓	✓	✓	Yes, comes back on its own

log — watch and report (the shipping default)

What it does: everything the other modes do, except acting. It follows the whole ladder and writes what it *would* have done to `crash.log`. It never runs your stop script, never touches the CPU frequency, never interrupts a listener.

What you must do: nothing — it is already running. After a week, open the **CRASH** tab in the admin panel, or run:

```
grep THERMAL crash.log
```

- **Nothing there?** Your cooling is fine and you can arm the guard with confidence.
- **STOP level reached ... NOT stopping?** The guard would have shut your server down. That is a real thermal problem you have been running with — fix the cooling *and* arm the guard.

Use it when: you have just installed the guard and do not yet know how your machine behaves. This is a shakedown mode, not a destination — a guard in `log` mode protects nothing.

throttle — slow the CPU down, never interrupt service

What it does: at the throttle threshold it lowers the maximum CPU frequency by 20 %, then restores it once the temperature is back below the warn line. Your WebSDR stays up throughout. Listeners may notice slightly worse performance under heavy load; most will notice nothing.

If the temperature keeps climbing past the stop threshold anyway, this mode **logs it and does nothing** — it will not stop the server.

What you must do:

1. Give the guard write access to the CPU frequency limit — it needs it, and by default only root has it. Run `./setup-cpufreq-perms.sh` once; see [section 8](#).
2. Confirm with `python3 thermal_guard.py --once` that it prints `cpufreq : writable, throttle stage available`. If it says *not writable*, this mode does nothing at all beyond what `log` does, and will say so in the log every time.

Use it when: you are chasing a marginal cooling problem and want the machine defended without ever dropping listeners. **Be aware this is not full protection** — if throttling is not enough, nothing else happens. Good as a first step; poor as a final answer.

stop — shut the server down and leave it down

What it does: at the stop threshold it runs your stop script, then holds the **lockout** — re-stopping the server every 2 seconds for as long as the machine is over-temperature, so nothing can revive it behind your back. When the CPU has been below the resume threshold for 5 minutes the lockout clears. **The server does not come back on its own.** You restart it when you are satisfied the cause is fixed.

What you must do:

1. Make sure the guard knows how to stop your server. In the admin panel that is the configured stop script (`./stop-websdr.sh` for the shipped launchers). If no stop script is set, the guard falls back to `SIGTERM` on the process name, then `SIGKILL` after a 10-second grace — which works, but a proper stop script is cleaner.
2. Accept that **your WebSDR will be offline until you notice**. Have some way of finding out: an uptime monitor, the CRASH tab, or simply checking daily.

Use it when: the machine matters more than the service, you are usually around, or a thermal event is serious enough that you want to inspect the box before it carries on. Also the right choice if you do not trust an automated restart on your hardware.

stop+restart — shut down, cool off, come back by itself

What it does: everything `stop` does, plus: once the CPU has held below the resume threshold for 5 minutes, it runs your start script and the WebSDR comes back on its own.

Flapping is prevented by `thermal_max_stops_hour` (default 2). After two thermal stops within an hour the guard **disables the automatic restart** and leaves the server down for you to look at. Note the important detail: the rate limit turns off the *restart*, never the *protection* — the lockout keeps stopping the server while it is hot, no matter how many times it has already tripped.

What you must do:

1. Make sure **both** scripts are configured and actually work standalone — the start script as well as the stop script. Test them by hand first: `./stop-websdr.sh` then `./start-rx888mk2.sh`.
2. Nothing else. This is the fire-and-forget mode.

Use it when: the WebSDR is public, unattended, or remote — which is most of them. This is the recommended mode for a normal installation.

5. Choosing a mode

Your situation	Mode
Just installed the guard, do not know the machine yet	<code>log</code> for a week
Public WebSDR, unattended, you want it to look after itself	stop+restart
Remote site you cannot easily reach	stop+restart
You want to inspect the machine after any thermal event	<code>stop</code>
Hardware you do not trust to survive an automatic restart	<code>stop</code>
Chasing marginal cooling, must not drop listeners	<code>throttle</code> , then move on to <code>stop+restart</code>
Machine has no trusted CPU sensor	none of them will work — fix the sensor first

For most sysops the answer is `stop+restart`, and the honest summary of the others is: `log` protects nothing, and `throttle` protects a little.

6. Configuration reference

All keys live in `admin_config.json` next to `thermal_guard.py`, and every one of them is also editable in **Settings → THERMAL GUARD**. The guard **re-reads the file on every tick**, so changes take effect within ~2 seconds — no restart.

Key	Default	Meaning
<code>thermal_enabled</code>	<code>true</code>	Master switch. <code>false</code> disables the guard entirely.
<code>thermal_mode</code>	<code>"log"</code>	<code>log</code> <code>throttle</code> <code>stop</code> <code>stop+restart</code> — see section 4 .
<code>thermal_warn</code>	<code>null</code>	Absolute warn threshold in °C. <code>null</code> = derive from the CPU's crit point.
<code>thermal_throttle</code>	<code>null</code>	Absolute throttle threshold. <code>null</code> = derived.
<code>thermal_stop</code>	<code>null</code>	Absolute stop threshold. <code>null</code> = derived.
<code>thermal_resume</code>	<code>null</code>	Absolute resume threshold. <code>null</code> = derived.
<code>thermal_sustain_s</code>	<code>60</code>	Seconds the stop threshold must be held before stopping. Minimum 5.
<code>thermal_warn_sustain_s</code>	<code>30</code>	Seconds for the warn and throttle stages. Minimum 5.
<code>thermal_resume_s</code>	<code>300</code>	Seconds below the resume threshold before the lockout clears. Minimum 30.
<code>thermal_max_stops_hour</code>	<code>2</code>	Thermal stops per hour after which the automatic restart is disabled. Protection continues.
<code>thermal_test_temp</code>	<code>null</code>	Pretend the CPU is this temperature. For testing only — see section 10 .

The standalone guard additionally uses `start_script`, `stop_script`, `sdr_process_name` and `sdr_base_dir` from the same file.

Overriding a threshold. Only do this if you have a reason. A common one: your machine legitimately runs hot during builds and you would rather not have that count. Raising `thermal_stop` buys you headroom at the cost of margin against the critical point — never set it above your crit value, or the CPU will hit its own emergency shutdown first and the guard becomes decoration.

7. Running it without the admin panel

Plenty of sysops never install the panel. The guard runs perfectly well on its own — same file, same behaviour.

Try it in the foreground first:

```
cd ~/PhantomSDR-Plus
python3 thermal_guard.py --once          # what would happen here?
python3 thermal_guard.py --mode log      # watch it live, Ctrl-C to quit
```

`--mode` overrides `thermal_mode` from the command line, so you never have to write JSON by hand:

```
python3 thermal_guard.py --mode stop+restart
python3 thermal_guard.py --config /etc/phantomsdr/thermal.json
```

Then make it permanent with the sample unit shipped in the repository:

```

sudo cp thermal-guard.service /etc/systemd/system/
sudo nano /etc/systemd/system/thermal-guard.service # set User= and paths
sudo systemctl daemon-reload
sudo systemctl enable --now thermal-guard
systemctl status thermal-guard

```

Read the comments at the top of `thermal-guard.service` before enabling it. Two things matter most:

- **User= must be an account that can actually run your start and stop scripts.** Running the guard as root is usually the wrong answer: it would run those scripts as root too.
- Either point it at an `admin_config.json` containing `start_script` and `stop_script`, or pass `--mode` on the `ExecStart` line. A minimal config is enough:

```

{
  "sdr_base_dir": "/home/youruser/PhantomSDR-Plus",
  "sdr_process_name": "spectrumserver",
  "start_script": "start-rx888mk2.sh",
  "stop_script": "stop-websdr.sh",
  "thermal_mode": "stop+restart"
}

```

Everything the guard does still goes to `crash.log` next to that config file, plus the systemd journal (`journalctl -u thermal-guard`).

8. Enabling the throttle stage without root

By default `python3 thermal_guard.py --once` reports:

```
cpufreq : not writable – throttle stage disabled
```

This is normal and it is not a bug. The kernel creates

```
/sys/devices/system/cpu/cpuN/cpufreq/scaling_max_freq
```

owned by `root:root`, mode `0644`. An admin panel running as an ordinary user cannot write it, so the guard skips the throttle stage and says so in the log rather than failing silently.

The wrong fix is running the guard as root. It also runs your start and stop scripts, and those should stay unprivileged.

The right fix is to hand write access on those specific files to a dedicated group. One script does it:

```

cd ~/PhantomSDR-Plus
./setup-cpufreq-perms.sh

```

It asks for your password once — and only once, at setup — then:

1. creates a system group `cpufreq` and adds you to it;
2. installs `/etc/tmpfiles.d/99-phantomsdr-cpufreq.conf` so the group ownership and `0664` mode are re-applied **on every boot** — `sysfs` permissions do not survive a reboot on their own;
3. applies the change immediately so you can test without rebooting.

Then, in order:

```
# 1. log out and back in (or reboot) – a new group only reaches your
#   processes through a fresh login
id | grep cpufreq

# 2. restart the panel; the guard tests for write access once, at startup
sudo systemctl restart phantomsdr-admin
#   (no logout needed this way – systemd rebuilds the group list at every
#   start. Only if the units are NOT installed: ./manage_admin.sh stop && ./manage_admin.sh start
#   – never both, they fight over port 3000)

# 3. confirm
python3 thermal_guard.py --once
#   cpufreq      : writable, throttle stage available
```

To undo it completely: `sudo ./setup-cpufreq-perms.sh --revoke`.

Notes

- The throttle stage still only acts if `thermal_mode` is `throttle` or higher. Granting permission changes nothing on its own.
- The guard lowers whatever limit is currently in force by 20 % and restores exactly that value afterwards. If you already cap your CPU (for example `MAX_SPEED` in `/etc/init.d/cpufrequtils`), your cap is what gets restored — the guard will not quietly hand you back a higher clock than you asked for.
- Some machines have no `cpufreq` driver exposed at all — many VPS, most containers. The script detects this and refuses to change anything. Use `stop` or `stop+restart` there; they do not depend on `cpufreq`.

9. Unattended operation

The most common question about the guard: *what happens when nobody is watching?* The short answer is that nobody watching is the case it was built for.

It does not need a browser open. The guard is a background thread inside `admin_server.py`, ticked from the graph sampler every 2 seconds. It runs whether or not anyone is logged into the panel, whether or not a browser is open, whether or not you are awake. In standalone mode it is a `systemd` service, which is even more independent. The dashboard card is a *window* onto the guard, not the guard itself.

It acts without asking. There is no confirmation dialog and no notification to wait for. At the stop threshold it stops the server, full stop. That is the entire point.

But only if you gave it permission. In `log` mode it will faithfully record a machine cooking itself and do nothing about it. If the reason you are reading this section is "*I am not around to watch it*", the mode you want is `stop+restart` — it is the only one that both protects the machine and brings the service back without you.

It only helps while the panel is alive. The guard is part of `admin_server.py`, so a panel that died at 02:00 takes the protection with it, and a reboot leaves the machine unguarded until you log in and start it again. If nobody is watching, let `systemd` watch: the repository ships `phantomsdr-admin.service` and `phantomsdr-proxy.service`, which start at boot and restart the panel within five seconds of a crash. For an unattended receiver this is **strongly recommended** — arming `stop+restart` without it protects you only until the next reboot. See [ADMIN_PANEL_SETUP.md](#).

How you find out afterwards. In rough order of usefulness:

Where	What you get
<code>crash.log</code> / the CRASH tab	Every action, timestamped. The permanent record.
Dashboard THERMAL GUARD card	Live state: current temperature, stage, lockout, stops in the last hour.
Graphs page (CPU temperature, 24 h)	The shape of the event — how fast it climbed, how long it took to cool.
<code>journalctl -u thermal-guard</code>	Standalone mode only.

A sensible routine for an unattended site: `stop+restart` armed, and `grep THERMAL crash.log` whenever you happen to log in. If it is empty, there is nothing to know.

A worked example. Fan fails at 03:00 on a machine in `stop+restart`:

```
03:14 temperature crosses 88 °C, holds 30 s → warn logged
03:16 crosses 92 °C, holds 30 s → throttle applied (if permitted)
03:19 crosses 95 °C, holds 60 s → STOP – stop script runs, lockout on
03:19 watchdog restarts the server → guard re-stops it, logs the lockout
03:26 CPU has been under 75 °C for 5 minutes → lockout clears, start script runs
03:41 it overheats and stops a second time → rate limit reached, no more restarts
server stays down, protection stays on
```

You wake up to a WebSDR that is down, a `crash.log` that says exactly why, and — most importantly — a CPU that never got near its critical point.

10. Testing it before you trust it

Do not wait for a real heatwave to find out whether your stop script works.

`thermal_test_temp` makes the guard believe the CPU is at a temperature you choose. Everything else behaves completely normally — the sustain timers, the ladder, the lockout, your actual scripts.

1. Admin panel → **Settings** → **THERMAL GUARD** → **TEST TEMPERATURE**.
2. Enter a value above your stop threshold (for example 97 on an Intel).
3. Save, and watch the dashboard card and the CRASH tab.
4. **Clear the field when you are done.** A test temperature left in place means the guard is no longer reading the real sensor.

In `log` mode this shows you the whole ladder with nothing at risk — the safe first test, and the one to run before arming anything.

In `stop+restart` mode this is a **live test**: your server really will stop, and really will come back about five minutes after you clear the field. Do it when nobody is listening. It is worth doing once, because it is the only way to know that your stop and start scripts work when called by something other than you.

Standalone equivalents:

```
python3 thermal_guard.py --once          # thresholds only, no action
python3 thermal_guard.py --mode log      # watch the ladder live
```

11. Reading the log

Everything is appended to `crash.log` next to the config file, one event per line, each prefixed `[THERMAL]`:

```
grep THERMAL crash.log
```

Lines are deliberately single-line and grep-friendly. What you will see:

Line	Meaning
warn: 88.4C (warn=88.0 crit=100) sustained 30s	First rung. Nothing has been done.
throttle: 92.1C sustained 30s – cpufreq max lowered 20%	CPU clock reduced.
throttle level reached: ... (no cpufreq write access – stage skipped)	See section 8 .
STOP level reached: ... mode is 'log', NOT stopping	It would have stopped the server. Arm it.
STOPPED server: 95.2C sustained 60s (stop=95.0 crit=100) – ran stop-websdr.sh	The real thing.
lockout: process reappeared at 96.0C – re-stopped (...) [12 time(s) so far]	Something is restarting your server; the guard is winning. Rate-limited to one line per minute.
rate limit: 2 thermal stops within the hour – automatic restart is now DISABLED	Stays down for you to investigate. Protection continues.
recovered: 54.0C held below 75.0 – lockout cleared	Cool again.
auto-restart: ran start-rx88mk2.sh	Back online.
back to normal: 59.0C (warn=88.0)	Dropped off the ladder without ever stopping.

12. Lockout and the rate limit

These are the two behaviours that surprise people, so they are worth stating plainly.

The lockout is what makes the guard work against *your* setup specifically. When it stops the server it does not simply issue one stop and hope. It marks itself locked, and on every 2-second tick, while the temperature is still above the resume threshold, it checks whether the server process is back — and if it is, stops it again. Your watchdog, systemd's `Restart=always`, a cron job, an impatient sysop: all of them lose that argument until the CPU cools. It clears itself automatically once the temperature has held below the resume threshold for `thermal_resume_s` (default 5 minutes).

The CLEAR LOCKOUT button on the dashboard card clears the lockout *and* the stops-per-hour counter, for when you have fixed the cooling and do not want to wait. It does **not** disarm the guard: if the machine is still hot, the very next check stops the server again. That is intended.

The rate limit (`thermal_max_stops_hour`, default 2) stops a broken machine from cycling up and down all night. After the limit is reached the guard **disables the automatic restart** and leaves the server down. Read that carefully: it disables the restart, not the protection. The lockout keeps re-stopping the server while it is over-temperature no matter how many times it has already tripped. A guard that went quiet exactly when the machine was at its worst would be worse than no guard at all.

13. Troubleshooting

sensor : NONE – no trusted CPU sensor on this machine The guard found no allow-listed sensor and refuses to guess. Try `sensors` (from `lm-sensors`; `sudo apt install lm-sensors && sudo sensors-detect`). Inside a VM or container there is often genuinely no CPU die sensor exposed — the guard cannot protect that machine and correctly reports itself disabled rather than pretending.

cpufreq : not writable – throttle stage disabled Expected unless you have run `./setup-cpufreq-perms.sh` — see [section 8](#). It only affects the throttle stage; `stop` and `stop+restart` are unaffected.

Ran `setup-cpufreq-perms.sh`, still says not writable Two likely causes: you have not logged out and back in since (a new group only reaches processes through a fresh login — check with `id`), or the panel has not been restarted since (the guard tests for write access once, at startup). Do both, in that order.

Thresholds look wrong for my CPU Check what the machine publishes:

`cat /sys/class/hwmon/hwmon*/temp*_crit`. If your crit is unusual, or absent, set `thermal_stop` and friends to absolute values.

It stopped my server and I do not think it was hot Check whether `thermal_test_temp` is still set from a test. That is by far the most common cause.

It never fires even though the machine gets hot Confirm the mode is not `log`, confirm `thermal_enabled` is `true`, and compare the temperature you are seeing against the thresholds from `--once`. Remember the threshold must be held *continuously* for the sustain time.

The server keeps restarting during a thermal event That is the lockout working, and the `lockout:` lines are it reporting success — your supervisor keeps trying, the guard keeps undoing it. Nothing to fix.

The panel is not running, so is the guard running? No — in panel mode the guard lives inside `admin_server.py`. If you want protection independent of the panel, use the standalone service in [section 7](#).

14. Turning it off or removing it

- **Pause it:** set `thermal_mode` to `log`. It keeps watching and reporting but never acts.
- **Disable it completely:** set `thermal_enabled` to `false`.
- **Standalone:** `sudo systemctl disable --now thermal-guard`.
- **Undo the cpufreq permissions:** `sudo ./setup-cpufreq-perms.sh --revoke`.

Deleting `thermal_guard.py` is also safe — `admin_server.py` imports it inside a `try`, and simply reports the guard as unavailable if it is missing.

See also

- [Admin Panel Setup](#) — the Thermal Guard tab in context
- [Installation](#) — thermal protection during a fresh install
- [Project Structure](#) — where `thermal_guard.py` sits

CHAPTER 5

Connection Limits

CONNECTION_LIMITS.md

Connection Limits — Sysop Manual

Protecting a public receiver from connection floods.

A WebSDR is a public service that hands anyone who asks a continuous stream of audio and spectrum. That is the point of it, and it is also the problem: nothing about a normal listener looks different, to the server, from someone opening sessions in a loop. On 16 September 2026 this receiver was hit by exactly that — one address opened 134 listener sessions inside a single minute and sat on 181 of them at once, while the station's genuine audience was around a dozen.

This manual is about the three layers that stop it, what each one can and cannot do, and how to tell whether yours are set sensibly.

In a hurry? The example `config.toml` already ships with sensible values, so a fresh installation is protected without you doing anything. If your own `.toml` predates this and has no `[limits]` keys beyond `audio/waterfall/events`, the per-IP limits are simply **off** — copy the block from the example to switch them on. The one command worth knowing afterwards is `grep Refused spectrumserver.log`, which tells you whether anyone is being turned away.

Contents

1. What you are actually defending against
2. The three layers
3. Per-IP limits — the listener policy
4. Idle connections — the silent attack
5. The kernel guard — `setup-firewall.sh`
6. Configuration reference
7. What a refused listener sees
8. Reading the logs and counters
9. Choosing your numbers
10. Traps worth knowing about
11. Turning it all off

1. What you are actually defending against

It is worth being precise, because the word "DDoS" covers two very different things and only one of them can be solved on your own machine.

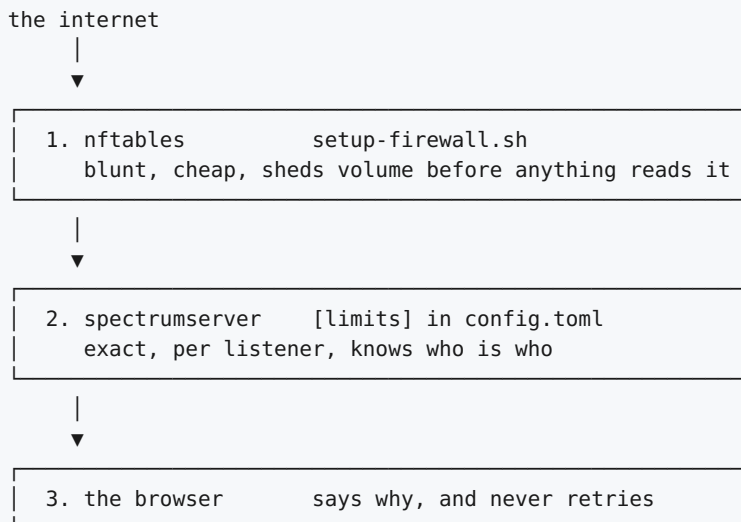
Single-source flooding. One host, or a small handful, opening connections as fast as it can. This is what happened here, and it is entirely fixable: the connections arrive from an address you can count, and counting is enough.

A genuine distributed attack. Thousands of hosts, often with forged source addresses, filling your uplink. By the time those packets reach your network card the bandwidth has already been spent.

Nothing described in this manual helps, and nothing you install on the server can — the only answer is a service upstream of you, which is a different conversation involving a real domain and a provider such as Cloudflare in front of the receiver.

Everything below addresses the first case. That is not a limitation worth apologising for: the first case is what actually happens to amateur stations.

2. The three layers



They are deliberately not redundant. The kernel layer is fast but stupid — it counts packets from an address and knows nothing about listeners. The server layer is precise but only sees a connection once it has been accepted and parsed, which costs something. The browser layer exists so that a refused listener is told what happened instead of staring at a page that never loads.

3. Per-IP limits

What counts as one listener

A listener is **one /audio connection** — or, where [kiwi_emulation] is enabled, **one Kiwi sound connection**, which counts exactly the same. That is the same unit the user count, `users.json` and the labels on the waterfall already use, so a Kiwi client consumes a slot and appears in the user list just as a browser does.

This matters because a single desktop browser tab opens **four** sockets to your server:

Socket	Purpose
/audio	the demodulated audio stream — this is the listener
/waterfall	the spectrum display
/events	user list, other listeners' frequencies
/chat	the chat panel

A phone on `/mobile` opens two (`/audio` and `/chat`), because it has no waterfall. So a limit that counted raw sockets would refuse everybody: three tabs is twelve sockets.

The three keys

```
[limits]
per_ip=3          # simultaneous listeners from one address
per_ip_sockets=0  # simultaneous sockets of any path; 0 derives per_ip * 4 + 4
per_ip_rate=40    # new sockets per minute from one address
per_ip_ban_s=600  # how long an address that breaks the rate is refused
```

`per_ip` is the headline number and the one you will think about. Three is a good starting point.

`per_ip_sockets` exists because `/audio` is not the only thing worth flooding. `/waterfall` is the heaviest stream your server produces, and a limit that counted only listeners could be sidestepped simply by opening waterfall sockets instead. Left at 0 it becomes `per_ip * 4 + 4`, which fits `per_ip` full desktop tabs with a little slack for sockets the kernel has not finished reaping after a page reload.

`per_ip_rate` is what actually ends a flood. A cap alone does not: refusing a connection is cheap but not free, and an attacker can keep reconnecting as fast as the refusals come back, burning your CPU on setup and teardown. The rate limit gives them a ban instead of an answer. Remember one tab costs four, so 40/minute is about ten page loads a minute from one address.

A deliberate ordering. The rate is charged *before* the simultaneous cap is tested. This looks backwards until you consider an address already sitting at its cap and reconnecting in a loop: if the cheap "you are at the cap" refusal answered it every time, the rate violations would never accumulate and the ban that ends the loop would never trigger.

Who is exempt

Loopback is never limited. The autorun spot-reporting tap, the admin panel and a browser opened on the server machine itself all arrive from `127.0.0.1`, and letting them consume slots would mean your own station locking out your listeners.

This also covers `proxy.py`: connections through the proxy reach `spectrumserver` from loopback, and the real client address arrives in the `X-Forwarded-For` header, which is what the limits are keyed on.

Since v5.0.0 that header is believed **only from a loopback peer** — a proxy on the same machine. Sent by anyone else it is ignored and the TCP address counts, because otherwise a direct visitor could write `X-Forwarded-For: 127.0.0.1` and pass as loopback, past every limit on this page. `proxy.py` also drops any `X-Forwarded-For` or `X-Real-IP` a client sends before adding its own, and answers the sysop kick (`/~~kick`) only for clients on the same machine. A station running several receivers behind the proxy is described in [Several Receivers](#); the limits there apply per receiver.

4. Idle connections

There is an attack that none of the above can see.

Open a TCP connection. Send nothing. Hold it.

Such a peer never completes a request, so it never reaches the code where the per-IP limits live — no listener exists, no client object exists, nothing has a name to count. Meanwhile it costs the attacker a few bytes and costs you a file descriptor and a slot for as long as it is held.

```
[limits]
idle_per_ip=8      # idle (pre-request) connections per address
idle_total=512     # ceiling across all addresses
```

The limit is on **how many**, not **how long**, and that distinction is the whole design. **There is no deadline on an idle connection, and deliberately so.** A deadline seems the obvious answer and is a trap (see §10): there are legitimate peers that connect and then sit silent for a long time before speaking. What separates them from an attack is quantity — a legitimate peer holds one such connection, an attacker holds thousands.

An idle connection therefore lives until it makes a request or dies, and its slot is released either way.

`idle_total` bounds a flood spread across many source addresses. When the ceiling is reached, **new** idle connections are refused rather than established ones evicted, so a legitimate long-lived connection is never what gets dropped to make room.

Both are enforced at the moment the socket is accepted, using the peer address — there is no request yet, so there is no `X-Forwarded-For` to consult. That is fine: loopback is uncounted, which covers everything arriving through `proxy.py`, and a direct flood carries its own address.

5. The kernel guard

`setup-firewall.sh` installs an `nftables` table that sheds volume before `spectrumserver` ever reads a byte. It is deliberately blunt, and its numbers sit well above the ones in `config.toml`, so a normal listener can never be caught by both.

```
./setup-firewall.sh --show      # print the rules, change nothing
sudo ./setup-firewall.sh --check # validate against your kernel
sudo ./setup-firewall.sh --apply # load, with a 60-second auto-rollback
sudo ./setup-firewall.sh --persist # reload at boot
sudo ./setup-firewall.sh --status # the per-rule packet counters
sudo ./setup-firewall.sh --remove # undo all of it
```

It covers four things: a concurrent-connection ceiling per source address on your proxy and receiver ports, a new-connection rate per source, an SSH brute-force brake, and Windows file sharing closed to everything outside private address ranges — `smbd` listens on `0.0.0.0` on many machines, and whether that is reachable depends on a router the script cannot see.

Ports are read from `admin_config.json`, the same file `proxy.py` uses, so it follows your setup.

Why it cannot lock you out

Two properties, both deliberate:

- The table's policy is **accept**, and it only ever drops specific named patterns. A rule that fails to match lets the packet through to whatever else you have configured. It cannot black-hole the machine.
- **Established connections are accepted in the very first rule.** The SSH session you are typing in is never affected by anything that follows.

On top of that, `--apply` arms an automatic rollback: load the rules, then confirm within 60 seconds or they are removed again. Say nothing, close the terminal, lose your connection — the ruleset disappears by itself.

Use that window properly. Check **from another device, off your own network** that the receiver still loads and that a *new* SSH session still opens. Testing with the session you already have proves nothing, because it was accepted by rule one.

6. Configuration reference

Every key below lives under `[limits]` in your `.toml` file. **All of them default to off or to generous values**, so a configuration that does not mention them behaves exactly as it always did.

Key	Default	Meaning
<code>per_ip</code>	0 (off)	Simultaneous listeners per address
<code>per_ip_sockets</code>	$0 \rightarrow \text{per_ip} * 4 + 4$	Simultaneous sockets of any path per address
<code>per_ip_rate</code>	0 (off)	New sockets per minute per address
<code>per_ip_ban_s</code>	600	Seconds an address is refused after breaking the rate
<code>idle_per_ip</code>	8	Idle, pre-request connections per address
<code>idle_total</code>	512	Idle connections across all addresses

Setting any key to 0 disables that particular check.

The three that are not limits

`[limits]` also holds `audio`, `waterfall` and `events`, inherited from upstream PhantomSDR, and **none of the three is enforced**. `waterfall` and `events` are read by no code at all. `audio` is read in exactly one place: it is posted as `max_users` in the registration JSON sent to the directories in `[websdr] register_urls`, so it is the figure `sdr-list.xyz` and the others display, and it does nothing whatsoever when `register_online=false`. If you want a cap on listeners, `per_ip` above is the one that acts.

One more, under `[server]`

```
[server]
min_client_version=0
```

Not a limit but a refusal, and it shares the same close code, so it belongs here. Pages announce the build they were loaded from as `/audio?v=N`, and anything below `min_client_version` is turned away with *"this page is out of date — please reload it"*.

It exists because the server cannot reach JavaScript that is already running in somebody's browser. A tab keeps the code it loaded until someone reloads it, so a change to the page only reaches listeners who happen to reload — and a station that has just changed how sessions behave may need the old pages gone now, not eventually.

Only `/audio` is checked, because that is the socket a session lives on, and loopback and the Kiwi paths are exempt so the autorun tap and Kiwi clients are untouched. 0 disables it and is the default. Leaving it on once the old pages have drained is fine: a current diversity client that is refused on a bare `/audio` retries with the marker, so what is still turned away is a diversity client on an older PhantomSDR-Plus build and any third-party tool that opens `/audio` without `?v=`. Set it back to 0 if another station reports that it can no longer use you as a diversity partner.

7. What a refused listener sees

Nothing ever reconnects

Start here, because it governs everything else: **a dropped `/audio` socket ends the session**. There is no automatic reconnection, by design.

A reconnect is the right instinct for a chat application and the wrong one for a receiver. `/waterfall` and `/events` never came back with it, so a retried session was a live audio socket bolted to a frozen waterfall; and stopping the server no longer cleared it, because every listener was back a few seconds later, filling the user list with sessions nobody was really in. The page goes quiet and the listener loads it again — which is what this receiver did before a reconnect was ever added.

For the limits in this manual that also removes a trap: against a rate limit, an automatic retry would extend the very ban it was trying to get around.

The two close codes

Code	Meaning	Who sends it
4003	refused — over a per-IP limit, or the page is out of date	spectrumserver
4001	kicked by the sysop	spectrumserver, when the admin panel calls <code>/~~kick</code>

Both are in the 4000–4999 private range, so neither can be confused with a protocol status, and both are treated as final by every page. The reason text travels in the close frame rather than as a message, because the first frame on `/audio` is reserved for the receiver's settings and anything sent ahead of it would break every normal connection.

On the page

Refused as the page loads — the listener never gets going, so both pages explain themselves. The desktop shows a panel: *"This receiver refused the connection"*, the reason, and the note that everyone sharing an internet connection counts as one address. `/mobile` shows the same in its notice area.

Refused or kicked mid-session — the desktop page simply stops, deliberately without a message: the waterfall's draw loop halts and that is all. `/mobile` shows one line telling the listener to reload, except after a kick, where it too stays silent.

That silence after a kick is intentional. **Kicking is the sysop ending a session, not a punishment** — nothing is announced, and whoever wants back in loads the page again and is an ordinary listener one second later. The only thing the 4001 code buys is that the browser knows this was not a dropped connection, so the kick actually takes instead of undoing itself.

The kick is done by spectrumserver itself, not by proxy.py: listeners connect to the receiver's own port directly, so the proxy owns none of their sockets and its version of a kick reaches nobody. The panel calls `/~~kick?ip=&secs=`, which is loopback-only and checks the real TCP peer rather than `X-Forwarded-For`, that being client-supplied. `secs` defaults to zero — no ban — but it accepts a length if you do want the address kept out for a while, and that reuses the same ban machinery as `per_ip_rate` so the reconnect is refused at the door.

8. Reading the logs and counters

The server

```
grep Refused spectrumserver.log
```

```
Refused /audio from 203.0.113.5: too many simultaneous connections
Refused /audio from 198.51.100.7: too many connection attempts
```

Nothing there means nobody is being turned away. Logging is throttled to at most one line per address every five seconds — without that, an attack would turn into unbounded disk writes, which is simply a slower way of taking the station down.

The kernel

```
sudo ./setup-firewall.sh --status
```

Read the **counter packets N** on each rule; that is the number actually dropped. All zeros means nothing has been blocked and your thresholds are comfortably loose.

Do not be alarmed by the `elements = { ... }` lists in the sets above the rules. Those are your ordinary listeners being *tracked* against the limits — an entry appears on an address's first connection and ages out a couple of minutes later. Tracking is not blocking.

9. Choosing your numbers

Guessing is unnecessary; your own logs will tell you. This is the measurement that set `per_ip=3` here, run over a week of `logs/users_*.jsonl`:

Concurrent sessions from one address	Number of addresses
1	682
2	54
3	13
more than 3	8

758 distinct addresses over seven days, and a limit of three would have touched eight of them. The flood sat at 181.

To repeat it on your own station, pair the session ids per address in `logs/users_*.jsonl` and take the maximum overlap. Note that the log records `tune` and `disconnect` events but has no `connect` event, so a session's start has to be inferred from its first `tune`.

Remember NAT. A club, a school, an office or a mobile carrier appears as a single address, and everyone behind it shares one allowance. This is inherent to any limit keyed on an address; the only mitigation is choosing a number you are comfortable with. If a listener ever reports being unable to connect, this is the first thing to suspect — check `grep Refused` before anything else.

10. Traps worth knowing about

Never put a deadline on an idle connection

There is no `handshake_s` knob any more, and this is why. A 30-second deadline on idle connections looks like the obvious defence against a slowloris. It was built, switched on, and produced this within a minute:

```
Idle deadline: closed 1 connection(s) idle past 30s: 192.87.173.88
```

192.87.173.88 is `etgd-websdr.ewi.utwente.nl` — **websdr.org's own callback host**. It connects back to your server and then sits idle, for well over 30 seconds, before sending its `GET /~~orgstatus`. The deadline cut it every time and registration stopped. The setting was removed rather than left as an option, because a footgun that breaks your directory listing is not worth the sixty lines that implement it.

The same reasoning applies to websocketpp's `open_handshake_timeout`, which is still there. The ten-minute value in `spectrumserver.cpp` looks absurd and is load-bearing: it is what allows the websdr.org callback to sit idle at all. There is a second reason too — the `/~~orgstatus` handler answers from a thread holding a `dup()` of the socket, and websocketpp ends a timed-out connection with `shutdown()`, a socket-level call that reaches through the duplicate and would break the callback even if the idle time were not an issue.

If you need to bound silent connections, bound their number, not their age. That is what `idle_per_ip` and `idle_total` are.

An nftables file must be idempotent

`nft -f` **appends** to a table that already exists rather than replacing it. A ruleset file without a `delete table` prologue therefore doubles the entire chain every time it is reloaded — which is what a naive systemd unit does on every service restart. `setup-firewall.sh` generates the prologue for you; if you write your own rules, do the same.

```
sudo ./setup-firewall.sh --status | grep -c dport    # should be 8, not 16
```

11. Turning it all off

The server limits: set the keys to `0`, or delete them from your `.toml`, and restart the receiver. There is no other state; nothing persists across a restart.

The firewall:

```
sudo ./setup-firewall.sh --remove
```

That deletes the table, removes the saved ruleset and disables the boot service. Your machine returns to exactly the state it was in before — which, on most installations, means no firewall at all. That is worth thinking about before you remove it.

CHAPTER 6

Several Receivers

MULTI_RECEIVER.md

Several Receivers — Sysop Manual

Running two or more receivers on one computer, with a receiver picker on the page.

Since v5.0.0 one PhantomSDR-Plus computer can run several receivers at the same time — for example an RX-888 for HF and an RTL-SDR for 2 m — and listeners switch between them with buttons in the page header, the way OpenWebRX offers its profiles. Every receiver keeps its own waterfall, chat, markers, listener list and station details; nobody on one receiver is ever disturbed by someone on another.

In a hurry? A station with one receiver needs none of this and sees no change. For a second receiver: create `instances/<name>/` with its own `config.toml`, start it with `INSTANCE=<name> ./start-<radio>.sh`, list both receivers in `receivers.toml`, and restart the proxy. The worked example in [section 3](#) goes through it line by line.

The easiest way: run `./add-receiver.sh` (the installers offer it at the end). It asks which receiver and what it should cover, installs its driver, writes everything described in [section 3](#) and [section 4](#), and offers to start the receiver and restart the proxy. The rest of this manual explains what it does, for when you want to change something by hand.

Contents

1. [How it works](#)
2. [Two ways to publish the receivers](#)
3. [Adding a second receiver](#)
4. [receivers.toml](#)
5. [Starting and stopping](#)
6. [What listeners see](#)
7. [The S-meter above 30 MHz](#)
8. [Security](#)
9. [How many receivers fit](#)
10. [Traps worth knowing about](#)

1. How it works

Each receiver is a complete spectrumserver of its own, with its own SDR front end, its own configuration and its own internal port. They never share samples, settings or listeners. What ties them together is `proxy.py`, the reverse proxy that already serves the admin panel: it reads `receivers.toml` and sends every request to one receiver.

```

listeners → proxy.py {
    → :8900 spectrumserver RX-888 (HF, the main receiver)
    → 127.0.0.1:9002 spectrumserver RTL-SDR (2 m, instance "vhf")

```

The proxy decides by these, in order:

1. `?rx=<id>` in the address — `http://your.host:8899/?rx=vhf`. The page adds it to every socket, poll and link it opens, so a tab stays on its receiver.
2. The `rx` cookie, which the proxy sets whenever a request carries a valid `?rx=`.
3. The Host header, if a receiver lists that name under `hostnames`.
4. Otherwise the default receiver.

The **main receiver** is the one started the usual way (`./start-rx888mk2.sh`). Every other receiver is a **named instance**: started with `INSTANCE=<name>`, configured in `instances/<name>/`, and given its own log (`logwebsdr-<name>.txt`), server log, watchdog lock and FIFO. Processes are told apart by a `PHANTOMSDR_INSTANCE` tag in their environment, not by their name, so two receivers may run the very same program — two `rtl_sdr`, or `rx_sdr` for both an RSP1A and an Airspy — and a restart of one never touches the other.

2. Two ways to publish the receivers

A — the main receiver keeps its own port. The main receiver stays exactly where it was (say :8900), listeners there notice nothing, and the other receivers are reached through the proxy's port (`proxy_port` in `admin_config.json`, say :8899) with `?rx=`. Nothing has to be moved and nothing goes off the air; it does mean two public ports.

```
HF: http://your.host:8900/
2 m: http://your.host:8899/?rx=vhf
```

B — one public port for everything. The proxy takes over the public port and the main receiver moves to an internal one. Set `[front] port` in `receivers.toml`, move the main receiver's `[server] port` and set `host = "127.0.0.1"` on it, set `public_port` in `admin_config.json` to the new internal port, and add `public_port = <the public port>` under `[websdr]` so the directory listing keeps announcing the right one. Everything then lives behind one port; the cost is a short outage while you move it.

```
HF: http://your.host:8900/
2 m: http://your.host:8900/?rx=vhf
```

Either way the receiver picker works the same. Layout A is the safer start; B can follow later without changing anything for the second receiver.

3. Adding a second receiver

add-receiver.sh does all of the steps below — run it and answer its questions; it also installs the driver (`setup-rtlsdr.sh` for an RTL-SDR, the matching `setup-*.sh` for the others), picks a free internal port and names the receiver after what it covers. The steps are written out here so that you can check its work or change it later.

The example adds an RTL-SDR Blog V4 for 2 m as instance `vhf`, next to an RX-888 that keeps port 8900 (layout A).

1. The driver. The RTL-SDR Blog V4 needs the RTL-SDR Blog driver rather than the distribution's `rtl-sdr` package, and the kernel's DVB-T driver must let go of the stick. `install.sh` (receiver option 2, "RTL-SDR Blog V4: yes") does both. Check with `rtl_test -t`, which must say `RTL-SDR Blog V4 Detected`.

2. The instance folder. Everything that belongs to the receiver lives in `instances/vhf/`, which is never committed and never overwritten by an update:

```
instances/vhf/
├─ config.toml          its spectrumserver configuration (required)
├─ instance.env         optional: receiver arguments and CPU pinning
├─ markers.json         its own markers
└─ www/                 its own copies of the page's per-station files
    ├─ site_information.json
    └─ wf-message.json
```

spectrumserver runs with `instances/vhf/` as its working directory, so its chat history, markers, FFTW wisdom and `logs/` are its own as well.

3. config.toml. Start from `config-rtl.toml` and change:

```
[server]
port=9002                # its own internal port
host="127.0.0.1"         # reachable only through the proxy
html_root="www/"         # its own few files...
html_fallback_root="../../frontend/dist/" # ...and the shared page for the rest

[input]
sps=2400000              # must match -s in instance.env
frequency=145000000      # must match -f in instance.env

[input.defaults]
frequency=145500000
modulation="FM"

[kiwi_emulation]
enabled = false          # see the traps below
```

`html_fallback_root` lets the instance keep only its own `site_information.json` and `wf-message.json` while everything else — the built page itself — comes from `frontend/dist`, so one frontend build reaches every receiver. A symlinked copy of `frontend/dist` does not work: the server refuses files that resolve outside its own folder.

4. instance.env. Read by the launcher after its own settings:

```
RX_ARGS="-f 145000000 -s 2400000 -g 29.7 -" # -f and -s match config.toml
SPECTRUM_CORES=8-11                       # keep it off the main receiver's cores
```

5. Its page details. Copy `frontend/site_information.json` to `instances/vhf/www/` and edit it: `siteReceiver`, `siteAntenna`, `siteSDRBaseFrequency` and `siteSDRBandwidth` (these choose which band buttons appear), and the keys described in [section 6](#).

6. Start it.

```
INSTANCE=vhf ./start-rtl.sh
```

7. List it in receivers.toml and restart the proxy (`sudo systemctl restart phantomsdr-proxy`). The second receiver is now at `http://your.host:8899/?rx=vhf`, and both pages show the picker.

4. receivers.toml

Copy `receivers.toml.example` to `receivers.toml`. Without that file the proxy serves a single receiver exactly as it always has.

```
# [front]
# port = 8900          # layout B only: the proxy also listens on this public port

[[receiver]]
id      = "hf"
name    = "HF 0-30 MHz (RX-888 MkII)"
port    = 8900
default = true
launcher = "start-rx888mk2.sh"
url      = "http://your.host:8900/"

[[receiver]]
id      = "vhf"
name    = "2 m (RTL-SDR Blog V4)"
port    = 9002
launcher = "start-rtl.sh"
instance = "vhf"
url      = "http://your.host:8899/?rx=vhf"
```

Key	Meaning
<code>id</code>	Short name used in <code>?rx=</code> ; letters, digits, <code>-</code> and <code>_</code>
<code>name</code>	The text on the receiver's button
<code>port</code>	Its spectrumserver's <code>[server] port</code>
<code>host</code>	Optional; where that spectrumserver runs (default <code>127.0.0.1</code>)
<code>default</code>	The receiver for requests that name none; at most one
<code>launcher</code>	Its <code>start-*.sh</code> , for <code>start-all.sh</code> ; leave it out for a receiver this computer does not start
<code>instance</code>	Its instance name; leave it out for the main receiver
<code>url</code>	Where its button sends a listener; without it, <code>/?rx=<id></code> on the proxy
<code>hostnames</code>	Optional DNS names routed straight to this receiver

The proxy also serves the list as `/receivers.json` (ids, names and links — never the internal ports), which is what the picker reads.

5. Starting and stopping

Command	Does
<code>./start-rx888mk2.sh</code>	Starts or restarts the main receiver
<code>INSTANCE=vhf ./start-rtl.sh</code>	Starts or restarts the <code>vhf</code> receiver
<code>./start-all.sh</code>	Starts every receiver in <code>receivers.toml</code> that has a <code>launcher</code>
<code>./stop-websdr.sh vhf</code>	Stops only <code>vhf</code>
<code>./stop-websdr.sh main</code>	Stops only the main receiver
<code>./stop-websdr.sh</code>	Stops every receiver, as it always did

While a receiver is stopped, its button disappears from the picker — the proxy lists only the receivers that answer, and open pages re-read the list once a minute — and comes back when the receiver starts again; with only one receiver left running, the *Receivers:* line disappears altogether. A stopped receiver does come back by itself whenever every receiver is started: `./start-all.sh`, the admin panel's **Restart** and a thermal-guard restart all start everything listed in `receivers.toml`. To keep one off for good, remove (or comment out) its `[[receiver]]` block and restart the proxy.

Set the admin panel's **Default start script** to `start-all.sh`. Its Stop already stops every receiver, and its Restart and the thermal guard use the start script, so with `start-all.sh` they bring back every receiver instead of only the main one.

The RADE sidecar belongs to the main receiver only. A named instance never starts or stops it.

6. What listeners see

The receiver picker. On a station with more than one receiver the page header gets a line of its own, *Receivers:*, with one button per receiver; the current one is shown in yellow. The /mobile page has the same buttons as a second row in its top bar, and they open the other receiver's /mobile page.

Each receiver's own details. A page opened on a second receiver loads that receiver's `site_information.json` before it starts, so the band buttons, the starting frequency, the users list, the station details and the Receiver and Antenna links are all its own. A plain visit to the main receiver makes no extra request.

Keys in `site_information.json` for this:

Key	Meaning
<code>siteReceiverId</code>	The receiver this page belongs to (its <code>id</code>); leave it out on the main receiver
<code>siteReceiversList</code>	Where the picker reads the list, e.g. <code>http://your.host:8899/receivers.json</code> ; needed on a receiver whose page is not served through the proxy
<code>siteReceiverURL</code>	Where the <i>Receiver</i> name in <i>Open Additional Info</i> links to
<code>siteAntennaURL</code>	Where the <i>Antenna</i> name links to; "" shows the name without a link

When the picker stays hidden. The picker only shows a list that names the host the page was opened on. Opened by LAN address (`http://192.168.1.10:8900/`) it stays hidden; opened as `http://your.host:8900/` it appears. That is deliberate: a station that copied someone else's `site_information.json` without editing it never shows that station's receivers as its own.

7. The S-meter above 30 MHz

The IARU Region 1 standard puts S9 at -73 dBm below 30 MHz and at **-93 dBm above**, six dB per S-unit on both. The S-meters (the analog needle, the digital bar and the /mobile bar) follow the tuned frequency: above 30 MHz they read VHF S-units, below it they read as before. The dBm and dBμV figures are never changed. If you prefer the meters to rest at 0 on an empty channel, like a VHF/UHF transceiver's, add `"siteSMeterGateDb": 6` to the receiver's `site_information.json` (no rebuild needed): above **60 MHz** the needle and the bar then come up only when a signal stands that many dB above the noise in its own neighbourhood (the passband's strongest point against the spectrum ± 100 kHz around it). It is **off by default** — a calibrated meter showing the real band noise is the more honest reading. The /mobile page, which has no waterfall, estimates the noise from the received level itself.

Calibrate a VHF receiver's dBm with `analog_smeter_offset` (needle and figures) and `smeter_offset` (digital bar) under `[input]` in its `config.toml`, then restart that receiver. Set the receiver's gain first: an RTL-SDR reports levels relative to its own full scale, so every gain change moves the readings. Without a signal generator, a 50 Ω terminator in place of the antenna, USB with a 2.7 kHz filter, should read about -136 dBm (thermal noise in 2.7 kHz is -139.7 dBm, plus the stick's noise figure).

For an RTL-SDR, `add-receiver.sh` writes a starting calibration: a fixed gain of `-g 29.7` in `instance.env` (the tuner's automatic gain cannot be calibrated) and `analog_smeter_offset=-55`, `smeter_offset=-38`, measured on an RTL-SDR Blog V4 at that gain with a -71 dBm (63 μ V) signal. Another stick of the same model reads within a few dB of that; check it with a known signal, and recalibrate if you change the gain.

8. Security

Putting the proxy in front of listeners closed two holes, both fixed in v5.0.0:

- **The sysop kick through the proxy.** `spectrumserver` allows `/~kick` only from its own computer, and to it every request relayed by the proxy comes from its own computer. Anyone who could reach the proxy port could therefore disconnect and ban any listener. The proxy now answers `/~kick` only for clients on the same computer.
- **Faked client addresses.** `spectrumserver` believed an `X-Forwarded-For` header from anyone, so a visitor could claim to be `127.0.0.1` and walk past the per-IP limits. The header is now believed only from a local proxy, and the proxy drops any copy a client sends before adding its own.

Bind every receiver that is reached through the proxy to `127.0.0.1` (`[server] host`), so its port cannot be used around the proxy. The per-IP limits in [Connection Limits](#) apply per receiver.

9. How many receivers fit

The software sets no limit; the hardware does. Narrowband receivers are cheap: an RTL-SDR at 2.4 Msps uses about 5–10 % of one core, a few dozen MB of memory, and about 38 Mbit/s of USB.

- **USB 2.0 is the usual ceiling.** Every USB 2 device shares one 480 Mbit/s bus, even in a blue USB 3 socket. Three to four RTL-SDRs per bus is a safe figure; add them one at a time and watch every waterfall for gaps.
- **Power:** use a powered hub beyond two sticks.
- **A second wideband receiver** (another RX-888, or a HackRF at 20 Msps) is a different matter: it competes with the first for USB 3, CPU and GPU.
- **Listeners cost more than receivers.** Each listener costs CPU on the receiver they are on and upload bandwidth, so the total number of listeners matters more than the number of receivers.

10. Traps worth knowing about

- **Two sticks of the same model have the same serial.** Every RTL-SDR Blog V4 reports `00000001`. Give each its own with `rtl_eeprom -s <serial>` (one stick plugged in at a time) and name it in `instance.env` (`RX_ARGS="-d <serial> ..."`), or after a reboot the sticks may swap receivers.
- **Kiwi clients cannot choose a receiver.** A KiwiSDR client dials a bare address and port; through the proxy it always lands on the default receiver. Keep `[kiwi_emulation]` off on the others.

- **Cookies are per host, not per port.** A visitor who used the second receiver on `:8899` carries `rx=vhf` to `:8900` too. That is why a page decides which receiver it is from `siteReceiverId`, never from the cookie.
- **Directory listings.** Give each receiver its own name in `[websdr]`, and on a receiver behind the proxy set `[websdr] public_port` to the port listeners use, or the listing advertises the internal one.
- **A frontend rebuild briefly shows "Not Found".** `frontend/dist` is rebuilt in place, so for a few seconds during a build the page is missing for every receiver.
- **Nothing starts the receivers at boot** unless you arrange it. After a reboot, `./start-all.sh` brings them all back.

CHAPTER 7

Statistics Server

sdr-stats/README.md

System Stats Installation Guide

A comprehensive guide to add real-time server monitoring to your PhantomSDR application using the automated installation script.

Overview

This feature adds a **Stats** button to your PhantomSDR interface that displays real-time system information:

- CPU usage, cores, frequency, temperature, and top processes
- Memory usage (used/total/percentage)

Stats automatically update every 5 seconds while the modal is open.

Not the same as the admin panel's Graphs page. This modal is a live snapshot that refreshes while it is open and keeps no history. The admin panel has a separate **Graphs** page that samples CPU frequency, load, temperature and users online every 2 seconds and plots them over the last 15 minutes to 24 hours — see the [Admin Panel guide](#).

Prerequisites

- **Linux server** (Ubuntu/Debian recommended)
- **Internet connection** (for downloading Node.js and packages)
- **Sudo access** (for systemd service setup)
- **PhantomSDR-Plus** already installed and running
- **Im-sensors** (recommended for Intel/AMD CPU temperature - automatically installed by script)

Installation Methods

NOTE

The main installer already offers this. `./install.sh` — and the four distro installers — run `install-stats-server.sh` by default as part of the normal PhantomSDR-Plus installation, so on a fresh machine the stats server is usually already in place. Use this page to install it separately, or to change its port, address or service afterwards.

Option 1: Automated Installation (Recommended)

The automated script handles everything for you.

Step 1: Run the script

```
cd ~/PhantomSDR-Plus
chmod +x install-stats-server.sh # only if it is not already executable
./install-stats-server.sh
```

IMPORTANT: Do NOT run as root or with sudo. Run as your regular user.

Step 2: Follow the interactive prompts

The script will guide you through:

1. Node.js Installation (if needed)

```
△ Node.js / npm is not installed
Would you like to install Node.js 22 via nvm now? (y/n): y
```

The script installs Node.js 22 through nvm, on any Linux distribution — no root and no apt source needed. It asks the same question when Node.js is present but older than 22.

2. Installation Directory

```
Installation directory [/home/user/sdr-stats-server]:
```

Press Enter for default, or specify a custom path.

3. Port Configuration (Enhanced!)

```
Port Configuration:
  ▫ Default port: 3001
  ▫ Common alternatives: 8080, 5000, 8888
Enter port number [3001]:
```

The script validates your port:

- Checks if it's a valid number
- Warns if port < 1024 (requires root privileges)
- Validates range (1-65535)
- Checks if port is already in use
- Shows which process is using it if occupied
- Allows you to choose a different port

Example scenarios:

Using default port:

```
Enter port number [3001]: ↵
✓ Port 3001 selected
```

Choosing custom port:

```
Enter port number [3001]: 8080
✓ Port 8080 selected
```

Port already in use:

```

Enter port number [3001]: 3001
△ Port 3001 is currently in use
Process using port 3001:
node    12345 user    20u  IPv6 123456  TCP *:3001 (LISTEN)
Choose a different port? (y/n): y
Enter port number [3001]: 3002
✓ Port 3002 selected

```

Invalid port:

```

Enter port number [3001]: abc
x Port must be a number
Enter port number [3001]: 99999
x Port must be between 1 and 65535
Enter port number [3001]: 3001
✓ Port 3001 selected

```

4. Server Address

```

What is your server's public address?
Examples: Your_site_IP, 192.168.1.100, localhost
Server address: Your_site_IP

```

Enter your server's public domain or IP address.

5. Confirmation

```

Please confirm your settings:
  Installation directory: /home/user/sdr-stats-server
  Port: 3001
  Server address: Your_site_IP
  Stats URL will be: http://Your_site_IP:3001
Continue with these settings? (y/n): y

```

6. Systemd Service Setup (Optional)

```

Would you like to set up the server as a system service (auto-start on boot)? (y/n): y

```

Recommended: Choose **y** to have the server start automatically on boot.

Step 3: Installation Complete! ✓

The script will display a summary:

```

Installation Complete! ✓
Open the port in the router!

```

```

Configuration Summary:
  Installation: /home/user/sdr-stats-server
  Port: 3001
  Server: Your_site_IP
  API URL: http://Your_site_IP:3001/api/system-stats

```

Configure Your PhantomSDR Application

After the script completes, you need to update your Svelte application.

Step 1: Update `site_information.json`

Since `site_information.json` was already edited during the initial setup, you only need to add the new `siteStats` line with your port:

```
{
  "siteSysop": "your name or callsign",
  "siteSysopEmailAddress": "mail@mail.net",
  "siteGridSquare": "QTH locator",
  "siteCity": "City Country",
  "siteInformation": "https://github.com/svlbtl/PhantomSDR-Plus",
  "siteHardware": "Hardware you are using, ",
  "siteSoftware": "Software you are using",
  "siteReceiver": "Receiver model",
  "siteAntenna": "Receiving Antenna.",
  "siteNote": "This is a bright new open-source WebSDR project, under active development.",
  "siteIP": "http://Your_site_IP:port",
  "siteStats": "http://Your_site_IP:3001", ← ADD THIS LINE (use your port)
  "siteSDRBaseFrequency": 0,
  "siteSDRBandwidth": 30000000,
  "siteRegion": 1,
  "siteChatEnabled": true
}
```

Note: Use the port you selected during installation!

Step 2: Update `App.svelte` (the latest GitHub version already includes the updated file)

Make these **4 changes** to your `App.svelte` file:

Change 1: Ensure that `siteStats` is declared (around line 50)

```
import {
  siteSysop,
  siteSysopEmailAddress,
  siteInformation,
  siteGridSquare,
  siteCity,
  siteHardware,
  siteSoftware,
  siteReceiver,
  siteAntenna,
  siteNote,
  siteIP,
  siteStats, // ← ADD THIS LINE
  siteSDRBaseFrequency,
  siteSDRBandwidth,
  siteRegion,
  siteChatEnabled,
} from "../site_information.json";
```

Change 2: Update systemStats object (around line 530)

```
let systemStats = {
  cpu: { usage: 0, cores: 0, temperature: null, frequency: null, topProcesses: [] }, // ← ADD
  topProcesses: []
  memory: { used: 0, total: 0, percent: 0 }
};
```

Change 3: Update fetch URL (around line 541)

```
async function fetchSystemStats() {
  try {
    const response = await fetch(`${siteStats}/api/system-stats`); // ← CHANGE THIS LINE
    if (response.ok) {
      systemStats = await response.json();
    } else {
      console.error('Failed to fetch system stats:', response.statusText);
    }
  } catch (error) {
    console.error('Error fetching system stats:', error);
  }
}
```

Change 4: Add top processes display (in CPU section, around line 3838)

Add this code after the temperature section:

```
{#if systemStats.cpu.frequency}
<div style="display: flex; justify-content: space-between; margin-bottom: 0.25rem;">
  <span>Frequency:</span>
  <span style="color: #4ade80;">{systemStats.cpu.frequency.current} GHz{#if
systemStats.cpu.frequency.max} <span style="opacity: 0.7;">(max {systemStats.cpu.frequency.max})</
span>{/if}</span>
</div>
{/if}

{#if systemStats.cpu.temperature !== null}
<div style="display: flex; justify-content: space-between; margin-bottom: 0.25rem;">
  <span>Temperature:</span>
  <span style="color: {systemStats.cpu.temperature > 70 ? '#fbbf24' :
'#4ade80'};">{systemStats.cpu.temperature}°C</span>
</div>
{/if}

<!-- ADD THIS ENTIRE SECTION: -->
{#if systemStats.cpu.topProcesses && systemStats.cpu.topProcesses.length > 0}
<div style="margin-top: 0.75rem; padding-top: 0.75rem; border-top: 1px solid
rgba(255,255,255,0.1);">
  <h4 style="margin: 0 0 0.5rem 0; font-size: 0.85rem; color: rgba(0, 225, 255, 0.8);">Top
Processes:</h4>
  {#each systemStats.cpu.topProcesses as process}
    <div style="display: flex; justify-content: space-between; margin-bottom: 0.25rem; font-size:
0.85rem;">
      <span style="overflow: hidden; text-overflow: ellipsis; white-space: nowrap; max-width:
70%;">{process.name}</span>
      <span style="color: {process.cpu > 50 ? '#ef4444' : process.cpu > 25 ? '#fbbf24' :
'#4ade80'};">{process.cpu}%</span>
    </div>
  {/each}
</div>
{/if}
```

Step 3: Rebuild Your Application

```
cd /path/to/your/phantomsdr-app
npm ./recompile.sh
```

Testing the Installation

Test 1: Check API Endpoint

```
curl http://localhost:3001/api/system-stats
```

Expected output (JSON):

```
{
  "cpu": {
    "usage": 45.2,
    "cores": 8,
    "coresUsed": 1.3,
    "temperature": 62.5,
    "frequency": {
      "current": 3.0,
      "max": 3.9,
      "limit": 4.4
    },
  },
  "topProcesses": [
    {"name": "node", "cpu": 12.5},
    {"name": "phantomsdr", "cpu": 8.3}
  ],
  "memory": {
    "used": 8.5,
    "total": 16,
    "percent": 53
  }
}
```

Test 2: Check Service Status

If you installed as a service:

```
sudo systemctl status sdr-stats.service
```

Should show: Active: active (running)

Test 3: Test in Browser

1. Open your PhantomSDR web interface
2. Scroll to **Additional Info** section
3. Click **Open Additional Info** button
4. Look for the **Stats** button next to PC info
5. Click it to open the stats modal
6. Verify stats are displaying and updating

Service Management

If you installed as a systemd service, use these commands:

```
# Start the service
sudo systemctl start sdr-stats.service

# Stop the service
sudo systemctl stop sdr-stats.service

# Restart the service (after updating files)
sudo systemctl restart sdr-stats.service

# Check status
sudo systemctl status sdr-stats.service

# View real-time logs
sudo journalctl -u sdr-stats.service -f

# View last 50 log entries
sudo journalctl -u sdr-stats.service -n 50

# Enable auto-start on boot
sudo systemctl enable sdr-stats.service

# Disable auto-start
sudo systemctl disable sdr-stats.service

# Edit service
sudo nano /etc/systemd/system/sdr-stats.service

# Delete service
sudo rm /etc/systemd/system/sdr-stats.service
```

Manual Installation (Option 2)

IMPORTANT

You do not need this section for a normal installation. `install-stats-server.sh` does all of it for you and is run by the main installer by default. Follow it only for a manual setup or to repair one piece by hand.

⚠ **Warning:** Manual installation may use outdated files. The **automated script is strongly recommended** as it:

- Installs the latest version with improved temperature detection
- Auto-installs and configures lm-sensors
- Handles all dependencies automatically

If you still prefer manual installation:

1. Install Node.js

```
# Any distribution - nvm needs no root and no apt source.
# NOT NodeSource: deb.nodesource.com now answers HTTP 403 on every path.
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.4/install.sh | bash
export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh"
nvm install 22 && nvm use 22 && nvm alias default 22
```

2. Install lm-sensors (Required for Intel/AMD temperature)

```
sudo apt-get update
sudo apt-get install -y lm-sensors
sudo sensors-detect --auto
```

3. Create Directory

```
mkdir ~/sdr-stats-server
cd ~/sdr-stats-server
```

4. Download Latest Files

Important: the server files are no longer kept as copies in the repository - the installer generates them. Instead, run the automated script once to generate the latest files, then copy them:

```
# Run the automated script from the repository root
cd ~/PhantomSDR-Plus
./install-stats-server.sh
# When prompted, use a temporary port like 9999
# After installation completes, copy the generated files
cp ~/sdr-stats-server/system-stats-server.js ~/your-manual-install-dir/
cp ~/sdr-stats-server/package.json ~/your-manual-install-dir/
```

OR manually create the files using the latest code from the installation script.

5. Edit Port (if needed)

```
nano ~/sdr-stats-server/system-stats-server.js
```

Change line 6:

```
const PORT = process.env.PORT || 3001; // Change this number
```

6. Install Dependencies

```
npm install
```

7. Test Run

```
npm start
```

8. Set Up Service (Optional)

Follow the systemd service setup from the automated installation section.

Changing the Port After Installation

If you need to change the port after installation:

Step 1: Edit the server file

```
nano ~/sdr-stats-server/system-stats-server.js
```

Change line 6:

```
const PORT = process.env.PORT || 3001; // Change this number
```

Step 2: Update site_information.json

```
"siteStats": "http://Your_site_IP:NEW_PORT"
```

Step 3: Restart the service

```
sudo systemctl restart sdr-stats.service
```

Step 4: Rebuild your Svelte app

```
cd /path/to/phantomsdr-app  
npm run build
```

Troubleshooting

Issue: Port already in use

Error:

```
Error: listen EADDRINUSE: address already in use :::3001
```

Solution:

1. Find what's using the port:

```
sudo lsof -i :3001
```

1. Stop that process:

```
kill <PID>  
# or  
sudo systemctl stop sdr-stats.service
```

1. Start the service again:

```
sudo systemctl start sdr-stats.service
```

Alternative: Run the installation script again and choose a different port when prompted.

Issue: Service won't start

Error:

```
status=217/USER
```

Solution:

1. Check the service file:

```
cat /etc/systemd/system/sdr-stats.service
```

1. Verify the `User=` line matches your username:

```
whoami
```

1. Edit if needed:

```
sudo nano /etc/systemd/system/sdr-stats.service
```

1. Reload and restart:

```
sudo systemctl daemon-reload  
sudo systemctl restart sdr-stats.service
```

Issue: Stats not showing in browser

Symptoms: Modal opens but no data, or shows 0 values

Solutions:

1. Check browser console (F12) for errors
2. Verify API is accessible:

```
curl http://localhost:3001/api/system-stats
```

1. Check service is running:

```
sudo systemctl status sdr-stats.service
```

1. Verify `siteStats` in `site_information.json` matches your server address and port
2. Check CORS errors - ensure server allows your domain

Issue: Temperature shows null

Cause: Temperature reading not available on your system

Solutions:

1. **For most Linux systems:** Check if thermal zone exists:

```
cat /sys/class/thermal/thermal_zone0/temp
```

1. **Install lm-sensors:**

```
sudo apt-get install lm-sensors
sudo sensors-detect
sensors
```

1. **Note:** Temperature may not be available on:
2. Virtual machines
3. Some VPS providers
4. Windows Subsystem for Linux (WSL)
5. Non-Linux systems

This is normal - the rest of the stats will still work!

Issue: Stats values don't match `top`

Cause: CPU sampling period was too short in older versions

Solution: The current version uses 1-second sampling for accurate readings. If you have an old installation:

1. Update the server file:

```
nano ~/sdr-stats-server/system-stats-server.js
```

1. Find line 18 and change:

```
}, 100); // Old value
```

to:

```
}, 1000); // New value (1 second)
```

1. Restart:

```
sudo systemctl restart sdr-stats.service
```

Or simply re-run the installation script to get the latest version.

Issue: Node.js not found

Error:

```
node: command not found
```

Solution:

1. Install Node.js:

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.4/install.sh | bash
export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh"
nvm install 22 && nvm use 22 && nvm alias default 22
```

1. Verify:

```
node --version
npm --version
```

What You'll See

Once installed and configured, the stats modal displays:

CPU Section

- **Usage:** Overall CPU utilization percentage
- **Cores:** How much of the CPU is actually in use, expressed as whole cores — e.g. 1.3 / 12 in use means the machine is doing 1.3 cores' worth of work. This is exact and needs no threshold. It is deliberately not a count of busy cores: the scheduler spreads one core's work across many, so any such count would only ever approximate this figure.
- **Frequency:** Average current CPU clock across all cores in GHz, with the fastest core in parentheses (if available)
- **Temperature:** CPU temperature in Celsius (if available), colour-coded: green up to 69 °C, orange 70-79 °C, red at 80 °C and above
- **Top Processes:** Top 5 CPU-consuming processes with color-coded percentages:
 - Green: up to 50% CPU
 - Orange: 51-80% CPU
 - Red: above 80% CPU

Memory Section

- **Used:** Memory in use (GB)
- **Total:** Total system memory (GB)
- **Usage:** Memory usage percentage (color-coded)

⚙️Auto-Update

- Stats refresh every **5 seconds** while modal is open
- Stops updating when you close the modal (saves resources)

Temperature Detection

The stats server uses advanced multi-method temperature detection:

Supported Systems:

- **Intel Processors:** Reads from coretemp sensors via lm-sensors
- **AMD Processors:** Reads from k10temp sensors (Tdie/Tctl)
- **ARM Processors:** Reads from thermal zones (Raspberry Pi, etc.)

Requirements:

- **Intel/AMD:** Requires `lm-sensors` package (auto-installed by script)
- **ARM/Raspberry Pi:** Works out of the box, no additional packages needed

Installation Priority:

1. Tries `sensors` command for Package/Core temperature (Intel/AMD)
2. Falls back to direct `coretemp` reading (Intel)
3. Falls back to `thermal_zone0` (ARM/Raspberry Pi)

If temperature shows `null`, install `lm-sensors`:

```
sudo apt-get install lm-sensors
sudo sensors-detect --auto
sudo systemctl restart sdr-stats.service
```

Features Summary

Real-time Monitoring - Live system stats from your server **CPU Metrics** - Usage, cores, temperature, top processes **Memory Tracking** - Used, total, and percentage **Auto-Updates** - Refreshes every 5 seconds **Color-Coded** - Visual indicators for warnings **Lightweight** - Minimal resource usage **Easy Installation** - Automated script handles everything **Port Validation** - Smart port selection with conflict detection **Service Management** - Auto-start on boot

Security Notes

- The stats server accepts connections from any origin (CORS: *)
- For production, consider restricting CORS to your domain only
- Port selection validates input and checks for conflicts
- CPU temperature reading is read-only and safe
- No sensitive system information is exposed

Support & Resources

Files Included:

In this folder (the installer itself lives in the repository root, as `PhantomSDR-Plus/install-stats-server.sh` - there is only ever one copy):

- `package.json` - Dependencies, identical to what the installer writes
- This document, also available in `de`, `el`, `es`, `fr`, `hr` and `ru`

Created by the installer in `~/sdr-stats-server/`:

- `system-stats-server.js` - Backend server, generated with the port you chose
- `package.json` - Dependencies
- `node_modules/` - Installed by `npm install`

Getting Help:

- Check the troubleshooting section above
- Review service logs: `sudo journalctl -u sdr-stats.service -n 50`
- Test API manually: `curl http://localhost:3001/api/system-stats`
- Verify configuration files

Uninstall stats-server:

- Delete the folder /sdr-stats-server
- If you've created a service, just delete it using:

```
sudo rm /etc/systemd/system/sdr-stats.service`
```

- Modify App.svelte: Find the part of the code:

```

        <!-- In case you don't want the Stats Button to appear, please comment this
button section (12 lines)-->
        <!-- System Stats Button -->
        <button
            type="button"
            class="glass-button text-white py-1 px-2 ml-2 rounded text-xs"
            on:click={openSystemStats}
            title="System Resources"
            aria-haspopup="dialog"
            aria-expanded={showSystemStats}
            aria-controls="system-stats-dialog"
            style="color:rgba(0, 225, 255, 0.993); font-size: 0.75rem;"
        >
            Stats
        </button>
    ...
    and replace with:

```

```

        <!-- In case you don't want the Stats Button to appear, please comment this button
section (12 lines)-->
        <!-- System Stats Button -->
        <!--
        <button
            type="button"
            class="glass-button text-white py-1 px-2 ml-2 rounded text-xs"
            on:click={openSystemStats}
            title="System Resources"
            aria-haspopup="dialog"
            aria-expanded={showSystemStats}
            aria-controls="system-stats-dialog"
            style="color:rgba(0, 225, 255, 0.993); font-size: 0.75rem;"
        >
            Stats
        </button>
        -->
    ...

```

- Recompile using the command:

```

    cd PhantomSDR-Plus
    ./recompile.sh
    ```

- Restart the PC and start the server

Author: Created for PhantomSDR-Plus
Version: 1.1 (Improved CPU temperature detection for Intel/AMD)
Updated: January 2026
License: MIT

Quick Command Reference

```bash
# Installation
cd ~/PhantomSDR-Plus
chmod +x install-stats-server.sh
./install-stats-server.sh

# Service Management
sudo systemctl start sdr-stats.service      # Start
sudo systemctl stop sdr-stats.service       # Stop
sudo systemctl restart sdr-stats.service    # Restart
sudo systemctl status sdr-stats.service     # Status
sudo journalctl -u sdr-stats.service -f     # Logs

# Testing
curl http://localhost:3001/api/system-stats # Test API
curl http://localhost:3001/api/health      # Health check

# Troubleshooting
sudo lsof -i :3001                          # Check port
whoami                                       # Check username
node --version                              # Check Node.js

```

Happy monitoring!

CHAPTER 8

Decoders

DECODERS.md

PhantomSDR-Plus — Decoder User Manual

This guide covers every built-in decoder available in PhantomSDR-Plus. All decoders share the same activation workflow, described below, followed by per-decoder setup instructions.

Table of Contents

1. How to Start a Decoder
 2. FT8
 3. FT4-FT2
 4. JS8
 5. CW — Morse Code
 6. QRSS Grabber
 7. WSPR
 8. HF FAX / WEFAX
 9. NAVTEX
 10. FSK / RTTY — including PSK31 and Olivia
 11. SSTV
 12. General Tips
-

1. How to Start a Decoder

All decoders are accessed from the **Decoder Options** section, located below the Audio Spectrogram controls on the main panel.

Steps:

1. Click the **Decoder: OFF** button to switch it to **Decoder: ON** (it turns blue when active).
2. Open the dropdown menu that appears to the right of the button and select the decoder you want.
3. The chosen decoder's panel will appear below the controls — follow the decoder-specific instructions in the relevant section of this guide.
4. To stop decoding, select — **Select decoder** — from the dropdown, or click the **Decoder: ON** button to turn it back OFF.

One-touch decoder buttons

The dropdown is not the only way in. The main panel carries a **Decoders** button row — directly below **Wheel Tuning Steps** — with one button per decoder:

Button	Decoder	Button	Decoder
FT8	FT8	SSTV	SSTV
FT4	FT4	NAVTEX	NAVTEX
FT2	FT2	RTTY	FSK / RTTY
CW	CW		
WSPR	WSPR	FAX	HF FAX / WEFAX

Pressing a button does the whole sequence in one go: it selects the decoder, switches the decoder ON, and scrolls its panel into view. The button turns blue while its decoder is running. **Press the same button again to switch the decoder off** — its window closes with it.

RADEL and **RADEU** are deliberately *not* in this row. They are digital-voice modes rather than text decoders, so they have their own pair of buttons next to the **Modes selector** heading, and inside the **Modes** and **Bands** popup windows. Those behave in exactly the same way — press to start, press again to stop. See the [RADE manual](#).

The buttons, the dropdown and the ON/OFF button all drive the same state, so whichever you use, the others follow.

Only one decoder can be active at a time. Switching to a different decoder automatically stops the previous one.

The **QRSS grabber** is not part of this dropdown — it has its own **QRSS** section and can run at the same time as a decoder.

Decoding runs in the background. SSTV, HF FAX, NAVTEX, FSK/RTTY and CW each run in a separate Web Worker thread, so the decoding work never competes with audio playback or the waterfall. Starting or stopping a decoder does not interrupt the audio, and the interface stays responsive while a picture or a page is being received. All decoders are fed raw audio taken *before* AGC, noise reduction and mute — so muting the receiver, or changing those settings to suit your ears, does not affect decoding.

A running decoder owns the mode and the passband. Normally the mode follows the band plan in `bands-config.js`: move the dial into a segment marked LSB or AM and the receiver switches to it. While a decoder is running that no longer happens. The decoder holds the mode it needs (USB for most, its own for RADE) and the passband it needs — PSK31 about ± 100 Hz, Olivia its full bandwidth, RTTY its shift — and both survive retuning, including a jump to another band. Without this, tuning FT8 on 40 m would flip the receiver to LSB the moment the dial moved, and the narrow decoder passbands would widen back to the full SSB filter.

The band's own mode returns the moment you switch the decoder off. You can still override the mode by hand at any time: the mode buttons are a deliberate choice and always win. The CW decoder is the exception to all of this — it decodes in whatever mode you are listening in and never takes the receiver over.

Decoder ID — recognising the mode automatically

Above the decoder buttons sits a **Decoder ID** frame. It answers the question "*what am I listening to?*" when you come across a digital signal you cannot place, and then offers the matching decoder in one click.

It is **off by default** and costs nothing while it is off. Press **On** and leave the signal tuned in. For about half a minute it shows *Listening...* with a progress bar — the timing measurements have to see two complete FT8 cycles before they mean anything — and then it names the mode:

Decoder ID [On] NAVTEX / SIT0R-B 99% 170 Hz shift · 100.00 Bd · 43 dB S/N [Use NAVTEX]

- The **mode** and a **confidence** figure. Green above 75%, amber above 50%, orange below — treat an orange answer as a hint, not a verdict.
- **What was actually measured:** occupied bandwidth, tone spacing, symbol rate, burst length, the UTC grid it fits, signal-to-noise. This is the evidence behind the verdict, so you can judge it for yourself.
- **Use** starts the right decoder, exactly as if you had pressed its button — and sets the receiver up to work that mode. For the modes that live in the FSK panel it also selects the correct variant — RTTY, Weather RTTY, PSK31 or Olivia.
- Runners-up are listed after **or**. Click one to try it instead, so a wrong first answer costs you nothing.
- **** means the frequency you are tuned to is a known calling frequency for that mode, and that this counted towards the ranking.

It never switches your decoder by itself. It only ever suggests. And once you start a decoder — from the button row, from the dropdown, or from Decoder ID's own **Use** button — Decoder ID switches itself off: its job was to choose one, and the decoder now wants the same audio.

It sets the receiver up for the mode. Choosing a mode is a decision to work it, so the receiver follows: the tuned frequency moves to the middle of the waterfall and the view becomes about 100 kHz wide around it, the sideband becomes the one that mode is worked on, and the passband narrows to the window that mode lives in — the whole 3 kHz sub-band for FT8, FT4, FT2 and JS8, 1350–1650 Hz for WSPR, 250–750 Hz for NAVTEX, 800–2700 Hz for HF FAX, 900–2600 Hz for SSTV, and ± 250 Hz either side of the dial for CW, which also switches the receiver to CW. A narrow passband keeps the neighbouring signals out of the decoder, which is what you would have done by hand. Two exceptions: SSTV keeps the sideband it chose for itself — LSB on 80 and 40 m by convention, where forcing USB would invert its tone mapping — and the FSK modes keep the window their own panel derives from the variant's shift or bandwidth, which is tighter than any fixed figure. Pressing **Use** for the decoder that is already running still switches it off, and then the receiver is left exactly as it was.

The slow modes take longer. WSPR sits on a two-minute grid and SSTV pictures run for a minute or two, so those two are named later than the rest — give them three or four minutes on frequency. HF FAX and SSTV are also recognised only when they are tuned the usual way — the picture band sitting between about 1500 and 2300 Hz in the audio, which is where their decoders expect it too. Retuning starts the measurement again from scratch, because everything gathered so far describes the frequency you have just left.

What it recognises: FT8, FT4, JS8, WSPR, CW, NAVTEX/SITOR-B, RTTY at 45.45 Bd, Weather RTTY (DWD), PSK31, Olivia, HF FAX, SSTV and DSC. There is no DSC decoder in this build, so DSC is named and marked *no decoder*.

Recentre & retry. Where the signal sits in the audio passband makes no difference between about 400 Hz and 2700 Hz. Outside that range the measurement degrades, and an amber **Recentre & retry** button appears. Pressing it moves the dial so that the signal lands in the middle of the passband, and starts a fresh measurement. It is a single calculated move, not a search.

Three honest limits:

- **FT8, JS8 and FT2 are the same signal.** The same 8-FSK modulation, the same tone spacing, and JS8 at Normal speed uses FT8's 15-second cadence. Nothing in the signal itself separates them. Frequency does: on an FT8 calling frequency it reports FT8, on a JS8 one it reports JS8, and anywhere else it honestly reports the family — *FT8 / JS8 / FT2*. Start either decoder and see which one produces text.

- **It goes quiet rather than guessing.** Below roughly 10 dB signal-to-noise it reports *Nothing above the noise* or *Not recognised* instead of naming a mode it cannot really see. This is deliberate: a confident wrong answer is worse than no answer at all.
- **A fading signal is read a different way.** WSPR transmits for 110 seconds without a break, which is longer than an ordinary QSB fade, so a deep fade cuts the transmission into pieces and its two-minute rhythm can no longer be read from them. It is then recovered by another route — matching the whole listening history against the two-minute grid instead of against single transmissions — and the evidence line says *120 s grid (through fading)*. That is weaker evidence than a transmission heard whole, so the confidence shown is deliberately lower. A fade deep enough to bury the signal in the noise cannot be recovered at all, and Olivia is stricter than the rest: it will not be named until its symbol rate has been measured, so it falls silent earlier than the other modes.

2. FT8

What it is: FT8 is a popular weak-signal digital mode used by amateur radio operators worldwide. Transmissions last exactly 15 seconds, and the protocol can copy signals up to 20–25 dB below the noise floor. It is the most commonly used mode for long-distance (DX) contacts.

Recommended frequencies (USB)

Band	Frequency
160m	1.840 MHz
80m	3.573 MHz
40m	7.074 MHz
30m	10.136 MHz
20m	14.074 MHz
17m	18.100 MHz
15m	21.074 MHz
12m	24.915 MHz
10m	28.074 MHz

Setup

1. Tune to an FT8 frequency listed above and set mode to **USB**.
2. Enable the decoder and select **FT8** from the dropdown. Or simply press the **FT8** button.
3. The **FT8 Messages** panel appears automatically below.

Reading the output

The message list shows decoded transmissions as they arrive. Each 15-second cycle produces a new batch of messages. The **Farthest** field (top-right of the panel) shows the greatest distance decoded in the current session in kilometres.

Typical message format: CQ DX AA1BB FN31 — a CQ call from callsign AA1BB located in grid square FN31.

The **SNR** shown with each decode is a real signal-to-noise ratio, measured on the decoded signal against the noise around it and referred to a 2500 Hz bandwidth, the way WSJT-X reports it. Weak stations read negative, down to about –20 dB, and strong ones positive; on the same recordings the two programs agree on average, typically within about 3 dB for a single decode. FT4 and FT2 use the same method.

After an update: the FT8/FT4/FT2 decoder is a separate file that browsers keep for a long time. If every SNR still sits between about 0 and +5 dB, reload the page with a hard refresh (**Ctrl+Shift+R**, or **Cmd+Shift+R** on a Mac).

Tip: FT8 is tightly time-synchronised. Your browser uses your computer clock; if your system clock drifts by more than a couple of seconds, decoding will fail. Keep your system time synchronised to NTP.

3. FT4-FT2

What it is: FT4 is a faster variant of FT8 designed for contest-style operation. Each transmission cycle is 7.5 seconds (half of FT8), making it twice as fast but requiring a slightly stronger signal. FT2 is an even faster variant of FT8. It is an ultra-fast 77-bit mode with TR periods of 3.75 seconds (= 1/2 of FT4) — still experimental.

Recommended frequencies (USB)

Band	Frequency FT4
80m	3.575 MHz
40m	7.047 MHz
30m	10.140 MHz
20m	14.080 MHz
15m	21.140 MHz
10m	28.180 MHz

Band	Frequency FT2
160m	1.843 to 1.846
80m	3.578 to 3.581
60m	5.360 (Check local regional regulations)
40m	7.052 to 7.062
30m	10.144
20m	14.084
17m	18.108
15m	21.144
12m	24.923
10m	28.184

Setup

1. Tune to an FT4 frequency and set mode to **USB**.
2. Enable the decoder and select **FT4** or **FT2** from the dropdown. Or simply press the **FT4 / FT2** button.
3. The **FT4 Messages** or **FT2 Messages** panel appears below, identical in layout to the FT8 panel.

Note: FT4 and FT8 use different spectral formats and are not interchangeable. Make sure you are on an FT4 frequency when using this decoder.

4. JS8

What it is: JS8 (the mode used by JS8Call) takes FT8's weak-signal engine and turns it into a keyboard-to-keyboard conversation mode. Where FT8 sends fixed 13-character exchanges, JS8 sends free text a few characters at a time and stitches consecutive transmissions back into whole sentences — so anything longer than about twelve characters arrives over several 15-second cycles. It decodes about as deep as FT8, which makes it usable when voice and CW are not.

Recommended frequencies (USB)

Band	Frequency
160m	1.842 MHz
80m	3.578 MHz
40m	7.078 MHz
30m	10.130 MHz
20m	14.078 MHz
17m	18.104 MHz
15m	21.078 MHz
12m	24.922 MHz
10m	28.078 MHz

Speeds

JS8 has five speeds. Every station in a conversation must use the same one. **Normal** is the calling speed and is where almost all activity is — start there.

Speed	Cycle	Bandwidth	When it is used
Slow	30 s	25 Hz	Very weak paths; the most sensitive
Normal	15 s	50 Hz	The standard calling speed
Fast	10 s	80 Hz	Quicker exchanges, needs a stronger signal
Turbo	6 s	160 Hz	Strong local signals
Ultra	4 s	250 Hz	Experimental, rarely seen on the air

Setup

1. Tune to a JS8 frequency and set mode to **USB**.
2. Enable the decoder and select **JS8** from the dropdown, or simply press the **JS8** button.
3. The **JS8 Decoder** panel appears below. Leave **Speed** on *Normal* unless you know the station you are after is using another.

Reading the output

The panel has two lists.

Messages still arriving appear at the top in green, with a blinking cursor. A JS8 message can take four cycles — a full minute on Normal — so this is where you watch a sentence build up. Nothing has gone wrong if it sits there for a while.

Completed messages fill the main list, one row each:

Column	Meaning
Mode	Always JS8
Hz	Audio frequency of the signal inside the passband
dB	Signal-to-noise ratio in a 2500 Hz reference bandwidth
Message	The decoded text; callsigns are shown in green

Typical messages:

- SV1BTL KM17: HB — a heartbeat: the station is announcing it is on the air, with its grid square.
- KN4CRD: K00G SNR -05 — a directed message: KN4CRD tells K00G it is hearing them at −5 dB.
- MP 100W 8M/BALUN JN58KH AUGSBURG, MARTIN — free text, in this case a station description that arrived over four cycles.

A message shown **dimmed and in italics** timed out before its last frame arrived. The text is real, but it may be cut short.

Sync offset

The **Sync offset** slider sets how long after the UTC cycle boundary the receiver starts capturing, which compensates for delay through the audio pipeline. Leave **Auto** ticked: the decoder measures the timing of the signals it hears and adjusts on its own. Only touch the slider if you have a reason to.

Spectrum marks

The small spectrum strip in the panel draws a yellow vertical line at the frequency of every signal decoded in the last cycle, so the Hz column can be read straight off the display.

Seeing your own spots

If the receiver is uploading JS8 spots to PSK Reporter (see the admin panel's Spot Reporting tab), the **JS8 map** button in the **Note:** panel opens PSK Reporter's map filtered to this receiver's callsign and mode JS8 — the same thing the **FT8 map** and **FT4 map** buttons do for those modes.

Tip: JS8 is much quieter than FT8. On a mid-band afternoon you may see one transmission every couple of minutes, and long stretches of nothing at all are normal. Give it five or ten minutes before concluding something is wrong. 20m (14.078) and 40m (7.078) are the usual places to look.

Tip: Like FT8, JS8 is tightly time-synchronised and uses your computer's clock. If your system time drifts by more than a second or two, nothing will decode. Keep it synchronised to NTP.

5. CW — Morse Code

What it is: The CW decoder listens for Morse code (continuous wave) signals and converts them to text in real time. It automatically tracks the signal frequency and adapts to the operator's sending speed.

Recommended frequencies

CW is active across amateur bands, typically at the lower portion of each band. Common spots:

Band	Segment
40m	7.000–7.040 MHz
20m	14.000–14.070 MHz
15m	21.000–21.080 MHz

Setup

1. Tune to a CW signal using **CW** or **CW-L** demodulation mode as appropriate.
2. Enable the decoder and select **CW** from the dropdown. Or simply press the **CW** button.
3. The **CW Decoder** panel appears below.

Reading the output

- The panel header shows the detected signal frequency in Hz (e.g. ≈ 700 Hz) and the estimated sending speed in words per minute (e.g. $\cdot 22$ WPM).
- If no signal is detected the header shows **scanning...**
- On an empty channel nothing prints, and the speed and tone are not shown. The decoder only starts printing when there is a clear difference between key-down and key-up, which plain noise does not produce, while a signal still readable at about -10 dB has plenty.
- Decoded text scrolls in amber monospaced text. The blinking cursor (█) marks where text is currently being written.
- Click **Clear** to erase the output buffer.

Tips: - Centre your passband on the CW tone. The decoder works best when the CW signal sits between roughly 400–900 Hz in the audio spectrum. - Speeds from about 8 to 40 WPM are copied, including Farnsworth sending (characters sent fast with longer gaps between them): the speed lock holds and letters are not split into separate words. Heavily hand-keyed (irregular) Morse may still reduce accuracy. - The decoder performs best on a single clean signal. Strong QRM from nearby signals on the same band may confuse it.

6. QRSS Grabber

What it is: QRSS is CW sent so slowly that a single dot lasts seconds instead of milliseconds. At that speed the signal occupies only a fraction of a hertz, so an analysis narrow enough to match it can pull the trace out from 20–30 dB *below* the noise floor. There is nothing to read by ear — QRSS is **looked at**, not listened to. The grabber runs its own very long FFT over the receiver audio and paints the classic grabber display: frequency on the vertical axis, time scrolling from left to right, the newest column always at the right-hand edge.

Most of what you will see comes from MEPT beacons (Manned Experimental Propagation Transmitter) — low-power transmitters, often a few hundred milliwatts into a wire, identifying continuously in very slow Morse or in patterned FSK.

Where to look

QRSS beacons live in narrow 100–200 Hz windows near the bottom of each band. You do not have to remember them: the grabber has a **Band** list of the windows below, and picking one tunes there and sets the receiver up for it (see *Opening the grabber*).

Band	Window centre	Note
30m	10,140.00 kHz	The main QRSS window — busiest by far, day and night
40m	7,039.90 kHz	
40m	7,000.85 kHz	Knights window, 7,000.8–7,000.9 kHz
20m	14,096.90 kHz	
80m	3,569.90 kHz	
80m	3,568.60 kHz	Older window
80m	3,500.85 kHz	Knights window, 3,500.8–3,500.9 kHz
160m	1,837.90 kHz	
160m	1,843.30 kHz	Older window
630m	476.10 kHz	
2200m	137.70 kHz	LF QRSS / DFCW
60m	5,288.55 kHz	
17m	18,105.90 kHz	
15m	21,095.90 kHz	
12m	24,925.90 kHz	
10m	28,125.70 kHz	
10m	28,000.85 kHz	Older window
10m	28,322.00 kHz	Alternative
6m	50,294.30 kHz	

Several bands appear twice because two conventions are genuinely in use. The modern windows sit 200 Hz below that band's WSPR frequency; the older *Knights* windows are somewhere else entirely, and on 40 and 80 m much lower down. If a band is quiet on one, try the other.

These are conventions, not regulations, and some are regional — the 10 m figures in particular vary. Treat the list as a starting point and follow local practice.

Opening the grabber

The QRSS grabber is **not** in the Decoder dropdown. It has its own **QRSS** section, alongside the Spectrogram and Decoder controls.

1. Click **Show**. The grabber canvas appears and its controls unfold next to the button.
2. Pick a window from the **Band** list and press **Tune**. This does everything the mode needs in one step: it tunes to the window, switches the receiver to **CW**, and sets a passband just wide enough for the slice you are watching. The dial then reads the true QRSS frequency, and the traces land on the centre line of the display.
3. Set **Speed** to match the beacon's dot length.
4. **Centre** and **Span** work as before; changing either re-shapes the passband to match, so the display and the receiver stay in step.
5. Click **Hide** to stop. The band's normal mode comes back.

You can still do it by hand instead — tune in **USB** about 1 kHz below the window so the traces land near 800 Hz of audio. The **Band** list only saves you the arithmetic.

While a window from the list is tuned, the grabber holds the receiver the way a decoder does: CW and its passband survive retuning, and the window follows the dial, so the trace stays on the centre line as you hunt along the band. It hands the receiver back when you hide the grabber, click a mode button, or start a decoder.

Speeds

Setting	Dot length	Resolution	Analysis window	New column every
QRSS 3	3 s	0.73 Hz	1.4 s	0.7 s
QRSS 6	6 s	0.37 Hz	2.7 s	1.4 s
QRSS 10	10 s	0.18 Hz	5.5 s	2.7 s
QRSS 30	30 s	0.09 Hz	10.9 s	5.5 s
QRSS 60	60 s	0.05 Hz	21.8 s	10.9 s

A setting faster than the beacon wastes sensitivity — the trace comes out thin and noisy. A setting slower than the beacon smears consecutive dots and dashes into one bar. The panel opens at **QRSS 6**, which is a reasonable place to start on an unknown signal: it is more sensitive than QRSS 3, and quicker to update and more forgiving of drift than QRSS 10. Once you can see the shape of the keying, move to the speed the beacon is actually sending — every step up in speed setting costs 3 dB of sensitivity.

Controls

Control	What it does
Band	The QRSS windows listed above. Tune goes to the selected one in CW with a matching passband.
Speed	Sets the transform length — the trade between frequency resolution and time resolution (table above).
Centre	Audio frequency at the middle of the display, 100–3000 Hz.
Span	Height of the displayed slice: 20, 50, 100 or 200 Hz, opening at 100 Hz — the width of a QRSS sub-band. A narrower span spreads each trace over more pixels. Wider spans fit more bins into the panel than it has pixel rows; each row then shows the strongest bin it covers, so nothing can hide between rows.
Gain	−10 to +40 dB. Shifts the colour map relative to the measured noise floor; raise it to brighten faint traces.
Color	Rainbow, Green or Grayscale.
Clear	Wipes the canvas and resets the noise reference.

Reading the display

Frequency ticks run down the left edge, highest frequency at the top. Under the canvas a status line shows the selected speed, the resolution actually in use (e.g. 0.183 Hz/bin · 5.5 s window) and the column rate (e.g. 2.7 s/column).

The colour map is referenced to the **current** noise, measured continuously from the empty parts of each column, not to an absolute level. Volume, AGC, antenna and band changes therefore do not require re-adjusting Gain — the display keeps a constant contrast against whatever the band noise happens to be. As with the decoders, the audio is taken before AGC, noise reduction and mute, so you can mute the receiver and keep watching.

What the shapes mean:

- A **steady carrier** draws a straight horizontal line.
- **Slow Morse** draws that line as a chain of short and long segments — dots and dashes read left to right.

- **Curved or drifting traces** are an unstabilised beacon oscillator warming up or cooling down. This is normal, and the drift signature is often how a regular grabber-watcher recognises a station.
- **Vertical streaks** across the whole slice are static crashes or local noise bursts, not signal.

Tips: - Be patient. At QRSS 30 a single column takes 5.5 seconds, so a complete callsign can take ten minutes or more to cross the screen. Leave it running. - Tuning from the **Band** list already narrows the passband around the window. If you tuned by hand, do it yourself: it does not change what the transform resolves, but it keeps strong neighbours from working the receiver's AGC. - The grabber is independent of the decoders — you can leave it running while a decoder works on something else. - A wide span at a slow speed is the heaviest combination; the whole transform runs in the browser, so on a modest machine prefer a 50 Hz span.

7. WSPR

What it is: WSPR (Weak Signal Propagation Reporter, pronounced "whisper") is an ultra-weak-signal beacon mode that maps HF propagation paths worldwide. Each transmission takes approximately 110 seconds and fits inside a 200 Hz-wide slot. The decoder waits for a complete 2-minute UTC-aligned slot before decoding.

Recommended frequencies (USB, dial)

Band	Dial frequency
160m	1.836.600 MHz
80m	3.568.600 MHz
40m	7.038.600 MHz
30m	10.138.700 MHz
20m	14.095.600 MHz
17m	18.104.600 MHz
15m	21.094.600 MHz

Setup

1. Tune to a WSPR dial frequency above and set mode to **USB**.
2. The WSPR signal occupies the 1400–1600 Hz audio range. Selecting **WSPR** does the rest for you: the receiver switches to USB and the passband narrows to 1350–1650 Hz, which covers the whole range the decoder searches. You do not need to adjust it further.
3. Enable the decoder and select **WSPR** from the dropdown. Or simply press the **WSPR** button.
4. The **WSPR-2 Decoder** panel appears below.

Reading the output

The panel shows a progress bar for the current 2-minute slot:

- **Cyan bar filling** — collecting signal data (0–116 s into the slot).
- **Amber bar pulsing** — decoding in progress (last ~4 s of the slot).
- **Empty bar** — waiting for the next even UTC minute.

Each successfully decoded spot is shown in a table with the following columns:

Column	Meaning
UTC	Time of the spot (even minute)
Callsign	The station that transmitted
Grid	Maidenhead locator of the transmitter
Power	Transmitted power in dBm
Freq	Exact audio frequency (Hz) within the WSPR passband
SNR	Signal-to-noise ratio in dB

Click **Clear** to erase the spot list.

Tip: WSPR decoding requires very accurate system time (within ± 1 second of UTC). The first slot after enabling the decoder will begin at the next even UTC minute — a short wait is normal.

8. HF FAX / WEFAX

What it is: HF Radiofax (also known as WEFAX) is used by coast guard and meteorological services worldwide to broadcast weather maps, sea-state charts, and surface analyses over shortwave. The decoder reconstructs the image line by line as it is received.

Setup

1. Enable the decoder and select **HF FAX / WEFAX** from the dropdown. Or simply press the **FAX** button.
2. The **HF FAX / WEFAX Receiver** panel appears below.
3. **Select a station** from the Station dropdown. Over 20 stations are available covering Europe, Asia, Oceania, and the Americas (e.g. DDH3/DDK3 Germany, SVJ4/GR Greece, JMH Japan, NMG USA New Orleans, CBM Punta Arenas Chile). The timetables follow the NOAA worldwide radiofax schedule (edition of 7 March 2025); Charleville (VMC, Australia) includes its extra broadcasts on 11030 kHz only.
4. If the station broadcasts on more than one frequency, select the desired frequency from the **Frequency** sub-dropdown.
5. Click ► **Tune** to automatically tune the waterfall to that station.
6. The mode is forced to **USB** automatically.

Broadcast schedule

When a station is selected, a **Next Transmissions** countdown table appears showing the next 4 scheduled broadcasts in UTC, with a live countdown:

- Normal (grey/green) — transmission is coming up.
- **Amber** — transmission starts within 3 minutes; arm the decoder now.
- **Red** ● — transmission starts within 30 seconds; reception is imminent.

Parameters

Most stations use the standard defaults (marked ★). Change only if you know the station uses non-standard settings.

Parameter	Default	Description
LPM	120 ★	Lines per minute — the drum rotation speed
IOC	576 ★	Index of cooperation — determines pixels per line
Shift	800 Hz ★	Frequency shift between black and white tones

Controls

- ⇄ **Auto-align** — enabled by default. Automatically synchronises to the phasing signal at the start of each image. Disable only if you are experiencing alignment problems on a known-good signal.
- ⬆ **Invert** — swaps black and white. Use if the image appears as a negative (white areas where black should be).
- ↺ **Refresh** — clears the canvas and resets the decoder. Use this between transmissions or if the image tears or drifts.
- ⬇ **Save PNG** — saves the current canvas as a PNG file to your computer.

Status indicators

At the bottom of the image, two tone indicators show:

- **300 Hz phasing** — lights cyan when the start-of-image phasing tone is detected.
- **450 Hz stop** — lights red when the end-of-image stop tone is detected.

Both signals are recognised by their rhythm across the whole width of a line, which the real start and stop signals always fill and chart content practically never does, so hatched areas in a chart are not taken for a start signal. They are recognised down to about 0 dB SNR, and on very weak signals the image is lined up from the average of several phasing lines, so the chart still starts at the left edge.

Note: The image scrolls upward — the newest received line always appears at the bottom of the canvas. If you see **[PHASING]** in the header, the decoder has locked onto a new image start.

9. NAVTEX

What it is: NAVTEX is the international maritime broadcast system for coastal safety information — navigational warnings, weather forecasts, and search-and-rescue notices. It uses 100-baud FSK (SITOR-B with FEC) and is received on dedicated channels worldwide.

Available channels

Channel	Frequency	Usage
International	518 kHz	English-language, international
Domestic	490 kHz	National language broadcasts
HF (×5)	4209.5 / 6314 / 8416.5 / 12579 / 16806.5 kHz	Long-range HF NAVTEX

Setup

1. Enable the decoder and select **NAVTEX** from the dropdown. Or simply press the **NAVTEX** button. The receiver switches to **USB** automatically.
2. The **NAVTEX Receiver** panel appears.
3. Select the desired channel from the **Station** dropdown (e.g. `International – 518 kHz`).

4. Click ⇒ **Tune & Set IF** to automatically tune the waterfall and narrow the passband to the correct audio window. Mode is set to **USB** automatically. The dial is placed 500 Hz below the channel centre, so the NAVTEX signal appears at 500 Hz audio.

Broadcast schedule

The schedule table lists all known stations on the selected channel with:

- Their ITU identifier letter and country flag
- Next broadcast time in UTC
- Live countdown to the next transmission
- **Amber** within 2 minutes / **Red** ● within 30 seconds — arm the decoder now

Reading the output

Decoded text appears in teal monospace. Message boundaries are clearly marked:

```
— ZCZC MA12 —
... message content ...
— NNNN —
```

ZCZC marks the start of a message. The four characters after it identify the station (M), subject (A = navigational warnings), and sequential number (12). The banner waits for the whole header, so it reads ZCZC MA12, not ZCZC ?????. NNNN marks the end.

If you tune in after a message has started, or its header is lost to noise, the text still appears, under a **(header missed)** — banner. That banner only opens once several characters in a row arrive with both of their NAVTEX transmissions agreeing, which noise practically never produces. It closes at NNNN as usual, with **(signal lost)** — if the signal disappears first, or when a new message begins.

Click **Clear** to erase the message buffer.

Note: NAVTEX operates on MF and LF frequencies (518/490 kHz). Reception range is typically 200–400 nautical miles from the transmitter. The HF channels (4–17 MHz) provide much greater range.

10. FSK / RTTY — including PSK31 and Olivia

What it is: A general-purpose decoder for narrow-band text modes, with five operating variants selected from one dropdown. Three are true FSK (Frequency-Shift Keying): Maritime FSK (SITOR), Weather RTTY and Amateur RTTY. The other two are not FSK at all but share the same window: **PSK31**, which is phase-shift keying, and **Olivia**, which is multi-tone FSK with forward error correction. Each variant comes with a preset tuned to its standard parameters.

Variants and presets

Variant	Center	Shift	Baud	Framing	Encoding
Maritime FSK / SITOR	500 Hz	170 Hz	100	7N1	CCIR-476
Weather RTTY	1000 Hz	450 Hz	50	5N1.5	ITA2
Amateur RTTY	1000 Hz	170 Hz	45.45	5N1.5	ITA2
PSK31 (BPSK)	1000 Hz	—	31.25	—	Varicode
Olivia (MFSK)	1000 Hz	—	see mode	—	7-bit + FEC

The panel adapts to the variant you choose. **Shift, Baud, Framing, Encoding, Invert mark / space** and **Auto shift detect** are hidden for PSK31 and Olivia, because neither mode has a mark/space tone pair or UART-style framing. In their place, **Center audio** becomes a free-entry number field (the carrier can sit anywhere in the passband), and Olivia gains a **Mode** selector and a **Squelch** slider.

Setup

1. Enable the decoder and select **FSK / RTTY** from the dropdown. Or simply press the **RTTY** button.
2. The decoder panel appears. Its title follows the variant — *FSK / RTTY Decoder*, *PSK31 Decoder* or *Olivia Decoder*.
3. Select the **Variant**. The parameters update automatically.
4. For Olivia, set **Mode** (tones / bandwidth) to match the transmission — see the Olivia notes below.
5. Use the **Known frequency** dropdown to select a common frequency for the chosen variant, then click **Tune** to jump to it.
6. Fine-tune frequency until decoded text becomes stable and readable.

Known frequencies by variant

Maritime FSK / SITOR

- 518.0 kHz — International NAVTEX
- 490.0 kHz — National NAVTEX
- 4209.5 / 6314.0 / 8416.5 / 12579.0 / 16806.5 / 22376.0 kHz — HF SITOR

Weather RTTY

- 4583.0 / 7646.0 / 10100.8 / 11039.0 / 14467.3 kHz — DWD (German Weather Service)

Amateur RTTY

- 3590 kHz (80m), 7043 kHz (40m), 10143 kHz (30m), 14083 kHz (20m), 21083 kHz (15m), 28083 kHz (10m)

PSK31

- 3580.15 kHz (80m), 7040.15 kHz (40m), 10142.15 kHz (30m), 14070.15 kHz (20m), 18100.15 kHz (17m), 21080.15 kHz (15m), 24920.15 kHz (12m), 28120.15 kHz (10m)

Olivia

- 3577.75 kHz (80m), 7073.75 kHz (40m), 10142.25 kHz (30m), 14075.5 kHz (20m), 18103.75 kHz (17m), 21075.75 kHz (15m), 24921.75 kHz (12m), 28123.75 kHz (10m)

Parameters

Parameter	Applies to	Description
Center audio (Hz)	all	The audio frequency of the mark/space midpoint; for PSK31 the carrier, for Olivia the centre of the tone block. A dropdown for the FSK variants, a free-entry field for PSK31 and Olivia
Shift (Hz)	FSK only	Frequency difference between mark and space tones
Baud	FSK only	Symbol rate
Framing	FSK only	Data bits, parity, stop bits (e.g. 7N1 = 7 data, no parity, 1 stop)
Encoding	FSK only	Character set (CCIR-476, ITA2/Baudot, or ASCII)
Invert mark / space	FSK only	Swaps mark and space tones
Auto shift detect	FSK only	Attempts automatic shift measurement from the incoming signal
Mode (tones / Hz)	Olivia only	Number of tones and bandwidth — must match the transmission exactly
Squelch (FEC S/N)	Olivia only	How strong the error-correction match must be before text is printed
Squelch (SNR)	Amateur and Weather RTTY	Stops noise printing as random letters. Starts at −5 dB for Amateur RTTY and −8 dB for Weather RTTY; the far left (off) disables it. The SNR figure turns grey while the squelch is closed

Signal metrics

The status bar shows live measurements, and the fields change with the variant:

Variant	Fields shown
FSK variants	Mark / Space (measured tone frequencies), SNR , Lock , Timing
PSK31	Carrier (Hz, after automatic frequency correction), IMD (dB), SNR , Lock , Timing
Olivia	Centre (Hz), Mode , S/N , FEC (%), Sync

Timing/Sync reads LOCKED/SYNCEd once the decoder is tracking, and SEARCH while it is still hunting.

SNR for the FSK variants and PSK31 is measured on the signal itself and given in a 3 kHz bandwidth, so the two read on the same scale: within about 1 dB of the true value from +20 dB down to −5 dB. An empty frequency reads about −12 to −15 dB, and for PSK31 0 dB is already a strong signal. Olivia's **S/N** is different: it is the error-correction quality figure.

Amateur RTTY (45.45 baud, 170 Hz) copies without errors down to about 0 dB, with about 1.5% of characters wrong at −3 dB and 20% at −6 dB.

Additional controls

- ⇒ **Set IF Band-Pass** — narrows the receiver passband to tightly bracket the signal. The width follows the variant: 300 Hz for Amateur RTTY (narrow enough to remove a stronger station 250 Hz away, while still copying a signal tuned 40 Hz off), 594 Hz for Weather RTTY, mark/space plus margin for Maritime FSK, about ±100 Hz for PSK31, and the full tone-block width plus margin for Olivia. The passband follows the decoder when **Auto-tune** or **Auto shift detect** moves the tones, and when you change the **Baud** setting.

- **🔊 Auto-tune Center** — automatic search for the signal. For the FSK variants it looks for a balanced pair of tones; for PSK31 it finds the carrier; for Olivia it finds the strongest block of the selected bandwidth. **Auto shift detect** only changes the shift when a real two-tone signal is present, so it no longer jumps on noise.

PSK31 notes

PSK31 is the most common keyboard-to-keyboard mode on HF. It is only 62 Hz wide, so several QSOs sit side by side within a few hundred hertz of the watering-hole frequency, and tuning is a matter of picking one trace out of the group on the waterfall.

- The decoder corrects its own tuning error automatically over roughly **±25 Hz**, so you only need to get close. **Carrier** in the metrics row shows where it actually settled.
- **IMD** measures transmitter quality, not reception: a clean signal reads about **−20 dB** or better. A poor reading means the other station is overdriving, not that you are mistuned.
- Text appears character by character with no error correction, so a weak signal degrades into occasional wrong letters rather than stopping.

Olivia notes

Olivia trades speed for robustness. It is far slower than PSK31 but decodes signals that are inaudible by ear, which makes it popular for weak-signal and long-distance contacts.

- **The Mode setting must match the transmission exactly.** A wrong tones/bandwidth combination decodes nothing at all — not garbled text, but silence. The panel opens on **8 / 250**, the narrow configuration left running on the calling frequencies; **16 / 500** and **32 / 1000** are the other two in common use, and **16 / 1000** is also offered.
- Olivia sends no preamble, so the decoder has to search for synchronisation. **Allow a few seconds** after tuning before text appears. The **Sync** field shows **SEARCH** until it locks.
- The error correction works on blocks, so text arrives in **bursts rather than a steady stream**, and there is a delay of several blocks between transmission and display.
- **Squelch (FEC S/N)** sets how confident the error correction must be before printing. The default of 4.0 keeps noise out; 3.0 is the floor, below which random noise starts printing occasional characters. A good signal reads 8–9 on the **FEC** meter, so there is plenty of room to raise the squelch on a busy band.

Mode note: The decoder takes control of the demodulation mode and IF passband while active. Both are restored automatically when you disable the decoder. All five variants use **USB**.

Polarity note (FSK variants only): For Weather RTTY you usually need to check **Invert mark / space**. For Maritime FSK (SITOR/NAVTEX-style) and Amateur RTTY, leave it unchecked — amateur RTTY sends mark as the higher radio frequency, and USB keeps it as the higher audio tone, which is the unchecked case. If decoded text is garbled, toggling this checkbox is the first thing to try. The checkbox is hidden for PSK31 and Olivia, which have no mark/space pair.

Letters/figures note (FSK variants only): Baudot carries letters and figures in two separate states, and noise can flip the decoder into the wrong one — which garbles every following character, not just the damaged one. The decoder therefore returns to letters on every space, standard practice that repairs a corrupted shift within a word or two instead of a whole line. The cost is that groups of digits separated by spaces need the sender to repeat the figures shift after each space, as transmitters normally do.

11. SSTV

What it is: Slow-Scan Television transmits still pictures over a normal SSB voice channel, one line at a time, as a frequency-modulated tone between 1500 Hz (black) and 2300 Hz (white). A full picture takes between 36 seconds and 4½ minutes depending on the mode.

Recommended frequencies (USB)

Band	Frequency	Mode	Notes
20m	14.230 MHz	USB	The main international SSTV calling frequency — by far the most active
20m	14.233 MHz	USB	Secondary, used when 14.230 is busy
15m	21.340 MHz	USB	
10m	28.680 MHz	USB	Active during band openings
40m	7.171 MHz	LSB	
80m	3.845 MHz	LSB	Regional, evenings

Sideband is chosen for you. Starting the decoder selects **LSB below 10 MHz** and **USB above it**, following normal amateur practice. On the wrong sideband the tone mapping is inverted and the picture will not decode. If you meet a station that ignores the convention, simply change the sideband by hand — the decoder stays running, clears the frame and starts fresh on the new setting.

Setup

1. Enable the decoder and select **SSTV** from the dropdown. Or simply press the **SSTV** button. The receiver switches sideband automatically — USB above 10 MHz, LSB below.
2. Tune so the picture tones fall in the middle of the passband. A correctly tuned signal has its sync pulses at 1200 Hz and the picture content between 1500 and 2300 Hz.
3. Leave **Mode** on **Auto** unless you already know what is being sent. The image builds line by line as it is received.

Modes

Setting	Picture	Duration
Auto	Detected automatically	—
Martin M1 / M2	320×256 colour	114 s / 58 s
Scottie S1 / S2	320×256 colour	110 s / 71 s
Scottie DX	320×256 colour	269 s
Robot 36 / 72	320×240 colour	36 s / 72 s

Auto works two ways: it reads the **VIS header** — the digital mode code sent in the first 300 ms of a transmission — and, if the header was missed (you tuned in late, or it was lost in QSB), it identifies the mode from the sync-pulse timing instead. Selecting a specific mode forces that mode, but the decoder still verifies the sync before it draws anything, so a wrong choice produces no picture rather than noise.

Reading the output

The status line under the controls reports what the decoder is doing:

Status	Meaning
Waiting for VIS / AUTO lock	Listening; nothing identified yet
Martin M1? verifying sync...	A candidate was found and is being confirmed against the next lines
Martin M1 lock (VIS)	Locked from the VIS header
Martin M1 lock (AUTO)	Locked from sync-pulse timing
Martin M1 lock (MANUAL)	Started by the ● Force button
Candidate rejected (no sync)	The candidate failed confirmation — normal on noise
– sync lost, resetting	The signal disappeared mid-picture
Frame complete	The full picture was received

The detected mode also appears in green next to the **SSTV** title, and the line counter shows progress.

Controls

Button	Action
► Start	Arms the decoder. It does not draw a picture by itself — a lock still has to come from the VIS header or from sync detection.
■ Stop	Stops decoding.
↺ Reset	Clears the canvas and re-arms in place. Use it between pictures, or after retuning.
● Force	Starts drawing immediately in the mode selected in the dropdown, skipping VIS and sync detection entirely. Only available when Mode is not Auto .
Save	Saves the current image as a PNG.

When to use Force. If you can see a picture in the waterfall but the decoder will not lock onto it — an unusual mode, a signal too weak or too distorted for the detectors, or a header you missed — select the mode by hand and press ● **Force**. The frame is anchored at the moment you click, and the mode badge shows **MANUAL** so you can tell it from a VIS or AUTO lock. The decoder still snaps to a real sync pulse if it finds one, so a click that is slightly early or late is corrected.

Because Force skips every safety check, it will happily paint noise if you press it on an empty channel, and it will not stop on its own — press ■ **Stop** or ↺ **Reset**.

Notes

- **Noise will not start the decoder.** Static crashes, sparks and mains hash used to be enough to trigger a decode. Every detection stage now tests that the tone is a genuine tone and confirms the candidate against the following lines before a single pixel is drawn. Expect the decoder to sit quietly on an empty band.
- **The picture drifts diagonally if you are off frequency.** SSTV is unforgiving of tuning error. If lines slant, adjust the dial in small steps and let the next picture start.
- Colour balance and sharpness are fixed; there are no adjustments to make.

12. General Tips

Decoder: ON must be engaged first. The dropdown is disabled (greyed out) until you click the Decoder button to turn it on.

One decoder at a time. Selecting a new decoder from the dropdown automatically stops the previously active one and resets any mode or passband changes it made.

Mode is managed for you. HF FAX and NAVTEX switch the receiver to USB as soon as you select them, and SSTV picks the sideband from the band (LSB below 10 MHz, USB above), and FAX, NAVTEX and FSK also adjust the passband when you click their Tune button. FT8, FT4, FT2, JS8 and WSPR do the same as soon as you select them — from the button row or the dropdown — switching the receiver to USB and setting their own passband: the whole 3 kHz sub-band for the FT8 family, 1350–1650 Hz for WSPR. When you stop the decoder, the band's default mode is restored.

Switching a decoder on no longer interrupts the audio. The decoders run on their own threads, so there is no gap, click or dropout when one starts, stops, or is swapped for another.

System clock accuracy matters. FT8, FT4, and WSPR are time-critical. They decode in fixed UTC-aligned windows. If your computer clock is off by more than 1–2 seconds, decode rates will drop significantly. Use an NTP client to keep your clock accurate.

The noise filters do not reach the decoders. NR, NB, NS and AN are listening aids for your ears only. Every decoder — FT8, FT4/FT2, CW, WSPR, FAX, NAVTEX, FSK/RTTY/PSK31/Olivia, SSTV and the QRSS grabber — taps the audio *before* those filters, so set them however sounds best without worrying about decode quality. For the same reason the decoders keep running while the receiver is muted or squelched: you can silence the speaker and leave a decoder or an overnight QRSS grab collecting. The one thing that does follow what you hear is the Audio Spectrogram, which is meant to show the filtered audio.

Signal quality beats signal strength. Most of these decoders are designed for weak signals. A quieter band with lower noise is often more productive than a loud, interference-filled signal. Use the waterfall and passband controls to identify and avoid QRM before enabling a decoder.

Use the Refresh or Clear buttons liberally. FAX images drift if the dial frequency is slightly off, and text decoders accumulate noise characters. A fresh start after tuning adjustments often produces much cleaner output.

Automatic spot reporting and server graphs

FT8, FT4 and WSPR decodes can also be uploaded automatically by the server itself — FT8/FT4 to PSK Reporter, WSPR to WSPRnet — by an autorun daemon the sysop starts from the admin panel. It is independent of the decoders in your browser: it keeps running whether or not anyone is listening, and nothing you decode in the browser is reported.

The sysop follows it with two counters, which are easy to mistake for each other: the per-decoder tiles count the spots uploaded since the daemon last started, while the number beside each band/mode checkbox is that slot's all-time total and survives restarts. The same panel has a **Graphs** page plotting CPU frequency, CPU load, CPU temperature and users online over the last 15 minutes to 24 hours. Both are described in the [Admin Panel guide](#).

CHAPTER 9

RADE README

RADE_README.md

RADE v1 Digital Voice for PhantomSDR-Plus

RADE (Radio AutoencoDEr) is FreeDV's flagship HF digital voice mode. It uses a Machine Learning / DSP hybrid (the FARGAN neural vocoder) to deliver high-quality speech over HF radio at SNRs as low as -2 dB, in just 1500 Hz of RF bandwidth — narrower than an SSB signal.

This document covers the complete integration of RADE v1 receive support into PhantomSDR-Plus, implemented as a Python sidecar (`rade_helper.py`) that bridges the browser to the `radae_rxe.py` + `lpcnet_demo` decode pipeline.

Installing it — the short way

```
cd ~/PhantomSDR-Plus
./install_rade.sh
```

That is the whole installation. The script installs the system packages through whichever package manager the machine has (`apt`, `pacman`, `dnf` or `zypper`), falls back to `pip` for anything a distribution does not ship, upgrades `websockets` if the distro version is older than the 11.0 the sidecar needs, clones and builds `radae`, checks the model weights, rebuilds the frontend and starts the sidecar. On Ubuntu 22.04 it hands over to `install_rade_ubuntu22.sh` for you. `./install.sh` offers to run it as part of the normal PhantomSDR-Plus installation, so on a fresh machine RADE is already there.

This document explains how RADE works and how to run it — it is not the install procedure. If you need to do the installation by hand, on a system the script does not cover or to repair one step, follow the manual guide linked below; it is the single place the steps are written down. Everything here — the architecture, the patched files, the port, using RADE in the browser, the environment variables, troubleshooting and concurrency measurement — applies to every installation however it was made.

- **Manual installation on Linux** - the by-hand route, step by step, covering Ubuntu, Debian, Fedora, Arch and Raspberry Pi OS

Table of Contents

1. Installing it — the short way
2. How It Works
3. Architecture Overview
4. RADEL vs RADEU
5. Prerequisites
6. Installing by hand
7. The Patched Files
8. Sidecar control
9. Port 8074
10. Using RADE in the Browser
11. Verification and Debugging

- 12. [Environment Variables](#)
 - 13. [Files Changed](#)
 - 14. [Signal Flow Summary](#)
 - 15. [Troubleshooting](#)
 - 16. [Measuring Concurrency on Your Own Hardware](#)
 - 17. [Updating RADE v1](#)
-

How It Works

RADE v1 cannot run in the browser — it requires PyTorch and the FARGAN neural vocoder, which are too large for WASM. The solution is a Python sidecar process (`rade_helper.py`) that runs on the same server as PhantomSDR-Plus.

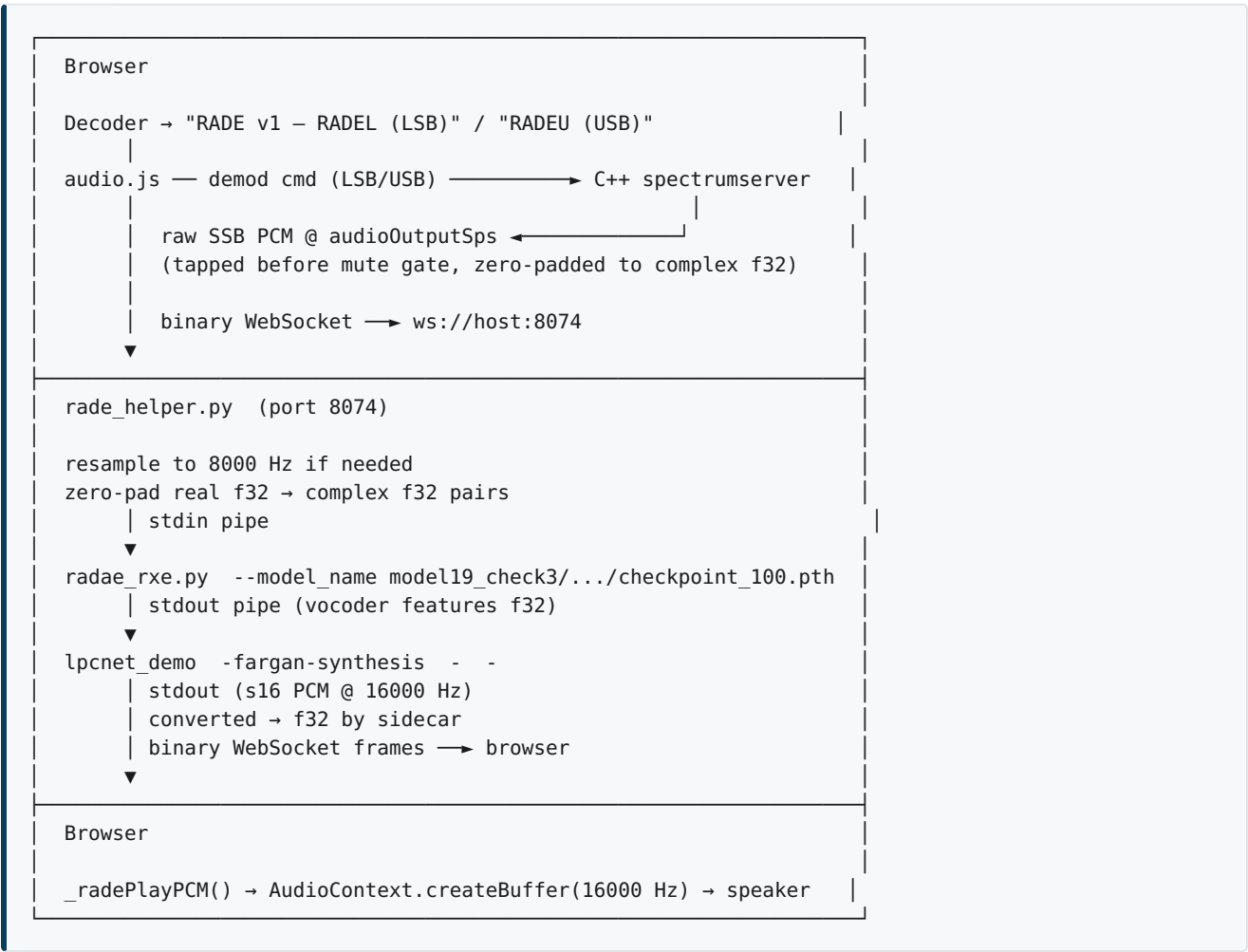
Important: `freedv_rx` from the `codec2` repository does **not** support RADEV1. The RADE decode pipeline lives entirely in the separate `radae` repository by David Rowe (VK5DGR). Do not attempt to use `codec2`'s `freedv_rx` for RADE.

When you select RADE in the browser:

1. The frontend sets the underlying demodulation to USB or LSB — the C++ server demodulates the SSB signal as normal.
2. The raw demodulated PCM is tapped in `audio.js` **before** any mute or squelch gating — `radae_rxe.py` needs continuous input to maintain frame sync.
3. Each PCM chunk is zero-padded from real f32 to complex f32 (real + 0.0 imaginary) — this is what `radae_rxe.py` expects on stdin.
4. The sidecar pipes those samples into `radae_rxe.py`, which outputs vocoder features.
5. `lpcnet_demo -fargan-synthesis` converts the features into s16 speech at 16000 Hz.
6. The sidecar converts s16 → f32 and sends it back to the browser as binary WebSocket frames.
7. The browser plays the decoded speech via the Web Audio API at 16000 Hz.

The C++ server (`spectrumserver.cpp`) is not modified at all.

Architecture Overview



RADEL vs RADEU

RADE v1 is always transmitted over SSB. By convention:

Mode	Sideband	Use on bands
RADEL	LSB	160 m, 80 m, 40 m (≤ 10 MHz)
RADEU	USB	20 m, 17 m, 15 m, 12 m, 10 m (> 10 MHz)

Prerequisites

Requirement	Version	Notes
PhantomSDR-Plus	any	with Vite/Svelte frontend
Linux	Ubuntu 24.04+ / Debian Bookworm+	tested
Python	3.8+	for <code>rade_helper.py</code> and <code>radae_rxe.py</code>
radae repo	latest master	provides <code>radae_rxe.py</code> and <code>lpcnet_demo</code>
cmake	3.10+	to build <code>lpcnet_demo</code> from <code>radae</code>
PyTorch	2.0+	required by <code>radae_rxe.py</code>
Node.js	16+	for <code>npm run build</code>
websockets (Python)	10–16+	<code>pip3 install websockets</code> — all versions supported
matplotlib	any	required by <code>radae_rxe.py</code> at import time
numpy	1.23+	required; also enables resampling
scipy	any	optional, enables accurate resampling

Installing by hand

`./install_rade.sh` does the whole installation. When you want to do it yourself — a system the script does not cover, or one step to repair — the procedure lives in [RADE v1 — Manual Installation Guide](#), and only there. It covers Ubuntu, Debian, Fedora, Arch and Raspberry Pi OS in one pass:

Step 1	System packages, and the Node.js 22+ check
Step 2	Python packages, including the PEP 668 flag and the CPU-only torch wheel
Step 3	Clone and build the radae repository, verify <code>lpcnet_demo</code> and the model weights
Step 4	Verify the decode pipeline offline, before wiring it to the browser
Step 5	Build the PhantomSDR-Plus frontend
Step 6	Sidecar control
Step 7	Open port 8074
Steps 8–10	Start the server, verify RADE, use it in the browser

The rest of this document assumes that is done.

The Patched Files

Copy the following files from the patch set into your PhantomSDR-Plus repository.

New file — place in the repository root, alongside the start script you use:

```
rade_helper.py
```

Patched frontend files — place in `frontend/src/`:

```
audio.js
App.svelte
```

What the patches do

audio.js — 5 changes:

- Constructor: 5 new RADE state fields (`decodeRADE`, `_radeSideband`, `_radeSocket`, `_radeCallback`, `_radeReady`, `_radeNextTime`)
- Pre-boost PCM save: `pcmArrayPreBoost` saved before the FLAC 300× gain boost so RADE receives original amplitude (the boost would saturate `rade_rxe.py`)
- `playAudio()`: RADE PCM tap using pre-boost audio, before mute/squelch gate
- `playAudio()`: `if (this.decodeRADE) return guard` — suppresses raw SSB playback while RADE decoded speech plays
- New methods: `setRADEDecoding()`, `setRADECallback()`, `_radePlayPCM()` with gapless scheduled playback via `_radeNextTime` clock (plays at **16000 Hz** — the output rate of `lpcnet_demo`)

Each Svelte variant — 6 changes:

- `demodulationDefaults`: `RADEL {type: 'LSB', offsets: [2200, -700]}`, `RADEU {type: 'USB', offsets: [-700, 2200]}` — passband starts 700 Hz from carrier, 1500 Hz wide
- State variables: `radeEnabled`, `radeConnected`, `radeSynced`, `radeSnr`, `_radeDeactivate()`
- `_radeDeactivate()`: stops the decoder and **restores the band default mode** from `bands-config.js` — identical behaviour to FAX, NAVTEX, FSK deactivation
- `_deactivateAll()`: RADE cleanup line
- `activateSelectedDecoder()`: `radel` and `radeu` branches
- Decoder dropdown: two new `<option>` entries
- Status panel: connection dot, sync/SNR status, error banner

Startup scripts

The receiver launchers (`start-rx888mk2.sh`, `start-airspyhf.sh`, `start-rtl.sh`, `start-rspla.sh`, `start-fobos-hf.sh`, `start-fobos.sh`, `start-hackrf.sh`) start the sidecar themselves once the server is up, and `stop-websdr.sh` stops it with the server. `rade.sh` remains for running the sidecar standalone — see [Sidecar control](#).

Sidecar control

You normally do not need a separate RADE control script. The receiver launchers start the sidecar once the server is confirmed up, restart it if it exits, and report its state in their closing summary — ✓ RADE sidecar running, or a • note explaining why it was not activated (not installed, `RADE_ENABLED=0`, or no python3). `./stop-websdr.sh` stops it along with the server. To run the server without RADE, start it with `RADE_ENABLED=0`.

`rade.sh` ships for standalone use — running the sidecar without the PhantomSDR-Plus launcher, or testing it on its own:

```
chmod +x ~/PhantomSDR-Plus/rade.sh
cd ~/PhantomSDR-Plus
./rade.sh start      # start sidecar with self-restarting watchdog
./rade.sh stop       # stop sidecar + watchdog + all lpcnet_demo children
./rade.sh restart    # stop then start cleanly
./rade.sh status     # show running / stopped + process info
```

Sidecar activity is logged to `~/PhantomSDR-Plus/rade.log`:

```
tail -f ~/PhantomSDR-Plus/rade.log
```

Do not mix the two while the server is running. If a launcher's watchdog is active it will bring the sidecar back within about five seconds, so `./rade.sh stop` looks like it failed. To stop RADE while the server runs, either start the server with `RADE_ENABLED=0`, or stop everything with `./stop-websdr.sh`.

If you do run `rade.sh` standalone, **always use** `./rade.sh stop` — `kill -f rade_helper.py` alone will not beat its own watchdog, which respawns the process within 3 seconds.

Port 8074

The sidecar listens on TCP port **8074** and the browser connects to it directly, so the port has to be reachable from outside. Opening it — `ufw`, `firewalld`, `iptables`, the router NAT rule, and the Nginx proxy to use when an ISP blocks the port outright — is [Step 7 of the manual guide](#).

Two things worth repeating here, because they cost the most time:

```
# Is the sidecar listening?
ss -tlnp | grep 8074
```

NAT hairpin warning: testing with `curl` from the server to its own public hostname often returns `Connection refused` even when the port is open — many routers do not loop traffic back. Always test from an outside machine, or use <https://portchecker.co>.

Using RADE in the Browser

1. Open your PhantomSDR-Plus web interface
2. Find active stations at qso.freedv.org
3. Tune to the station's dial frequency
4. In **Decoder Options**, select:
5. **RADE v1 — RADEL (LSB)** for 40 m / 80 m / 160 m
6. **RADE v1 — RADEU (USB)** for 20 m / 17 m / 15 m / 12 m / 10 m
7. Click **Decoder: ON**

The RADEL / RADEU buttons (one-touch)

Steps 4 and 5 can be replaced by a single press. A pair of **RADEL** / **RADEU** buttons is available in three places:

- next to the **Modes selector** heading on the main panel;

- inside the **Modes** popup window;
- inside the **Bands** popup window.

Pressing one selects the decoder, switches the decoder ON, closes the popup you pressed it in, and scrolls the RADE panel into view. The button turns blue while RADE is running. **Press the same button again to switch RADE off** — the panel closes and mode and passband revert as described below. The buttons, the **Decoder Options** dropdown and the ON/OFF button all share the same state.

Panel indicator states

Indicator	Meaning
Red — "Connecting to sidecar..."	Port 8074 unreachable or sidecar not running
Yellow — "Searching for signal..."	Sidecar connected, no RADE frame detected yet
Green — "Synced · SNR x.x dB"	Decoding — speech is playing

When you turn the decoder OFF

Mode and passband automatically revert to the correct default for the current frequency as defined in `bands-config.js` — identical to FAX, NAVTEX, FSK.

Verification and Debugging

Is the sidecar running?

```
ps aux | grep rade_helper | grep -v grep
ss -tlnp | grep 8074
```

Watch live activity

```
tail -f ~/PhantomSDR-Plus/rade.log
```

When a browser connects:

```
[RADE] client connected: x.x.x.x:XXXXX
[RADE] x.x.x.x:XXXXX sps=12000 sideband=LSB
[RADE] x.x.x.x:XXXXX spawning pipeline (torch_threads=1)
```

Quick WebSocket smoke test

```
python3 - &&& 'EOF'
import asyncio, websockets, json

async def test():
    async with websockets.connect('ws://localhost:8074') as ws:
        await ws.send(json.dumps({'type': 'init', 'sps': 8000, 'sideband': 'LSB'}))
        print(await ws.recv()) # expect: {"type": "status", "connected": true}

asyncio.run(test())
EOF
```

Browser console (F12)

```
[RADE] ► ENABLED LSB @ 12000 Hz → helper ws://localhost:8074
```

Good — WebSocket opened. The Hz value matches your server's `audioOutputSps` (typically 8000–12000 Hz).

```
[RADE] sidecar socket error
```

Port 8074 is unreachable — check firewall and router NAT rule.

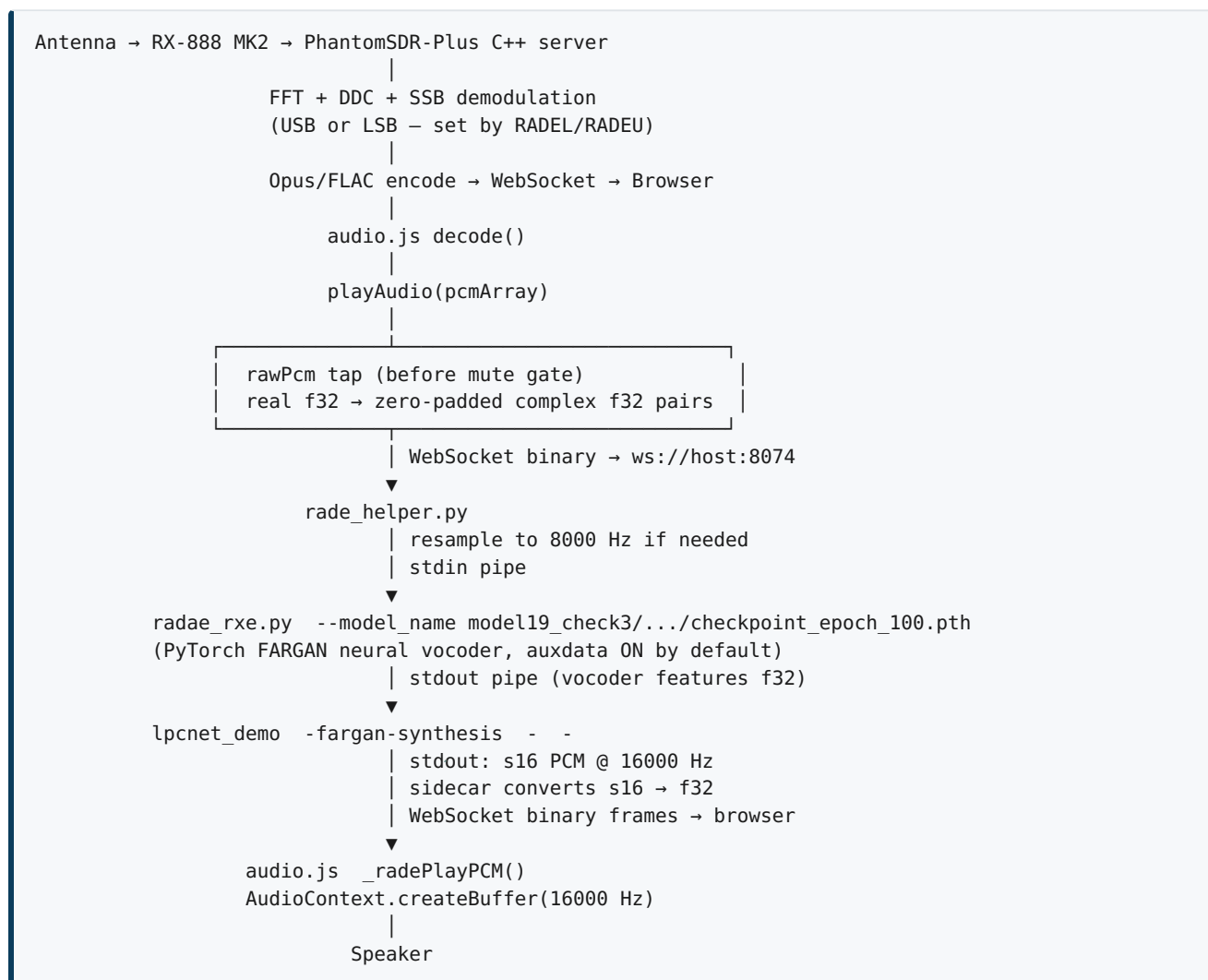
Environment Variables

Variable	Default	Description
<code>RADE_HELPER_PORT</code>	8074	TCP port the sidecar listens on
<code>RADE_HELPER_HOST</code>	0.0.0.0	Bind address (127.0.0.1 if behind a proxy)
<code>RADAE_DIR</code>	<code>~/radae</code>	Root of the radae repository
<code>RADE_MODEL</code>	<code>RADAE_DIR/model19_check3/checkpoints/checkpoint_epoch_100.pth</code>	Model weights
<code>LPCNET_DEMO</code>	<code>RADAE_DIR/build/src/lpcnet_demo</code>	<code>lpcnet_demo</code> binary
<code>RADE_AUXDATA</code>	1	Set to 0 to pass <code>--noauxdata</code> to <code>radae_rxe.py</code>
<code>RADE_TORCH_THREADS</code>	1	PyTorch/OpenBLAS threads per <code>radae_rxe.py</code> instance — limits CPU usage
<code>RADE_PIN_CORES</code>	1	Pin each decode pipeline to its own core set; 0 disables pinning
<code>RADE_CORES_PER_CLIENT</code>	2	Cores allocated per client when pinning is enabled

Files Changed

File	Type	Notes
rade_helper.py	New	Python sidecar: WebSocket server + two-process decode pipeline
frontend/src/audio.js	Modified	4 patches
frontend/src/App.svelte	Modified	6 patches
rade.sh	New	RADE sidecar control script for standalone use (start/stop/restart/status)
start-rx888mk2.sh	Modified	starts, watchdogs and reports the sidecar; same for start-airspyhf.sh, start-rtl.sh, start-rspla.sh, start-fobos-hf.sh, start-fobos.sh, start-hackrf.sh
stop-websdr.sh	Modified	stops the sidecar along with the server
spectrumserver.cpp	Unchanged	No C++ modifications required

Signal Flow Summary



Troubleshooting

Red banner — "Sidecar not reachable"

```
# Is sidecar running?
ps aux | grep rade_helper | grep -v grep

# Start it manually for testing
python3 ~/PhantomSDR-Plus/rade_helper.py &

# Check port is open externally – use portchecker.co NOT curl from the server
# (curl from the server uses NAT loopback and gives false "Connection refused")
```

Port 8074 closed on portchecker.co despite iptables rule

The ISP may be filtering the port, or the router NAT rule is missing. Options:

1. Try a different port: `RADE_HELPER_PORT=8080 python3 rade_helper.py`
2. Proxy through your existing public port via Nginx (see [Step 7 of the manual guide](#))

`radae_rxe.py: error: unrecognized arguments: model_path`

The model path must use `--model_name`, not a positional argument:

```
# WRONG – positional argument
python3 radae_rxe.py model19_check3/checkpoints/checkpoint_epoch_100.pth

# CORRECT – named argument
python3 radae_rxe.py --model_name model19_check3/checkpoints/checkpoint_epoch_100.pth
```

`ModuleNotFoundError: No module named 'matplotlib'`

```
pip3 install matplotlib
```

`radae_rxe.py` imports `matplotlib` unconditionally at the top of the file.

`ModuleNotFoundError: No module named 'torch'`

```
pip3 install torch
```

`lpcnet_demo: No such file or directory`

The `radae` build did not complete. Rebuild:

```
cd ~/radae/build
cmake .. && make -j$(nproc)
ls src/lpcnet_demo      # should exist now
```

`size mismatch error in inference.sh`

`model19_check3` requires `--auxdata` when encoding with `inference.sh`:

```
./inference.sh model19_check3/checkpoints/checkpoint_epoch_100.pth \
wav/brian_g8sez.wav /dev/null \
--rate_Fs --pilots --pilot_eq --eq_ls --cp 0.004 \
--write_rx rx.f32 --auxdata # ← required for model19_check3
```

Note: when **decoding** with `radae_rxe.py`, `--auxdata` is the default — do not pass it.

Yellow indicator — "Searching for signal" — never syncs

- Confirm the correct sideband: RADEL for ≤ 10 MHz, RADEU for > 10 MHz
- Check qso.freedv.org to confirm a station is currently transmitting
- RADE v1 uses 30 carriers in 1500 Hz BW — it appears as a compact cluster on the waterfall
- Allow up to 1.5 seconds for acquisition

underrun!!! from aplay during the offline pipeline test

Expected during offline file testing only. `radae_rxe.py` processes slower than file I/O, causing the audio buffer to starve. This does not occur during live reception because the browser feeds audio at real-time rate.

ConnectionClosedError: received 1011 (internal error)

The sidecar accepted the WebSocket connection but crashed internally before responding. This is caused by an incompatible version of `rade_helper.py` — an older version attempted to pass an `asyncio StreamReader` as a subprocess stdin, which fails silently and closes with error 1011.

Fix: Replace `rade_helper.py` with the current version from the patch set. The current version uses `os.pipe()` for the inter-process pipe and is compatible with websockets 10.x through 16.x+.

Multiple simultaneous users

Each browser connection spawns its own independent `radae_rxe.py` + `lpcnet_demo` pair, so each user can tune to a different frequency freely.

By default `radae_rxe.py` uses **all available CPU cores** for PyTorch matrix operations, causing ~900% CPU usage per instance. `rade_helper.py` limits this with two optimisations:

1. `OMP_NUM_THREADS=1` (and MKL/OpenBLAS equivalents) — limits PyTorch to 1 thread
2. Numpy-vectorised audio conversion functions — eliminates Python loop overhead

Result: each instance uses roughly **~8-10%** of one core, sufficient for real-time RADE decode. If you hear audio dropouts, raise to 2 threads:

```
RADE_TORCH_THREADS=2 ./start-rx888mk2.sh # or your receiver's launcher
# standalone: RADE_TORCH_THREADS=2 ./rade.sh restart
```

Simultaneous RADE users	Approximate CPU
1	~9%
5	~45%
10	~90%
20	~180%

On an i5-12450H (12 threads) with spectrumserver using ~42% and rx888_stream ~11%, you have roughly **1000% headroom** — enough for **20+ simultaneous** RADE users before CPU becomes a concern.

The table above assumes CPU cost scales linearly with users. Measurement says it does not. A `rade_loadtest.py` run on that same i5-12450H showed per-listener CPU well below this table up to ~24 listeners, then rising sharply above it, with the machine stopped by **package temperature at 32 listeners** — not by CPU and not by bandwidth. Treat these figures as a rough guide for small numbers only and measure your own box: see [Measuring Concurrency on Your Own Hardware](#). Note the load test drives the pipeline with noise rather than a real RADE signal, so absolute CPU under genuine traffic may differ; the *shape* of the curve is the reliable part.

Measuring Concurrency on Your Own Hardware

The numbers above are from one specific box. `rade_loadtest.py` (repository root) measures the same thing on **yours** — it ramps synthetic RADE listeners against the sidecar in steps and reports the last listener count your machine sustained before something gave way.

After each step it lets the box settle, then samples package temperature, total CPU, available RAM, swap, and the combined RSS of every `radac_rxe.py` and `lpcnet_demo` process. It stops at the first **knee** — whichever limit is breached first — and prints the last stable count.

Requirements

```
sudo apt install python3-websockets python3-psutil
```

Run it **on the SDR host** (it reads local sensors and process memory), with spectrumserver and the RADE sidecar already running:

```
cd ~/PhantomSDR-Plus
python3 rade_loadtest.py
```

Ctrl-C tears down all connections cleanly at any point.

Before the first run

Open the file and check the two blocks at the top:

- **CONFIG** — `WS_URL` defaults to `ws://127.0.0.1:8074`; change it if you moved the sidecar off its default port. `RADE_SPS` (12000) and `RADE_SIDE BAND` (USB) should match what your frontend reports.
- **handshake_messages()** — the single `init` frame it sends must match the RADE `init` in `frontend/src/audio.js`. If that handshake has drifted, every listener fails to connect and the test ends immediately with connect failures.

Knee thresholds

Tune these in `CONFIG` to taste — they are deliberately conservative:

Threshold	Default	Stops when
TEMP_KNEE_C	85.0	Package temperature reaches this ceiling
MIN_AVAIL_MB	800	Available RAM falls below this
SWAP_GROWTH_MB	50	Swap grows past the baseline taken at startup
FAIL_LIMIT	3	This many listeners fail to connect in one step
MAX_LISTENERS	60	Hard stop even if no knee is ever hit

Ramp shape is controlled by `STEP` (listeners added per step, default 2), `SETTLE_S` (25 s of settling before sampling — this is what lets heat build) and `SAMPLE_S` (5 s CPU averaging window). A full run to the default 60-listener ceiling therefore takes roughly 15 minutes.

Reading the results

Each step prints a line and appends a row to `rade_loadtest.csv`:

Column	Meaning
listeners	Synthetic listeners connected at this step
connect_fails	Listeners that failed to establish or hold a connection
pkg_c	Package temperature, median of 5 samples (spike-resistant)
cpu_pct	Whole-system CPU percent, not per-listener
avail_mb	Available RAM
swap_mb	Swap in use
dec_rss_mb	Combined RSS of all decoder processes
dec_procs	Decoder process count — expect 2 per listener (<code>rade_rxe.py</code> + <code>lpcnet_demo</code>)

A measured run

On the i5-12450H referenced above, a full run ended like this:

```
KNEE at n=32: temp 91.0°C ≥ 85.0
Last stable concurrency: 30 RADE listeners
```

Heat was the binding constraint, and it was not close. At the knee there was still 7004 MB of RAM available — 6.2 GB clear of the `MIN_AVAIL_MB` cutoff — and zero connect failures on every step.

Two columns are worth reading carefully:

dec_rss_mb overstates real memory cost. It grows at a very steady 295 MB per listener (292, 294, 294 ... 295 across all 16 steps), but *available RAM* only falls about **202 MB per listener**. The two processes behind each listener share library pages, and RSS counts those pages once per process. Extrapolating the `avail_mb` slope, the memory knee would arrive near **63 listeners** — past the 60-listener hard stop, so on this box it can never trip. Size memory from `avail_mb`, not from `dec_rss_mb`.

cpu_pct is flat and then it is not. It sits at ~5.3% of the whole machine through 24 listeners, then goes superlinear:

Listeners	24	26	28	30	32
cpu_pct	6.7%	18.4%	30.1%	49.6%	65.2%

That is a 10× rise in CPU while listener count grows by a third. The inflection reproduces at the same point across runs, so treat it as a property of the box rather than noise — per-listener CPU cost is roughly 3.4% of one core below the inflection and about 20% above it. Do not extrapolate a per-user CPU figure taken at low listener counts.

The test has no CPU knee. The four cutoffs are temperature, available RAM, swap growth and connect failures — CPU is recorded but never ends the run. In the run above CPU reached 65% and would have kept climbing had temperature not tripped first. If you care about a CPU ceiling, watch the column yourself or add a threshold.

What this does and does not prove. Each synthetic listener streams low-amplitude random noise, not a real RADE signal. That is enough to make the full decode pipeline run and consume resources, so the resource figures are meaningful — but the test says nothing about decode *quality* or audio continuity under load. Confirm those with real listeners on a real signal.

If cores saturate before the knee

RADE pins one thread hard per instance, so individual cores can reach ~90 °C while total CPU still looks idle. `rade_helper.py` spreads pipelines across cores round-robin to counter this. If a run shows early thermal knees, widen the core allocation per client:

```
RADE_CORES_PER_CLIENT=3 ./rade.sh restart
```

No file edit is needed — this is an environment variable (see [Environment Variables](#)); `RADE_PIN_CORES=0` disables pinning entirely.

Updating RADE v1

The PhantomSDR-Plus integration (`rade_helper.py`, frontend patches) is a bridge only — all RADE decode logic lives in the `radae` repository. Updates are therefore almost always a simple `git pull` + rebuild with no changes to PhantomSDR-Plus itself.

Standard update (new code, same model)

```
# 1. Pull latest radae code
cd ~/radae
git pull

# 2. Rebuild lpcnet_demo (in case C code changed)
cd build
cmake ..
make -j$(nproc)

# 3. Restart the sidecar – no server restart needed
cd ~/PhantomSDR-Plus
./rade.sh restart
```

That is all. No frontend rebuild, no server restart, no file edits.

New model weights only

If a new checkpoint is released (e.g. `model20`) without code changes, point the sidecar at the new weights using the env var — no file editing required:

```
# One-off: start with new model
RADE_MODEL=~/.rade/model20/checkpoints/checkpoint_epoch_100.pth ./rade.sh start

# Or permanently – add to your shell profile (~/.bashrc):
export RADE_MODEL=~/.rade/model20/checkpoints/checkpoint_epoch_100.pth
```

What each type of change requires

What changed in radea	Action required
New model weights (new .pth file)	Set RADE_MODEL env var, ./rade.sh restart
Code change in radea_rxe.py	git pull, ./rade.sh restart
lpcnet_demo C code changed	git pull, rebuild, ./rade.sh restart
radea_rxe.py renamed or moved	Update RADAЕ_RX path in rade_helper.py (one line)
--model_name argument renamed	Update radea_cmd in rade_helper.py (one line)
New output sample rate (≠ 16000 Hz)	Update SPS_OUT in rade_helper.py + createBuffer() in audio.js
RADE v2 uses a different binary	Update RADAЕ_RX in rade_helper.py (one line)

Verifying the update worked

After restarting, confirm the new code is running:

```
# Check sidecar picked up new radea_rxe.py
./rade.sh status

# Tail log to see startup lines
tail -20 ~/PhantomSDR-Plus/rade.log
```

The log should show the model path you expect:

```
[RADE] model : /home/sv1btl/radea/model19_check3/checkpoints/checkpoint_epoch_100.pth
```

Staying up to date

Subscribe to the radea repository releases page to be notified of new models and code updates:

```
https://github.com/drowe67/radea/releases
```

Check the FreeDV blog for announcements about new RADE waveforms and models:

```
https://freedv.org/blog/
```

Developed and tested on PhantomSDR-Plus (sv1btl/PhantomSDR-Plus fork) with RX-888 MK2 on Ubuntu 24.04. The C++ server is not modified.

RADE is developed by David Rowe VK5DGR and the FreeDV team. See freedv.org/radio-autoencoder.

Full uninstall and fresh re-install

Here's the full teardown before re-running `install_rade.sh`:

1. Stop the sidecar `cd ~/PhantomSDR-Plus && ./rade.sh stop`
2. Remove the radae repo and build `rm -rf ~/radae`
3. Remove torch (installed in user local) `pip3 uninstall -y torch` `rm -rf ~/.local/lib/python3.11/site-packages/torch*`
4. Remove apt Python packages (optional — skip if used by other things) `sudo apt-get remove -y python3-numpy python3-scipy python3-matplotlib python3-websockets` `sudo apt-get autoremove -y`
5. Verify everything is gone `python3 -c "import torch" 2>&1` # should say `ModuleNotFoundError` `ls ~/radae 2>&1` # should say `No such file or directory`
6. Fresh install `chmod +x ~/PhantomSDR-Plus/install_rade.sh` `~/PhantomSDR-Plus/install_rade.sh`

CHAPTER 10

RADE General Install (Linux)

RADE_General_INSTALL_MANUAL_LINUX.md

RADE v1 — Manual Installation Guide

All Linux — Ubuntu / Debian / Fedora / Arch / Raspberry Pi OS · PhantomSDR-Plus

Pre-condition: The patched files (rade_helper.py, rade.sh, audio.js, App.svelte) are already placed in the PhantomSDR-Plus directory tree. This guide builds everything around them.

IMPORTANT

You do not need this guide for a normal installation. ./install_rade.sh in the PhantomSDR-Plus folder does every step below for you — system packages, the Python modules, the radae build, the model-weights check and the sidecar start — and it works on apt, pacman, dnf and zypper systems alike. ./install.sh (and the four distro installers) offer to run it for you as part of the normal installation. Follow this guide only for a manual setup, for a system the script does not cover, or to repair one step by hand.

Raspberry Pi users: this is your guide too. Raspberry Pi OS is Debian, so follow the Debian blocks throughout, and read the **Raspberry Pi / Bookworm** notes where they appear — they cover the two things that differ on a Pi: PEP 668 and the CPU-only torch wheel.

Step 1 — System packages

Choose the block that matches your distribution.

Ubuntu / Debian / Raspberry Pi OS

```
sudo apt update
sudo apt install -y \
    build-essential cmake git \
    python3 python3-pip \
    nodejs npm \
    alsa-utils
```

Fedora / RHEL / Rocky

```
sudo dnf install -y \
    gcc gcc-c++ cmake git \
    python3 python3-pip \
    nodejs npm \
    alsa-utils
```

Arch / Manjaro

```
sudo pacman -Sy --needed \
    base-devel cmake git \
    python python-pip \
    nodejs npm \
    alsa-utils
```

Node.js version check (all distros)

RADE's frontend build requires Node.js 22 or later:

```
node --version
```

If below 22.x, install it with `nvm` (any distro, no root needed):

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.4/install.sh | bash
export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh"
nvm install 22 && nvm use 22 && nvm alias default 22
```

Not NodeSource. `deb.nodesource.com` and `rpm.nodesource.com` now answer HTTP 403 on every repository path, so the old `curl -fsSL https://deb.nodesource.com/setup_NN.x | sudo -E bash -` line no longer works — and on Debian/Ubuntu it leaves an `apt` source behind that breaks every later `apt` update. The install scripts use `nvm` for the same reason.

Step 2 — Python packages

Ubuntu 24.04 and earlier / Fedora / Arch

Plain `pip3 install` works on these systems:

```
pip3 install websockets matplotlib numpy scipy

# torch — CPU-only build (~150–250 MB, avoids the ~3 GB CUDA wheel)
# Use this if the server has no GPU, which is the typical case for a WebSDR
pip3 install torch --index-url https://download.pytorch.org/whl/cpu
```

If your server **does** have an NVIDIA GPU and CUDA installed, you can omit `--index-url` to get the full CUDA build. There is no performance benefit for RADE — `radar_xe.py` uses PyTorch for CPU matrix ops only.

Ubuntu 23.04+ / Debian Bookworm+ / Raspberry Pi OS (PEP 668 systems)

These distributions block bare `pip3 install`. Add the flag:

```
# All packages except torch
pip3 install --break-system-packages websockets matplotlib numpy scipy

# torch — CPU-only build
pip3 install --break-system-packages torch \
  --index-url https://download.pytorch.org/whl/cpu
```

Raspberry Pi / Bookworm — two rules that bite:

1. Every `pip3 install` requires `--break-system-packages` (PEP 668 enforcement). Without it the install is blocked entirely.
2. Plain `pip3 install torch` downloads the CUDA wheel (~3 GB). On a Pi there is no CUDA — the `--index-url` above is what gets you the lean ARM64 wheel (~150 MB) instead.

Alternatively, use a virtual environment to avoid the flag entirely:

```
python3 -m venv ~/rade-venv
source ~/rade-venv/bin/activate
pip install websockets matplotlib numpy scipy torch \
  --index-url https://download.pytorch.org/whl/cpu
```

If using a venv, prefix all subsequent `python3` calls in this guide with `source ~/rade-venv/bin/activate` or use the full path `~/rade-venv/bin/python3`.

Verify

```
python3 -c "import websockets, matplotlib, torch, numpy, scipy; print('All OK')"
```

Expected output:

```
All OK
```

Step 3 — Clone and build the radae repository

The RADE decoder lives in a separate repository from codec2. `freedv_rx` from codec2 does **not** support RADE v1.

```
# Remove any previous incomplete clone
rm -rf ~/radae

# Clone
git clone https://github.com/drowe67/radae.git ~/radae
cd ~/radae

# Build
mkdir build && cd build
cmake ..
make -j$(nproc)
```

Verify lpcnet_demo was built

```
ls -la ~/radae/build/src/lpcnet_demo
```

Expected: the binary is present and executable.

Verify model weights are present

```
ls ~/radae/model19_check3/checkpoints/checkpoint_epoch_100.pth
```

Expected: the `.pth` file is listed. The weights are bundled in the repository — no separate download needed.

Step 4 — Verify the decode pipeline

This step confirms the full offline pipeline works before wiring it to the browser. Run from `~/radae`:

```
cd ~/rade
```

4a — Generate a RADE-encoded test signal

```
./inference.sh model19_check3/checkpoints/checkpoint_epoch_100.pth \
wav/brian_g8sez.wav /dev/null \
--rate_Fs --pilots --pilot_eq --eq_ls --cp 0.004 \
--write_rx rx.f32 --auxdata
```

Wait for it to finish. The last lines printed should be:

```
loss: 0.741 Auxdata BER: 0.012
```

4b — Decode and play

```
cat rx.f32 \
| python3 rade_rxe.py \
--model_name model19_check3/checkpoints/checkpoint_epoch_100.pth \
| ./build/src/lpcnet_demo -fargan-synthesis - - \
| aplay -f S16_LE -r 16000
```

You should hear a voice. The output shows sync acquisition:

```
1 state: search    ...
5 state: sync      ... SNRdB: 4.03 uw_err: 0
Playing raw data 'stdin' : Signed 16 bit Little Endian, Rate 16000 Hz, Mono
```

underrun!!! messages during this test are expected and harmless. They occur because `radae_rxe.py` processes slower than file I/O. They do not appear during live reception — the browser feeds audio at real-time rate.

4c — Verify the sidecar starts correctly

```
python3 ~/PhantomSDR-Plus/rade_helper.py
```

Expected output (no WARNING lines):

```
[RADE] helper starting on ws://0.0.0.0:8074
[RADE] rade_rx.py      : /home/<user>/rade/rade_rxe.py
[RADE] model          : /home/<user>/rade/model19_check3/checkpoints/checkpoint_epoch_100.pth
[RADE] lpcnet_demo    : /home/<user>/rade/build/src/lpcnet_demo
[RADE] auxdata        : ON (default)
[RADE] torch threads  : 1 per instance (RADE_TORCH_THREADS to override)
[RADE] architecture   : per-connection (each user tunes independently)
[RADE] listening      - waiting for PhantomSDR-Plus clients
```

Press **Ctrl+C** to stop.

`<user>` in the paths above is whichever account you are logged in as — on a stock Raspberry Pi OS image that makes them `/home/pi/rade/...`

Step 5 — Build the PhantomSDR-Plus frontend

```
cd ~/PhantomSDR-Plus/frontend

# Only if node_modules is missing:
npm install

cd ~/PhantomSDR-Plus
./recompile.sh
```

Watch for Vite/acorn parser errors. The patched files deliberately avoid `?.`, `??`, and bare `catch {}` to comply with the acorn constraint.

Step 6 — Sidecar control

You normally do not need a separate RADE control script. The start scripts (`start-rx888mk2.sh`, `start-airspyhf.sh`, `start-rtl.sh`, `start-rspla.sh`, `start-fobos-hf.sh`, `start-fobos.sh`, `start-hackrf.sh`) start the sidecar themselves once the server is up, restart it if it exits, and report its state; `stop-websdr.sh` stops it along with the server. Go straight to Step 7.

`rade.sh` still ships for standalone use — running the sidecar without the PhantomSDR-Plus launcher, or testing it on its own. Its commands, its log, and the watchdog caveats that go with it are in [Sidecar control](#).

Step 7 — Open port 8074

The sidecar listens on TCP port **8074**. The browser connects directly to this port. You must open it manually.

ufw (Ubuntu / Debian / Raspberry Pi OS)

```
sudo ufw allow 8074/tcp && sudo ufw reload
```

firewalld (Fedora / RHEL / Rocky)

```
sudo firewall-cmd --add-port=8074/tcp --permanent
sudo firewall-cmd --reload
```

iptables (any distro, permanent)

```
sudo iptables -I INPUT -p tcp --dport 8074 -j ACCEPT

# Ubuntu / Debian / Raspberry Pi OS — persist across reboots:
sudo apt install iptables-persistent
sudo netfilter-persistent save

# Fedora / RHEL — persist:
sudo service iptables save
```

Router

Add a NAT/port-forward rule: **TCP 8074 → server LAN IP : 8074**

Test from outside

Use <https://portchecker.co> and check port 8074 against your public hostname. Do **not** test with `curl` from the server itself — NAT hairpin gives false "Connection refused" results even when the port is open.

Alternative — Nginx proxy (if port 8074 is blocked by ISP)

Add inside your existing `server {}` block:

```
location /rade {
    proxy_pass          http://127.0.0.1:8074;
    proxy_http_version 1.1;
    proxy_set_header    Upgrade      $http_upgrade;
    proxy_set_header    Connection  "upgrade";
    proxy_set_header    Host         $host;
    proxy_read_timeout  3600s;
}
```

```
sudo nginx -t && sudo nginx -s reload
```

Then edit `audio.js` inside `setRADEDecoding()`:

```
// Change:
var helperUri = uri || ('ws://' + window.location.hostname + ':8074');
// To:
var helperUri = uri || ('ws://' + window.location.host + '/rade');
```

Rebuild the frontend after this change (`./recompile.sh`).

Step 8 — Start the server

Start PhantomSDR-Plus as usual. The launcher starts RADE for you, once the receiver and `spectrumserver` are both confirmed running:

```
cd ~/PhantomSDR-Plus
./start-rx888mk2.sh      # or start-airspyhf.sh,
                        # start-rtl.sh, start-rspla.sh,
                        # start-fobos-hf.sh, start-fobos.sh,
                        # start-hackrf.sh
```

Use the launcher that matches your receiver. Each one includes its own watchdog and log; `./stop-websdr.sh` stops whichever is running, RADE included.

The launcher mirrors its log to the terminal and finishes with a summary line for the sidecar — `✓ RADE sidecar running`, or a `•` note explaining why it was not activated (not installed, `RADE_ENABLED=0`, or no python3). A missing sidecar never stops the server from starting.

Step 9 — Verify RADE

The sidecar was already started by Step 8. Confirm it:

```
cd ~/PhantomSDR-Plus
pgrep -af rade_helper.py    # should print one python3 process
tail -f rade.log
```

Expected log:

```
[RADE] sidecar starting at ...
[RADE] helper starting on ws://0.0.0.0:8074
[RADE] radae_rx.py      : /home/<user>/radae/radae_rxe.py
[RADE] model           : /home/<user>/radae/model19_check3/checkpoints/checkpoint_epoch_100.pth
[RADE] lpcnet_demo     : /home/<user>/radae/build/src/lpcnet_demo
[RADE] auxdata         : ON (default)
[RADE] torch threads   : 1 per instance
[RADE] architecture   : per-connection (each user tunes independently)
[RADE] listening       - waiting for PhantomSDR-Plus clients
```

Step 10 — Using RADE in the browser

1. Open your PhantomSDR-Plus web interface
2. Find active stations at <https://qso.freedv.org>
3. Tune to the station's dial frequency
4. In **Decoder Options**, select:
5. **RADE v1 — RADEL (LSB)** for 40 m / 80 m / 160 m (≤ 10 MHz)
6. **RADE v1 — RADEU (USB)** for 20 m / 17 m / 15 m / 12 m / 10 m (> 10 MHz)
7. Click **Decoder: ON**

Steps 4 and 5 can be replaced by a single press: **RADEL / RADEU** buttons sit next to the **Modes selector** heading and inside the **Modes** and **Bands** popup windows. One press selects the decoder, switches it ON and scrolls the RADE panel into view; press the same button again to switch RADE off. See the [RADE manual](#).

Indicator	Meaning
Red — "Connecting to sidecar..."	Port 8074 unreachable or sidecar not running
Yellow — "Searching for signal..."	Sidecar connected, no RADE frame detected yet (allow ~1.5 s)
Green — "Synced · SNR x.x dB"	Decoding — speech is playing

CPU note

Each RADE user uses ~8–10 % of one core (PyTorch thread count capped at 1 by the sidecar). If you hear audio dropouts, raise to 2 threads:

```
RADE_TORCH_THREADS=2 ./start-rx888mk2.sh      # or your receiver's launcher
```

The launcher passes its environment through to the sidecar, so setting the variable on the start command is enough. (Standalone: `RADE_TORCH_THREADS=2 ./rade.sh restart.`)

Simultaneous users	Approximate CPU
1	~9 %
5	~45 %
10	~90 %
20	~180 %

These figures were measured on x86_64. A Raspberry Pi decodes RADE fine, but the per-user cost is higher and the table above does not transfer — measure your own board with `top` while a user is synced before advertising a user limit.

Updating RADE in future

```
cd ~/rade
git pull
cmake -S . -B build && make -C build -j$(nproc)

cd ~/PhantomSDR-Plus
./start-rx888mk2.sh          # or your receiver's launcher
```

Re-running the launcher restarts the sidecar along with the server. No frontend rebuild is needed unless `rade_helper.py` itself changed.

Tested on Ubuntu 24.04 (x86_64) and on Raspberry Pi 4 / Raspberry Pi OS Bookworm (ARM64, Python 3.11). PhantomSDR-Plus fork: [sv1bt/PhantomSDR-Plus](https://github.com/sv1bt/PhantomSDR-Plus). RADE developed by David Rowe VK5DGR and the FreeDV team — <https://freedv.org>

CHAPTER 11

Editing Variants

EDITING_VARIANTS.md

PhantomSDR-Plus — How to Edit the Frontend Variants

(after the four `App__*_smeter_.svelte` files were merged into one `App.svelte`)

1. What the Variants Are

Two independent choices, giving four combinations:

```
smeter = "analog"    the moving-needle meter face
      = "digital"    the segmented bar-graph meter

layout = "v1"        Modes selector above, Band selector below
      = "v2"        Band selector above, Modes selector below
                    (also moves the mobile fine-tuning button row)
```

All four live in **one build**, served at `/`. The `☞` selector at the top right switches between them inside the running page: it assigns `App.svelte`'s `smeter/layout` props, so nothing reloads and audio, the waterfall and the decoders keep running. The choice is stored per browser in `localStorage` under `phantom.variant`, and that stored value wins over the build default on the next visit.

The variant the sysop picks in `recompile.sh` is therefore the **starting** variant — what a first-time visitor sees before touching the menu.

URL	what it is
<code>/</code>	the desktop page, all four variants, switchable at runtime
<code>/mobile/</code>	separate page, built by <code>build-mobile.sh</code> — not a variant

Until 2026-08-10 each variant was its own build under `/analog/`, `/digital/`, `/v2-analog/` and `/v2-digital/`, and the selector navigated between them — which is why switching used to reload the page and drop the audio. Those builds are gone; each of the four paths now holds only a small `index.html` that redirects to `/`, written by `frontend/make-redirect-stubs.sh` on every build so old bookmarks keep working.

The variant is **not** written into any source file. `src/main.js` is a fixed entry point that hands two build-time defines to the component:

```
const app = new App({
  target: document.getElementById('app'),
  props: { smeter: __PHANTOM_SMETER__, layout: __PHANTOM_LAYOUT__ }
})
```

`vite.config.js` fills those in at build time, taking the first of these that is set:

1. **PHANTOM_SMETER** / **PHANTOM_LAYOUT** — environment variables. Nothing sets them now that the per-variant builds are gone; they still work for a one-off build with a different starting variant.

2. `frontend/variant.json` — the site default, written by `recompile.sh`. The root (`/`) build passes no variant, so it lands here.
3. `analog / v1` — the fallback.

Those two values are only the **seed**. `App.svelte` overwrites them at init from `localStorage` when the visitor has already chosen a variant, and the `☛selector` assigns them again on every switch.

Inside `App.svelte` the props are turned into two flags at lines 20-21:

```
$: isAnalog = smeter !== "digital";
$: isV2      = layout === "v2";
```

Use those two flags — never `smeter/layout` directly — so the whole file reads the same way.

2. Which File Does What

`frontend/src/main.js` — 12 lines

The entry point. Nothing but the App instantiation and the two `__PHANTOM_*__` defines it passes as props. It is **constant** — neither `recompile.sh` nor the build scripts rewrite it any more, and keeping it that way is what lets the variant builds run at the same time.

Do not put application code here. It is the entry file, not a place for logic.

`frontend/src/App.svelte` — ~17130 lines

The whole application: waterfall, tuning, decoders, bookmarks, panels, desktop layout AND the responsive phone layout. Almost every edit you will ever make goes here.

`frontend/src/lib/SMeterAnalog.svelte` — 824 lines

The moving-needle meter, complete: face drawing, needle, the light/dark face toggle and its `localStorage` key, and the needle smoothing.

- Input prop: `dbm` — the CALIBRATED power in dBm.
- Also takes: `mobile` — smaller canvas attributes for the phone layout.
- Tuning knob: `SMOOTH_TIME_MS` at line 55. Lower = faster needle, higher = smoother. Currently 16 (one frame at 60 fps, effectively the floor).

The three meter faces — and `smeter_theme.sh`

The analog meter has three background faces, all drawn by the same code:

```
dark      the original dark brushed metal
amber     a light, pale grey face
vintage   a warm aged-amber face
```

Do not confuse this with the `smeter = analog / digital` variant above. The variant chooses *which meter component* is shown; the face chooses *how the analog one is painted*. They are stored under different `localStorage` keys (`phantom.variant` vs `smeterTheme`) and are set in different places.

The visitor's side. The meter canvas is a button (`role="button"`, also Enter/Space from the keyboard, with a "click to switch style" hint under it). Each click cycles dark → amber → vintage and saves the choice in `localStorage.smeterTheme`. It is read back on every page load, so the visitor's pick survives

refreshes and browser restarts indefinitely. It is deliberately **not** a cookie: a cookie would be sent to the server on every waterfall and audio request for a value the server never reads.

The sysop's side. Two things in `SMeterAnalog.svelte` control what everyone else gets:

Line	What it does
<code>let smeterTheme = 'dark';</code>	The face a browser with no saved choice gets.
<code>const SMETER_PREF_VERSION = 2;</code>	A counter. When a browser's stored version is behind this, it forgets its saved face once and takes the default above. Bump it to move visitors who have already chosen.

Editing the default alone therefore only reaches brand-new visitors. Both edits together move everybody — that is what `smeter_theme.sh` in the repository root is for:

```
./smeter_theme.sh           # report the current default, then pick one from a menu
./smeter_theme.sh vintage   # set it straight away, then offer the rebuild
./smeter_theme.sh dark --build # run frontend/build-all.sh without asking
./smeter_theme.sh amber --no-reset # new visitors only – leave existing users alone
```

It reports the current default, sets the new one, bumps the reset counter (always upward — **never lower it**, a lower number simply stops the reset from firing), and then offers the same frontend build choice `recompile.sh` gives. It does **not** touch the backend and does **not** rewrite `variant.json`.

A change reaches a connected listener only when their page reloads — the default is compiled into the JS bundle and the reset runs at load time. There is no live push. An ordinary reload is enough, since vite hashes the bundle filenames.

The counter's one-time wipe overrides everyone, including visitors who deliberately clicked their own face. That is intended: it is the only way to bring the whole site back onto one look. Use `--no-reset` when you would rather leave them alone.

`frontend/src/lib/SMeterDigital.svelte` — 184 lines

The segmented bar meter, complete.

- Input props: `rawDb` — the RAW `audio.getPowerDb()`
- `smeterOffset` — `audio.smeter_offset`
- `mobile` — smaller canvas attributes

NOTE: this meter is deliberately driven from the RAW power plus the offset, NOT from the calibrated dBm the analog face uses. It has always done its own mapping. Feeding it the calibrated value would shift every reading on every digital site. Do not "unify" these two inputs.

`frontend/src/lib/BandSelector.svelte` — 69 lines

The band buttons grid. Rendered twice in `App.svelte` (once per layout).

`frontend/src/lib/ModesSelector.svelte` — 75 lines

The mode buttons grid. Rendered twice in `App.svelte`.

`frontend/src/lib/StatusIndicators.svelte` — 122 lines

The status lamps (mute, squelch, NR, NB, NS, AN, CTCSS). Takes a `wide` prop: the digital layout uses wider lamps than the analog one. Both Tailwind classes (`w-8` and `w-10`) are written out in full inside the

component — Tailwind's scanner only sees literal class names, so never build them by string concatenation.

All other files in `frontend/src/lib/` (Spectrogram, PassbandTuner, FrequencyInput, VersionSelector, ...) predate this work and are unchanged.

3. Making a Change That Affects Only Some Variants

Wrap it in the flag. In markup:

```
{#if isAnalog}
  ...only the analog sites see this...
{:else}
  ...only the digital sites see this...
{/if}

{#if isV2} ... {/if}      {#if !isV2} ... {/if}
```

In a class attribute:

```
class="p-4 {isAnalog ? 'text-xs' : 'text-sm'} rounded-md"
```

Write both alternatives out as whole literals, as above. Tailwind scans the source for complete class names; `text-{size}` or a concatenated name is purged from the CSS and the style silently vanishes.

In the `<script>` block: try NOT to branch at all. The script deliberately computes the inputs of BOTH meters on every tick —

```
smeterDbm      = powerDb;           // analog face
smeterRawDb    = audio.getPowerDb(); // digital face
smeterOffset   = audio.smeter_offset; // digital face
```

— so only the markup has to choose. That is why there are just 15 variant switches in a 17000-line file. Keep it that way; script-level branching is what made the four old files drift apart.

4. Where the Variant Switches Are

Fifteen places in `App.svelte`. Line numbers drift when you edit — the reliable way to find them all is:

```
cd frontend/src && grep -n 'isAnalog\|isV2' App.svelte
```

As of this writing:

Line	Flag	What it switches
20		isAnalog declared
21		isV2 declared
8062	isV2	desktop: Band selector first (v2 ordering)
8090	isV2	desktop: panel heading size
8686	isV2	desktop: Modes selector first (v1 ordering)
8717	isAnalog	meter panel min-width (needle is narrower)
8730	isAnalog	analog-only date/time line above the meter
8819	isAnalog	analog-only status lamps + divider
8834	isAnalog	digital-only time line, wide lamps, and the <SMeterDigital> / <SMeterAnalog> choice itself
8861	isAnalog	vertical spacing of the frequency block
8864	isAnalog	top margin of the fine-tuning block
12288	isV2	phone: fine-tuning row above (v2 ordering)
12347	isAnalog	phone: which meter to render
12433	isV2	phone: fine-tuning row below (v1 ordering)
13204	isAnalog	noise-gate button text size

Everything else in the file is shared by all four variants.

5. Other Knobs Worth Knowing

Location	Constant	Purpose
App.svelte line 5034	const visualGain = 1.1;	Analog meter calibration. Each 0.10 is roughly 5 dBm on the reading.
SMeterAnalog.svelte line 55	const SMOOTH_TIME_MS = 16;	Needle smoothing time constant, in milliseconds.
SMeterAnalog.svelte line 64	let smeterTheme = 'dark';	Default analog meter face for a browser with no saved choice. Set it with ./smeter_theme.sh, not by hand.
SMeterAnalog.svelte line 69	const SMETER_PREF_VERSION = 2;	One-time reset counter for the saved face. Bump to move existing visitors onto the default; only ever goes up.
SMeterDigital.svelte line 149	const DIGITAL_BAR_TRIM = 0;	Shifts the bar by whole segments.
SMeterDigital.svelte line 31	const numberOfDots = 35;	Segment count of the bar.

6. After You Edit — Rebuilding

Easiest, from the repository root:

```
./recompile.sh
```

Menu **[2]** rebuilds the frontend and asks which variant the site root (/) should serve. That choice is recorded in `frontend/variant.json` — no source file is copied or rewritten. (It used to regenerate `VersionSelector.svelte` from a heredoc as well; that was removed when the selector stopped navigating, because it would have reverted the runtime switching on every run.) Menu **[1]** is backend only, **[3]** is both.

Or directly, from `frontend/`:

```
./build-all.sh      builds the desktop page AND /mobile/ &lt;-- normally use this
./build-default.sh   just the desktop page
./build-mobile.sh    just /mobile/
```

`build-default.sh` lets vite empty `dist/`, which removes `dist/mobile/` until the next `build-all.sh` or `build-mobile.sh`. That is why `build-all.sh` is the normal choice.

`build-all.sh` has its own copy of the build logic (`build_version SMETER LAYOUT NAME OUTDIR BASE LOG`) rather than calling `build-default.sh`. If you change how the page is built, change it in BOTH places or the two paths will disagree.

A few details worth knowing:

- It sets `PHANTOM_KEEP_OUTDIR=1` so Vite does not empty the output directory, and cleans `dist/` itself instead — deleting the contents but keeping `users.json`, which the running `spectrumserver` rewrites on every listener connect and disconnect.
- It builds `/mobile` last, after the desktop output is complete.
- Its parallel-build machinery (`PHANTOM_BUILD_JOBS`, default 3) is left over from when there were five desktop builds. With one, nothing queues.

Executable bits: `recompile.sh` runs `chmod +x` on a script before running it, and `build-all.sh` deliberately invokes `bash build-mobile.sh` rather than `./build-mobile.sh`, so a missing bit no longer breaks a build. Uploading a script through the GitHub web interface still resets it to 644 though, so if you run one directly:

```
chmod +x frontend/build-*.sh
```

7. How to Check Your Edit Before Trusting It

A green vite build is **NOT** enough. The build cannot see an identifier that went missing when code moved between files — that is a runtime `ReferenceError`, and the page simply does not load. It has happened twice.

1) Undefined-identifier check (catches exactly that failure)

```
cd frontend
npx eslint --no-eslintrc \
  --parser svelte-eslint-parser \
  --rule '{"no-undef":"error"}' \
  --env browser,es2022 \
  src/App.svelte src/lib/SMeter*.svelte
```

One known-benign hit is expected:

```
'dBmCalOffset' is not defined
```

`typeof x === 'number'` on an undeclared name never throws, so that expression already evaluates to 0. Anything else is a real problem.

2) Build, then **LOAD THE PAGE AND SWITCH THROUGH ALL FOUR VARIANTS**

Not just the one you changed — an edit under `isAnalog/isV2` can break the other three while yours works. Open `/`, step through all four entries in the ⚙️ menu, then open `/mobile/`.

Open the browser console on each. A blank or half-drawn page with a `ReferenceError` in the console is the signature of the bug above.

3) **Never leave `frontend/dist/` in a partial state**

A bare `npm vite build` empties the output directory and takes `dist/mobile/` and the server's live `dist/users.json` with it — the mobile page 404s until the next successful build. Always build through the scripts: `build-all.sh` sets `PHANTOM_KEEP_OUTDIR=1` and does the cleaning itself, preserving `dist/users.json`. If in doubt, just run `./build-all.sh` again.

CHAPTER 12

User Guide

USER_GUIDE.md

PhantomSDR-Plus User Guide

Welcome to PhantomSDR-Plus! This guide will help you get the most out of your WebSDR listening experience.

Table of Contents

1. Introduction
 2. Getting Started
 3. Interface Overview
 4. Basic Operations
 5. Advanced Features
 6. Demodulation Modes
 7. Digital Decoders
 8. Keyboard Shortcuts
 9. Bookmarks
 10. Mobile Usage
 11. Tips and Best Practices
 12. Troubleshooting
 13. Frequently Asked Questions
-

Introduction

What is PhantomSDR-Plus?

PhantomSDR-Plus is a web-based Software Defined Radio (SDR) that allows you to listen to radio signals over the internet. No special software or hardware is required on your end—just a modern web browser!

What Can You Listen To?

Depending on the WebSDR's configuration, you can tune into:

- **Amateur Radio:** Ham radio operators worldwide
- **Broadcast Stations:** AM/FM radio, shortwave broadcasting
- **Aviation:** Air traffic control, aircraft communications
- **Marine:** Ship-to-shore, maritime weather
- **Weather Satellites:** NOAA, METEOR-M
- **Digital Modes:** FT8, JS8, RTTY, PSK31, and more
- **Utility Stations:** Time signals, military, government

System Requirements

- **Browser:** Chrome/Edge (recommended), Firefox, Safari
 - **Connection:** Broadband internet (1+ Mbps recommended)
 - **Audio:** Working speakers or headphones
 - **Optional:** Mouse with scroll wheel for easier tuning
-

Getting Started

1. Access the WebSDR

Open your browser and navigate to the WebSDR URL provided by the operator.

Example: `http://websdr.example.com:9002`

2. Initial Page Load

When the page loads, you'll see:

- A colorful waterfall display showing radio activity
- Control panel with frequency display and buttons
- S-meter showing signal strength
- User count indicator

3. Start Listening

1. **Click on a signal** in the waterfall display
 2. **Audio will automatically start** playing
 3. **Adjust volume** using your browser's volume control or the on-screen slider
 4. **Fine-tune frequency** by clicking precisely on the signal
-

Interface Overview

Main Components

WebSDR <callsign>, located in <QTH> ⌘ [Analog S-Meter ▼] email · Keyboard Shortcuts · Users · Stats · Other servers 1 2 3 Frequency Search: MW SW HamDash PSK Reporter WSPRnet · Callsign Search [Open Additional Info]			
Spectrum + Waterfall band-plan strip along the bottom			
Audio & Buffer volume · SQ · zoom AGC Auto Fast Med Slow Compressor Equalizer	Frequency 7,120.00 band · VFO	S-Meter dBm dBµV SNR · NF	Waterfall Controls Min · Max · zoom colour-map strip Zoom In Out Max Min
Filters NR NB NS AN CTCSS Modes selector ... RADEL RADEU USB LSB CW AM QUAM FM Band selector	Fine Tuning (kHz) VFO A · Modes · Bands · IF Filters Wheel Tuning Steps Decoders FT8 FT4 FT2 JS8 CW WSPR FAX SSTV NAVTEX RTTY ...	Waterfall Spectrum Auto Adj. Height [profile ▼] [Bookmarks]	

The three panels below the waterfall scroll as one page; on a narrow window they stack instead of sitting side by side. **RADEL** and **RADEU** sit on the *Modes selector* heading line itself, to its right.

1. Waterfall Display

The waterfall is a visual representation of radio signals:

- **Horizontal axis:** Frequency
- **Vertical axis:** Time (scrolls downward)
- **Colors:** Signal strength
- **Dark blue/black:** No signal (noise floor)
- **Green/yellow:** Weak to moderate signals
- **Orange/red:** Strong signals
- **White:** Very strong signals

2. Frequency Display

Shows the currently tuned frequency in various formats:

- **MHz:** 7.100.000 MHz (HF bands)
- **kHz:** 14200 kHz
- **Hz:** 145500000 Hz (VHF/UHF)

3. S-Meter (analog or digital)

Displays signal strength:

- **S1-S9:** Standard signal strength scale
- **+dB:** Signals above S9 (e.g., S9+20dB)
- **Analog or Digital:** Depending on configuration

4. Mode Buttons

Select demodulation mode:

- **AM**: Amplitude Modulation
- **FM**: Frequency Modulation
- **USB**: Upper Sideband
- **LSB**: Lower Sideband
- **CW**: Morse Code (Continuous Wave)
- **WBFM**: Wideband FM (broadcast)
- **QUAM**: C-QUAM AM stereo — the label turns green when a stereo pilot is detected

Next to the **Modes selector** heading sit the **RADEL** and **RADEU** buttons, which start RADE v1 digital voice in one press — see [Digital Decoders](#).

5. Control Panel

Additional controls:

- **AGC**: Automatic Gain Control
- **NR**: Noise Reduction (spectral)
- **NB**: Noise Blanker (impulse)
- **NS**: Background Noise Suppression
- **AN**: Auto Notch
- **CTCSS**: Sub-audible tone squelch (FM). Pressing it opens a small window beside the button with *Any tone* and the 49 standard subtones (67.0–254.1 Hz); pick one and the button shows it (for example 88.5, or *ANY*), and the audio then passes only while that tone is received — it takes about 0.7 s to open. Press the button again to switch CTCSS off; Escape or a click outside closes the window without changing anything. Bookmarks keep the chosen tone.
- **SQL**: Squelch
- **AI**: AI noise reduction (voice modes)
- **Zoom**: Waterfall zoom level
- **Wheel Tuning Steps**: step size used by the mouse wheel
- **Decoders**: one-touch buttons that start and stop each decoder

6. Band Plan Overlay

Colored bars showing frequency allocations:

- Different colors for different services
- Helps identify what's allowed on each frequency

Basic Operations

Tuning to a Frequency

Method 1: Click on Waterfall

1. Click directly on a signal in the waterfall
2. The receiver will tune to that frequency
3. Audio will begin playing

Method 2: Type Frequency

1. Click on the frequency display
2. Type the desired frequency
3. Press Enter

Examples:

- 7100 → 7.100 MHz
- 14200.0 → 14.200 MHz
- 145.500 → 145.500 MHz

Method 3: Use Mouse Wheel

1. Hover over the frequency display
2. Scroll wheel up to increase frequency
3. Scroll wheel down to decrease frequency

Method 4: Step Buttons

1. Use the ▲ and ▼ buttons next to frequency
2. Step size varies by mode:
3. **AM/FM**: 1 kHz steps
4. **SSB**: 100 Hz steps
5. **CW**: 10 Hz steps

Scanner

The scanner walks the receiver across a range of channels and stops on the first one carrying a signal. It sits on the **Fine Tuning (kHz)** row, hard right:



Control	What it does
◀▶	Scan down / up. While parked on a signal, an arrow resumes the scan.
■	Stop.
⊗	Lock this channel out of the scan — for a birdie or a permanent carrier.
30 dB · 4 ▼	The threshold, with the level being measured right now beside it. Opens the settings.

The text to the left shows what the scanner is doing: **Scanner** when idle, the frequency it has reached while scanning, ● 3s while counting down to resume, ● 30·22s while the channel is still busy, ● hold when it will stay put.

Settings

- **Range** — *Scan Band* sweeps the band the scan starts in; *Scan Visible* sweeps exactly what the waterfall shows, and follows it if you zoom or drag.
- **Scan** — *Every channel* tunes and listens to every channel in turn; *Skip empty* reads the spectrum and jumps straight to the signals.
- **Stop at** — how far above the band noise floor a channel must be to stop the scan, in dB. The floor is tracked continuously, so one setting works by day and by night. An empty channel does not read 0 dB — watch the live figure on the button for a moment and set the threshold above it.

- **Resume after** — how long a channel must stay quiet before the scan moves on. Pauses in speech do not restart it. *Hold* stays until you press a button.
- **Max stay** — moves on after this long even if the signal is still there, so a permanent carrier cannot hold the scan for ever.

The step follows the mode — 1 kHz in SSB, 0.1 kHz in CW, 5 kHz in AM, 9 or 10 kHz on medium wave and 9 kHz on long wave — and stops land on the channel grid. The scan stays inside its range and inside what the receiver can tune; at an edge it wraps round and carries on.

Settings and the locked-out channels are remembered in your browser.

Selecting Demodulation Mode

Choose the appropriate mode for the signal:

For Voice Communications:

- **AM**: Aviation, broadcast AM, some amateur
- **FM**: VHF/UHF repeaters, FM broadcast
- **USB**: HF ham radio (20m, 17m, 15m, 12m, 10m)
- **LSB**: HF ham radio (160m, 80m, 40m, 30m)

For Data/Digital:

- **USB**: Most digital modes (FT8, PSK31, RTTY)
- **LSB**: Some digital modes on lower HF bands

For Morse Code:

- **CW**: Telegraph/Morse code signals

Adjusting Volume

- **On-screen slider**: Drag the volume slider
- **Browser volume**: Use browser's media controls
- **System volume**: Adjust your computer's volume
- **Keyboard**: Use + and - keys (if supported)

Using the S-Meter

The S-meter shows signal strength:

- **S0-S3**: Very weak signal, difficult to copy
- **S4-S6**: Weak to fair signal
- **S7-S9**: Good to strong signal
- **S9+**: Extremely strong signal

Tip: For best audio, tune to signals showing S7 or higher.

Changing the meter face: on the analog (needle) meter, click the meter itself — or focus it and press Enter or Space — to cycle through three backgrounds: dark brushed metal, a light pale-grey face, and a warm vintage amber face. Your browser remembers the choice, so it is still there after a refresh or a restart. It is per browser and per address, so opening the receiver by hostname and by IP gives two separate settings, and a private window always starts from the site default.

Above 30 MHz the meter follows the VHF standard of IARU Region 1: S9 is -93 dBm there instead of -73 dBm on HF, still six dB per S-unit. The needle and the bar switch by themselves with the tuned frequency; the dBm and dB μ V figures are always the measured level. Some stations also set their VHF/

UHF receivers to rest at **0 on an empty channel**, like a VHF/UHF transceiver's meter: above 60 MHz the needle and the bar then come up only when a signal stands out of the noise. The dBm, dBμV, SNR and NF figures always show the real values.

Choosing a receiver

Some stations run more than one receiver — for example one for HF and one for the 2 m band. They show a line *Receivers*: in the page header with one button per receiver; the one you are on is shown in yellow. Pressing another button opens that receiver's page, with its own waterfall, bands, chat and listener list. On the /mobile page the same buttons sit in the top bar. A station with a single receiver shows no such line.

Advanced Features

Auto Gain Control (AGC)

AGC automatically adjusts audio levels:

- **Off**: No automatic gain adjustment
- **Slow**: Gradual level changes (best for SSB)
- **Medium**: Balanced response
- **Fast**: Quick adjustment (best for AM)

Recommendation: Start with "Fast" for AM, "Slow" for SSB.

The four noise controls

NR, NB, NS and AN are separate on/off buttons that each attack a different kind of noise. They are independent — switching one on does not switch on another — and they can be combined freely.

None of them reach the decoders: FT8, CW, WSPR, SSTV, FAX, NAVTEX, RTTY/PSK31/ Olivia and the QRSS grabber all read the audio *before* these filters, so you can set them purely by ear without affecting what decodes. See the [Decoder manual](#).

Noise Reduction (NR)

Spectral noise reduction. It estimates the noise level in each part of the audio spectrum and turns those parts down, leaving what stands above the noise alone.

Use when: You hear steady hiss or white noise behind the signal.

Steady tones — a CW note, a carrier — are recognised as signal and protected, so NR does not eat a CW signal the way a naive filter would. Typical effect is a 10 dB drop in the noise between speech, for a few tenths of a dB off the speech itself.

Noise Blanker (NB)

Removes impulse noise: clicks, pops, static crashes, ignition and power-line noise. It watches the audio envelope and silences only the samples that spike far above it — about a millisecond per crash, ramped in and out so the blanking does not itself click.

Use when: You hear clicks or crashes from power lines, motors, thunderstorms or vehicle ignition.

The 50 Hz and 60 Hz mains hum notches follow this button.

Background Noise Suppression (NS)

Measures the band's own noise floor over several seconds and applies a fixed cut to whatever sits at that floor, deeper at high audio frequencies than low. Where NR reacts moment to moment, NS is a slow, steady hand: it lowers the band hiss without changing how the signal itself sounds.

Use when: The band is quiet but hissy and you want the noise pushed down without the "underwater" quality of aggressive NR.

Active in USB, LSB and AM only — CW, FM and digital modes are left untouched. It re-measures the floor whenever you retune or change mode, and settles within a few seconds.

Auto Notch (AN)

Automatically finds and removes steady interfering tones — heterodynes, carriers, birdies — without you having to place a notch by hand. It adapts continuously, so several tones can be removed at once and a drifting one stays notched.

Use when: You hear a whistle or tone on top of the signal you want.

Bypassed in CW, where the wanted signal *is* a steady tone.

Delay note: NR and NS each add about 40 ms of audio delay while switched on (both together, about 80 ms). NB and AN add none. This affects listening only, never the decoders.

Auto Squelch (SQL)

Mutes audio when no signal is present:

- **Off:** Always playing (hear static)
- **Auto:** Automatically sets threshold
- **Manual:** Adjust threshold manually

Use when: Monitoring a frequency for activity.

AI Noise Reduction (AI)

The **AI** button, with a strength slider beside it, sits in the Audio & Buffer panel right under SQ. On the / mobile page it is in the Audio tab, under Squelch. It removes band noise from speech with RNNoise, a small neural network trained on voice.

Everything happens in your own browser: nothing is sent to an outside server, there is no account, and the receiver does no extra work.

- **On/off:** Click **AI**; the button turns blue. The first click downloads the module (about 1.3 MB), and the button pulses while it loads.
- **Strength:** The slider sets how strongly it works while someone is talking — lower values keep more of the original sound. The top of the scale is deliberately softer than a full cut, so that speech is not damaged even at 100%, which still cleans more than 50% does. Between words it works harder on its own, so the hiss drops further there. The default is 50%. You can set it while AI is off.
- **Status label:** The AI label under the frequency display lights cyan while AI is working, and shows half-dimmed when AI is on but the current mode is not a voice mode.

Use when: Listening to SSB or AM voice on a noisy band. The hiss between words typically drops by about 10 dB at the default 50%, and more at 100%, up to about 20 dB, while the speech keeps its level.

Voice modes only — USB, LSB, AM and SAM. In CW, FM, the data modes and C-QUAM the audio passes through untouched, because the network treats a CW tone or music as noise. Like the four noise controls, it never reaches the decoders. On weak stations (below about +6 dB SNR) it measures the signal-to-noise ratio itself and smooths its decisions more, so speech does not take on the bubbling,

"watery" sound this kind of filter can give; strong stations are processed exactly as before. If a very weak signal still sounds processed, lower the slider. It adds about 50 ms of audio delay.

Audio Buffer and Delay

Audio reaches the browser in small packets, 35 ms each on a typical receiver, and the browser keeps a little of it in hand so that a packet arriving late does not cause a gap. That reserve is the **buffer**, and it is most of the delay between the receiver and your loudspeaker.

- **Desktop:** in the Audio & Buffer panel, the button with the network icon moves the buffer one step up per click (it turns green above the lowest step), and the slider beside it picks a step from 1 to 6 directly. The lowest step suits a wired connection; phones start on step 2.
- **/mobile:** the **BUF** button and the **Buffer** slider in the Audio tab do the same.

The buffer adjusts itself. The page measures how irregularly your connection delivers the audio and keeps only as much in hand as the last minute showed was needed, plus the safety margin of the step you chose: 20 ms on step 1, 50 ms on step 2, more on the higher steps. After a moment of network trouble the extra delay drains away again within a minute or two, and a small difference between your sound card's clock and the receiver's no longer leads to a break every few minutes. The adjustments are a few milliseconds at a time, made where the sound repeats itself, and cannot be heard. The decoders and your recordings are never affected: they receive every sample.

If the audio breaks up, move the buffer one step higher. On a phone over home Wi-Fi the delay from the receiver to the loudspeaker is typically about 100–150 ms; the network between you and the receiver adds to it.

Chat

When the operator has enabled the chat, it has its own panel on the desktop page and a **Chat** tab on /mobile. Enter your name or callsign, type your message and send it.

To answer a particular message, click ↩ **Reply** next to its time. A line above the input box shows who and what you are answering; type and send as usual, or press × or Esc to cancel. The answer appears indented under the message it answers, with a short quote line. A reply to a reply joins the same thread. The server keeps the last 20 messages; when the original has already dropped out of that history, a reply shows only the name and time it answers.

Zoom Function

Magnifies the waterfall display:

- **1x:** Normal view (wide coverage)
- **2x:** 2× magnification
- **4x:** 4× magnification
- **8x:** 8× magnification

Use when: You need to see signals more clearly or tune precisely.

Transceiver control (CAT)

Your own transceiver and the receiver page can be kept on the same frequency: turn the rig's dial and the waterfall follows, or click a signal on the waterfall and the rig tunes to it. It only moves *your* listening session — nobody else on the receiver is affected.

With Desktop PhantomSDR+ (4.0 or later). The desktop application has a **Rig** menu. *Rig → Rig control...* opens a window where you choose your rig and how to reach it, and the menu itself switches the sync on and off. It syncs frequency, mode and filter width, in either direction or both, and can mute

the receiver while you transmit. Besides PhantomSDR-Plus it drives **KiwiSDR**, **PA3FWM WebSDR** and **UberSDR** receivers the same way. It reaches the rig in one of four ways:

- **Built-in** — no other software: Icom (CI-V), Yaesu (new CAT and the FT-817/857/897), Kenwood, Elecraft, FlexRadio SmartSDR CAT, QRP Labs and other Kenwood-compatible rigs.
- **Hamlib** — every rig Hamlib knows, more than 300 of them, picked from a searchable list. The Windows installers include Hamlib; on Linux install `libhamlib-utils`.
- **rigctld on the network** — a `rigctld` that is already running.
- **flrig** — for a rig that flrig already shares with fldigi, WSJT-X or a logger.

The application, its installers and its full manual are at [Desktop PhantomSDR+ \(Dropbox\)](#).

With a web browser. The [CATsync Tool for WebSDRs](#) (Windows) couples a rig to the receiver page open in your browser. It syncs frequency and mode.

Filter width and mute on transmit work on KiwiSDR, WebSDR and UberSDR receivers, and on a PhantomSDR-Plus receiver running 4.1.0 or later. On an older PhantomSDR-Plus frequency and mode still sync; the filter does not.

The full manual — every setting, the supported rigs, how the two sides are kept from fighting, and troubleshooting — is [Rig Control](#).

For developers. Every receiver page offers these functions on `window`, which is what both tools use:

Function	Does
<code>catsync_getFrequency()</code> / <code>catsync_setFrequency(hz)</code>	Tuned frequency, in Hz
<code>catsync_getMode()</code> / <code>catsync_setMode(mode)</code>	USB, LSB, CW, CW-L, AM, FM, WBFM ...
<code>catsync_getBandwidth()</code> / <code>catsync_setBandwidth(hz)</code>	Whole passband width, in Hz. Set it after the mode: changing mode resets the passband
<code>catsync_getMute()</code> / <code>catsync_setMute(on)</code>	Mute, through the page's own mute button
<code>catsync_ready</code>	<code>true</code> once the functions are installed

The last three rows are new in 4.1.0; check that a function exists before calling it.

Demodulation Modes

AM (Amplitude Modulation)

Used for:

- Aviation communications
- AM broadcast radio
- Some amateur radio
- Maritime communications

Characteristics:

- Wide bandwidth (typically 10 kHz)
- Susceptible to noise
- Easy to tune (just click on signal)

Best Practices:

- Use Fast AGC
- Enable Noise Blanker if you hear clicks
- Tune to peak of signal on waterfall

QUAM (C-QUAM AM Stereo)**Used for:**

- Medium-wave broadcasters transmitting AM stereo

Characteristics:

- 10 kHz wide, decoded as a stereo pair rather than mono AM
- The button label turns **green** on its own when the 25 Hz stereo pilot is present, so you can see which stations are actually in stereo before selecting it
- From AM, pressing the button again switches to synchronous AM on the carrier; the label turns **yellow** and reads SAM. A third press returns to plain AM
- C-QUAM audio is carried with Opus; every other mode uses FLAC

Best Practices:

- Look for the green label on strong medium-wave signals at night
- If stereo sounds unstable, SAM is the steadier choice on a weak carrier

FM (Frequency Modulation)**Used for:**

- VHF/UHF amateur radio repeaters
- Public service (police, fire, EMS)
- Commercial two-way radio
- Some satellite communications

Characteristics:

- Narrow bandwidth (typically 12.5 or 25 kHz)
- Excellent noise immunity
- "Capture effect" (strongest signal wins)

Best Practices:

- Tune precisely to center of signal
- Use squelch to mute when idle
- Disable noise reduction (not needed)

USB (Upper Sideband)**Used for:**

- HF amateur radio (above 10 MHz)
- Most HF digital modes
- Maritime communications (above 8 MHz)

Characteristics:

- Narrow bandwidth (typically 2.4 kHz)

- Efficient use of spectrum
- Requires precise tuning

Best Practices:

- Use Slow AGC
- Tune to lower edge of signal on waterfall
- Enable Noise Reduction if needed

LSB (Lower Sideband)

Used for:

- HF amateur radio (below 10 MHz)
- Some HF digital modes
- Maritime communications (below 8 MHz)

Characteristics:

- Same as USB but mirror image
- Convention: LSB on lower HF bands

Best Practices:

- Use Slow AGC
- Tune to upper edge of signal on waterfall
- Enable Noise Reduction if needed

CW (Continuous Wave / Morse Code)

Used for:

- Amateur radio telegraphy
- Navigational beacons
- Time signal stations

Characteristics:

- Very narrow bandwidth (100-500 Hz)
- High efficiency
- Requires learning Morse code to understand

Best Practices:

- Use narrow filter (400-500 Hz)
- Tune precisely to tone center
- Enable audio filter for better tone

CW-L (CW, Lower Sideband)

The same ± 250 Hz Morse filter as **CW**, but the tone is taken from below the carrier instead of above it. Use it when a signal is easier to copy on the low side, or when an interfering carrier sits just above the one you want.

There is no CW-L button on the desktop page. Its mode row is USB · LSB · CW · AM · QUAM · FM. CW-L is offered on the simplified `/mobile` page, and on the desktop it can still be set by a link or a bookmark that names it.

WBFM (Wideband FM)

Used for:

- FM broadcast radio (88-108 MHz)
- Some satellite downlinks

Characteristics:

- Very wide bandwidth (200 kHz)
- Excellent audio quality
- High fidelity

Best Practices:

- Tune to exact center frequency
- No squelch needed for broadcast
- Enjoy high-quality audio!

Digital Decoders

PhantomSDR-Plus includes built-in decoders for digital modes. For a complete guide, see [Decoders](#).

The quickest way in is the **Decoders** button row on the main panel, just below **Wheel Tuning Steps**: ten buttons — **FT8, FT4, FT2, JS8, CW, WSPR, FAX, SSTV, NAVTX, RTTY** — that each start their decoder, switch the master Decoder toggle ON and scroll the decoder's window into view in a single press. The button stays blue while the decoder runs; press it again to stop the decoder and close its window. The **Decoder Options** dropdown still works exactly as before and stays in sync with the buttons.

RADEL and **RADEU** (RADE v1 digital voice) have their own buttons next to the **Modes selector** heading and inside the **Modes** and **Bands** popup windows, and work the same press-on / press-off way.

The SSTV, HF FAX, NAVTEX, FSK/RTTY and CW decoders each run on their own background thread, so switching a decoder on or off never interrupts the audio, and the waterfall stays smooth while decoding. They are fed audio taken before AGC, noise reduction and mute — you can mute the receiver and decoding continues unaffected.

While a decoder runs it holds the mode and the passband. Normally the mode follows the band plan — tune into a segment marked LSB or AM and the receiver switches to it. A running decoder overrides that, keeping the mode and the narrow passband it needs even if you tune to another band. The band's own mode returns as soon as you switch the decoder off, and the mode buttons still work at any time if you want to take over. The CW decoder is the exception: it decodes in whatever mode you are listening in.

FT8, FT4 Decoder

What is FT8, FT4?

- Popular amateur radio digital mode
- Weak signal communication
- 15-second transmissions for FT8, 7.5-second transmissions for FT4

How to use:

1. Tune to FT8 or FT4 frequencies
2. Select USB mode

3. Enable the FT8 decoder — press the **FT8** button, or pick it from the menu
4. Watch decoded messages appear

Common FT8 Frequencies (USB):

- 160 meters: 1.840 MHz
- 80 meters: 3.573 MHz
- 60 meters: 5.357 MHz
- 40 meters: 7.074 MHz
- 30 meters: 10.136 MHz
- 20 meters: 14.074 MHz
- 17 meters: 18.100 MHz
- 15 meters: 21.074 MHz
- 12 meters: 24.915 MHz
- 10 meters: 28.074 MHz
- 6 meters: 50.313 MHz (50.323 MHz for DX)

Common FT4 Frequencies (USB):

- 80m: 3.575 MHz
- 40m: 7.0475 MHz
- 30m: 10.140 MHz
- 20m: 14.080 MHz
- 17m: 18.104 MHz
- 15m: 21.140 MHz
- 12m: 24.919 MHz
- 10m: 28.180 MHz
- 6m: 50.318 MHz

JS8 Decoder

What is JS8?

- FT8's weak-signal engine used for keyboard-to-keyboard conversation
- Free text instead of fixed exchanges, so long messages arrive over several cycles
- Five speeds; **Normal** (15 s) is the calling speed and carries almost all traffic

How to use:

1. Tune to a JS8 frequency
2. Select USB mode
3. Enable the JS8 decoder — press the **JS8** button, or pick it from the menu
4. Leave **Speed** on *Normal* and leave the sync offset on **Auto**

Reading the panel:

- Messages still arriving show at the top in green with a blinking cursor — a message can take a full minute to complete on Normal
- Completed messages list below as Mode | Hz | dB | Message, with **callsigns in green**
- A dimmed, italic row timed out before its last frame arrived: the text is real but may be cut short

Common JS8 Frequencies (USB):

- 160m: 1.842 MHz
- 80m: 3.578 MHz
- 40m: 7.078 MHz
- 30m: 10.130 MHz
- 20m: 14.078 MHz
- 17m: 18.104 MHz
- 15m: 21.078 MHz
- 12m: 24.922 MHz
- 10m: 28.078 MHz

JS8 is far quieter than FT8 — one transmission every few minutes is normal, and quiet spells are not a fault. See [Decoders](#) for the full guide.

CW Decoder

Just press the CW button and the decoded text will appear. Press the CW button once more to clear the window and restart the decoder.

WSPR

WSPR (Weak Signal Propagation Reporter, pronounced "whisper") is an ultra-weak-signal beacon mode that maps HF propagation paths worldwide. Each transmission takes approximately 110 seconds and fits inside a 200 Hz-wide slot. The decoder waits for a complete 2-minute UTC-aligned slot before decoding. Enable the decoder and select **WSPR** from the dropdown. Or simply press the **WSPR** button.

HF FAX / WEFAX

HF Radiofax (also known as WEFAX) is used by coast guard and meteorological services worldwide to broadcast weather maps, sea-state charts, and surface analyses over shortwave. The decoder reconstructs the image line by line as it is received.

NAVTEX

NAVTEX is the international maritime broadcast system for coastal safety information — navigational warnings, weather forecasts, and search-and-rescue notices.

FSK / RTTY, PSK31 and Olivia

A general-purpose decoder for narrow-band text modes, with five variants in one window: Maritime FSK (SITOR), Weather RTTY, Amateur RTTY, **PSK31** (phase-shift keying, 31.25 baud) and **Olivia** (multi-tone FSK with forward error correction). The panel adapts to the variant — the shift, baud and framing controls disappear for PSK31 and Olivia, and Olivia adds a Mode selector and a squelch slider.

Amateur RTTY and Weather RTTY have a **Squelch (SNR)** slider that stops noise from printing as random letters, and every variant except Olivia shows a real SNR on the same 3 kHz scale. Two things to know: PSK31 corrects its own tuning error over about ± 25 Hz, so you only need to get close; Olivia needs its **Mode** (tones / bandwidth) set to exactly match the transmission, sends no preamble, and therefore takes a few seconds to synchronise before any text appears. The panel opens on Olivia **8 / 250**.

QRSS Grabber

QRSS is CW sent so slowly that a single dot lasts seconds, and it is watched rather than heard — the grabber paints the trace on its own display. It is not in the Decoder dropdown; it has its own **QRSS** section and can run at the same time as a decoder.

Press **Show**, then pick a window from the **Band** list and press **Tune**: the receiver goes to that frequency in **CW** with a passband sized for the mode, and the trace lands on the centre line of the display. 30 m (10,140.00 kHz) is the busiest window. Set **Speed** to match the beacon — **QRSS 10** if you do not know — and be patient: a callsign can take ten minutes to cross the screen. Full details in [Decoders](#).

SSTV

Slow-Scan Television sends still pictures over a normal SSB channel, one line at a time. Enable the decoder, select **SSTV**, and tune to an SSTV frequency — 14.230 MHz is the main international calling frequency. Leave **Mode** on **Auto**: the decoder reads the transmission's VIS header, and falls back to identifying the mode from the sync timing if you tuned in after the header. Martin, Scottie and Robot modes are supported, and the picture builds line by line as it arrives. Or simply press the **SSTV** button.

Automatic spot reporting and server graphs

FT8, FT4 and WSPR decodes can also be uploaded automatically by the server itself — FT8/FT4 to PSK Reporter, WSPR to WSPRnet — by an autorun daemon the sysop starts from the admin panel. It is independent of the decoders in your browser: it keeps running whether or not anyone is listening, and nothing you decode in the browser is reported.

The sysop follows it with two counters, which are easy to mistake for each other: the per-decoder tiles count the spots uploaded since the daemon last started, while the number beside each band/mode checkbox is that slot's all-time total and survives restarts. The same panel has a **Graphs** page plotting CPU frequency, CPU load, CPU temperature and users online over the last 15 minutes to 24 hours. Both are described in the [Admin Panel guide](#).

Keyboard Shortcuts

Keyboard shortcuts for faster operation.

Frequency Control

The frequency display is a row of digits you drive directly. **Click a digit first** — that selects it, and everything below acts on the selection.

- **Arrow Left / Right**: move the selection to the next digit (eight digits, 100 MHz down to 10 Hz)
- **Arrow Up / Down**: step the *selected* digit up or down by its own place value — so the step is 1 MHz on the MHz digit, 10 Hz on the last one
- **0 - 9**: type that digit straight into the selected position

Mouse wheel over the frequency display: steps by the band's tuning step (1 kHz unless the band plan sets another). Hold **Shift** for 1 kHz, **Alt** for 10 kHz.

Mouse wheel over the waterfall: zooms. Hold **Ctrl** (or **Cmd**) or **Shift** to tune instead of zooming, and **Shift + Ctrl** together to snap to the nearest whole kHz.

There are no Page Up / Page Down shortcuts.

Bookmarks

Save your favorite frequencies for quick access. You can also export the list of bookmarks and save it locally, then import the list into any other PhantomSDR.

When bookmarks and markers overlap:

Blue bookmarks appear on top

Yellow markers appear underneath

Bookmark clicks take priority

Adding a Bookmark

1. Tune to desired frequency
2. Click "Bookmarks" button
3. Click "Add Bookmark"
4. Enter description
5. Click "Save"

	Bookmark name	Label (optional)	[Add]

How to Use:

Add bookmark:

- Name: "Local News Station"
- Label: "NEWS"
- Click Add

View on waterfall:

- Zoom in until markers appear
- See marine blue box with "NEWS" in bold yellow

Click bookmark:

- Tunes to frequency
- Sets demodulation mode
- Works exactly like clicking a marker

Managing Bookmarks

- **Edit:** Click pencil icon next to bookmark
- **Delete:** Click trash icon next to bookmark
- **Export:** Download bookmarks as JSON file
- **Import:** Upload bookmarks from JSON file

Sharing Bookmarks

1. Click "Export Bookmarks"
2. Share the JSON file with others
3. Recipients click "Import Bookmarks"
4. Select your file

Mobile Usage

PhantomSDR-Plus works great on mobile devices!

Two mobile views

There are two ways to use the receiver on a phone:

- **`http://your_server:PORT/mobile`** — the simplified page. No waterfall, so it uses roughly half the data. Frequency entry, tuning steps, modes, S-meter, bands, bookmarks, users and chat.
- **The extended view** — the main interface's own phone layout, with the waterfall and the full set of controls.

Switch between them with the buttons at the bottom of `/mobile` (**Mobile extended view**, **Full desktop view**) and the **Simplified mobile** button in the extended layout.

Your frequency follows you. Switching view keeps you on the same signal — the frequency and mode travel in the link, so you no longer land back on the receiver's default frequency. The address bar tracks your tuning as well, which means reloading the page, bookmarking it, or sending the link to someone else all return to that exact frequency.

The mode follows the band plan on both pages. Tune into a segment marked AM, LSB, USB or CW — by typing a frequency, stepping to it or pressing a band button — and the receiver switches to that mode, on the simplified page as well as the full interface. A mode you pick by hand stays put while you move around inside the same segment, and outside the defined bands your mode is left alone. While RADE (RADEL/RADEU) is running it keeps the receiver, so tuning does not interrupt it.

When you switch between the two views your current mode travels with you, but if the view you are switching to has no equivalent for it, the mode for that frequency is taken from the band plan — a broadcast frequency arrives in AM, 40 m in LSB, a CW segment in CW. SAM on the simplified page becomes AM with the synchronous detector in the extended view, and back again. RADEL and RADEU exist only on the simplified page, so switching away from one of those keeps your frequency and lets the band plan pick the mode.

A frequency outside the receiver's coverage is pulled back to the nearest edge, so an old link can never strand you off-band.

Mobile-Specific Features

- **Touch-friendly controls:** Large buttons and sliders
- **Swipe to tune:** Swipe left/right on waterfall
- **Pinch to zoom:** Pinch waterfall to zoom in/out
- **Landscape mode:** Rotate for better view

Mobile Tips

1. **Use WiFi:** Streaming audio uses data
2. **Landscape orientation:** Better waterfall view
3. **Headphones:** Better audio quality
4. **Bookmark favorites:** Easier to revisit stations
5. **Close other apps:** Ensure smooth performance

Mobile Browser Recommendations

- **Android:** Chrome or Samsung Internet
- **iOS:** Mozilla

- **Both:** Ensure browser is up to date
-

Tips and Best Practices

For Best Reception

1. **Choose strong signals:** Look for orange/red on waterfall
2. **Tune precisely:** Click directly on signal center
3. **Select correct mode:** Match the signal type
4. **Adjust AGC:** Fast for AM, Slow for SSB
5. **Use NR/NB:** Help with noisy conditions

Finding Activity

1. **Watch the waterfall:** Colors show signal strength
2. **Listen on popular frequencies:**
3. 40m: 7.100-7.300 MHz (LSB)
4. 20m: 14.200-14.350 MHz (USB)
5. 2m: 145.200-145.600 MHz (FM)
6. **Check band plan overlay:** Shows frequency allocations
7. **Use bookmarks:** Quick access to active frequencies

Understanding Band Conditions

Daytime HF (High Frequency):

- Higher bands work better (20m, 15m, 10m)
- Long-distance (DX) communication possible
- Broadcast stations audible

Nighttime HF:

- Lower bands work better (80m, 40m)
- Different propagation patterns
- Different stations audible

VHF/UHF:

- Mostly line-of-sight
- Local communications
- More consistent conditions

Etiquette

1. **Don't tie up the receiver:** Others want to listen too
 2. **Use chat respectfully:** Be courteous to other users
 3. **Report problems:** Help the operator maintain the station
 4. **Don't ask for tech support:** This is a listening platform. Send a message to the Sysop for help.
-

Troubleshooting

No Audio

Possible causes:

1. Browser muted → Check browser volume controls
2. System muted → Check computer volume
3. Weak signal → Tune to stronger signal (S7+)
4. Wrong mode → Try different demodulation modes

Solutions:

1. Click a strong signal (orange/red on waterfall)
2. Check browser isn't muted (look for mute icon in tab)
3. Try a different frequency
4. Reload the page (F5)

Distorted Audio

Possible causes:

1. Overdriven signal → Signal too strong
2. Wrong mode → AM signal on SSB mode, etc.
3. Interference → Adjacent signals bleeding over

Solutions:

1. Reduce volume
2. Try different demodulation mode
3. Use narrower filter bandwidth
4. Tune away from interfering signals

Audio Breaks Up or Lags Behind

Possible causes:

1. Irregular connection → packets arrive late and the buffer runs dry
2. Busy phone or computer → the browser cannot keep up

Solutions:

1. Move the buffer one step higher (see *Audio Buffer and Delay*)
2. Prefer Wi-Fi or a cable to mobile data
3. Close other tabs and apps

For a closer look, open the page with `?diag=1` added to its address (for example `http://your_server:PORT/mobile?diag=1`). The `/mobile` page then shows the delay, how irregular the connection is and how the buffer is doing under its controls; the desktop page writes the same line to the browser console (F12) every ten seconds. Useful when reporting a problem to the station operator.

Waterfall Not Updating

Possible causes:

1. Network issue → Slow or interrupted connection
2. Browser performance → Too many tabs open

3. Server overload → Too many users

Solutions:

1. Check internet connection
2. Close unnecessary browser tabs
3. Reload page (F5)
4. Try again later when fewer users online

Can't Tune to Frequency

Possible causes:

1. Frequency out of range → SDR doesn't cover that frequency
2. Typing wrong format → Use correct format (e.g., "14200" not "14.200.000")

Solutions:

1. Check SDR's frequency coverage (shown on page)
2. Use frequency format examples provided
3. Click on waterfall instead

Stuttering/Choppy Audio

Possible causes:

1. Slow internet connection
2. High server load
3. Browser performance issues

Solutions:

1. Close other applications using bandwidth
2. Try again during off-peak hours
3. Close unnecessary browser tabs
4. Use wired connection instead of WiFi

Frequently Asked Questions

General Questions

Q: Do I need special equipment to use WebSDR? A: No! Just a computer or mobile device with internet access.

Q: Is WebSDR free to use? A: Yes, most WebSDRs are free. They're operated by volunteers.

Q: Can I transmit with WebSDR? A: No, WebSDR is receive-only. You cannot transmit.

Q: What frequencies can I listen to? A: Depends on the WebSDR's configuration. Check the station info.

Q: Can I record audio? A: Some browsers allow recording. Check your browser's features.

Technical Questions

Q: What sample rate does the SDR use? A: Varies by station. Check the station information page.

Q: What's the latency? A: Usually a few tenths of a second between the radio signal and your speakers: the receiver's processing, the network, and the browser's playback buffer, which adjusts itself to your connection (about 100–150 ms on a phone over home Wi-Fi). See *Audio Buffer and Delay*.

Q: Can I use multiple instances? A: Usually yes, but it may strain the server. Be considerate.

Q: Does it work offline? A: No, WebSDR requires internet connection.

Q: What browsers are supported? A: Chrome, Firefox, Edge, Safari (all recent versions)

Usage Questions

Q: How many people can listen at once? A: Depends on server capacity. Often 50-200+ users.

Q: Can I see what others are listening to? A: If enabled, yes. Look for "other users" indicators.

Q: Can I chat with other listeners? A: If enabled by the operator. Look for the chat box; ↩ **Reply** answers a particular message (see *Chat* above).

Q: Why do some frequencies show nothing? A: No signals on that frequency at the moment. Try others!

Q: What are the colored bands on the waterfall? A: Band plan overlay showing frequency allocations.

Resources

Learning More About Radio

- **Band Plans:** Search "amateur radio band plan" + your region
- **Propagation:** Learn about HF radio wave propagation
- **Digital Modes:** Research FT8, PSK31, RTTY
- **Ham Radio:** Consider getting an amateur radio license!

Finding More WebSDRs

- **WebSDR Directory:** <http://sdr-list.xyz>
- **WebSDR.org:** <http://websdr.org>
- **KiwiSDR:** <http://kiwisdr.com/public/>

Getting Help

1. **Station operator:** Check contact info on page
2. **User chat:** Ask other listeners (if available)
3. **Online forums:** Search for WebSDR communities
4. **Documentation:** Refer to this guide!

Appendix: Common Frequencies

HF Amateur Radio Bands

Band	Frequency Range	Mode	Activity
160m	1.800-2.000 MHz	LSB	Night/Local
80m	3.500-4.000 MHz	LSB	Night/Regional
40m	7.000-7.300 MHz	LSB	Day/Night/DX
30m	10.100-10.150 MHz	USB	Data/CW only
20m	14.000-14.350 MHz	USB	Daytime/DX
17m	18.068-18.168 MHz	USB	Daytime/DX
15m	21.000-21.450 MHz	USB	Daytime/DX
12m	24.890-24.990 MHz	USB	Daytime/DX
10m	28.000-29.700 MHz	USB	Sporadic/DX

VHF/UHF Amateur Bands

Band	Frequency Range	Mode	Activity
6m	50.000-54.000 MHz	USB/FM	Sporadic
2m	144.000-148.000 MHz	FM	Very Active
70cm	420.000-450.000 MHz	FM	Active

Broadcast Bands

Service	Frequency Range	Mode
AM Radio	530-1710 kHz	AM
Shortwave	2.3-26.1 MHz	AM
FM Radio	88-108 MHz	WBFM

Aviation

Service	Frequency Range	Mode
Air Traffic Control	118-137 MHz	AM
ACARS (data)	130-136 MHz	Data

Maritime

Service	Frequency Range	Mode
Marine VHF	156-162 MHz	FM
Marine HF	2-22 MHz	USB

Glossary

AGC: Automatic Gain Control - adjusts audio levels automatically

AM: Amplitude Modulation - voice mode used for aviation and broadcast

Bandwidth: The frequency range of a signal

CW: Continuous Wave - Morse code signals

DX: Long distance communication

FFT: Fast Fourier Transform - converts time to frequency domain

FM: Frequency Modulation - voice mode for VHF/UHF

HF: High Frequency (3-30 MHz) - long distance bands

kHz: Kilohertz (1,000 Hz)

LSB: Lower Sideband - voice mode for lower HF bands

MHz: Megahertz (1,000,000 Hz)

NB: Noise Blanker - removes impulse noise

NR: Noise Reduction - reduces background noise

PSK: Phase Shift Keying - digital mode

RTTY: Radio Teletype - digital text mode

S-meter: Signal strength meter

SDR: Software Defined Radio

SQL: Squelch - mutes audio when no signal present

SSB: Single Sideband (USB or LSB)

USB: Upper Sideband - voice mode for higher HF bands

VHF: Very High Frequency (30-300 MHz) - line-of-sight

UHF: Ultra High Frequency (300-3000 MHz) - line-of-sight

Waterfall: Visual display of radio spectrum over time

Enjoy exploring the radio spectrum with PhantomSDR-Plus!

73 (Best regards) de SV1BTL & SV2AMK

For installation instructions, see [INSTALLATION.md](#). For technical details, see [PROJECT_STRUCTURE.md](#).

CHAPTER 13

KiwiSDR Client Emulation

Aether_config.md

KiwiSDR Client Emulation

Connecting AetherSDR, kiwiclient and other KiwiSDR software to PhantomSDR-Plus

Since v4.1.0 PhantomSDR-Plus can also answer the **KiwiSDR protocol**, so software written for a KiwiSDR — **AetherSDR**, `kiwiclient` and the rest — connects to your receiver directly. It is a bridge inside the same server process, on the same host and the same port you already publish: no second daemon, no second port, no proxy.

It is off until you enable it. Nothing about the bridge runs — not a socket, not a check — while `[kiwi_emulation]` is missing or `false`, which is the state a fresh tree is in.

Table of Contents

1. What a Kiwi client gets
2. Installing the bridge
3. Switching it on
4. Configuration reference
5. Connecting a client
6. Audio level
7. The S-meter
8. Waterfall and spectrum
9. If something looks wrong
10. How far this has been verified

1. What a Kiwi client gets

Audio	<code>ws://<host>:<port>/kiwi/<id>/SND</code> — demodulated audio in the Kiwi's own SND framing
Waterfall	<code>ws://<host>:<port>/kiwi/<id>/W/F</code> — the spectrum, in Kiwi W/F frames
Retuning	real, not cosmetic: <code>SET mod=...</code> from the client changes frequency, sideband and passband on the PhantomSDR side, exactly as a browser listener would
S-meter	driven from the demodulator's own per-block power, on the same scale as the web S-meter

Retuning is supported for a **real** input (`signal = "real"` in your `.toml`). For an IQ input the bin-to-frequency relation differs and is not implemented, so an IQ receiver will serve audio and waterfall but will not follow a Kiwi client's tuning.

2. Installing the bridge

The bridge is a set of patches to the backend sources plus one new header, `src/kiwi_bridge.h`. There are two ways to get them in.

With the installer. `install.sh` (and `install_arch.sh`, `install_fedora.sh`, `install_opensuse.sh`) have a step of their own for it — step 17, *Kiwi client emulation*. It is offered, not forced: answer `n` and nothing is patched. For an unattended run, `PHANTOM_KIWI=y|n` decides it, and the default is yes.

On a tree that is already installed, run the installer script that ships in the repository root:

```
cd ~/PhantomSDR-Plus
./kiwi_install.sh
./recompile.sh          # choose [1] Backend only
```

`kiwi_install.sh` backs every file it touches up into `backup_kiwi_bridge_<timestamp>/` first, and applies each patch by exact text match — if an anchor is not where it expects it (a locally modified tree, a different version), it stops there and then, names the file, and leaves that file untouched. It is idempotent: run it on a tree that already has the bridge and it reports each patch as already applied and changes nothing.

If you have just applied an update archive that already contains the bridge, you do not need to run it. The patched sources are in the drop; `kiwi_install.sh` would only tell you every patch is already in place.

The files it patches are `src/client.h`, `src/signal.cpp`, `src/waterfall.cpp`, `src/spectrumserver.h`, `src/spectrumserver.cpp`, `src/websocket.cpp` and `src/http.cpp`, and it copies `kiwi_bridge.h` into `src/`.

It installs the bridge; it does not upgrade one. If a patch ever *fails* on a tree where the bridge already works, the source has moved ahead of the script and it is the script that needs updating — not your tree.

3. Switching it on

Add this to the `.toml` your receiver actually runs with — the same file you pass to `spectrumserver`, the one named in your `start-<radio>.sh`. Not `config.example.*`:

```
[kiwi_emulation]
enabled = true
```

Then restart the receiver:

```
./stop-websdr.sh
./start-<your-radio>.sh
```

`kiwi_install.sh` adds the block for you, with `enabled = true`, to whichever of `config.toml`, `config-rx888mk2.toml`, `config-airspyhf.toml`, `config-rtl.toml`, `config-rspla.toml`, `config-fobos.toml`, `config-fobos-hf.toml` and `config-hackrf.toml` exist in the repository root. It cannot know about a config you keep elsewhere or under another name, and **update archives never ship a `.toml`**, so after an update the block is yours to add by hand. The shipped `config.example.*.toml` files carry it documented and set to `false`.

4. Configuration reference

Everything the bridge reads lives in one `[kiwi_emulation]` section. Only `enabled` is required; the four trims all default to the calibrated value, so a block with nothing but `enabled = true` is already correct.

Key	Default	What it does
<code>enabled</code>	<code>false</code>	Answer the KiwiSDR protocol at all. Off = completely inert.
<code>audio_gain</code>	<code>0</code>	dB. Output gain for Kiwi clients only. See Audio level .
<code>smeter_offset</code>	<code>input.analog_smeter_offset</code>	dB. Overrides the S-meter offset for Kiwi clients only. See The S-meter .
<code>wf_cal</code>	<code>0</code>	dB. Display trim for the waterfall and spectrum. See Waterfall and spectrum .
<code>wf_fps_max</code>	<code>23</code>	Waterfall frames per second. See Waterfall and spectrum .

A fully populated block, with the values this project's own receiver runs:

```
[kiwi_emulation]
enabled      = true
audio_gain   = 60.0    # dB, Kiwi clients only. 0 = untouched.
wf_cal       = -10.0   # dB, display taste. 0 = agrees with the S-meter.
wf_fps_max   = 28      # frames/s. 23 is the protocol maximum.
# smeter_offset = 5.0  # dB. Defaults to input.analog_smeter_offset.
```

Every key here is read at startup. Changing any of them is a **restart**, not a rebuild — `./stop-websdr.sh` then your start script. You never need to recompile to retune these.

Integer and decimal forms are both accepted: `28` and `28.0` are read identically. Only a key that is genuinely absent falls back to its default.

5. Connecting a client

Point the client at the same address and port your listeners use — no path, no prefix. AetherSDR asks for a host and a port; `kiwiclient` takes `-s` and `-p`:

```
# kiwiclient, recording 30 seconds of 7100 kHz LSB
python3 kiwirecorder.py -s your.receiver.example -p 8073 -f 7100 -m lsb --tlimit=30
```

The client tunes, the waterfall fills, and the S-meter reads. If nothing happens at all, the bridge is almost certainly still disabled — see [section 3](#).

6. Audio level

The first thing most people notice is that a Kiwi client sounds thinner than the PhantomSDR web page. That is not the bridge losing anything: one buffer feeds every encoder, and measuring the same frequency and passband through the browser's own audio path and through `/SND` gives identical samples. The web page is loud because `audio.js` rebuilds the sound in the browser — bass boost, bandpass, a presence lift, a compressor with makeup gain and the volume slider. A Kiwi client has none of that and cannot be given it.

`audio_gain` closes the gap, for Kiwi clients only:

```
[kiwi_emulation]
enabled      = true
audio_gain = 55.0      # dB, Kiwi clients only. 0 = untouched.
```

Start around 55-60 dB. 55 dB is not arithmetic: it is where a Kiwi client and the PhantomSDR web page were measured level with each other, listening to a strong local signal on 729 kHz. In practice a little more is often preferred — this receiver settled on 60 — so pick it by ear.

You cannot break the audio with it. A **look-ahead peak limiter** sits between the gain and the 16-bit clamp: every sample is delayed by 4 ms while the gain is worked out from audio that has not been sent yet, so a loud passage arrives to a gain that has already come down for it, and peaks fold instead of being flattened. Driven to 60 and even 70 dB — far past the headroom the raw gain has — not one sample in a quarter of a million was clipped. Below the limiter's threshold the gain is exactly 1.0, so a receiver that is not driving the audio hard gets the samples untouched.

What a too-high value costs is loudness, not damage. Above about 60 dB the extra gain is simply limited away: measured here, the level rose about 1 dB from 55 to 60 and hardly moved from 60 to 70. Past that point you are buying compression rather than volume.

None of this touches the web page, and none of it moves the S-meter — that is derived from the demodulator's power, not from the audio samples, so turning the volume up cannot make the meter lie.

7. The S-meter

The Kiwi S-meter is not calibrated separately: it reproduces what your web page displays, stage for stage. The demodulator's per-block power is the starting point for both, and the page then applies `input.analog_smeter_offset`, expands the result about -130 dBm, and adds its own display offset. The bridge does the same, so a Kiwi client and the page show the same signal at the same strength — verified on a strong local station, where both read -37 dBm.

If, and only if, you want Kiwi clients to read differently from your page, give the bridge an offset of its own:

```
[kiwi_emulation]
enabled      = true
smeter_offset = 5.0      # replaces input.analog_smeter_offset for Kiwi clients only
```

Compare the two meters on the same filter, or not at all. The reading comes from the power in the passband, so a wider filter collects more noise and reads higher — on a quiet band the same frequency measured 5.6 dB stronger at 9 kHz wide than at 2.4 kHz. A Kiwi client with a 6 kHz filter against a web page on 2.4 kHz will disagree by several dB no matter how either is calibrated. Match the mode and the bandwidth first.

This is the page's scale, not a physical one. The web meter expands its range for readability, so 10 dB of real signal change is displayed as about 11. Matching it means Kiwi clients agree with your receiver and differ from a genuine KiwiSDR by an amount that grows with signal strength. That is the right trade when your own page is the reference everyone compares against; if you would rather the bridge stayed physically honest, this is the section to change.

8. Waterfall and spectrum

The dB scale — `wf_cal`

Each waterfall bin is converted to a real dBm before it is sent, on the same scale the S-meter is calibrated to. A carrier therefore reads the **same level at every zoom**, which is what a genuine KiwiSDR gives you, and the spectrum agrees with the web page's own S-meter over the same passband.

```
[kiwi_emulation]
enabled = true
wf_cal = 0.0          # dB, Kiwi clients only
```

`wf_cal` is not a calibration. 0 is the value at which the spectrum agrees with your S-meter, and that is what to leave it at if you want the numbers to mean something. It exists because how *strong* a spectrum should look is a matter of taste and of what your client does with the range — this project's receiver runs `-10` simply because that is what looks right in AetherSDR. One value moves the spectrum and the waterfall together: they are the same bytes on the wire, and the Kiwi protocol cannot scale them apart. If you want them to differ, that has to come from the client's own display controls.

To check the scale, compare the waterfall summed over the passband against the S-meter on a **quiet** frequency. On a carrier an SSB passband excludes the carrier itself and the two are not comparable.

The frame rate — `wf_fps_max`

A Kiwi client asks for the fastest waterfall it can have (`SET wf_speed=4`) and paces its scroll — and its spectrum averaging — on the rate the server says it will deliver. If the two disagree the display looks sluggish even though nothing is actually late.

The receiver produces $2 \times \text{sps} / \text{fft_size}$ spectra per second. Kiwi clients are served from every one of them and thinned to the rate they asked for; the web page keeps its own slower cadence and is unaffected.

```
[kiwi_emulation]
enabled = true
wf_fps_max = 23      # frames per second
```

23 is the KiwiSDR protocol's own declared maximum and the safe default. **The lower of `wf_fps_max` and the receiver's own rate always wins**, so raising it does something only if your FFT is fast enough to have frames to spare. Worked example, for a receiver at 60 Msps with a 4194304-point FFT — $2 \times 60000000 / 4194304 = 28.6$ frames a second:

<code>wf_fps_max</code>	delivered
absent (default 23)	23.0 fps
28	28.0 fps
40	28.6 fps — the receiver's own ceiling wins

`SET wf_speed` from the client is honoured as well: 0 off, 1 = 1 fps, 2 a quarter of the maximum, 3 a half, 4 the maximum. A client on a thin link can therefore ask for less.

Whether raising it helps is the client's decision, not the server's. A client that draws each frame as it arrives gives you a smoother scroll; a client that paces itself at the 23 it expects will simply queue the extra frames, and the latency will grow. Try it, and if the waterfall feels laggier rather than smoother, put it back to 23.

What sets the rest of the delay

If you are chasing latency, most of it is not in the bridge. The analysis window is `fft_size / sps` wide — 70 ms on a 4194304-point FFT at 60 Msps — and a sample also waits up to half of that for the hop to fill. Halving `fft_size` halves both and doubles the frame rate, but it also halves your frequency resolution, which is what the narrowband decoders (WSPR, FT8, CW) live on. On most receivers that is a poor trade for a few tens of milliseconds. It is a configuration decision in `[input]`, not something the bridge can do anything about.

9. If something looks wrong

Every command a Kiwi client sends is written to `/tmp/kiwi_retune.log` — the tuning requests, the mode changes, the authentication, and anything the bridge did not recognise. It costs nothing measurable and is meant to be left in place. When a client behaves oddly, that file shows what it actually asked for, which is usually the whole answer.

Symptom	Look at
Client connects and immediately drops	<code>[kiwi_emulation] enabled = true</code> missing from the config the server was launched with
No audio, no waterfall, no log lines	the backend was not rebuilt after <code>kiwi_install.sh</code> — run <code>./recompile.sh</code> , option <code>[1]</code>
<code>kiwi_install.sh</code> stops on a patch	the message names the file and the anchor; the tree has diverged from what the patch expects, and that file was left untouched
Audio present but thin and quiet	expected — set <code>audio_gain</code> , see section 6
Waterfall scrolls sluggishly	<code>wf_fps_max</code> , see section 8
Tuning in the client does nothing	an IQ input (<code>signal = "iq"</code>); retuning is implemented for real inputs only
A changed setting had no effect	these keys are read at startup — restart the receiver

AGC is client-side. SET `agc=` from a Kiwi client is accepted and ignored — the audio arrives with PhantomSDR's own AGC applied, and the client's AGC controls act on what it receives.

10. How far this has been verified

The wire format has been checked against a live receiver and read line by line against the source of both clients it targets — AetherSDR's `KiwiSdrProtocol.cpp` and `kiwiclient's kiwi/client.py`. Frame layout and size, sequence numbers, byte order, the S-meter scale, retuning and the waterfall all agree. The bridge has also been **confirmed working with the AetherSDR desktop application** itself, by the maintainer, against this receiver.

Measured on that receiver, with a test client on the live sockets: retuning lands in 35–50 ms, audio runs at real time with no drift, and the waterfall delivers the configured rate with no stalls or dropped frames.

If something does look wrong on your own installation, `/tmp/kiwi_retune.log` is the place to look first, and a report either way is welcome.

See also: [Installation Guide](#) · [Project Structure](#) · [User Guide](#)

CHAPTER 14

Receive Diversity

RECEIVE_DIVERSITY.md

PhantomSDR-Plus — Receive Diversity

Combine your receiver with a **second receiver somewhere else** and listen to whichever of the two currently has the better signal. When one site drops into a fade, the other usually has not, and you keep copy through it.

The second receiver can be another PhantomSDR-Plus, a KiwiSDR, an UberSDR or a WebSDR. For the first three, everything runs in the browser: no server changes, no configuration files, nothing to install, and no special access to the other receiver. A WebSDR is the exception and needs one small piece of software on your own server — see [Using a WebSDR](#) below.

What it does, and what it does not

Diversity here means **selection**: at any moment you are listening to one site or the other, with a short crossfade when it switches. It is not a phased array and it does not add the two signals together.

That is a deliberate choice, and the reason is physics rather than software. Two receivers hundreds of kilometres apart hear the same transmission over different ionospheric paths. The two audio waveforms arrive with unrelated phase and slightly different Doppler, so adding them sounds hollow and comb-filtered — the classic "phasey" artefact. True coherent combining needs two receivers on a **common clock**, sample-aligned; two independent receivers connected over the internet can never provide that.

So what you gain is **continuity, not raw signal strength**:

- On a stable path where one receiver is simply better, you will hear that receiver and gain nothing. This is normal.
- On a fluttery path with deep QSB, fades at two distant sites are largely uncorrelated. Site A drops out for three seconds, site B does not, and the audio carries on. Expect "*I stopped losing words*", not "*S3 became S7*".
- The second win is often **local noise and QRM**. Two sites have different neighbours, different power lines and different broadcast splatter. On a noisy band this can matter more than the fading.

There is a cost: you listen **one to three seconds behind real time**, because the earlier stream has to be delayed to match the later one. That is irrelevant for listening and decoding, and unhelpful if you are trying to work a station.

NOTE

The digital decoders (FT8, JS8, WSPR, RADE and the rest) deliberately keep using your **local** receiver, not the combined audio. Those modes integrate coherently across a whole transmission slot, and switching sites in the middle of a slot is a phase discontinuity that can cost you the very decode diversity was meant to save. Diversity serves the speaker; the decoders keep the continuous local stream.

Quick start

1. Open the **Receive Diversity** panel — it sits just below the decoder windows, collapsed, showing `Receive Diversity – off`. Click it to expand.
2. Choose what the second receiver is: **PhantomSDR+**, **KiwiSDR**, **UberSDR** or **WebSDR**.
3. Type its address — or pick one you have saved — and press **Start**.
4. Tune your own receiver normally. The second receiver **follows automatically** — frequency, mode and passband — every time you retune.

That is the whole setup. There is nothing to configure on the other receiver, and no separate tuning control: it always tracks you.

Addresses

Second receiver	What to type	Notes
PhantomSDR+	<code>host:8900</code>	The receiver's normal port
KiwiSDR	<code>host:8073</code>	The KiwiSDR's own port
UberSDR	<code>host</code>	Its normal web port , not 8073
WebSDR	<code>host:8901</code>	The site's own port. Needs the relay — see below

A bare hostname, an `http://` URL or a full `ws://` URL are all accepted. If the other receiver is served over HTTPS, type the hostname and it will use a secure connection.

Saved receivers

Every address you connect to is remembered, in a list of its own for each receiver type — a KiwiSDR address is never a useful suggestion when the selector says WebSDR. The lists live in your own browser: they are not sent anywhere, and every listener has their own.

The ☰ button beside the address box opens the list.

- **Names.** Any entry can be given a name — `Twente` reads better than `websdr.ewi.utwente.nl:8901`. Named entries appear in green above their address, and the name shows beside the address in the drop-down as you type. An entry is identified by its address, so naming one never puts the same station in the list twice. While diversity is running, the receiver it is running against shows its name **bold and slowly blinking**, so the list also says which one you are actually hearing.
- **add** enters a receiver by hand, without connecting to it first.  corrects a name and address in place,  removes one entry, and **clear all** empties the list for the type on screen. Enter saves, Escape cancels.
- ******** opens that receiver's own web page in a new tab. The address stored is the one the software dials, so it is turned back into a browsable one first — `ws://` becomes `http://`, and the `/audio`, `/ws` or KiwiSDR `/kiwi/.../SND` path each kind of receiver needs is trimmed away, since the site's own page is at the root. It is a real link, so a middle click or a long press behaves as usual.
- **The order is yours.** Move a row with ▲▼, or drag it and drop it where it belongs. That is the order in which the drop-down offers the addresses.
- **export and import.** The lists are held in the browser's own storage, which is thrown away when you clear the site's data — this is what protects against that. Export writes all four lists, names included, to a JSON file you can keep or carry to another browser or machine. Import reads it back and asks whether to **merge** it with what is saved or **replace** everything.

-  **QR**. Draws every list as a QR code on the screen, for carrying it to a phone — see "On a phone" below. A list too large to scan falls back to the text beside it, which can be copied and pasted anywhere.

There is no limit on how many receivers you keep.

On a phone

The `/mobile` page has the same feature in a **Div** tab, beside Audio, Bands, Marks, Users and Chat: the receiver type, the address, Start and Stop, the saved receivers as a tap-to-pick list, the live figures and the SNR trim. Each row also carries the `****` link to that receiver's own page. The second receiver follows the phone's tuning by itself, exactly as on the desktop page. What it leaves out is the editing — renaming, reordering, deleting — which stays on the desktop page, where there is room for it.

The saved list belongs to one browser, so a phone starts with an empty one however many receivers are saved on the desktop. That is what the QR code is for: press  then  **QR** on the desktop, scan the code with the phone's camera, and paste the text into **Import** on the Div tab. It asks whether to merge or replace, and Export on the phone sends a list back the other way.

Using a WebSDR

A WebSDR — the software written by Pieter-Tjerk de Boer, PA3FWM, that runs at Twente and several hundred other sites — can be used as the second receiver, but not directly from the browser.

The reason is a deliberate check on their side. A WebSDR refuses its audio connection unless the `Origin` header names its own site, and `Origin` is a *forbidden header name*: the browser sets it from the page you are on and no script is permitted to change it. There is no client-side way around this, and there should not be.

So the connection is made by a small program on your own server instead, `websdr_relay.py`. Your browser talks to the relay, and the relay talks to the WebSDR.

Installing the relay

The four distribution installers offer it as an optional step. To add it to an existing installation:

```
cd ~/PhantomSDR-Plus
./setup_websdr_relay.sh
```

It asks for a port, takes your callsign and site address from `frontend/site_information.json`, and offers to install a systemd service so the relay starts at boot. Settings live in `websdr_relay.json` and can be changed at any time.

One thing the installer cannot do for you: forward the relay's port on your router. Visitors' browsers connect to the relay directly — it is not proxied through the receiver — so without that forward, WebSDR diversity works only from inside your own network. The other three receiver types are unaffected.

If the admin panel is installed, its dashboard shows a **WebSDR Diversity Relay** card with the relay's state, the port, how many sessions are running and to which sites.

Being a good guest

The relay connects to other people's receivers from your server rather than from each listener's own address, so it is built to behave:

- **A cap of ten concurrent sessions to any one WebSDR**, and sixty overall. This is what stops a busy day at your site from looking like an attack on someone else's.
- **A User-Agent naming your station and its operator**, so an operator who would rather not be used this way knows exactly who to write to. Please fill it in honestly.
- **Only tuning commands are forwarded**. The connection cannot be used to send anything else.
- **Private, loopback and carrier-NAT addresses are refused**, so a visitor cannot aim the relay at something inside your machine or your LAN.

IMPORTANT

The Origin check exists because the WebSDR authors did not want their receivers driven from other people's pages. Using the relay is a considered decision, not an oversight on their part. Keep the cap where it is, keep the User-Agent honest, and stop if an operator asks you to.

What is different about a WebSDR

- **Many sites cover a few narrow slices of spectrum**, not a continuous range — a 256 kHz window on 40 m, for example. Diversity engages only inside them, and the panel reads the site's own band list to know where those are.
- **The audio sample rate varies** with the filter width and sits further from nominal than the other receiver types, so alignment can take a little longer.
- **CW is not adjusted** for WebSDR's convention, which places the passband entirely below the carrier. SSB and AM are correct.

Reading the panel

Once running, the panel shows a small table:

Row	Meaning
link	The connection to the second receiver. <code>ready</code> is what you want. A failure shows the server's own error text and the WebSocket close code
aligned	The measured delay between the two streams once they are locked together, or <code>searching</code>
corr	How strongly the two streams correlate. Only meaningful once aligned
remote audio	Decoded audio arriving from the second receiver. <code>none</code> in amber means nothing is being decoded
SNR local / remote	The two signal-to-noise estimates being compared
switches	How many times it has changed site

Beside the title, one word tells you where you stand:

off (grey)	not running
Please wait... (amber)	connecting, or connected and still aligning
Ready (green)	linked, locked and following the better site
the receiver's own error (red)	it has failed and will not recover on its own

The distinction between amber and red is the one that saves time: amber means keep waiting, red means stop waiting and read the message.

The coloured dot adds which site you are actually hearing — green for your own receiver, cyan for the remote.

Alignment takes about 15 seconds

This is normal and worth expecting. The two streams are aligned by correlating their **audio envelopes**, and that needs a window of audio to work with. The search runs in two stages: a narrow ± 1.5 second search can start once about 9 seconds of audio has been gathered, and a second, independent measurement five seconds later has to **agree** with the first before the alignment is trusted. When the two sites are further apart in time than that — a second or two of extra buffering somewhere along the path — the full ± 4 second search takes over at about 14 seconds and confirmation follows at 19.

The confirmation step matters. When two sites fade in antiphase — never both carrying the signal at the same moment — a single correlation will happily lock onto a confident, completely wrong delay, which sounds like an echo. Requiring two independent measurements to agree rejects that.

A second thing has to be right before a lock can hold: the two receivers' sample rates. Two receivers are two clocks and two decimation chains, so their audio streams do **not** arrive at quite the same rate even when both declare 12 kHz — 2% apart is normal, which is 240 samples a second of creep, far more than the alignment tolerance. The remote stream is therefore continuously resampled to match your own receiver's rate. That ratio is measured by **counting samples**, not by correlating, so it needs no lock of its own and is settled within the first few seconds — before the first alignment search even runs. A weak or intermittent signal can still take two or three tries to align, so 30 seconds is not a cause for concern; a minute with nothing to show is.

While **searching**, both SNR figures read 0.0. That is by design: the two signal-to-noise estimates are measured on **content-aligned** samples, so nothing is measured until there is an alignment. 0.0 / 0.0 means "not locked yet", not "no signal".

Remote SNR trim

This slider biases the choice between the two sites. It is added to the remote receiver's measured SNR before the comparison — positive favours the remote, negative favours your own. It changes **nothing else**: not the audio level, not the combining, only which site wins.

It exists because the two SNR figures are not always comparable. Both are measured the same way — a percentile spread of the audio — but a receiver whose codec floors its own noise reads lower than it deserves. A KiwiSDR applies strong gain and a limiter, which compresses the spread; a lossy codec sets a noise floor the signal never gets below.

How to set it:

1. Tune both receivers to a signal that is reliably present, and wait for **aligned**.
2. Watch the two SNR figures for half a minute. Note the typical gap.

3. Use the extremes as a listening test: **+15** forces the remote site live, **-15** forces your own. Listen to each for a few seconds. This is the only way to hear one site in isolation.
4. If the remote reads, say, 5 dB lower but sounds just as good, set **+5**. You are making the two readings agree when the two sites sound equal.
5. Watch the **switches** counter over ten minutes of listening. Constant flipping means the trim is too close to a tie; never switching even when your own receiver audibly fades means it is too far the other way.

Starting points: **0** for another PhantomSDR-Plus, which uses an identical audio path and is comparable by construction; a small **positive** value for a KiwiSDR or UberSDR.

TIP

If you need more than about ± 8 dB, stop trimming. At that point the honest explanation is usually that one receiver really is worse for that path, and biasing past a real difference just means listening to the weaker site.

The setting applies immediately and is remembered in your browser.

Choosing a second receiver

Distance matters. Fades decorrelate with distance — a few hundred kilometres on HF is a good target. Two receivers in the same city fade together and buy you nothing. Too far apart and the second receiver may not hear your signal at all.

Both must actually hear the signal. This is the requirement no software can work around. If the second receiver cannot hear what you are listening to, the correlation stays low and it will refuse to lock — deliberately, because a wrong alignment sounds worse than no diversity at all.

Coverage. A second PhantomSDR-Plus or a KiwiSDR announces the frequency range it covers, and the panel warns you if you tune outside it. A WebSDR announces its bands, which are often narrow slices rather than a continuous range. UberSDR does not announce coverage, so there the refusal to lock is your only signal.

IMPORTANT

Public receivers are run by volunteers and have a limited number of listener slots. A diversity session occupies one of them for as long as it runs, just as a human listener would. Please be considerate about leaving one connected indefinitely, and ask the operator if you intend to use their receiver heavily.

Troubleshooting

Link never leaves connecting, or cycles connect/close The address or port is probably wrong. Check the table above — in particular, UberSDR uses its normal web port, not 8073. The panel shows the WebSocket close code, and the browser console (F12) shows the underlying error.

Link shows an error with text after it That text comes from the other receiver. It is usually specific: a refused session, a full server, or a receiver that does not accept outside clients.

KiwiSDR refuses the connection Some KiwiSDRs require a password, and on an UberSDR the KiwiSDR-compatible endpoint is switched off by default. For an UberSDR, use the **UberSDR** source type instead — that path is always available.

WebSDR: the WebSDR relay is not reachable The relay is not running, or your browser cannot reach its port. It must be reachable at the **same hostname the receiver page is served from**, because the browser connects to it directly rather than through the receiver. Open `http://<that hostname>:<relay port>/status` in the same browser to see which. If it answers on your LAN but not from outside, the port is not forwarded on the router.

WebSDR: an error mentioning a refused connection That site has locked things down further than the standard `Origin` check, or its operator has blocked this station.

Link is ready but remote audio shows none Audio is not being decoded. Reload the page; if it persists, the browser console will show which decoder failed.

Audio arrives but it never aligns The two receivers are not hearing the same thing. Try a strong broadcast station, or a band where both sites have good propagation. A quiet band with only noise at both ends will never correlate — and should not.

It switches back and forth constantly The two sites are too close in SNR. Adjust the trim slightly so one is consistently preferred.

It never uses the second receiver Check `remote audio` is climbing and that the SNR figures look sensible. If the remote reads much lower than it sounds, that is what the trim is for.

Limitations

- **Not coherent.** No phasing, no nulling, no direction finding. Cancelling a noise source needs two antennas on one clock at one site; this is a different technique for a different problem.
- **One to three seconds of delay**, unavoidably.
- **Two receivers only.**
- **Mono.** If your receiver is in a stereo mode such as C-QUAM, the audio passes through untouched and diversity does not engage.
- **Alignment needs signal.** Under about 10 dB SNR at one of the two sites, expect it to sit at `searching`.
- **A WebSDR needs the relay** on your own server, and its port forwarded. The other three receiver types need neither.

CHAPTER 15

Rig Control

RIG_CONTROL.md

PhantomSDR-Plus — Rig Control (CAT)

Keep **your own transceiver** and a **PhantomSDR-Plus receiver** on the same frequency, mode and filter. Turn the rig's dial and the waterfall follows; click a signal on the waterfall and the rig tunes to it. Key the rig, and the receiver can fall silent so it does not play your own signal back to you.

It works with **PhantomSDR-Plus, KiwiSDR, PA3FWM WebSDR and UberSDR** receivers, yours or someone else's, and only ever moves *your* listening session — nobody else on the receiver hears or sees a thing. There is nothing for the receiver's operator to install or configure.

What you need

Two ways to connect a rig, and one requirement on the receiver:

Piece	What it is	Syncs
Desktop PhantomSDR+ 4.0 or later	The desktop application, with a Rig menu. Linux (PC and Raspberry Pi) and Windows.	Frequency, mode, filter width, mute on transmit — in either direction or both
CATsync Tool for WebSDRs	A separate Windows program that couples a rig to the receiver page in your browser.	Frequency and mode
TCI-CAT on the receiver page	A button on the PhantomSDR-Plus page itself, in any browser. Talks TCI to ExpertSDR, AetherSDR or Thetis, or to any Hamlib rig through a small bridge. Only on PhantomSDR-Plus receivers running 4.1.0 or later.	Frequency, mode and filter width both ways; mute on transmit
The receiver	PhantomSDR-Plus, KiwiSDR, PA3FWM WebSDR or UberSDR. A PhantomSDR-Plus receiver needs 4.1.0 or later for filter width and mute.	An older PhantomSDR-Plus still syncs frequency and mode

Most of this manual describes Desktop PhantomSDR+. The TCI-CAT button has its own section, [TCI-CAT on the receiver page](#). The CATsync Tool has its own documentation on its website.

Supported receivers

The app recognises the kind of receiver in a station window and drives it through that page's own controls. The Rig control window shows which kind it found next to the station's name.

Receiver	Frequency and mode	Filter width	Mute on transmit
PhantomSDR-Plus	Yes	With 4.1.0 or later	With 4.1.0 or later
KiwiSDR (including Web-888)	Yes	Yes	Yes
PA3FWM WebSDR	Yes, switching band on a multi-band site	Yes	Yes
UberSDR	Yes	Yes	Yes

A web receiver has fewer modes than most rigs, so some rig modes share one receiver mode: KiwiSDR and WebSDR have a single CW for both CW and CW-R. The receiver modes named in this manual are PhantomSDR-Plus's; the other receivers use their nearest equivalent. A WebSDR keeps a CW filter under 1 kHz and other filters at 1 kHz or wider, because that is how the page itself tells CW apart, and a frequency outside every band a WebSDR site covers is left alone. UberSDR sets its filter in slider steps, so a width can land up to about 50 Hz from the rig's. Any other kind of page — OpenWebRX, for example — shows *not a receiver this app can drive*, and nothing is synced.

What it does, and what it does not

- It syncs **one rig with one receiver window** at a time.
- It reads both sides several times a second and, when they disagree, sets one to match the other. It does **not** transmit, key the rig, or send audio anywhere.
- Moving the receiver changes only your own session. Other listeners on the same receiver are not affected, and the receiver's operator does not have to allow it.
- A serial port can be opened by **one program at a time**. If WSJT-X, a logger or a manufacturer's utility already has the rig's port, use the **flrig** or **rigctld on network** choice to share the rig instead of fighting over the port.

Quick start

1. Open a station in Desktop PhantomSDR+ as usual.
2. **Rig → Rig control...**
3. Under **Connection**, choose **Built-in** if your rig is in the list, otherwise **Hamlib (all rigs)**.
4. Pick your rig, the serial port, and the speed your rig's own CAT or CI-V menu is set to.
5. Under **Sync**, leave **Both directions** selected.
6. Press **Connect**. The two readouts at the top — transceiver and receiver — should show the same frequency within a second.

Every setting is saved as you change it. Next time, **Rig → Connect** is enough, or tick **Connect when the app starts**.

Choosing how to reach the rig

Choice	Use it when	Needs
Built-in	Your rig is in the list below.	Nothing else
Hamlib (all rigs)	Your rig is anything else — Hamlib knows more than 300 rigs. The app starts Hamlib's <code>rigctld</code> for you, on a private local port, and stops it when you disconnect.	Windows: nothing, Hamlib is included. Linux: <code>sudo apt install libhamlib-utils</code>
rigctld on network	A <code>rigctld</code> is already running, on this computer or another one on your network.	The host and port (4532 by default)
flrig	flrig already controls the rig for fldigi, WSJT-X or a logger.	flrig running, with its XML-RPC port (12345 by default)

Rigs with a built-in driver

The speed and CI-V address shown are the factory defaults the app fills in. **They are only a starting point — set them to what your rig's menu says.**

Family	Rigs	Default speed	Notes
Icom CI-V	IC-7300, IC-7610, IC-705, IC-9700, IC-905, IC-7760, IC-7851, IC-7100, IC-7410, IC-9100, IC-7600, IC-7200, IC-7700, IC-7000, IC-7800, IC-756PROIII, IC-756PROII, IC-R8600, and any other CI-V rig	19200	CI-V address filled in per model (IC-7300 94, IC-705 A4, IC-9700 A2, IC-7610 98 ...)
	IC-746PRO, IC-718, IC-R75	9600	
Xiegu (CI-V)	G90, X6100	19200	Address 70; check the rig's menu
Yaesu new CAT	FTDX101D/MP, FTDX10, FT-710, FT-991/A, FT-891, FTDX5000, FTDX3000, FTDX1200, FT-950, FT-2000, FT-450/450D	38400	
Yaesu classic CAT	FT-817/818, FT-857/857D, FT-897/897D	38400	2 stop bits; tunes in 10 Hz steps
Kenwood	TS-990S, TS-890S, TS-590S/SG	115200	
	TS-480, TS-2000, TS-870S	57600	
Elecraft	K4, K3/K3S, KX3, KX2	38400	Filter width syncs
Kenwood-compatible	FlexRadio SmartSDR CAT (virtual port), QRP Labs QMX/QMX+/QDX, (tr)uSDX, Lab599 Discovery TX-500, other Kenwood-compatible rigs	9600–38400	

A rig that ought to be compatible but will not talk to a built-in driver usually works with **Hamlib**, which copes with far more variations.

Serial settings

Setting	What to put there
Serial port	The rig's port. USB adapters and rigs with a USB port are listed first. Other / network address... takes a port the list does not show — COM7, /dev/ttyUSB1 — or tcp://host:port for a serial port served over the network by ser2net or similar.
Speed (baud)	Exactly what the rig's CAT / CI-V baud-rate menu says. A mismatch looks like a rig that never answers.
Stop bits	1 for almost everything; 2 for the FT-817/857/897 family.
CI-V address	Icom only, in hexadecimal (94, not 148). Must match the rig's CI-V address menu.
DTR / RTS	Leave off unless your interface needs them. Many CAT cables key the transmitter, or reset the rig, on one of those lines.
Hardware flow control	Leave off unless the rig's manual asks for RTS/CTS.

For **Hamlib** the same settings are passed on to `rigctl`. Stop bits there has a *Rig default* choice, and **Extra rigctl options** takes anything else `rigctl` accepts, for example `--set-`

`conf=post_write_delay=10`. **rigctl** program lets you point at a particular `rigctl` if you have more than one installed.

Sync

Direction

Choice	What happens
Rig → receiver	The receiver window follows the rig. A change made on the waterfall is put back to the rig's frequency.
Receiver → rig	The rig follows the receiver window. Turning the rig's dial is undone.
Both directions	Whichever side you touched last wins. At the moment you connect, before either has been touched, the rig wins.

The direction can also be changed from the **Rig** menu while connected.

How the two sides are kept from fighting

Every value the app writes shows up a moment later as a change on the other side. If that were taken at face value, the rig and the receiver would chase each other for ever. The app avoids it in three ways:

- The receiver page applies a change at once, so it is read back straight after writing, and that reading becomes the new starting point.
- A rig applies a change a little later, so each value sent to it is remembered. When the rig reports that value, it is recognised as the app's own write rather than a hand on the dial.
- A value the rig refuses — wide FM on an HF rig, for example — is sent **twice** and then left alone until the source side changes, instead of being repeated several times a second.

Retuning the receiver page can make it choose its band's default mode (LSB below 10 MHz, for instance). When the rig is leading, the app puts the rig's own mode back straight away, so a rig in USB on 40 m keeps the receiver in USB.

Which receiver window

Receiver window chooses which station follows the rig:

- **The station window last in front** (default) — with two stations open, click into one and the rig follows that one.
- **A particular station** — pin the rig to it, whether or not it is in front. If that station is not open, nothing is synced until it is.

Update rate

Update every sets how often both sides are read: 150 ms, 300 ms (default), 500 ms or 1 s. Faster feels more immediate on the dial; slower is kinder to an old rig at 4800 or 9600 baud, where each read takes real time on the wire.

Modes

Rig mode	Receiver listens in
USB, LSB	USB, LSB
CW	CW
CW-R (reverse)	CW-L
AM, synchronous AM, DSB	AM
FM, narrow FM	FM
Wide FM	WBFM
RTTY / FSK	LSB
RTTY-R / FSK-R	USB
Data modes (USB-D, DATA-U, PKTUSB, DIG)	USB
Data LSB, data FM	LSB, FM

Receiver mode	Rig is set to
USB, LSB	USB, LSB
CW	CW
CW-L	CW-R
AM, QUAM	AM
FM	FM
WBFM	WFM — most HF rigs refuse it, and are left alone after two tries
RADE (upper / lower)	USB / LSB

A rig in a **data mode** stays in it: the receiver's USB is treated as agreeing with the rig's USB-D, so the receiver never knocks the rig out of data mode.

Filter width

Tick **Sync filter width** to keep the passbands matched. Two differences of less than 60 Hz are treated as equal, since no two filters step alike.

Rig connection	Filter width
Hamlib	Yes, where Hamlib supports it for that rig
flrig	Yes
Built-in Icom CI-V	Yes — 50 Hz steps up to 500 Hz, then 100 Hz steps up to 3.6 kHz; AM in 200 Hz steps up to 10 kHz; not in FM
Built-in Elecraft	Yes, in 10 Hz steps
Built-in Kenwood, Yaesu, FT-817 family	No — these rigs select filters from per-model tables. Use Hamlib if you need the filter

KiwiSDR, WebSDR and UberSDR receivers always have the control. A PhantomSDR-Plus receiver must run **4.1.0** or later; on an older one frequency and mode still sync, and the Rig control window says why the filter does not.

Mute while transmitting

Tick **Mute receiver while transmitting**. While the rig is keyed the receiver window is muted, and when it is unkeyed the sound comes back. The receiver's own mute button shows it, and you can still unmute by hand.

If you had already muted the receiver yourself, it is left muted afterwards too.

It needs a rig connection that reports transmit state — all the built-in drivers, flrig, and Hamlib for most rigs — and, on a PhantomSDR-Plus receiver, 4.1.0 or later.

Frequency offset

Frequency offset is added to the rig's frequency to get the receiver's:

receiver frequency = rig frequency + offset

Setup	Offset
2 m transverter on a 10 m rig (144.100 MHz shows as 28.100 MHz)	116000000
70 cm transverter on a 2 m rig (432 → 144)	288000000
No transverter	0

The Rig menu

Item	Does
Rig control...	Opens the Rig control window
Connect / Disconnect <i>rig name</i>	Starts or stops the sync; for Hamlib this also starts or stops rigctld
Rig to receiver / Receiver to rig / Both directions	Sync direction
Sync filter width	On / off
Mute receiver while transmitting	On / off
Status line	<i>Not connected, Connecting..., Connected: rig name</i> , or the last error

The live frequency readout is in the Rig control window rather than in the menu, which would close itself every time it changed.

Linux

Serial port permission. Serial ports belong to the `dialout` group. A user outside it gets *Could not open ttyUSB0*. Add yourself once, then log out and back in:

```
sudo usermod -aG dialout $USER
```

Hamlib. Install it from your distribution:

```
sudo apt install libhamlib-utils
```

The `.deb` package of Desktop PhantomSDR+ recommends it, so `sudo apt install ./phantomsdr-plus-desktop_4.0.0_amd64.deb` brings it along; `dpkg -i` does not install recommended packages. The built-in drivers and flrig need no Hamlib.

Windows

Hamlib's own `rigctl.exe` is included in both the 64-bit and the 32-bit installer. COM ports appear in the port list by name (COM3). If a rig's USB driver is needed, install the manufacturer's driver first — the port does not exist until it is.

TCI-CAT on the receiver page

A PhantomSDR-Plus receiver page can also follow a transceiver **by itself**, with no desktop application: the browser talks straight to a **TCI** server on your own computer. TCI is the WebSocket control protocol of ExpertSDR2/ExpertSDR3 (SunSDR), AetherSDR (FlexRadio) and Thetis (Apache Labs ANAN, Hermes). For a rig without TCI, a small bridge included with PhantomSDR-Plus makes any Hamlib rig look like a TCI server.

It needs a PhantomSDR-Plus receiver running **4.1.0 or later**. KiwiSDR, WebSDR and UberSDR pages do not have it.

The TCI-CAT button

The button sits in the row with **VFO**, **Modes**, **Bands** and **IF Filters** — **TCI-CAT** in the wide layout, **CAT** in the compact one. Its dot is **green** while a TCI server is connected and **grey** otherwise. It opens a window with: Until 4.2.0 the button was labelled **QRG Sync**.

Control	What it does
Status	<i>Active</i> — TCI (port 50001) when connected; <i>Inactive</i> with <i>Wait</i> while it searches
CAT Sync	Frequency, mode and filter sync on or off. The mute on transmit works either way
Host	The computer running the TCI server: <code>localhost</code> for this computer, or its address on your network, e.g. <code>192.168.1.42</code> . Press Enter or click away to reconnect

Nothing needs choosing beyond that: ports **50001** (ExpertSDR3, AetherSDR) and **40001** (ExpertSDR2, Thetis) are tried at the same time and retried every few seconds, so the page connects as soon as the software starts. **CAT Sync** and **Host** are remembered by the browser.

Browsers that ask before a web page may reach your local network — recent Chrome and Edge do — ask once, the first time the page connects. Allow it, or the dot stays grey.

What is kept in step

	Behaviour
Rig → receiver: frequency	Only the dial moves. Zoom, brightness and contrast stay as you left them; the waterfall scrolls only when the frequency is off screen, and the band plan does not change the mode
Rig → receiver: mode	Follows when the rig's mode changes — USB, LSB, CW, AM, FM; data modes as USB or LSB. A mode the receiver lacks leaves it alone. A running decoder keeps its own mode
Rig → receiver: filter width	The receiver's passband takes the rig's filter width: turning the rig's filter, or pressing its FIL button, changes the passband. A running decoder keeps its own passband
Receiver → rig: frequency	Typing a frequency, a label or bookmark, a click on the waterfall and a passband drag all move the rig's VFO. A drag sends only changed frequencies
Receiver → rig: mode	Every mode change on the page — a mode button, the band plan, a decoder, a bookmark — sets the rig's mode (USB, LSB, CW, AM, FM; RADE as USB or LSB). A rig in a data mode (USB-D) stays in it
Receiver → rig: filter width	The IF Filters buttons, the IF slider, a passband drag and a mode change all set the rig's filter; a drag sends only the width it settles on. On Icom rigs the bridge selects FIL1, FIL2 or FIL3 instead — see Icom filters
No echo	A change that came from the rig is never sent back to it, and a report of the rig's previous mode or filter that arrives just after the page changed it is ignored
Mute on transmit	Always on, even with CAT Sync off: the page falls silent while the rig transmits. Moving the volume slider during transmit keeps it silent; on receive the sound returns at the slider's current level
Not handled	Split, VFO B, RIT/XIT and transverter offsets. Only VFO A of the first receiver is followed

Mode and filter width travel as TCI's own commands (`modulation`, `rx_filter_band`), so ExpertSDR, AetherSDR and Thetis should follow them too; only the Hamlib bridge has been tried. For a transverter offset, use Desktop PhantomSDR+ instead. Do not use both on the same rig at once — two controllers fight over the dial.

Which transceivers it works with

Transceiver	Works	How
SunSDR (ExpertSDR2/3), FlexRadio (AetherSDR), ANAN/Hermes (Thetis)	Yes	Directly — enable the TCI server in the software. Not yet tried with these programs; tested against a simulated TCI server
A radio with a CAT port that Hamlib supports — most Icom, Yaesu, Kenwood, Elecraft, Xiegu, QRP Labs	Yes	Through the Hamlib bridge below. Tried on an Icom IC-7300 , on Linux and Windows; other drivers may differ in mode names or in reporting PTT
A radio with no CAT port, or one Hamlib does not support	No	There is nothing to read the frequency from

The Hamlib bridge (IC-7300 and other rigs without TCI)

`tci-bridge/tci-rigctld.mjs` in the PhantomSDR-Plus tree turns any radio that Hamlib can control into a TCI server for the page. It runs on **your** computer — the one wired to the radio, next to the browser — not on the receiver. Four times a second it asks Hamlib for frequency, mode, filter width and transmit state, sends the page only what changed, and passes the page's frequency, mode and filter changes to the rig.

It reaches Hamlib in one of two ways:

Way	Chain	Use it when
--rigctl (recommended)	transceiver → rigctl → tci-rigctld.mjs → page	Normally. The bridge starts Hamlib's rigctl itself: one window, no network port. Use this on Windows , where rigctld.exe is often refused with <i>Access is denied</i>
rigctld	transceiver → rigctld → tci-rigctld.mjs → page	Another program, such as WSJT-X, must use the radio at the same time — see Sharing the radio

What it needs

	Linux	Windows
Hamlib	<code>sudo apt install libhamlib-utils</code> , or your distribution's hamlib package	hamlib-w64-....zip from github.com/Hamlib/Hamlib/releases , extracted to C:\hamlib — rigctl.exe is in C:\hamlib\bin
Node.js	18 or newer	The LTS <i>Windows Installer (.msi)</i> from nodejs.org , with the default options
The ws package	Found automatically inside the PhantomSDR-Plus tree	Copy tci-rigctld.mjs to a folder such as C:\tci-bridge and run <code>npm install ws</code> there once
The radio's port	<code>ls /dev/serial/by-id/</code> . Join the dialout group once: <code>sudo usermod -aG dialout \$USER</code> , then log out and back in	Install the radio's USB driver from the manufacturer; the COM number is in Device Manager → Ports (COM & LPT)

Only one program can hold the rig's CAT port. Close WSJT-X, flrig, JS8Call, RS-BA1 or Desktop PhantomSDR+'s rig connection before starting the bridge.

Example: Icom IC-7300

On the radio: **MENU → SET → Connectors → CI-V — CI-V USB Baud Rate 115200, CI-V Transceiver ON**. Hamlib's model number for the IC-7300 is 3073.

Linux. The radio is the line containing IC-7300 in `ls /dev/serial/by-id/`, usually also `/dev/ttyUSB0`.

1. Check that Hamlib reaches the radio — it must print the frequency, e.g. 14280000: `bash rigctl -m 3073 -r /dev/ttyUSB0 -s 115200 f`
2. Start the bridge and leave it running: `bash cd ~/PhantomSDR-Plus/tci-bridge node tci-rigctld.mjs --rigctl rigctl -m 3073 -r /dev/ttyUSB0 -s 115200`

Windows. The radio appears in Device Manager as *Silicon Labs CP210x USB to UART Bridge (COM4)* — use your own COM number. In a Command Prompt:

1. Check that Hamlib reaches the radio — it must print the frequency: `bat C:\hamlib\bin\rigctl.exe -m 3073 -r COM4 -s 115200 f`
2. Start the bridge and leave the window open: `bat cd C:\tci-bridge node tci-rigctld.mjs --rigctl C:\hamlib\bin\rigctl.exe -m 3073 -r COM4 -s 115200` The line is long: make sure it really ends in `-s 115200`, or rigctl stops with *Type: rigctl --help*.

On both systems the bridge prints, within a second:

```
rig answered through rigctl
rig -> page 14280000 Hz
rig -> page mode USB
rig -> page RX
```

Then open the receiver page in a browser on the same computer, open **TCI-CAT** and switch **CAT Sync** on. The dot turns green and the bridge prints `page connected`. Turn the VFO and the receiver follows; click the waterfall and the radio follows (`page -> rig ... Hz`); key the radio and the page goes silent. **Ctrl+C** stops the bridge, and `rigctl` with it.

Everything after `--rigctl` is the `rigctl` program and its own options — exactly the ones that worked in the check. That is the rule for every rig: **when `rigctl ... f` prints the frequency, the bridge works with the same options.**

Icom filters (FIL1, FIL2, FIL3)

Icom rigs such as the IC-7300 do not take just any filter width: they have three filters, **FIL1**, **FIL2** and **FIL3**, each with a width set in the rig's menu. Given a width, Hamlib would select one of them and also overwrite its width — and on the IC-7300 that width can land on the filter selected before, scrambling the rig's settings. So for Icom rigs the bridge never sends a width: it picks the filter whose reference width is nearest to the page's passband and only **selects** it, with the rig's own CI-V command. The widths set on the rig are never changed.

Mode	FIL1	FIL2	FIL3
USB, LSB (and their data modes)	2700 Hz	2400 Hz	1800 Hz
CW	1200 Hz	500 Hz	250 Hz
AM	9000 Hz	6000 Hz	3000 Hz
FM	15000 Hz	10000 Hz	7000 Hz

On the IC-7300 (`-m 3073`) this is automatic. Set the rig's SSB filters to match — FIL1 2.7 kHz, FIL2 2.4 kHz, FIL3 1.8 kHz (hold **FIL** on the radio) — and choosing 2.7 / 2.4 / 1.8 kHz in **IF Filters** selects FIL1 / FIL2 / FIL3, while pressing **FIL** on the radio sets the page's passband to that filter's width. The bridge prints, for example, `page -> rig filter 2398 Hz LSB → FIL2 sent`.

Option	Use
<code>--filters 3000,2400,1800</code>	Other SSB reference widths, in the order FIL1,FIL2,FIL3; also switches filter selection on for another Icom rig
<code>--civ A4</code>	The rig's CI-V address in hex, when it is not 94 (IC-705 A4, IC-9700 A2, IC-7610 98)
<code>--filters off</code>	Send widths through Hamlib instead, as for other makes

Both options go before `--rigctl`, e.g. `node tci-rigctld.mjs --filters 3000,2400,1800 --civ A4 --rigctl rigctl -m 3085 -r /dev/ttyACM0 -s 19200`. Tried on an IC-7300 only. Other makes — Yaesu, Kenwood, Elecraft and the rest — receive the page's width through Hamlib, which sets it as closely as the rig allows.

Other rigs

The same steps work for any rig Hamlib supports; only the options change.

1. **Prepare the rig.** In its menu, note the CAT (or CI-V) speed, and switch on any *CAT over USB* or *CI-V transceive* setting it has. On Windows, install the manufacturer's USB driver.
2. **Find the Hamlib model number** in the list `rigctl -l` prints:
3. Linux: `rigctl -l | grep -i 991`
4. Windows: `C:\hamlib\bin\rigctl.exe -l | findstr /i 991`
5. **Find the port.** Linux: `ls /dev/serial/by-id/`. Windows: Device Manager. Some rigs create **two** ports — Yaesu's FT-991A, FTDX10 and FT-710 name them *Enhanced* and *Standard*; CAT is on the **Enhanced** one.

6. **Check, then start the bridge** with `-m <model> -r <port> -s <speed>`, as in the IC-7300 example:
 first `rigctl -m ... -r ... -s ... f`, then the same options after `--rigctl`.

Some model numbers, from Hamlib 4.5 — confirm them with `rigctl -l`, as another Hamlib version may number a rig differently:

Rig	-m	Rig	-m
Icom IC-7300	3073	Yaesu FT-991 / FT-991A	1035
Icom IC-705	3085	Yaesu FTDX10	1042
Icom IC-7610	3078	Yaesu FT-710	1049
Icom IC-9700	3081	Yaesu FT-891	1036
Xiegu G90	3088	Yaesu FT-817	1020
Xiegu X6100	3087	Kenwood TS-590SG	2037
Elecraft K3 / K3S	2029	Kenwood TS-890S	2041
Elecraft KX3	2045	Kenwood TS-2000	2014
Elecraft K4	2047	QRP Labs QCX / QDX	2052

Extra options, only when needed:

- **An Icom rig with a changed CI-V address:** add `-c` and the address in *decimal* — 94h is `-c 148`.
- **A setting Hamlib offers for that rig:** `rigctl -m <model> -L` lists them; set one with `-C name=value`, e.g. `-C post_write_delay=10` for a slow interface.
- **A rig that keys up or resets when the port opens:** some CAT cables use DTR or RTS for PTT; add `-C dtr_state=OFF -C rts_state=OFF`.

Example: Yaesu FT-991A

Not tried on a real FT-991A — it follows Hamlib's settings for the rig; the `rigctl ... f` check tells you at once whether it works.

On the radio, set **CAT RATE** (menu 031) to `38400`. Hamlib's model number for the FT-991 and FT-991A is `1035`.

The FT-991A's USB cable creates **two** serial ports. CAT is on the **Enhanced** one:

- **Linux:** `ls /dev/serial/by-id/` shows two lines for the radio; the one ending in `-if00-port0` is Enhanced, usually `/dev/ttyUSB0`.
- **Windows:** Device Manager shows *Silicon Labs Dual CP2105 USB to UART Bridge: Enhanced COM Port (COM5)* and a *Standard COM Port*; use the Enhanced one's number.

Linux:

```
rigctl -m 1035 -r /dev/ttyUSB0 -s 38400 f
cd ~/PhantomSDR-Plus/tci-bridge
node tci-rigctld.mjs --rigctl rigctl -m 1035 -r /dev/ttyUSB0 -s 38400
```

Windows:

```
C:\hamlib\bin\rigctl.exe -m 1035 -r COM5 -s 38400 f
cd C:\tci-bridge
node tci-rigctld.mjs --rigctl C:\hamlib\bin\rigctl.exe -m 1035 -r COM5 -s 38400
```

If the check prints nothing useful, the usual cause is the Standard port instead of the Enhanced one, or a CAT RATE that differs from `-s`.

Example: Kenwood TS-590SG

Not tried on a real TS-590SG — it follows Hamlib's settings for the rig; the `rigctl ... f` check tells you at once whether it works.

In the radio's menu, set the baud rate of the **USB** port to `115200` (the menu number is in the TS-590SG manual). Hamlib's model number is `2037`; the older TS-590S is `2031`. On Windows, install Kenwood's virtual COM port driver for the USB port first.

The USB cable creates **one** serial port:

- **Linux:** the radio's line in `ls /dev/serial/by-id/`, usually `/dev/ttyUSB0`.
- **Windows:** Device Manager → Ports (COM & LPT), for example `COM6`.

Linux:

```
rigctl -m 2037 -r /dev/ttyUSB0 -s 115200 f
cd ~/PhantomSDR-Plus/tci-bridge
node tci-rigctld.mjs --rigctl rigctl -m 2037 -r /dev/ttyUSB0 -s 115200
```

Windows:

```
C:\hamlib\bin\rigctl.exe -m 2037 -r COM6 -s 115200 f
cd C:\tci-bridge
node tci-rigctld.mjs --rigctl C:\hamlib\bin\rigctl.exe -m 2037 -r COM6 -s 115200
```

If the check fails, the usual cause is a USB baud rate that differs from `-s`, or a cable in the rig's RS-232 (COM) socket while `-r` names the USB port.

More examples

None of these has been tried on real hardware. Each row gives the options to put after `rigctl` for the check and after `--rigctl rigctl` for the bridge; set the rig's menu to the same speed. The Linux port is the usual one — confirm it with `ls /dev/serial/by-id/`. On Windows, replace the port with the COM number from Device Manager and `rigctl` with `C:\hamlib\bin\rigctl.exe`.

Rig	Rig's menu	Options (Linux)	Notes
Icom IC-705 (USB)	CI-V USB Baud Rate 19200	-m 3085 -r /dev/ttyACM0 -s 19200	Two ports appear; CI-V is the first (ttyACM0)
Icom IC-7100 (USB)	CI-V USB Baud Rate 19200	-m 3070 -r /dev/ttyUSB0 -s 19200	Hamlib's highest speed for this rig is 19200
Icom IC-7610	CI-V USB Baud Rate 115200	-m 3078 -r /dev/ttyUSB0 -s 115200	
Icom IC-9700	CI-V USB Baud Rate 38400	-m 3081 -r /dev/ttyUSB0 -s 38400	
Yaesu FTDX101D / FTDX101MP	CAT RATE 38400	-m 1040 -r /dev/ttyUSB0 -s 38400	-m 1044 for the FTDX101MP. Two ports; use Enhanced
Yaesu FTDX10	CAT RATE 38400	-m 1042 -r /dev/ttyUSB0 -s 38400	Two ports; use Enhanced, as on the FT-991A
Yaesu FT-710	CAT RATE 38400	-m 1049 -r /dev/ttyUSB0 -s 38400	Two ports; use Enhanced, as on the FT-991A
Yaesu FT-891	CAT RATE 38400	-m 1036 -r /dev/ttyUSB0 -s 38400	Two ports; use Enhanced, as on the FT-991A
Yaesu FT-450D	CAT RATE 38400	-m 1046 -r /dev/ttyUSB0 -s 38400	RS-232 socket: use a USB-serial adapter
Yaesu FT-817 / FT-818	CAT RATE 38400	-m 1020 -r /dev/ttyUSB0 -s 38400	-m 1041 for the FT-818. Needs a CAT cable on the ACC socket
Yaesu FT-857 / FT-897	CAT RATE 38400	-m 1022 -r /dev/ttyUSB0 -s 38400	-m 1023 for the FT-897. Needs a CAT cable
Kenwood TS-890S (USB)	USB baud rate 115200	-m 2041 -r /dev/ttyUSB0 -s 115200	
Kenwood TS-990S (USB)	USB baud rate 115200	-m 2039 -r /dev/ttyUSB0 -s 115200	
Kenwood TS-590SG / TS-590S (USB)	USB baud rate 115200	-m 2037 -r /dev/ttyUSB0 -s 115200	Worked example above
Kenwood TS-480	COM port baud rate 57600	-m 2028 -r /dev/ttyUSB0 -s 57600	RS-232 socket: use a USB-serial adapter
Kenwood TS-2000	COM port baud rate 57600	-m 2014 -r /dev/ttyUSB0 -s 57600	RS-232 socket; Hamlib's highest speed for this rig is 57600
Elecraft K4 (USB)	RS232 speed 115200	-m 2047 -r /dev/ttyUSB0 -s 115200	
Elecraft K3 / K3S	RS232 speed 38400	-m 2029 -r /dev/ttyUSB0 -s 38400	K3S: USB; K3: serial port or KUSB cable
Elecraft KX3	RS232 speed 38400	-m 2045 -r /dev/ttyUSB0 -s 38400	KXUSB cable
Elecraft KX2	RS232 speed 38400	-m 2044 -r /dev/ttyUSB0 -s 38400	KXUSB cable
Xiegu G90	CI-V baud rate 19200	-m 3088 -r /dev/ttyUSB0 -s 19200	Hamlib uses the G90's default CI-V address

Rig	Rig's menu	Options (Linux)	Notes
Xiegu X6100	CI-V baud rate 19200	<code>-m 3087 -r /dev/ttyUSB0 -s 19200</code>	Hamlib's highest speed for this rig is 19200
Lab599 TX-500	—	<code>-m 2050 -r /dev/ttyUSB0 -s 9600</code>	Hamlib uses 9600 only
ELAD FDM-DUO	—	<code>-m 33001 -r /dev/ttyUSB0 -s 115200</code>	
QRP Labs QDX	—	<code>-m 2052 -r /dev/ttyACM0 -s 9600</code>	A USB serial port; the speed does not matter but must be given

Complete Linux command for the IC-705, as an example of reading a row:

```
rigctl -m 3085 -r /dev/ttyACM0 -s 19200 f
node tci-rigctld.mjs --rigctl rigctl -m 3085 -r /dev/ttyACM0 -s 19200
```

Rigs run by other software

Some radios are already run by a program that can do the job — sometimes with no bridge at all. None of these has been tried.

Radio and program	What to do
SunSDR with ExpertSDR2/3, FlexRadio with AetherSDR, Apache Labs ANAN / Hermes with Thetis	No bridge. Switch on the program's TCI server and use TCI-CAT directly — ports 50001 and 40001 are found automatically
FlexRadio with SmartSDR (Windows)	Add a port in SmartSDR CAT . It speaks Kenwood CAT, so a serial port there works as a TS-2000: <code>-m 2014 -r COM8 -s 57600</code> . A TCP port works with Hamlib's FlexRadio model: <code>-m 2036 -r 127.0.0.1:<port></code>
Any rig controlled by flrig	Leave flrig running and use Hamlib's flrig model; the rig stays shared with fldigi, WSJT-X and loggers: <code>-m 4 -r 127.0.0.1:12345</code>
A running rigctld , or a program offering a <i>Hamlib NET rigctl</i> server	Start the bridge without <code>--rigctl</code> , with that address: <code>node tci-rigctld.mjs 50001 127.0.0.1:4532</code>

As everywhere, the options go after `rigctl` for the check and after `--rigctl rigctl` for the bridge. A network address such as `127.0.0.1:12345` needs no `-s`.

Sharing the radio: the rigctld way

When WSJT-X or a logger must use the radio while the bridge runs, start Hamlib's server `rigctld` with the same options, leave it running, and connect both programs to it:

```
rigctld -m 3073 -r /dev/ttyUSB0 -s 115200
node tci-rigctld.mjs
```

Without `--rigctl` the bridge looks for `rigctld` at `127.0.0.1:4532` and prints `rigctld connected`; `node tci-rigctld.mjs 50001 192.168.1.50:4532` uses a `rigctld` on another computer. In WSJT-X choose *rig Hamlib NET rigctl* at the same address. On Windows the server is `C:\hamlib\bin\rigctld.exe` with the same options; if Windows answers *Access is denied*, use `--rigctl` and close the other program while the bridge runs.

Ports. The bridge serves TCI on port 50001; another port goes first on the command line, e.g. `node tci-rigctld.mjs 40001 --rigctl ...`. If the bridge and the browser are on different computers, put the bridge computer's address in **Host**.

TCI-CAT troubleshooting

Symptom	Likely cause	What to do
Dot stays grey	TCI server or bridge not running; wrong Host ; the browser's local-network permission was refused	Enable TCI in ExpertSDR/Thetis/AetherSDR, or start the bridge; check Host ; allow local network access in the site settings
<code>rigctl ... f</code> gives an error or waits	Wrong port or speed; no dialout permission (Linux) or no USB driver (Windows); another program has the port	Match <code>-s</code> to the rig's menu; check the port (<code>ls /dev/serial/by-id/</code> , Device Manager); close other rig programs
Bridge prints <i>rigctl stopped ... Type: rigctl --help</i>	The command is incomplete — usually the speed after <code>-s</code> is missing	Type the whole line again
Bridge prints <i>The value after -s is missing</i>	The command line was cut short when it was pasted	Type the end of the line again, e.g. <code>-s 115200</code>
The filter does not follow in one direction or the other	CAT Sync is off, a decoder is running on the page, or (IC-7300) the rig's FIL widths differ from 2700 / 2400 / 1800 Hz	Switch CAT Sync on; stop the decoder; set the FIL widths on the rig, or give <code>--filters</code> with the rig's own widths
Bridge prints <i>rigctl stopped ... rig_open: error</i>	<code>rigctl</code> cannot open the port	As for <code>rigctl ... f</code> above
Bridge prints <i>the rig is not answering</i>	The port opens but the rig does not reply: wrong speed or model, CAT switched off in the rig's menu, or a changed Icom CI-V address	Repeat the <code>rigctl ... f</code> check; add <code>-c</code> for a changed CI-V address
Bridge prints <i>rigctld not reachable</i>	Started without <code>--rigctl</code> , and no <code>rigctld</code> is running	Add <code>--rigctl ...</code> , or start <code>rigctld</code> first
<code>rigctld.exe</code> says <i>Access is denied</i> (Windows)	Windows refuses to run it	Use <code>--rigctl</code> — it needs only <code>rigctl.exe</code>
Connected, but the receiver does not move	CAT Sync is off	Switch it on — the mute on transmit works without it
Rig mode does not follow	A mode the receiver has no equivalent for, or the Hamlib driver reports an unusual name	Frequency still syncs; set the mode on the page
No mute on transmit	The rig or its Hamlib driver does not report transmit state	Nothing to set on the page
The dial jumps back and forth	Desktop PhantomSDR+ or another program is also syncing the rig	Use one controller at a time

For receiver operators

Nothing to configure. Rig control uses a small JavaScript interface every PhantomSDR-Plus page already carries; it involves no server setting, no open port and no admin permission. The filter and mute functions came with version 4.1.0 — after applying it, rebuild the frontend (`./recompile.sh`, option 2); the receiver does not need to be stopped. KiwiSDR, WebSDR and UberSDR receivers need nothing either: the app uses the controls their pages already have.

The **TCI-CAT** button came with 4.1.0 too, again a frontend rebuild only. It opens no port on the receiver: the connection runs from each listener's browser to their own computer. The Hamlib bridge, `tci-bridge/tci-rigctld.mjs`, is for listeners to run at home; the receiver does not use it.

For developers: the page interface

Both Desktop PhantomSDR+ and the CATsync Tool use these functions, which every PhantomSDR-Plus page puts on `window` once it has loaded (KiwisDR, WebSDR and UberSDR pages are driven through their own, different controls):

Function	Returns / does
<code>catsync_ready</code>	<code>true</code> once the functions below are installed
<code>catsync_getFrequency()</code>	Tuned frequency, Hz
<code>catsync_setFrequency(hz)</code>	Tune to hz
<code>catsync_getMode()</code>	USB, LSB, CW, CW-L, AM, QUAM, FM, WBFM, RADEU, RADEL
<code>catsync_setMode(mode)</code>	Set the mode; resets the passband to the mode's default
<code>catsync_getBandwidth()</code>	Whole passband width, Hz
<code>catsync_setBandwidth(hz)</code>	Set the width — grows upwards in USB, downwards in LSB, evenly otherwise. Call it after <code>catsync_setMode</code>
<code>catsync_getMute()</code>	<code>true</code> when muted
<code>catsync_setMute(on)</code>	Mute or unmute, through the page's mute button

The last four arrived with 4.1.0, so test before calling:

```
if (window.catsync_ready) {
  window.catsync_setFrequency(7074000)
  window.catsync_setMode('USB')
  if (typeof window.catsync_setBandwidth === 'function') window.catsync_setBandwidth(2400)
}
```

Setting the frequency retunes audio, so call a setter only when the value has actually changed — polling a setter with the same value is audible. The older KiwiSDR/WebSDR-style entry points (`setfreq`, `set_mode`, `freqset_complete`) are still there for tools that expect them.

Troubleshooting

Symptom	Likely cause	What to do
<i>Could not open ttyUSB0 (Linux)</i>	Not in the dialout group, or another program has the port	<code>sudo usermod -aG dialout \$USER</code> , log out and in; close WSJT-X, loggers, rig utilities
The receiver readout says <i>not a receiver this app can drive</i>	Another kind of web receiver, or the page is still loading	Supported are PhantomSDR-Plus, KiwiSDR, PA3FWM WebSDR and UberSDR; give a slow page a few seconds
<i>The rig did not answer</i>	Wrong speed, wrong rig type, wrong CI-V address, rig off	Match the speed to the rig's menu; check the CI-V address; try Hamlib
<i>Hamlib is not installed</i>	No rigctld found	Linux: <code>sudo apt install libhamlib-utils</code> . Or set rigctld program to its path
<i>rigctld stopped: ...</i>	Hamlib could not open the rig — its own message follows	Usually the port or speed; the text after the colon is Hamlib's reason
<i>flrig is not running at ...</i>	flrig closed, or its XML-RPC port differs	Start flrig; check its port in flrig's configuration
Connected, but the receiver does not move	No station window open, or Receiver window pinned to a station that is closed	Open the station, or choose <i>The station window last in front</i>
The rig transmits when connecting	DTR or RTS keys the rig through your interface	Untick DTR on and RTS on
Filter does not follow	Receiver older than 4.1.0, or a built-in Kenwood/Yaesu driver	Frequency and mode still sync; use Hamlib for the filter on Kenwood/Yaesu
Mute on transmit does nothing	Receiver without the update, or the rig does not report transmit state	As above
<i>Lost the rig ... reconnecting</i>	The cable was pulled, the rig was switched off, or rigctld died	Nothing — it retries every 3 seconds and carries on when the rig is back
The two sides keep jumping	Two programs are controlling the rig at once	Let only one program set the rig, or share it through flrig

Known limitations

- One rig, one receiver window at a time.
- Receivers other than PhantomSDR-Plus, KiwiSDR, PA3FWM WebSDR and UberSDR — OpenWebRX, for example — are not supported.
- The built-in drivers follow the manufacturers' published protocols and were tested against simulated rigs and real Hamlib; for a rig that behaves differently, Hamlib is the fallback.
- Split operation, VFO B, RIT/XIT and memory channels are not synced — only the frequency of the active VFO.
- TCI-CAT on the receiver page does not sync split, VFO B, RIT/XIT or transverter offsets, and was tried only on an IC-7300 through the Hamlib bridge.
- On Linux the packages use the distribution's Hamlib; none is bundled.

CHAPTER 16

Project Structure

PROJECT_STRUCTURE.md

PhantomSDR-Plus Project Structure

This document provides a comprehensive overview of the PhantomSDR-Plus directory structure, file organization, and component relationships.

Table of Contents

1. Directory Tree
 2. Root Directory
 3. Source Code (src/)
 4. Frontend (frontend/)
 5. Frequency Lists (frequencylist/)
 6. Configuration Files
 7. Build System
-

Directory Tree

```

PhantomSDR-Plus
├── add-receiver.sh          # adds one more receiver: driver, instances/<name>/, receivers.toml
├── see docs/MULTI_RECEIVER.md
├── admin_config.json
├── admin_server.py
├── autorun
│   ├── audiotap.js
│   ├── bandplan.js
│   ├── decodeworker.js
│   ├── index.js
│   ├── manager.js
│   ├── package.json
│   ├── pool.js
│   ├── probe-ft8.js
│   ├── probe-js8.js
│   ├── pskreporter.js
│   ├── spotparse.js
│   ├── wasm-shim.js
│   └── wsprnet.js
├── cf32_to_real.c          # Fobos HF: keeps I from rx_sdr's CF32, scaled to s16 (built by
├── setup-fobos.sh)
├── chat_history.txt
├── check-go.sh             # legacy: watchdog for the go.sh chain
├── config-airspyhf.toml
├── config.example.hackrf.toml
├── config.example.rtlsdr.toml
├── config-fobos-hf.toml
├── config-fobos.toml
├── config-hackrf.toml
├── config-rspla.toml
├── config-rtl.toml
├── config-rx888mk2.toml
├── config.toml
├── connection_impl.hpp
├── docs
│   ├── ADMIN_PANEL_SETUP.md
│   ├── Aether_config.md
│   ├── CONNECTION_LIMITS.md
│   ├── de
│   │   ├── ADMIN_PANEL_SETUP.md
│   │   ├── Aether_config.md
│   │   ├── CONNECTION_LIMITS.md
│   │   ├── DECODERS.md
│   │   ├── EDITING_VARIANTS.md
│   │   ├── INSTALLATION.md
│   │   ├── MULTI_RECEIVER.md
│   │   ├── PhantomSDR-Plus-Documentation-DE.pdf
│   │   ├── PROJECT_STRUCTURE.md
│   │   ├── RADE_General_INSTALL_MANUAL_LINUX.md
│   │   ├── RADE_README.md
│   │   ├── README.md
│   │   ├── RECEIVE_DIVERSITY.md
│   │   ├── RIG_CONTROL.md
│   │   ├── THERMAL_GUARD.md
│   │   └── USER_GUIDE.md
│   ├── DECODERS.md
│   ├── EDITING_VARIANTS.md
│   └── el
│       ├── ADMIN_PANEL_SETUP.md
│       ├── Aether_config.md
│       ├── CONNECTION_LIMITS.md
│       ├── DECODERS.md
│       ├── EDITING_VARIANTS.md
│       └── INSTALLATION.md

```

- └─ MULTI_RECEIVER.md
- └─ PhantomSDR-Plus-Documentation-EL.pdf
- └─ PROJECT_STRUCTURE.md
- └─ RADE_General_INSTALL_MANUAL_LINUX.md
- └─ RADE_README.md
- └─ README.md
- └─ RECEIVE_DIVERSITY.md
- └─ RIG_CONTROL.md
- └─ THERMAL_GUARD.md
- └─ USER_GUIDE.md
- └─ es
 - └─ ADMIN_PANEL_SETUP.md
 - └─ Aether_config.md
 - └─ CONNECTION_LIMITS.md
 - └─ DECODERS.md
 - └─ EDITING_VARIANTS.md
 - └─ INSTALLATION.md
 - └─ MULTI_RECEIVER.md
 - └─ PhantomSDR-Plus-Documentation-ES.pdf
 - └─ PROJECT_STRUCTURE.md
 - └─ RADE_General_INSTALL_MANUAL_LINUX.md
 - └─ RADE_README.md
 - └─ README.md
 - └─ RECEIVE_DIVERSITY.md
 - └─ RIG_CONTROL.md
 - └─ THERMAL_GUARD.md
 - └─ USER_GUIDE.md
- └─ fr
 - └─ ADMIN_PANEL_SETUP.md
 - └─ Aether_config.md
 - └─ CONNECTION_LIMITS.md
 - └─ DECODERS.md
 - └─ EDITING_VARIANTS.md
 - └─ INSTALLATION.md
 - └─ MULTI_RECEIVER.md
 - └─ PhantomSDR-Plus-Documentation-FR.pdf
 - └─ PROJECT_STRUCTURE.md
 - └─ RADE_General_INSTALL_MANUAL_LINUX.md
 - └─ RADE_README.md
 - └─ README.md
 - └─ RECEIVE_DIVERSITY.md
 - └─ RIG_CONTROL.md
 - └─ THERMAL_GUARD.md
 - └─ USER_GUIDE.md
- └─ hr
 - └─ ADMIN_PANEL_SETUP.md
 - └─ Aether_config.md
 - └─ CONNECTION_LIMITS.md
 - └─ DECODERS.md
 - └─ EDITING_VARIANTS.md
 - └─ INSTALLATION.md
 - └─ MULTI_RECEIVER.md
 - └─ PhantomSDR-Plus-Documentation-HR.pdf
 - └─ PROJECT_STRUCTURE.md
 - └─ RADE_General_INSTALL_MANUAL_LINUX.md
 - └─ RADE_README.md
 - └─ README.md
 - └─ RECEIVE_DIVERSITY.md
 - └─ RIG_CONTROL.md
 - └─ THERMAL_GUARD.md
 - └─ USER_GUIDE.md
- └─ INSTALLATION.md
- └─ MULTI_RECEIVER.md
- └─ PhantomSDR-Plus-Documentation-EN.pdf
- └─ PROJECT_STRUCTURE.md

```

├── RADE_General_INSTALL_MANUAL_LINUX.md
├── RADE_README.md
├── README.md
├── RECEIVE_DIVERSITY.md
├── RIG_CONTROL.md
├── ru
│   ├── ADMIN_PANEL_SETUP.md
│   ├── Aether_config.md
│   ├── CONNECTION_LIMITS.md
│   ├── DECODERS.md
│   ├── EDITING_VARIANTS.md
│   ├── INSTALLATION.md
│   ├── MULTI_RECEIVER.md
│   ├── PhantomSDR-Plus-Documentation-RU.pdf
│   ├── PROJECT_STRUCTURE.md
│   ├── RADE_General_INSTALL_MANUAL_LINUX.md
│   ├── RADE_README.md
│   ├── README.md
│   ├── RECEIVE_DIVERSITY.md
│   ├── RIG_CONTROL.md
│   ├── THERMAL_GUARD.md
│   └── USER_GUIDE.md
├── sdr-stats
│   ├── package.json
│   ├── readme_de.md
│   ├── readme_el.md
│   ├── readme_es.md
│   ├── readme_fr.md
│   ├── readme_hr.md
│   ├── README.md
│   └── readme_ru.md
├── THERMAL_GUARD.md
├── USER_GUIDE.md
├── websdr2.png
├── websdr3.png
├── websdr.png
├── favicon.ico
├── fftw_wisdom
├── fix_local_geo.py
├── frequencylist
│   ├── 0.TXT
│   ├── A26all00.TXT          # not in the repo: unzipped from a26allx2.zip by update-markers.sh
│   ├── a26allx2.zip
│   ├── admin.txt
│   ├── antenna.txt
│   ├── broadcas.txt
│   ├── curl-output.txt
│   ├── currentUpdateFile.txt
│   ├── fmorg.txt
│   ├── generate-current-shortwave.py
│   ├── language.txt
│   ├── mymarkers.json
│   ├── README.md
│   ├── shortwavestations.json
│   ├── site.txt
│   └── update-markers.sh
├── frontend
│   ├── build-all.sh
│   ├── build-default.sh
│   ├── build-mobile.sh
│   ├── debug-title.sh
│   ├── favicon.ico
│   ├── fix-title-python.py
│   ├── index.html
│   └── jsconfig.json

```

```

├─ LICENSE
├─ make-redirect-stubs.sh
├─ mobile
│   └─ index.html
├─ package.json
├─ package-lock.json
├─ pnpm-lock.yaml
├─ postcss.config.cjs
├─ .prettierrc.json      # Prettier formatting rules for the frontend sources
├─ public
│   ├── analyze_users.py
│   ├── decoders
│   │   ├── ft8_lib.wasm
│   │   ├── js8_dict.bin  # JS8 word dictionary, fetched only when needed
│   │   └─ js8.wasm      # JS8 decoder, built from jsdsp/js8_wasm
│   ├── logo.jpg
│   ├── stats.html
│   ├── users.html
│   └─ wf-message.json
├─ README.md
├─ site_information.json
├─ src
│   ├── app.css
│   ├── App.svelte
│   ├── assets
│   │   ├── amateurfrequencies.json
│   │   ├── background.jpg
│   │   ├── shortwavestations.json
│   │   ├── SSTV.png
│   │   ├── SSTV.svg
│   │   └─ svelte.png
│   ├── audio.js
│   ├── audio-stream-worklet.js
│   ├── bands-config.js
│   ├── broadcastSchedules.js
│   ├── clientVersion.js
│   ├── cwDecoder.js
│   ├── cw.worker.js
│   ├── cwWorkerProxy.js
│   ├── decoder.worker.js
│   ├── diversity.js
│   ├── diversityList.js
│   ├── eventBus.js
│   ├── events.js
│   ├── fax.js
│   ├── fax.worker.js
│   ├── faxWorkerProxy.js
│   ├── fft.js
│   ├── flacLowLatency.js      # FLAC decoder that plays each packet as it arrives
│   ├── fsk.js
│   ├── fsk.worker.js
│   ├── fskWorkerProxy.js
│   ├── kiwiSource.js
│   ├── lib
│   │   ├── backend.js
│   │   ├── BandSelector.svelte
│   │   ├── catsync.js
│   │   ├── chatReply.js      # chat reply marker and threading
│   │   ├── CheckButton.svelte
│   │   ├── colormaps.js
│   │   ├── Counter.svelte
│   │   ├── DiversityPanel.svelte
│   │   ├── fftRadix2.js
│   │   ├── freedv-reporter.js
│   │   └─ FreeDVReporter.svelte

```

```

├─ FrequencyInput.svelte
├─ FrequencyMarkers.svelte
├─ FtxSpectrum.svelte
├─ hammeractions.js
├─ LineThroughButton.svelte
├─ Logger.svelte
├─ MagicEyeIndicator.svelte
├─ ModeIdChip.svelte
├─ ModesSelector.svelte
├─ opusMlDecoder.js
├─ PassbandTuner.svelte
├─ Popover.svelte
├─ QrssPanel.svelte
├─ receivers.js          # receiver picker data (/receivers.json)
├─ rnnoise.js           # loads RNNoise for the AI noise reduction
├─ rx.js                # which receiver a page belongs to (?rx=)
├─ SMeterAnalog.svelte
├─ SMeterDigital.svelte
├─ Spectrogram.svelte
├─ StatusIndicators.svelte
├─ storage.js
├─ sUnits.js            # S-unit scale by frequency (S9 = -93 dBm above 30 MHz)
├─ Tooltip.svelte
├─ VersionSelector.svelte
├─ VideoAreaSelector.svelte
├─ wrappers.js
├─ main.js
├─ mobile
│   └─ backend.js
│   └─ bookmarks.js
│   └─ main.js
│   └─ Mobile.svelte
│   └─ tuning.js
├─ modeId.js
├─ modeId.worker.js
├─ modeIdWorkerProxy.js
├─ modePriors.js
├─ modules
│   └─ decode.wasm
│   └─ encode.wasm
│   └─ ft4.js
│   └─ ft8.js
│   └─ js8-decoder.js
│   └─ js8-format.js
│   └─ js8.js
│   └─ js8-reassembler.js
│   └─ js8-slots.js
│   └─ js8-tables.js
│   └─ package.json
│   └─ phantomsdrdsp_bg_fallback.js
│   └─ phantomsdrdsp_bg.js
│   └─ phantomsdrdsp_bg.wasm
│   └─ phantomsdrdsp_bg.wasm.d.ts
│   └─ phantomsdrdsp.d.ts
│   └─ phantomsdrdsp.js
│   └─ phantomsdrdsp_router.js
│   └─ wspr.js
├─ olivia.js
├─ playoutControl.js     # measures the connection, keeps the playback buffer small
├─ psk31.js
├─ refused.js
├─ remoteSource.js
├─ scanner.js
├─ siteInfo.js          # site_information.json, overlaid with the receiver's own

```

copy

```

├── sstv.js
├── sstv.worker.js
├── sstvWorkerProxy.js
├── uberSource.js
├── unused
│   ├── AudioProcessor.js
│   ├── decoder.js
│   ├── decoding.js
│   └── modules-emscripthen
│       ├── dav1d.js
│       ├── dav1dnoWasm.js
│       ├── dav1dnoWasm.js.mem
│       ├── dav1d.wasm
│       ├── decode_ft8.js
│       ├── decode_ft8.wasm
│       ├── FoxenFlac.js
│       ├── jsDSP.js
│       ├── jsDSPnoWasm.js
│       ├── jsDSPnoWasm.js.mem
│       ├── jsDSPnoWasm.wasm
│       ├── jsDSP.wasm
│       ├── libzstd.js
│       ├── LiquidDSP.js
│       ├── opus.js
│       ├── opusnoWasm.js
│       ├── opusnoWasm.js.mem
│       ├── opus.wasm
│       ├── redsea.js
│       └── redsea.wasm
├── unused.js
├── wrappers.js
├── videoRecorder.js
├── vite-env.d.ts
├── waterfall.js
├── webSdrCodec.js
├── webSdrSource.js
├── stats.html
├── svelte.config.js
├── tailwind.config.cjs
├── vite.config.js
├── go.sh # legacy launcher chain, superseded by start-<radio>.sh
├── install_arch.sh
├── install_fedora.sh
├── install_opensuse.sh
├── install_rade.sh
├── install_rade_ubuntu22.sh
├── install.sh
├── install-stats-server.sh
├── instances # extra receivers, one folder each (not in git) – see docs/
├── MULTI_RECEIVER.md
├── instructions-for-airspy
├── instructions-for-rspla
├── jsdsp
│   ├── compilejs.sh
│   ├── configureredsea.sh
│   ├── extract_EXPORTED_FUNCTIONS.js
│   ├── ft8_wasm
│   │   ├── build_ft8_wasm.sh
│   │   ├── README.md
│   │   └── wasm_wrapper.c
│   └── include
│       ├── avif
│       │   ├── avif.h
│       │   └── internal.h
│       └── liquid

```

```

├── liquid.h
├── js8_wasm          # JS8 decoder in C, built to WebAssembly, with its tests and
reference harness – see its README
├── build_js8_wasm.sh
├── check_against_reference.sh
├── extract_js8_constants.py
├── extract_js8_varicode.py
├── js8_ab_test.mjs
├── js8_autorun_test.mjs
├── js8_chain_test.mjs
├── js8_constants.c
├── js8_constants.h
├── js8_decode.c
├── js8_decode.h
├── js8_decode_test.c
├── js8_encode.c
├── js8_encode.h
├── js8_frames_test.mjs
├── js8_osd.c
├── js8_osd.h
├── js8_reassembly_test.mjs
├── js8_roundtrip.c
├── js8_slots_test.mjs
├── js8_subtract.c
├── js8_subtract.h
├── js8_wasm_test.mjs
├── js8_wasm_wrapper.c
├── README.md
├── reference
├──   ├── build_in_container.sh
├──   ├── build_reference_decoder.sh
├──   ├── build_reference_frames.sh
├──   ├── Containerfile
├──   ├── Containerfile.decoder
├──   ├── ref_decode.f90
├──   ├── ref_decode.txt
├──   ├── ref_frames.cpp
├──   └── ref_frames.txt
├── ref_gen.f90
├── run_tests.sh
├── lib
├──   ├── ANR.c
├──   ├── arm_funcs.h
├──   ├── CMSIS_DSP
├──   ├──   ├── BUILDING.txt
├──   └──   └── LICENSE.txt
├──   ├── dav1d.cpp
├──   ├── NB.c
├──   ├── NR_spectral.c
├──   └── types.h
├── redsea.js
├── redsea.wasm
├── src
├──   ├── index.js
├──   ├── libzstd.js
├──   ├── LiquidDSP.js
├──   ├── NoiseProcessing.js
├──   └── wbfmpll.cpp
├── kill.sh          # legacy: kills the server processes, called by go.sh
├── kiwi_install.sh  # installs the KiwiSDR bridge into an existing tree
├── LICENSE
├── logproxy         # rotated copies of the panel / proxy / autorun logs
├── logrotate
├──   └── phantomsdr  # logrotate config for proxy.log + admin.log (edit paths before
installing)

```

```

├─ manage_admin.sh
├─ markers.json
├─ meson.build
├─ meson_options.txt
├─ phantom_fftw_wisdom
├─ phantomsdr-admin.service # sample systemd unit for the admin panel (starts at boot, restarts
after a crash)
├─ phantomsdr-proxy.service # sample systemd unit for the proxy, installed alongside the panel
unit
├─ phantomsdr-websdr-relay.service # sample systemd unit for the WebSDR diversity relay
├─ proxy.py
├─ rade_helper.py
├─ rade_loadtest.csv # output of rade_loadtest.py (one row per ramp step)
├─ rade_loadtest.py # RADE concurrency knee test – see docs/RADE_README.md
├─ rade.sh
├─ README.md
├─ recompile.sh
├─ receivers.toml.example # receiver list for proxy.py – copy to receivers.toml
├─ _relaunch.sh # legacy: delayed re-launch helper for the go.sh chain
├─ request.hpp
├─ setup_admin.sh
├─ setup-airspyhf.sh # Airspy HF+ driver chain: libairspyhf + SoapyAirspyHF + rx_sdr +
udev
├─ setup-cpufreq-perms.sh # grants group write on scaling_max_freq so the guard can throttle
without root
├─ setup-firewall.sh # optional nftables flood guard – see docs/CONNECTION_LIMITS.md
├─ setup-fobos.sh # Fobos driver chain: libfobos + SoapyFobosSDR + rx_sdr +
cf32_to_real + udev
├─ setup-hackrf.sh # HackRF: the distribution's hackrf package + udev (nothing built)
├─ setup-rspla.sh # RSP1A driver chain: libmirisdr-5 + SoapyMiri + rx_sdr + msi2500
blacklist + udev
├─ setup-rtlsdr.sh # RTL-SDR driver: the distribution's package, or the Blog V4 driver
from source + DVB blacklist + udev
├─ setup-rx888-udev.sh
├─ setup-sdr-common.sh # shared helpers sourced by the three setup scripts above
├─ setup_websdr_relay.sh # installs the WebSDR diversity relay (port, identity, systemd)
├─ smeter_theme.sh
├─ src
├─ │ audio.cpp
├─ │ audio.h
├─ │ chat.cpp
├─ │ chat.h
├─ │ client.cpp
├─ │ client.h
├─ │ compression.cpp
├─ │ compression.h
├─ │ crash_handler.cpp
├─ │ crash_handler.h
├─ │ events.cpp
├─ │ events.h
├─ │ fft.cpp
├─ │ fft_cuda.cu
├─ │ fft.h
├─ │ fft_impl.cpp
├─ │ fft_mkl.cpp
├─ │ http.cpp
├─ │ kiwi_bridge.h # KiwiSDR protocol bridge – see docs/Aether_config.md
├─ │ listing
├─ │ │ software_info.cpp
├─ │ │ software_info.h
├─ │ samplerreader.cpp
├─ │ samplerreader.h
├─ │ signal.cpp
├─ │ signal.h
├─ │ spectrumserver.cpp

```

```

├── spectrumserver.h
├── utils
│   ├── audioprocessing.cpp
│   ├── audioprocessing.h
│   ├── dsp.cpp
│   └── dsp.h
├── utils.cpp
├── utils.h
├── waterfallcompression.cpp
├── waterfallcompression.h
├── waterfall.cpp
├── waterfall.h
├── websocket.cpp
└── websocket.h
├── start-airspyhf.sh
├── start-all.sh           # starts every receiver in receivers.toml
├── start-fobos-hf.sh
├── start-fobos.sh
├── start-hackrf.sh
├── start-rspla.sh
├── start-rtl.sh
├── start-rx888mk2.sh
├── stop-websdr.sh
├── subprojects
│   ├── fftw3.wrap
│   ├── flac.wrap
│   ├── glaze.wrap
│   ├── libcds.wrap
│   ├── libflac.wrap
│   ├── libvolk.wrap
│   ├── ogg.wrap
│   ├── opus.wrap
│   ├── tomlplusplus-3.4.0
│   │   ├── CHANGELOG.md
│   │   ├── cmake
│   │   │   ├── install-rules.cmake
│   │   │   ├── project-is-top-level.cmake
│   │   │   ├── tomlplusplusConfig.cmake
│   │   │   ├── tomlplusplusConfig.cmake.meson.in
│   │   │   ├── tomlplusplusConfigVersion.cmake.meson.in
│   │   │   └── variables.cmake
│   │   ├── CMakeLists.txt
│   │   ├── CODE_OF_CONDUCT.md
│   │   ├── CONTRIBUTING.md
│   │   ├── cpp.hint
│   │   └── docs
│   │       ├── images
│   │       │   ├── badge-awesome.svg
│   │       │   ├── badge-C++17.svg
│   │       │   ├── badge-gitter.svg
│   │       │   ├── badge-license-MIT.svg
│   │       │   ├── badge-TOML.svg
│   │       │   ├── banner.ai
│   │       │   ├── banner.png
│   │       │   ├── banner.svg
│   │       │   ├── favicon.ico
│   │       │   ├── logo.ai
│   │       │   └── logo.svg
│   │       ├── pages
│   │       │   └── main_page.md
│   │       └── poxy.toml
│   └── examples
│       ├── benchmark_data.toml
│       ├── CMakeLists.txt
│       └── error_printer.cpp

```

```

├─ error_printer.vcxproj
├─ examples.hpp
├─ example.toml
├─ merge_base.toml
├─ merge_overrides.toml
├─ meson.build
├─ parse_benchmark.cpp
├─ parse_benchmark.vcxproj
├─ simple_parser.cpp
├─ simple_parser.vcxproj
├─ toml_generator.cpp
├─ toml_generator.vcxproj
├─ toml_merger.cpp
├─ toml_merger.vcxproj
├─ toml_to_json_transcoder.cpp
├─ toml_to_json_transcoder.vcxproj
├─ include
├─   └─ meson.build
├─   └─ toml++
├─     └─ impl
├─       └─ array.hpp
├─       └─ array.inl
├─       └─ at_path.hpp
├─       └─ at_path.inl
├─       └─ date_time.hpp
├─       └─ formatter.hpp
├─       └─ formatter.inl
├─       └─ forward_declarations.hpp
├─       └─ header_end.hpp
├─       └─ header_start.hpp
├─       └─ json_formatter.hpp
├─       └─ json_formatter.inl
├─       └─ key.hpp
├─       └─ make_node.hpp
├─       └─ node.hpp
├─       └─ node.inl
├─       └─ node_view.hpp
├─       └─ parse_error.hpp
├─       └─ parse_result.hpp
├─       └─ parser.hpp
├─       └─ parser.inl
├─       └─ path.hpp
├─       └─ path.inl
├─       └─ preprocessor.hpp
├─       └─ print_to_stream.hpp
├─       └─ print_to_stream.inl
├─       └─ simd.hpp
├─       └─ source_region.hpp
├─       └─ std_except.hpp
├─       └─ std_initializer_list.hpp
├─       └─ std_map.hpp
├─       └─ std_new.hpp
├─       └─ std_optional.hpp
├─       └─ std_string.hpp
├─       └─ std_string.inl
├─       └─ std_utility.hpp
├─       └─ std_variant.hpp
├─       └─ std_vector.hpp
├─       └─ table.hpp
├─       └─ table.inl
├─       └─ toml_formatter.hpp
├─       └─ toml_formatter.inl
├─       └─ unicode_autogenerated.hpp
├─       └─ unicode.hpp
├─       └─ unicode.inl

```

```

├── value.hpp
├── version.hpp
├── yaml_formatter.hpp
├── yaml_formatter.inl
├── toml.h
├── toml.hpp
├── LICENSE
├── meson.build
├── meson_options.txt
├── README.md
├── src
│   ├── meson.build
│   └── toml.cpp
├── tests
│   ├── at_path.cpp
│   ├── conformance_burntsushi_invalid.cpp
│   ├── conformance_burntsushi_valid.cpp
│   ├── conformance_iarna_invalid.cpp
│   ├── conformance_iarna_valid.cpp
│   ├── cpp.hint
│   ├── for_each.cpp
│   ├── formatters.cpp
│   ├── impl_toml.cpp
│   ├── leakproof.hpp
│   ├── lib_catch2.hpp
│   ├── main.cpp
│   ├── manipulating_arrays.cpp
│   ├── manipulating_parse_result.cpp
│   ├── manipulating_tables.cpp
│   ├── manipulating_values.cpp
│   ├── meson.build
│   ├── odr_test_1.cpp
│   ├── odr_test_2.cpp
│   ├── parsing_arrays.cpp
│   ├── parsing_booleans.cpp
│   ├── parsing_comments.cpp
│   ├── parsing_dates_and_times.cpp
│   ├── parsing_floats.cpp
│   ├── parsing_integers.cpp
│   ├── parsing_key_value_pairs.cpp
│   ├── parsing_spec_example.cpp
│   ├── parsing_strings.cpp
│   ├── parsing_tables.cpp
│   ├── path.cpp
│   ├── settings.hpp
│   ├── tests.cpp
│   ├── tests.hpp
│   ├── user_feedback.cpp
│   ├── using_iterators.cpp
│   ├── visit.cpp
├── vs
│   ├── odr_test.vcxproj
│   ├── test_debug_x64_cpplatest_noexcept_unrel.vcxproj
│   ├── test_debug_x64_cpplatest_noexcept.vcxproj
│   ├── test_debug_x64_cpplatest_unrel.vcxproj
│   ├── test_debug_x64_cpplatest.vcxproj
│   ├── test_debug_x64_noexcept_unrel.vcxproj
│   ├── test_debug_x64_noexcept.vcxproj
│   ├── test_debug_x64_unrel.vcxproj
│   ├── test_debug_x64.vcxproj
│   ├── test_debug_x86_cpplatest_noexcept_unrel.vcxproj
│   ├── test_debug_x86_cpplatest_noexcept.vcxproj
│   ├── test_debug_x86_cpplatest_unrel.vcxproj
│   ├── test_debug_x86_cpplatest.vcxproj
│   └── test_debug_x86_noexcept_unrel.vcxproj

```

```

├── test_debug_x86_noexcept.vcxproj
├── test_debug_x86_unrel.vcxproj
├── test_debug_x86.vcxproj
├── test_release_x64_cpplatest_noexcept_unrel.vcxproj
├── test_release_x64_cpplatest_noexcept.vcxproj
├── test_release_x64_cpplatest_unrel.vcxproj
├── test_release_x64_cpplatest.vcxproj
├── test_release_x64_noexcept_unrel.vcxproj
├── test_release_x64_noexcept.vcxproj
├── test_release_x64_unrel.vcxproj
├── test_release_x64.vcxproj
├── test_release_x86_cpplatest_noexcept_unrel.vcxproj
├── test_release_x86_cpplatest_noexcept.vcxproj
├── test_release_x86_cpplatest_unrel.vcxproj
├── test_release_x86_cpplatest.vcxproj
├── test_release_x86_noexcept_unrel.vcxproj
├── test_release_x86_noexcept.vcxproj
├── test_release_x86_unrel.vcxproj
├── test_release_x86.vcxproj
├── windows_compat.cpp
├── toml++.code-workspace
├── toml.hpp
├── toml++.natvis
├── toml++.props
├── toml++.sln
├── toml-test
│   ├── meson.build
│   ├── README.md
│   ├── tt_decoder.cpp
│   ├── tt_decoder.vcxproj
│   ├── tt_encoder.cpp
│   ├── tt_encoder.vcxproj
│   └── tt.hpp
├── toml++.vcxproj
├── toml++.vcxproj.filters
├── tools
│   ├── ci_single_header_check.py
│   ├── clang_format.bat
│   ├── generate_conformance_tests.py
│   ├── generate_single_header.bat
│   ├── generate_single_header.py
│   ├── generate_windows_test_targets.py
│   ├── requirements.txt
│   ├── utils.py
│   └── version.py
├── vendor
│   ├── catch.hpp
│   ├── json.hpp
│   └── README.md
├── tomlplusplus.wrap
├── websocketpp.wrap
├── zlib.wrap
├── zstd.wrap
├── tci-bridge
│   └── tci-rigctld.mjs          # TCI server for Hamlib rigs (IC-7300...), run on the listener's PC –
see docs/RIG_CONTROL.md
├── thermal_guard.py             # CPU over-temperature guard used by the admin panel (also runs
standalone)
├── thermal-guard.service       # sample systemd unit for the guard, for installs without the admin
panel
├── tmpfiles
│   └── phantomsdr-logs.conf    # keeps admin.log + proxy.log owned by the panel's user (edit paths
before installing)
├── update.sh
└── waterfall.sh

```

```

└─ websdr_relay.json.example # its config template (port, caps, station identity)
└─ websdr_relay.py           # the relay itself – see docs/RECEIVE_DIVERSITY.md
└─ websocketpp_asio_connection.hpp # Boost >= 1.87 shim: ws_post instead of io_service::post
└─ websocketpp_asio_endpoint.hpp  # Boost >= 1.87 shim: ws_work / ws_restart,
max_listen_connections
└─ websocketpp_asio.hpp           # Boost >= 1.87 shim for websocketpp (io_context,
executor_work_guard)
└─ xgo.sh                        # legacy: starts spectrumserver, called by check-go.sh

```

Root Directory

Configuration Files

File	Purpose	When to Modify
config.toml	Default configuration	Initial setup, testing
config-rtl.toml	RTL-SDR specific config	Using RTL-SDR device
config-rspla.toml	SDRplay RSP1A config	Using RSP1A device
config-airspyhf.toml	Airspy HF+ config	Using Airspy device
config-fobos-hf.toml	RigExpert Fobos SDR, HF1/HF2 direct sampling (0-25 MHz)	Using a Fobos on HF
config-fobos.toml	RigExpert Fobos SDR, RF path (25-6000 MHz)	Using a Fobos above 25 MHz
config-hackrf.toml	HackRF One config	Using a HackRF
config-rx888mk2.toml	RX888 MK2 config	Using RX888 device
config.example.hackrf.toml	HackRF One example	Using HackRF device

Start/Stop and Maintenance Scripts

Each `start-*.sh` below is a **self-contained launcher + watchdog + logger**: it stops any running instance, brings up the receiver + `spectrumserver`, detaches into the background, auto-restarts the chain if it dies, and logs to `logwebsdr.txt`. They share one stop script and a single `flock` lock (only one receiver runs at a time). Edit only the **RECEIVER CONFIGURATION** block at the top of each (receiver args / config / process name). The six launchers built from the same template (`start-airspyhf.sh`, `start-rtl.sh`, `start-rspla.sh`, `start-fobos.sh`, `start-fobos-hf.sh`, `start-hackrf.sh`) are identical below that block; `start-rx888mk2.sh` is a script of its own. Since v5.0.0 several receivers can run side by side: start a launcher with `INSTANCE=<name>` and that receiver gets its own folder in `instances/<name>/`, its own log, lock and FIFO — see [Several Receivers](#).

Script	Purpose
<code>install.sh</code>	Automated installation and build
<code>add-receiver.sh</code>	Adds one more receiver to this computer: asks which receiver and what it covers, installs its driver, creates <code>instances/<name>/</code> and the <code>receivers.toml</code> entry, and offers to start it and restart the proxy. The installers offer it at the end — see Several Receivers
<code>start-rtl.sh</code>	Launch + watchdog server with RTL-SDR (<code>rtl_sdr</code>)
<code>start-rsp1a.sh</code>	Launch + watchdog server with SDRplay RSP1A (<code>rx_sdr</code>); uses <code>libmirisdr-5</code> , or SDRplay's API when that is installed (<code>RX_DRIVER</code> forces one)
<code>start-airspyhf.sh</code>	Launch + watchdog server with Airspy HF+ (<code>rx_sdr</code>)
<code>start-fobos-hf.sh</code>	Launch + watchdog server with RigExpert Fobos SDR, HF1/HF2 direct sampling (<code>rx_sdr \ cf32_to_real</code>)
<code>start-fobos.sh</code>	Launch + watchdog server with RigExpert Fobos SDR, RF path (<code>rx_sdr</code>)
<code>start-hackrf.sh</code>	Launch + watchdog server with HackRF One (<code>hackrf_transfer</code>)
<code>start-rx888mk2.sh</code>	Launch + watchdog server with RX888 MK2 (<code>rx888_stream</code>)
<code>start-all.sh</code>	Starts every receiver listed in <code>receivers.toml</code> through its own launcher — set it as the admin panel's start script on a station with several receivers; see Several Receivers
<code>stop-websdr.sh</code>	Stop the server + its watchdog — shared by all receivers
<code>stop-websdr.sh</code> <code><instance></code>	Stops one receiver only (main for the main one); without an argument every receiver stops, as before
<code>setup-rx888-udev.sh</code>	Install udev rules so RX-888 <code>rx888_stream</code> runs without <code>sudo</code>
<code>setup-rsp1a.sh</code> / <code>setup-fobos.sh</code> / <code>setup-airspyhf.sh</code>	Install the SoapySDR driver chain for the RSP1A, Fobos or Airspy HF+ — SoapySDR from the distribution, the driver and <code>rx_tools</code> built into <code>sdr_drivers/</code> , a udev rule — on any of the four supported distributions. Installer options 3, 5 and 6 run them; they also run on their own. Shared code in <code>setup-sdr-common.sh</code> — see Installation Guide
<code>setup-hackrf.sh</code>	Installs the distribution's <code>hackrf</code> package and a udev rule for the HackRF One, on any of the four supported distributions; installer option 7 runs it
<code>setup-rtlsdr.sh</code>	Installs the RTL-SDR driver on any of the four supported distributions — the distribution's <code>rtl-sdr</code> package, or with <code>RTL_V4=y</code> the RTL-SDR Blog V4 driver built from source — blacklists the DVB-T driver and adds a udev rule; <code>add-receiver.sh</code> runs it
<code>setup-firewall.sh</code>	Optional kernel-level flood guard: loads an <code>nftables</code> table with a per-source connection ceiling and rate on the receiver ports, an SSH brute-force brake, and Windows file sharing closed outside private ranges. Needs root, cannot lock you out (policy accept, established accepted first) and <code>--apply</code> rolls itself back unless confirmed within 60 s — see Connection Limits
<code>setup-cpufreq-perms.sh</code>	Grants a <code>cpufreq</code> group write access to the per-CPU frequency limit, so the thermal guard's throttle stage works without running the panel as root. Installs a <code>tmpfiles.d</code> rule so it survives a reboot; <code>--revoke</code> undoes it
<code>update.sh</code>	Update the installation from the published tree, leaving your configuration, markers, frequency list and local edits alone — see Installation Guide
<code>recompile.sh</code>	Rebuild the backend and/or the frontend, and pick the variant served at /
<code>smeter_theme.sh</code>	Set the default analog S-meter face (dark / amber / vintage) for all users, and offer the frontend rebuild — see Editing Variants

Script	Purpose
waterfall.sh	Change the default minimum waterfall level (dB) in waterfall.js + App.svelte — see README
kiwi_install.sh	Install the KiwiSDR client-emulation bridge into a tree that has not got it: patches the backend sources, copies src/kiwi_bridge.h, and adds a documented [kiwi_emulation] block to the config files in the repository root. Idempotent, and backs up every file it touches — see KiwiSDR Client Emulation
tci-bridge/tci-rigctld.mjs	Not used by the receiver. A small Node.js program a listener runs next to their own transceiver: it reads frequency, mode and transmit state from Hamlib's rigctld and serves them as TCI on port 50001, so the page's TCI-CAT button can drive a rig with no TCI of its own, such as the IC-7300 — see Rig Control

Legacy launcher chain. go.sh, xgo.sh, check-go.sh, kill.sh and _relaunch.sh are the previous generation of launcher, watchdog and stop scripts. Everything they did is now inside each start-<radio>.sh, which is what you should use. They are kept on disk because existing installations still reference them, and are not maintained.

Data Files

File	Purpose	Format
markers.json	Frequency bookmarks and markers	JSON
chat_history.txt	User chat messages	Plain text
favicon.ico	Website icon	ICO image
fftw_wisdom	FFT optimization data	FFTW binary
phantom_fftw_wisdom	Additional FFT optimization	FFTW binary

Admin Panel and Spot Reporting

File	Purpose
admin_server.py	The admin panel itself. Besides the management pages it runs the Graphs sampler: a background thread takes CPU frequency, load, temperature and users online every 2 seconds and keeps them in memory only — 1 hour at full resolution plus 24 hours of 30-second averages. Nothing is written to disk, so the history is lost on restart.
admin_config.json	Admin panel settings (password hash, ports, options, thermal guard thresholds)
thermal_guard.py	CPU over-temperature guard. Stops the server when the CPU gets too hot and restarts it once cool, deriving its thresholds from the critical trip point your own CPU publishes instead of a fixed number. It never identifies what supervises the server — while over-temp it re-issues the stop every 2 s, so a watchdog, systemd unit or cron job that revives it is undone until the machine cools. Stdlib-only; imported by admin_server.py (ticked from the Graphs sampler) and also runnable standalone for installations without the panel. Defaults to log-only, so it acts on nothing until enabled — see the Thermal Guard manual
autorun/	The spot-reporting daemon — see Autorun Spot Reporter
autorun.json	Which bands/modes to decode, identity and destinations
autorun-status.json	Live state of the running daemon — feeds the per-run spot counters (the per-decoder tiles), which reset on Stop/Start
autorun-totals.json	All-time uploaded spots per band+mode — the number beside each checkbox; written by the daemon so the counts survive restarts

Build System Files

File	Purpose
meson.build	Main build configuration
meson_options.txt	Configurable build options
.gitattributes	Git repository attributes

Source Code (src/)

The `src/` directory contains the C++ backend implementation.

Key Components

1. Main Application (main.cpp)

- Parses command-line arguments
- Loads configuration file
- Initializes server components
- Starts event loop

2. Spectrum Server (spectrumserver.cpp)

- Coordinates all components
- Manages user connections
- Distributes spectrum data
- Handles user requests

3. SDR Drivers (drivers/)

- Abstract interface for SDR hardware
- Read and format sample data
- Handle device-specific features

4. DSP Engine (dsp/)

- FFT calculation (CPU/GPU accelerated)
- Demodulation (AM, FM, SSB, CW, etc.)
- Audio filtering and resampling
- AGC and noise reduction

5. Web Server (server/)

- WebSocket communication
- HTTP static file serving
- User session management
- Real-time data streaming

6. Audio Encoding (audio/)

- FLAC compression
- Opus compression
- Streaming optimization

Frontend (`frontend/`)

The web-based user interface built with Svelte and Vite.

1. Main App (`App.svelte`)

- Top-level component
- Layout structure
- Component orchestration
- **Decoders button row** — one button per decoder, on the main panel directly under the Modes selector; press to start the decoder and open its window, press again to stop. It replaced the former Bandwidth row. RADEL/RADEU live in `lib/ModesSelector.svelte` instead, beside the Modes selector and inside the Modes and Bands pop-ups.

2. Waterfall Display (`waterfall.js` + `lib/`)

- Canvas-based spectrum and waterfall rendering, colormaps and the adaptive auto-adjust, all in `waterfall.js` (plain JS, not a component)
- Interactive tuning through `lib/PassbandTuner.svelte`
- Band plan and marker overlays through `lib/FrequencyMarkers.svelte`
- The audio spectrogram is a separate component, `lib/Spectrogram.svelte`

3. Controls (`App.svelte` + `lib/`)

- Frequency input/display — `lib/FrequencyInput.svelte`
- Mode selection (AM/FM/SSB/CW) — `lib/ModesSelector.svelte`, band switching — `lib/BandSelector.svelte`
- AGC/NR/NB and the remaining controls live in `App.svelte` itself; there is no separate `Controls.svelte`

3a. Scanner (`scanner.js`)

- Channel scanner: walks the receiver across a range and stops on the first channel carrying a signal. Plain JS, not a component — `App.svelte` supplies the VFO, the mode, the band plan and the tuning call, and receives the UI state back through one callback
- Threshold is dB over the band noise floor rather than an absolute level, reusing the floor `waterfall.js` already tracks (`snrNoiseDb`), so it needs no calibration of its own
- Two ways to cross the band: tune and listen to every channel, or screen the spectrum first and only tune to what it cannot rule out. The spectrum may skip a channel but never stop on one — every stop comes from a real dwell
- Range is either the band plan's band or exactly what the waterfall shows; auto-resume, a max-stay timeout and a lockout list live here too, persisted in `localStorage`

4. Audio System (`audio.js`)

- WebSocket audio stream
- FLAC/Opus decoding. FLAC goes through `flacLowLatency.js`, which decodes each packet the moment it arrives and corrects the stream's true rate (for example 12016.3 Hz) to 12000 Hz with a 64-tap resampler (2.7 ms of delay). The WASM decoder in `modules/` released audio in fixed 1024-sample chunks and held back about 160 ms; it is still used for Opus, for diversity sources, and for FLAC when the page is opened with `?flac=wasm`
- Audio playback control
- Playback buffer control (`playoutControl.js`): `ArrivalMeter` measures how irregularly packets arrive against the stream's own timeline, and the drift between the receiver's sample clock and the listener's sound card; `CushionController` keeps the smallest buffer seen in the last minute close to the buffer preset's margin, by cutting or repeating one pitch-matched period (`spliceOut` / `spliceIn`)

of the audio going to the loudspeaker. Decoders and recordings get the untouched stream. Applies to the scheduled-buffer path that every listener on a plain-http station uses; `?drain=0` switches it off, `?diag=1` shows its figures

- Distributes raw PCM (taken before AGC, noise reduction and mute) to the mode decoders
- AI noise reduction: `lib/rnnoise.js` loads the RNNoise WebAssembly module (`@jitsi/rnnoise-wasm`) on first use; `audio.js` runs it on voice modes only, after the decoder tap, so the decoders never hear it

4a. Mode decoders and their workers

Each of the heavy mode decoders runs in its own Web Worker so that decoding never blocks audio playback or the waterfall. They follow one common pattern — three files per decoder:

Decoder	Engine	Worker	Main-thread proxy
SSTV	<code>sstv.js</code>	<code>sstv.worker.js</code>	<code>sstvWorkerProxy.js</code>
HF FAX	<code>fax.js</code>	<code>fax.worker.js</code>	<code>faxWorkerProxy.js</code>
NAVTEX + FSK/RTTY + PSK31 + Olivia	<code>fsk.js</code> , <code>psk31.js</code> , <code>olivia.js</code>	<code>fsk.worker.js</code>	<code>fskWorkerProxy.js</code>
CW	<code>cwDecoder.js</code>	<code>cw.worker.js</code>	<code>cwWorkerProxy.js</code>

- The **engine** is plain DSP code with no knowledge of workers, so it can also be run directly (unit tests, or the in-thread fallback).
- The **worker** holds one engine instance and forwards its events verbatim.
- The **proxy** mirrors the engine's method surface, so `audio.js` calls it exactly as it would call the decoder. It creates the worker lazily on first enable and falls back to running in-thread if workers are unavailable.

Two details are load-bearing: PCM is **copied** into a fresh buffer before being transferred to the worker (transferring a view into the audio accumulator would detach it and kill playback), and the worker's `init` message re-applies configuration to an already-running engine rather than assuming a fresh one.

`fsk.js` serves both NAVTEX and FSK/RTTY from one engine, selected per instance by a `role` field; each instance owns its own state, so the two can run independently.

The `fsk` role additionally hosts two decoders that are not FSK at all. Selecting the `psk31` or `olivia` variant makes `fsk.js` hand the audio to `psk31.js` or `olivia.js` instead of its own discriminator chain, while still borrowing its configuration, worker and event plumbing — so `fsk.worker.js`, `fskWorkerProxy.js` and `audio.js` need no knowledge of either mode, and the UI consumes the same `char/status/metrics` events throughout.

- `psk31.js` — BPSK31: complex baseband, matched filter, differential detection and varicode, with a spectral coarse acquisition plus a fine AFC covering about ± 25 Hz.
- `olivia.js` — Olivia MFSK: a port of Pawel Jalocho's MFSK receiver from `fldigi` (`pj_mfsk.h`, GPL-3, as is this project), including the Walsh/Hadamard FEC and the blind synchronisation search over block phase and frequency offset.
- `broadcastSchedules.js` — the UTC timetables the FAX, NAVTEX and RTTY decoders offer as presets, taken from the NOAA/NWS marine radiofacsimile schedules and the published NAVTEX station lists

4b. Receive Diversity (`diversity.js`)

- Combines the local receiver with a second one elsewhere and follows whichever site currently has the better signal. Plain JS, not a component — it sits on one seam in `audio.js`, which hands it the local PCM and plays back what it returns

- **Selection, not summing.** Two sites hear the same transmission over different ionospheric paths, so their waveforms have unrelated phase; adding them sounds comb-filtered. Coherent combining would need a common clock, which two receivers over the internet cannot share. The two streams are only ever mixed during a 30 ms crossfade
- Alignment correlates the two **audio envelopes** (log power at 100 Hz), never the waveforms — the envelope survives both the path and any codec. A lock is only trusted once a second, independent search agrees with it, which rejects the confident-but-wrong delay two antiphase-fading sites would otherwise produce
- The remote is **rate-locked** to the local stream first. Two receivers are two clocks and two decimation chains, so their audio arrives up to 2% apart even when both declare 12 kHz — 240 samples a second of creep, which no correlation can hold. The ratio is measured from how many samples each side actually delivers and applied by a resampler that carries its fractional phase across blocks, so an arbitrary ratio holds indefinitely
- Site choice uses a percentile SNR measured on **content-aligned** samples, with hysteresis, a dwell timer and a fast escape hatch for a collapsed incumbent. Levels are matched noise-to-noise, so a switch does not change the background hiss
- Decoders keep the **local** stream: FT8, JS8, WSPR and RADE integrate coherently across a slot, and a mid-slot switch is a phase discontinuity that can cost the decode

4c. Diversity sources (`remoteSource.js`, `kiwiSource.js`, `uberSource.js`, `webSdrSource.js`)

- One contract — `onPcm` / `onState` / `tune` / `canReceive` — so `diversity.js` never learns what is on the other end. Adding a receiver type is one new file
- `remoteSource.js` — another PhantomSDR-Plus over `/audio` (cbor + FLAC), reusing `createDecoder()` from `lib/wrappers.js`
- `kiwiSource.js` — a KiwiSDR: SND frames, big-endian PCM, with the 10-byte GPS timestamp a stereo packet inserts before the audio
- `uberSource.js` — an UberSDR over its native `/ws`: Opus in a 21-byte header, retuning over the open socket. The session id must be registered with `POST /connection` first and must be a UUID
- `webSdrSource.js` — a WebSDR, through `websdr_relay.py` on this server: the browser cannot connect directly, because WebSDR checks the `Origin` header and no script may change it. Tuning is a text frame on the same socket; band coverage comes from the relay
- `webSdrCodec.js` — the WebSDR audio format: a tagged byte stream whose compressed blocks drive a 20-tap leaky-LMS predictor. Ported from the WebSDR client and verified sample-for-sample against it
- `diversityList.js` — the saved-receiver list and the address rules for the four types, shared by the desktop panel and the mobile page so one format serves both. Also the transfer — a compact blob, its QR code, and a parser that accepts every shape the two pages have ever written — and `browseUrl()`, which walks a dialled address back into one a browser can open
- `lib/DiversityPanel.svelte` — the UI: endpoint, source type, SNR trim and live status, plus the saved receivers — named, reorderable, and exported to and imported from a JSON file, kept per source type in `localStorage`. Everything is edited inside the panel: `prompt()` and `confirm()` block the main thread, which is where audio frames are pushed into the playback worklet. A  **QR** button draws the list as a scannable code, because `localStorage` is per browser and a phone starts empty. `mobile/Mobile.svelte` carries the same feature in a **Div** tab, in that page's own plain CSS. See [Receive Diversity](#)

4d. Mode identifier (`modeId.js`, `modePriors.js`)

- Answers "what am I listening to?". It reads the same raw PCM tap as the decoders and ranks the likely modes, so the operator can pick the right decoder instead of trying all ten. It never decodes anything: it measures physical properties of the signal and scores them against a table of known modes

- Occupied bandwidth is the contiguous –15 dB width around the peak, deliberately not a 99 %-power figure: keying splatter puts long tails on the power integral and made every narrow mode read several times too wide. Symbol rate comes from the **instantaneous frequency** rather than tone energies, because no integration length can both resolve a 170 Hz shift and a 100 Bd symbol. Keying rate comes from the on/off envelope, and doubles as a CW speed readout
- `modePriors.js` adds the one clue the audio cannot carry — where you are tuned. A 100 Bd / 170 Hz signal on 518 kHz is NAVTEX; the identical signal on 14.070 MHz is not. It only reweights what the signal already supported, and never invents a candidate
- FT8, JS8 and FT2 are reported as one group on purpose: they are not separable on bandwidth and tone spacing alone, and pretending otherwise would be a confident wrong answer
- Below roughly 10 dB SNR it stays quiet rather than guessing
- Runs in its own Web Worker (`modeId.worker.js` + `modeIdWorkerProxy.js`), following the same engine/worker/proxy pattern as the decoders above; the result is the chip in `lib/ModeIdChip.svelte`

4e. Connection refusals (`refused.js`, `clientVersion.js`)

- `refused.js` is the shared vocabulary for a connection the server turns away: close code **4003** (over a per-IP limit, or the page is older than `[server] min_client_version`) and **4001** (a sysop kick). Both are final. `audio.js`, `waterfall.js` and `events.js` all import `isRefusal()` from it, because all three open a socket that can be refused and all three must settle their init promise when it is — otherwise the page waits forever on a connection that is never coming
- Nothing retries. A dropped `/audio` socket ends the session by design: `/waterfall` and `/events` never came back with a reconnect, so a retried session was live audio attached to a frozen waterfall, and against a rate limit each retry would extend the very refusal it was working around. The desktop page explains a refusal that happens at page load and simply stops when one happens mid-session; `/mobile` shows one line asking the listener to reload
- `clientVersion.js` holds a single integer, `CLIENT_VERSION`, which the page appends to its audio socket as `/audio?v=N`. The server refuses anything below `[server] min_client_version`, which is how a station forces tabs still running an older build to reload — the only lever there is, since the server cannot reach JavaScript already running in a browser. Bump it when a frontend change must not keep being ignored by open tabs
- Full reference: [Connection Limits](#)

4f. Chat replies (`lib/chatReply.js`)

- A reply is an ordinary chat message that starts with a marker naming the message it answers, `[RE:<timestamp>|<name>]`; the server stores and relays it like any other line and needs no change
- `chatReply.js` builds and reads that marker and groups replies under their original, one level deep. `App.svelte` and `mobile/Mobile.svelte` both use it, so the two pages thread the chat the same way
- When the original has already left the server's 20-message history, the reply stays where it arrived and shows only the name and time it answers

5. State Management (`stores/`)

- Reactive data stores
- Shared application state
- Event handling

Key Features

- Real-time audio processing
- Digital mode decoding (FT8, RTTY, etc.)
- Audio filtering

- Spectrum analysis

Frequency Lists (`frequencylist/`)

The on-waterfall frequency markers. `mymarkers.json` is the list the receiver actually shows; the rest is the raw material `update-markers.sh` turns into it, refreshed from the online schedules. `README.md` in that directory explains the update in all seven languages.

```
frequencylist/
├─ mymarkers.json           # the markers the receiver shows – hand-edited, survives an update
├─ shortwavestations.json   # shortwave broadcasters, generated by generate-current-shortwave.py
├─ 0.TXT                   # the A26 global HF schedule, unpacked and parsed by update-
markers.sh
├─ a26allx2.zip            # source archive downloaded by update-markers.sh
├─ admin.txt · antenna.txt · broadcas.txt · fmorg.txt · language.txt · site.txt
├─                          # supporting lists from the same source (sites, languages, admins,
antennas)
├─ generate-current-shortwave.py
├─ update-markers.sh       # refreshes everything above from the online schedules
└─ README.md
```

Format (`mymarkers.json`, `shortwavestations.json`)

```
[
  { "frequency": 77500,   "name": "DCF77", "mode": "CW" },
  { "frequency": 2485000, "name": "Vanuatu Broadcasting", "mode": "AM" }
]
```

Configuration Files

Server Configuration (`.toml` files)

Structure of configuration files:

```

[server]
# Web server settings
port = 9002
html_root = "frontend/dist/"
threads = 2
otherusers = 1

[limits]
# Per-IP connection limits – all default to off or to a sensible value, so a
# config without them behaves as it always did. See CONNECTION_LIMITS.md.
per_ip = 3          # simultaneous listeners from one address
per_ip_rate = 40    # new sockets per minute from one address

[websdr]
# Online registration
register_online = true
name = "WebSDR Name"
antenna = "Antenna Type"
grid_locator = "AB12cd"
hostname = "domain.com"

[input]
# SDR input settings
sps = 2048000        # Sample rate
fft_size = 131072    # FFT size
frequency = 145000000 # Base frequency
signal = "iq"         # Signal type: "iq" or "real"
audio_sps = 12000     # Audio sample rate
audio_compression = "opus" # "flac" or "opus"
accelerator = "opencl" # "none", "cuda", "opencl"

[input.driver]
# Driver settings
name = "stdin"
format = "u8"          # Sample format

[input.defaults]
# User interface defaults
frequency = 145500000
modulation = "FM"

```

Site Information (`site_information.json`)

```

{
  "siteSysop": "Operator Callsign",
  "siteSysopEmailAddress": "email@example.com",
  "siteGridSquare": "AB12cd",
  "siteCity": "City, Country",
  "siteInformation": "https://github.com/sv1btl/PhantomSDR-Plus",
  "siteHardware": "Hardware specs",
  "siteSoftware": "Software version",
  "siteReceiver": "SDR model",
  "siteAntenna": "Antenna description",
  "siteNote": "Additional notes",
  "siteIP": "http://domain.com:9002",
  "siteSDRBaseFrequency": 0,
  "siteSDRBandwidth": 2048000,
  "siteRegion": 1,
  "siteChatEnabled": true
}

```

Build System

Meson Build Configuration

meson.build (Root)

Defines:

- Project metadata
- Dependencies
- Compiler options
- Source file lists
- Build targets

meson_options.txt

Available options:

```
option('opencl', type: 'boolean', value: false, description: 'Enable OpenCL support')
option('cuda', type: 'boolean', value: false, description: 'Enable CUDA support')
option('optimization', type: 'string', value: '3', description: 'Optimization level')
```

File Dependencies

Backend Build Dependencies

```
spectrumserver binary depends on:
├─ C++ source files (src/**/*.cpp)
├─ External libraries:
│   ├── FFTW3
│   ├── WebSocket++
│   ├── FLAC
│   ├── Opus
│   ├── Liquid-DSP
│   ├── Boost
│   ├── zlib
│   ├── zstd
│   └─ OpenCL/CUDA (optional)
└─ Subproject headers:
    ├── nlohmann/json
    └─ toml11
```

Frontend Build Dependencies

```
frontend/dist/ depends on:
├─ Source files (frontend/src/**/*.*)
├─ npm packages (node_modules/):
│   ├── Svelte
│   ├── Vite
│   ├── @wasm-audio-decoders/opus-ml
│   └─ ...
└─ Static assets (frontend/public/)
```

Data Flow

Server Operation Flow

1. SDR Hardware → rtl_sdr/hackrf_transfer/etc.
↓
2. Sample Stream → stdin → spectrumserver
↓
3. spectrumserver:
 - FFT calculation (waterfall)
 - Demodulation (audio)
 - Compression (FLAC/Opus)
 ↓
4. WebSocket → Browser Client
↓
5. Browser:
 - Render waterfall
 - Decode and play audio
 - Display controls

User Interaction Flow

1. User clicks on waterfall
↓
2. JavaScript sends frequency change request
↓
3. WebSocket → spectrumserver
↓
4. spectrumserver:
 - Updates demodulator frequency
 - Sends new audio stream
 ↓
5. Browser receives and plays new audio

File Modification Guide

When you modify backend code (src/**):

```
cd PhantomSDR-Plus
meson compile -C build
# Server restart required
```

When you modify frontend code (frontend/src/**):

```
cd PhantomSDR-Plus/frontend
npm run build
cd ..
# Server restart required (for static files)
```

When you modify configuration (.toml, .json):

```
# Restart server
./stop-websdr.sh
./start-rtl.sh # (or appropriate start script)
```

When you modify markers (`markers.json`):

```
# Reload page in browser
# No server restart needed
```

Important Paths

Runtime Paths

- **Configuration:** `./config-*.toml`
- **HTML Root:** `./frontend/dist/`
- **Markers:** `./markers.json`
- **Chat History:** `./chat_history.txt`
- **FFTW Wisdom:** `./fftw_wisdom`, `./phantom_fftw_wisdom`

Build Paths

- **Binary Output:** `./build/spectrumserver`
- **Frontend Output:** `./frontend/dist/`
- **Node Modules:** `./frontend/node_modules/`

Source Paths

- **Backend Source:** `./src/`
- **Frontend Source:** `./frontend/src/`
- **DSP Libraries:** `./jsdsp/`

Common File Operations

Adding a New SDR Configuration

1. Copy existing config: `cp config-rtl.toml config-mydevice.toml`
2. Edit parameters: `nano config-mydevice.toml`
3. Create start script: `cp start-rtl.sh start-mydevice.sh`
4. Edit start script: `nano start-mydevice.sh` — change only the **RECEIVER CONFIGURATION** block at the top (`RX_LABEL`, `RX_COMM` = the receiver's process name, `RX_ARGS`, `CONFIG`, `FIFO`, and the `prestart` hook if the device needs one). The launcher/watchdog/logging logic below it is generic and needs no changes.
5. Make executable: `chmod +x start-mydevice.sh`

`stop-websdr.sh` already stops any `start-*.sh --watchdog`; if your receiver uses a process name other than `rx888_stream/rx_sdr/rtl_sdr/hackrf_transfer/cf32_to_real`, add a `killall -9 <name>` line for it there too.

Customizing the Frontend

1. Modify source: `nano frontend/src/App.svelte`
2. Rebuild: `cd frontend && npm run build && cd ..`
3. Restart server: `./stop-websdr.sh && ./start-rtl.sh`

Adding Custom Markers

1. Edit markers file: `nano markers.json`
 2. Format:

```
json { "markers": [ { "frequency": 145500000, "label": "2m Calling", "mode": "FM" } ] }
```
 3. Reload browser (no server restart needed)
-

Version Control

Files to Track in Git

- Source code (`src/`, `frontend/src/`, `jsdsp/`)
- Configuration examples (`config.example*.toml`)
- Build system (`meson.build`, `meson_options.txt`)
- Documentation (`*.md`, `docs/`)
- Scripts (`*.sh`)

Files to Ignore (`.gitignore`)

- Build output (`build/`, `frontend/dist/`)
 - Dependencies (`frontend/node_modules/`)
 - User data (`chat_history.txt`)
 - Personal configs (`config-rtl.toml` if customized)
 - Binary data (`*.o`, `*.so`)
-

This structure documentation should help you navigate and understand the PhantomSDR-Plus codebase.

For setup instructions, see [INSTALLATION.md](#). For usage information, see [USER_GUIDE.md](#). For the KiwiSDR client bridge, see [Aether_config.md](#).

73 de SV1BTL & SV2AMK