# Software Architecture for an HF FT8/JS8 Decoder System

## Introduction

We propose a lightweight, high-performance software system to **decode in-band FT8 and JS8 digital mode messages** from an HF radio (2.5–50 MHz) and log the decoded text and signal quality. The system will run on a Raspberry Pi 3B+ connected to an **RTL-SDR v4** USB dongle, capturing weak-signal HF transmissions and extracting the message content plus a robust SNR estimate for each decode. FT8 is a time-synchronized 15 s FSK digital mode that encodes 77-bit structured messages (callsigns, grids, etc.) with strong forward error correction[[1]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=other%20digital%20modes%20pioneered%20in,specifi%20cally%20for%20this%20application)[[2]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=FT4%20and%20FT8%20transmissions%20always,targeted%20purposes%2C%20we%20allocate%20three). JS8 (JS8Call) is a derivative of FT8’s waveform designed for **keyboard-to-keyboard chat** and relay networking, essentially “FT8 for conversations”[[3]](https://www.sigidwiki.com/wiki/JS8#:~:text=Location%20Worldwide%20Short%20Description%20JS8Call,Raw%20Recording%20%E2%80%94%20Audio%20Sample) – it uses the same 8-tone modulation and weak-signal robustness, but allows longer free-form text messages and networking features. The software will decode both FT8 and JS8 signals in real time, store each decoded message (and its metadata) into a local SQLite database, and provide a minimal **web-based GUI** over the LAN for monitoring the traffic and controlling the band/frequency. The entire design emphasizes **efficient DSP and decoding algorithms** (using C++ or Rust with optimized libraries) to meet the Raspberry Pi’s CPU constraints. In summary, the system will function as a headless HF digital-mode receiver: tuning the SDR, processing the signal, decoding all FT8/JS8 messages in the passband, and allowing the user to view the messages and manage the receiver remotely.

## Requirements Summary

Key requirements and features of the system include:

* **Supported Modes**: Receive and decode **FT8** and **JS8** digital modes on HF. FT8 uses 15 s time-slot transmissions (8-tone CPFSK at 6.25 Hz tone spacing) with structured 77-bit payloads and strong LDPC FEC[[1]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=other%20digital%20modes%20pioneered%20in,specifi%20cally%20for%20this%20application). JS8 uses the same FT8 modulation and time-slot structure but enables longer free-text messaging and network relays[[3]](https://www.sigidwiki.com/wiki/JS8#:~:text=Location%20Worldwide%20Short%20Description%20JS8Call,Raw%20Recording%20%E2%80%94%20Audio%20Sample). Both modes occupy ~50 Hz bandwidth signals in a ~3 kHz audio channel.
* **Radio Interface**: Use the **RTL-SDR v4** dongle (device index -d 1) as the RF source. The software must control the tuner frequency and sample IQ data. The user can **select the band/frequency** (e.g. 3.578 MHz for 80 m JS8, 7.074 MHz for 40 m FT8, etc.) via the UI. The tuning range is 2.5 MHz to 50 MHz, covering all HF bands up to 6 m. There are no additional RF front-end controls (no tunable filters or mixers aside from the SDR itself) – **SDR tuning** is the only frequency control needed.
* **Concurrent Mode Decoding**: If the SDR bandwidth permits, the system should decode **FT8 and JS8 simultaneously** on the selected band. Many bands have the JS8 frequency only ~4–5 kHz away from the FT8 frequency[[4]](https://www.sigidwiki.com/wiki/JS8#:~:text=FT8%20%20JS8Call%20160m%20,136%20%20All), so a single IQ capture can cover both. The design will attempt to decode both modes in parallel (configurable), so the user can monitor FT8 and JS8 activity concurrently on one band. (If CPU or bandwidth becomes a bottleneck, the user may opt to decode only one mode at a time via settings.)
* **Performance**: The decoder must be **optimized for low CPU usage and real-time operation** on a Raspberry Pi 3B+ (1.4 GHz quad-core ARM). Implement critical signal processing in C++ or Rust for native speed, utilize SIMD-optimized libraries (FFTW, liquid-dsp, etc.), and parallelize tasks to use multiple cores. The system should keep up with the 15 s cycle of FT8/JS8 without dropping data (i.e. decode each frame before the next one arrives).
* **Accuracy and Sensitivity**: Achieve performance comparable to WSJT-X: decode signals down to ~-20 dB SNR (in 2.5 kHz BW) and handle multiple simultaneous signals. Use robust decoding algorithms including **synchronization on FT8 Costas sync sequences**, LDPC forward error correction, and CRC checks, to reliably extract messages from noisy signals[[5]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=Tone%20patterns%20known%20as%20Costas,bit%20fi%20elds%20listed%20in)[[6]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=77,CRC%20word). Provide a **quantitative SNR estimate** for each decoded message, using the standard FT8 metric of dB relative to noise in 2500 Hz bandwidth[[7]](https://ham.stackexchange.com/questions/22477/why-is-snr-for-ft8-defined-using-the-noise-power-in-the-2500-hz-channel-containi#:~:text=But%20it%27s%20easy%20to%20change,based%20on%20625%20etc%20etc). This SNR should be computed in a robust way (e.g. measuring signal vs. background noise power during decode) to closely match the reports from WSJT-X.
* **Data Logging**: Store all decoded messages locally in a **SQLite database**. Each record will include at least: timestamp, band/frequency, mode (FT8/JS8), the decoded message text (e.g. the full FT8 message or JS8 chat line), and the measured SNR (plus potentially other metadata like time offset, audio frequency, etc. as needed). Using SQLite ensures the data is easily queryable and persists between sessions without requiring a separate database server.
* **Networking and UI**: Provide a minimal **server interface** accessible over the LAN for **monitoring and control**. This will likely be a small web application (for example, an HTTP server embedded in the decoder or a lightweight Flask web app) that serves a dashboard page. The UI should display a live table of recent decoded messages (with time, mode, SNR, etc.) and allow the user to change bands (and toggle modes) remotely. It should also indicate system status – in particular, show when the decoder is actively processing a timeslot (e.g. a “Decoding…” indicator during the few-second interval when the software is crunching data). This lets the user know the system is online and working, even if no messages are decoded in a given cycle. The web UI need not be fancy; a simple, mobile-friendly page with the essential info is sufficient, given CPU constraints.

All components should use **open-source libraries** (GPL/AGPL is acceptable for this non-commercial use) and the final software itself can be released under GPL. Next, we detail the system architecture meeting these requirements.

## System Overview

At a high level, the system consists of the following components working together: **(1) an RF Input module** to control the RTL-SDR and stream raw IQ samples, **(2) a Signal Processing & Decoder engine** that converts IQ data into decoded text messages (using sub-modules for FT8 and JS8), **(3) a Data Storage layer** using SQLite to log messages, and **(4) a Network/GUI module** providing a web-based control panel and live feed. These components are structured to operate concurrently and efficiently, as shown below:

* **RTL-SDR Input Thread** – Initializes the RTL-SDR (select device #1), sets the sampling rate and center frequency according to the chosen band, and continuously reads IQ samples from the dongle. IQ data is buffered for processing. This thread also handles band tune commands (retuning the frequency when the user selects a new band).
* **DSP & Decode Engine** – Processes buffered IQ data in 15-second frames aligned to the FT8/JS8 timing. It performs filtering, demodulation, and decoding for both FT8 and JS8 signals found in the passband. This engine may be internally split into multiple threads or tasks (e.g. one per mode or one for detection and one for FEC decoding) to exploit multiple cores. After each timeslot, it produces a list of decoded messages (with metadata like time, SNR, etc.).
* **Database Logger** – Upon each decode cycle, the results are written to the SQLite database. This could be done in the decoding thread or by a separate thread that consumes decoded message objects and inserts them into the DB asynchronously. The database uses a simple schema (e.g. a table messages) to store each message record.
* **Web Server / UI** – A lightweight HTTP server runs (either as a separate thread in C++ or as an external minimal Python server) to serve a webpage and API endpoints. The UI allows user actions (like changing band or mode) which the server relays to the appropriate components (e.g. notifying the SDR thread to retune, or enabling/disabling JS8 decoding in the engine). It also provides an endpoint to fetch recent decoded messages from the SQLite DB (or from an in-memory cache) to update the live view. Optionally, the server can push real-time updates using WebSockets or Server-Sent Events so new messages and status indicators appear without manual refresh.

This architecture cleanly separates concerns: **real-time signal capture**, **digital signal processing and decode**, **data persistence**, and **user interface**. The components interact via thread-safe queues or signals – for example, the SDR thread pushes IQ blocks to a decode queue, the decode thread pushes decoded message objects to a results queue, and the web UI thread periodically queries or is notified of new results to display. Below we describe each component in detail, including the libraries and techniques used to meet performance goals.

## RF Front-End and Tuning Control

The **RF Input module** manages the RTL-SDR dongle using the official **librtlsdr** (from the RTL-SDR open-source project). This C library (GPL licensed) provides functions to open the device, set center frequency, gain, sample rate, and read samples. We will configure the dongle for **direct sampling mode or Q-branch sampling** if needed to cover HF frequencies down to ~2.5 MHz (the RTL-SDR v4 has built-in HF capability). The typical sampling rate of an RTL-SDR is on the order of 2–3 MSPS; however, to minimize CPU load and focus on the narrow FT8/JS8 band, we will use a **lower sample rate with decimation**. For example, we might sample at 240 kS/s or 192 kS/s and then digitally down-sample to a 12 kHz bandwidth around the target frequency. FT8/JS8 signals fit in a ~3 kHz audio passband, but to decode both FT8 and JS8 concurrently (which can be ~4–5 kHz apart), we may need ~8–10 kHz bandwidth. A 12 kHz final sample rate (the same rate WSJT-X uses for its decoder) is ideal[[8]](http://www.kk5jy.net/ft8modem/#:~:text=The%20ft8modem%20program%20takes%20two,of%20available%20options%20and%20sound) – it’s high enough to capture both modes on one band and low enough for efficient FFT processing. We will likely set the SDR to a base frequency centered between the FT8 and JS8 channel frequencies (e.g. for 40 m, center ~7.076 MHz to cover 7.074 MHz FT8 and 7.078 MHz JS8 equally). The input thread will then apply a digital band-pass filter to isolate roughly a ±5 kHz band around center and decimate to 12 kHz IQ stream. We can use **Liquid-DSP** (MIT-licensed DSP library) for implementing the filters and decimator, or implement a custom FIR filter + decimation if needed. Liquid-DSP also provides utility functions for frequency translation, which could help center the passband.

**Time Synchronization**: FT8 and JS8 decoding requires knowing the start of each 15-second interval. We will ensure the Raspberry Pi system clock is synchronized via NTP. The decoder thread will align its processing with system time modulo 15 s. The SDR thread can run continuously, writing samples into a ring buffer, while the decode thread wakes up at each 15 s boundary to copy a block of samples (e.g. 15 s \* 12 kHz = 180,000 samples) for processing. This ensures we always analyze data starting at the proper epoch. Minor clock offset of a fraction of a second can be tolerated by the decoder (we can search a range of time offsets, as WSJT-X does, e.g. ±2 s). The system will log the nominal UTC timestamp of each frame for reference.

**Band Switching**: The UI will provide a list of band presets (e.g. 80 m JS8, 40 m FT8, etc., or just 80 m, 40 m, 20 m etc. with assumption to cover both modes). When the user selects a band, the web server triggers the SDR thread to retune: it calls the librtlsdr API to set the new frequency (and possibly adjusts gain if different bands require it). We maintain an internal map of band names to frequencies for FT8/JS8 (based on standard dial frequencies[[4]](https://www.sigidwiki.com/wiki/JS8#:~:text=FT8%20%20JS8Call%20160m%20,136%20%20All)). For example: *40 m* might map to 7.074 MHz center (to get FT8 at 7.074 and JS8 at 7.078 MHz), *20 m* to 14.076 MHz, etc. The system could also allow direct frequency entry if custom frequencies are desired. Tuning happens near-instantly; the decode engine will detect the change and likely reset its sync (clearing any old buffered data). We ensure a smooth transition by flushing buffers and reinitializing the next 15 s cycle after a band change.

Because the RTL-SDR provides raw samples, no other hardware tuning is needed (the **SDR’s internal tuner** covers the RF and the software covers the rest). We will, however, implement **automatic gain control (AGC)** or allow manual gain setting to optimize dynamic range. This can be done via librtlsdr (setting tuner gain or enabling AGC mode). Given we are dealing with weak signals, we may set a fixed moderate gain and rely on 16-bit DSP processing to maintain precision.

In summary, this module ensures we always have a stream of properly centered and filtered baseband samples ready for the decoder. It abstracts the hardware so that upper layers just request “band = 20 m” and the RF module handles the rest.

**Libraries/Tools**: librtlsdr for SDR control; liquid-dsp or similar for filtering/decimation; possibly FFT routines (though main FFT usage is in decoder stage).

## Signal Processing and Decoder Engine

This is the core of the system: it takes 12 kHz complex baseband samples for each 15 s interval and produces decoded text messages. We break the decoder engine into a pipeline of stages: **frequency-domain analysis**, **signal detection & synchronization**, **demodulation and decoding**, and **error-check/SNR measurement**. We will implement two decoding paths – one for **FT8** and one for **JS8** – which share many underlying DSP steps but differ in message format handling.

**1. Frequency-Domain Processing & Sync:** After receiving a 15 s block of samples, the first task is to find the presence of any FT8/JS8 signals and align to them in time and frequency. Both FT8 and JS8 use structured synchronization signals called **Costas arrays** embedded in the transmission. For FT8, a 7-tone Costas sequence (tone pattern 3,1,4,0,6,5,2) is sent at the start, middle, and end of the 15 s frame[[5]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=Tone%20patterns%20known%20as%20Costas,bit%20fi%20elds%20listed%20in), allowing the decoder to detect these as unmistakable markers. JS8 being derivative likely uses a similar sync (possibly also FT8’s Costas at start of each frame). We leverage this: the decoder performs a series of FFTs on the time block to detect the 7-tone pattern. Concretely, we can perform a **frequency vs. time search**:  
- Compute the power spectrum over the 15 s window. One approach is doing a short-term FFT (e.g. 1920-point FFTs which correspond to 0.16 s segments, since FT8 symbols are 0.16 s each) sliding through the frame to create a time-frequency “waterfall”. We will effectively replicate the WSJT-X sync detection algorithm: correlating the received spectral data with the expected Costas tone pattern across time[[9]](http://www.kk5jy.net/ft8modem/#:~:text=The%20decode%20,when%20each%20transmission%20actually%20starts). High correlation peaks identify a candidate signal, with specific start time offset (within the 15 s) and frequency offset (audio frequency within the passband). For each detected candidate, we also get an initial signal strength estimate.

This process might find multiple concurrent signals (e.g. dozens of FT8 signals can be active in one band). The decoder will collect all candidates above a certain SNR threshold. We will likely implement this using an optimized FFT library like **FFTW** (GPL, very fast on ARM with NEON support) or **kissFFT** (lightweight). The computation involves many FFTs but it’s feasible on a Pi (WSJT-X can decode ~50 signals per pass on a Pi 3B+ with optimized code). We will also account for slight frequency drift or time drift: the algorithm can try small shifts to maximize correlation.

**2. Demodulation and Bit Extraction:** Once a candidate is detected, the next step is to demodulate the 8-tone frequency-shift keyed signal into a sequence of symbol bits. FT8 uses **8-tone continuous-phase FSK** where each symbol carries 3 bits (tone 0–7 corresponds to a 3-bit Gray-coded value)[[10]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=FT8%20messages%20are%20transmitted%20using,only%20one%20bit%20position%2C%20thereby). There are 79 symbol intervals total in FT8’s 15 s (including sync symbols)[[11]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=of%20the%20information%20symbols%20by,%EF%80%BD%20%EF%81%BB%20%EF%81%BD%203%2C2%2C0%2C1). We will use the detection info to mix the signal to baseband (if the signal is at e.g. 1200 Hz audio, we shift that down to 0 Hz) and then sample the symbol intervals. By doing another set of FFTs or Goertzel filters aligned to each symbol period, we can determine which of the 8 tones was transmitted for each symbol. Essentially, for each symbol interval, we find the strongest tone bin (0–7) – this yields the raw symbol sequence. We discard the known Costas symbol slots (they are used for sync, not part of the message bits) and concatenate the rest. This gives us the sequence of 58 data symbols (for FT8) which correspond to 174 transmitted bits[[10]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=FT8%20messages%20are%20transmitted%20using,only%20one%20bit%20position%2C%20thereby)[[5]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=Tone%20patterns%20known%20as%20Costas,bit%20fi%20elds%20listed%20in). JS8 will be similar if it also uses 8-FSK and the same symbol length; likely JS8 also has 79 symbol frames (or possibly longer if they use extended periods for more text – but since JS8Call “normal” speed is the same 15 s period, we assume 79 symbols as well). We’ll treat JS8 similarly in demodulation.

Throughout demodulation, we refine time and frequency offsets for better accuracy – e.g., the initial sync gives a coarse alignment, then we can adjust the timing to maximize tone SNR, and correct any frequency offset (Doppler or mis-tuning) by tracking phase. Since FT8 is CPFSK, phase continuity can be used to aid detection (but a non-coherent approach with magnitude FFTs is simpler and usually sufficient given the strong coding).

**3. Forward Error Correction (FEC) Decoding:** After demodulating, we obtain a sequence of bits for each message candidate. Due to noise, there will be bit errors; we rely on FT8’s powerful error-correcting code to recover the true message. FT8 (and JS8 if inherited) uses a **Low-Density Parity-Check (LDPC) code** with parameters (174,91): 91 data bits (77 payload + 14 CRC) are expanded to 174 coded bits[[6]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=77,CRC%20word)[[10]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=FT8%20messages%20are%20transmitted%20using,only%20one%20bit%20position%2C%20thereby). Our decoder will implement an LDPC decoder for this code. We can use the same parity-check matrix as WSJT-X (it’s known from their source) and run an iterative belief-propagation decoder. This is computationally the most intensive step, but it’s necessary for sensitivity. We will likely integrate an existing **LDPC decoding library or code** from WSJT-X. For example, we might use the open-source FT8 decoder library *ft8\_lib* (by YL3JG) which includes an LDPC decoder optimized for FT8, or borrow code from *rtmrtm/ft8mon* which implements the FT8 LDPC in C[[12]](https://git.jerryxiao.cc/rpitx/ft8_lib#:~:text=Decoding%20is%20is%20still%20work,of%20RAM%20for%20that%20purpose)[[13]](https://git.jerryxiao.cc/rpitx/ft8_lib#:~:text=Thanks%20to%20Robert%20Morris%2C%20AB1HL%2C,various%20parts%20of%20the%20code). These libraries are GPL-compatible and can be linked. The LDPC decoder will output the best guess at the 91-bit message+CRC. We then compute the CRC (14-bit CRC as specified by FT8 protocol[[14]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=77,matrix%20used%20to%20compute%20the)) to verify if the decode is correct. Only messages with a valid CRC are accepted as valid decodes (this greatly reduces false decodes). If the CRC fails, we can optionally attempt “ordered-statistics decoding” (a technique to try flipping some bits and re-running LDPC, which WSJT-X does to decode marginal signals). Depending on CPU budget, we may implement a limited version of this to catch a few extra messages.

For JS8, it likely also uses LDPC (given it’s derived from FT8’s physical layer). However, JS8’s message might be longer than 77 bits – it might split a long text message across multiple transmissions. Each single transmission could still use the standard FT8 77-bit+CRC+LDPC encoding (perhaps sending 13 characters at a time, since FT8 free text supports 13-char message[[15]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=FT4%20and%20FT8%20are%20digital,radio%20activity%20on%20the%20high)[[2]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=FT4%20and%20FT8%20transmissions%20always,targeted%20purposes%2C%20we%20allocate%20three)). We will assume each JS8 15 s frame is decodable with the same process: demodulate 8-FSK, run the LDPC decoder. It should yield 77 payload bits (which for JS8 are not callsigns and reports, but part of a longer text). The output might look like a fragment of text with a sequence or message ID. To truly reconstruct multi-frame JS8 messages, we would need to assemble successive frames from the same sender. This is a higher-layer task that JS8Call does internally. In our architecture, as a first cut, we will **log each frame as its own message**. Since JS8 transmissions are conversational, often a single 15 s transmit *is* a complete sentence or phrase within a chat. Our system will capture those. (In future, we could enhance it to reassemble multi-frame messages by tracking station IDs and sequence numbers, but that’s beyond the minimal scope.)

**4. Message Interpretation:** Once we have the 77-bit user payload (for FT8) or the corresponding bits for JS8, we translate it to human-readable text. FT8 has a defined source encoding: most messages are “structured” (e.g. *CQ K1ABC FN20* or *K1ABC K2XYZ -07*). These are encoded in the 77 bits using compression (callsign hashing, grid encoding, etc.)[[2]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=FT4%20and%20FT8%20transmissions%20always,targeted%20purposes%2C%20we%20allocate%20three)[[16]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=FT4%20and%20FT8%20are%20digital,popularity%2C%20by%20some%20measures%20soon). We will utilize reference code or libraries to decode these bits into text. For example, the WSJT-X code or ft8\_lib includes routines to map 77-bit payloads to text (taking into account message type bits). We will incorporate those. That yields a string like "CQ K1ABC FN20" or "K1ABC K2XYZ R-07". FT8 also allows a **free text** message up to 13 characters[[16]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=FT4%20and%20FT8%20are%20digital,popularity%2C%20by%20some%20measures%20soon)[[17]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=c28%20Standard%20callsign%2C%20CQ%2C%20DE%2C,h12%20Hashed%20callsign%2C%2012%20bits), which the library can decode (these are indicated by a message type bit and then 13 characters from a limited alphabet). We’ll ensure free text is handled so that things like “HELLO WORLD” are output properly if ever sent in FT8. For JS8, since it is basically *all* free text, we might leverage the same free-text decoder but note that JS8 expands the character set (likely full ASCII or a larger set than FT8’s limited alphabet). We’ll need to confirm JS8’s character encoding – if it’s different, we may implement a custom translation for JS8 frames. (If JS8 uses multiple frames, possibly each frame carries a part of the text; but since we log frame by frame, we’ll just output whatever text is in that frame.)

**5. SNR and Metadata Calculation:** For each decoded message, we calculate an SNR value (and optionally the frequency offset and timing offset). SNR will be reported in **dB referenced to a 2.5 kHz noise bandwidth**, as is conventional for FT8[[7]](https://ham.stackexchange.com/questions/22477/why-is-snr-for-ft8-defined-using-the-noise-power-in-the-2500-hz-channel-containi#:~:text=But%20it%27s%20easy%20to%20change,based%20on%20625%20etc%20etc). We will compute this by measuring the signal power vs. background noise in the passband. A practical method: during the sync detection stage, we obtain an initial SNR estimate (correlation amplitude vs. average background). We can refine it by, say, taking the demodulated signal’s amplitude and comparing it to the baseline spectrum level away from the tones. The WSJT-X methodology is to take the ratio of signal power in the tone bin to the noise power in a 50 Hz bin, then adjust to 2500 Hz bandwidth (which is a +17 dB correction since 50 Hz vs 2500 Hz)[[18]](https://news.ycombinator.com/item?id=22830504#:~:text=The%20indicated%20FT8%20signal%20to,1%20dB%20in%2050)[[19]](https://ham.stackexchange.com/questions/22477/why-is-snr-for-ft8-defined-using-the-noise-power-in-the-2500-hz-channel-containi#:~:text=Matter%20of%20opinion%2C%20but%20it%27s,have%20a%20different%20occupied%20bandwidth). We will emulate this approach to produce an SNR in dB. For example, a very weak decode might show -20 dB (meaning 20 dB below noise in 2.5 kHz), whereas a strong signal might be +5 dB. This SNR is stored and displayed with the message. We will also note the **audio frequency** of the signal (e.g. 1500 Hz within the passband) which can be useful to the user. The **time offset** (difference between actual signal start and the expected 15.0 s boundary) can also be measured (FT8 often reports e.g. “DT 0.3 s”). These details are not explicitly required by the user but are trivial to extract, so we may include them in the database for completeness. At minimum, SNR is required and will be provided in the output.

**Multi-Signal Decoding:** The decoder engine will decode **all signals in the band** each cycle, not just one. This means after the initial detection, it might identify, say, 10 FT8 signals and 2 JS8 signals in one 15-second frame. We will loop through candidates and attempt decode for each. To optimize, candidates can be sorted by strength and decoded strongest-first. If a decode is successful (CRC OK), we could optionally **subtract** that signal’s contribution from the FFT (to help decode weaker overlapping signals). WSJT-X does something similar (it subtracts discovered signals to reduce interference). We can include an interference cancellation step if needed for crowded band performance, though on a Pi we must be mindful of the processing cost. At least, the architecture supports decoding many signals – our data structures will hold a list of decodes per interval. The concurrency can be exploited by using multiple threads for FEC decoding: LDPC decoding of each message can be done in parallel (since each candidate decode is independent). On a 4-core Pi, we might decode 2–4 signals in parallel threads to speed up the cycle. This is an optional optimization we’ll consider if necessary to complete decoding within the 15 s window. Alternatively, a vectorized single-thread approach might suffice given typical number of signals.

We will integrate existing **open-source implementations** where possible: for instance, *ft8mon* (by Robert Morris) is a C++ console FT8 decoder using FFTW and has demonstrated decoding multiple signals on Linux[[20]](https://github.com/rtmrtmrtmrtm/ft8mon#:~:text=You%20should%20see%20output%20like,this). We can draw on its code for the signal detection and decoding logic (it’s MIT licensed). Similarly, **WSJT-X’s source (in Fortran/C++)** is available under GPL and could be referenced, but directly integrating WSJT’s Fortran decoder into our program via a C wrapper (as some have done[[21]](https://www.hydrogen18.com/blog/calling-wsjtx-encoding-decoding-from-c.html#:~:text=Unfortunately%20for%20myself%2C%20decoding%20is,call%20a%20Fortran%20subroutine%20called)[[22]](https://www.hydrogen18.com/blog/calling-wsjtx-encoding-decoding-from-c.html#:~:text=call%20my_ft8,napwid%2Cmycall%2Chiscall)) might be too complex. Instead, we lean on re-implementing the algorithms in C++ using references like the QEX paper and existing smaller libraries. For JS8, since no standalone decoder library exists, we’ll use the **JS8Call open-source code** (GPL-3.0) as a reference for how to interpret JS8 messages. JS8Call’s code (C++ with Qt) contains a JS8.cpp and related logic for message parsing. We won’t run the whole JS8Call GUI, but we can reuse the bits of logic that map demodulated bits to text (ensuring compliance with its license in our non-commercial context).

Summing up, the decoder engine will produce a list of decoded messages every 15 seconds. Each message entry includes: **Timestamp**, **Mode** (FT8/JS8), **Text** (e.g. "CQ K1ABC FN20" or "KN4XYZ: HELLO HOW ARE YOU" for JS8), **SNR (dB)**, and possibly frequency (audio Hz) and/or sending station callsign fields if we choose to break them out. These results are then handed off to be logged and displayed.

## Data Storage and SQLite Database Design

For persistent storage of decoded messages, we use **SQLite**, a serverless SQL database that is lightweight and fast enough for our needs. The Pi will be writing a relatively low volume of data (at most a few dozen decodes every 15 seconds, which SQLite can easily handle). We choose SQLite to avoid the overhead of a separate DB process and because it allows the user to easily query or export the data later (using standard SQL or tools like sqlite3). The database file will reside on the Raspberry Pi’s filesystem (likely on the SD card; we should be mindful of excessive writes, but the volume here is small and we can periodically vacuum or rotate the DB if it grows large).

**Schema**: We define a table (e.g. messages) with columns capturing the decode information. For example:

messages(  
 id INTEGER PRIMARY KEY AUTOINCREMENT,  
 timestamp DATETIME, -- UTC timestamp of the start of the message/frame  
 band TEXT, -- e.g. "40m"  
 frequency REAL, -- dial frequency in MHz (or audio frequency in Hz)  
 mode TEXT, -- "FT8" or "JS8"  
 snr\_db REAL, -- SNR in dB (relative to 2.5 kHz noise BW)  
 message TEXT -- decoded text content of the message  
);

We can derive band from frequency or store it explicitly. frequency could be stored as the dial freq (e.g. 7.074 MHz) or the audio offset (in Hz) of the signal. Storing the dial frequency of the receiver plus the audio offset might be useful to reconstruct exact signal frequency: e.g. 7078000 Hz for a JS8 on 40m. For simplicity, we might log just the band or base frequency and the audio offset separately. The **message text** is the main content – this will include whatever the decoded message is (callsigns and report for FT8, or the actual chat line in JS8). Since FT8 messages often have structured components (two callsigns, grid, etc.), we could also parse and store fields (like callsign\_tx, callsign\_rx, grid, report) in separate columns for advanced analysis. However, the user’s request focuses on text content and SNR, so we can leave the message as one text string exactly as decoded. This preserves all info and is simplest.

Upon each decode cycle, the system will insert new rows into this table. This can be done via the SQLite C API (if our main code is C/C++) – linking against libsqlite3.so and using parameterized INSERT statements. We should perform inserts in batches if possible (e.g. one transaction per cycle for all decodes) to minimize I/O overhead. If 10 messages are decoded in one go, we wrap those inserts in a single transaction. SQLite on the Pi can easily handle tens of writes per minute.

We will also consider the database access by the web server: The server might query recent messages to display. We can index the table on timestamp to facilitate queries like “last N messages” or by band/mode. Given the potentially large number of decodes over time (could be thousands per day), we might implement a **pruning strategy** or simply let the DB grow and advise the user to archive data occasionally. Since this is non-commercial and local, we assume disk space is sufficient for our purposes (text records are small).

**Thread Safety**: Our design likely has the decode thread performing inserts. SQLite can handle multiple readers/writers but in our single-process, we can also let the web server thread query the DB. SQLite uses file locking to serialize access. To avoid contention, one simple approach is to funnel all DB operations to a single thread (e.g. the decode thread does inserts, and the web UI only reads). Reads can be done on a connection safely even while inserts occur, thanks to SQLite’s locking (the UI might see data up to the last commit). We may use a **mutex** around DB operations if needed to ensure consistency in our program.

In summary, the SQLite storage ensures that even if the program or Pi restarts, past decoded messages are saved. The database can also be used for further data analysis (e.g. the user could later analyze what stations were heard, etc., though that’s outside current scope). Our architecture focuses on writing and reading this data seamlessly during runtime.

## Web Server and User Interface

For user interaction, we implement a **minimal web-based GUI** accessible from any device on the LAN (desktop browser, smartphone, etc.). This avoids running a heavy local GUI on the Pi (which could consume CPU/GPU) and leverages the fact that most users can connect via network. The web interface provides two main functions: **control** (band/mode selection) and **monitoring** (displaying decoded message log and status). We aim to keep this interface simple and low-bandwidth.

**Web Server Implementation**: We have a few options: we could write the server in the same C++ program using an embedded HTTP library (like Mongoose/CivetWeb – MIT licensed – or cpp-httplib). This would avoid needing a separate process and can be very efficient. Alternatively, since the decoding is self-contained, we could run a small Python Flask app that queries the SQLite DB and calls a control script to change band. However, to minimize complexity, **embedding the server** in the main program (C++ or Rust) is preferred. For example, using **CivetWeb**, we can serve static files (HTML/JS/CSS) and handle a couple of API endpoints via callback functions. The overhead is minimal.

We will designate a port (say http://<pi-address>:8080) for the interface. No authentication is planned (assuming a trusted LAN). If needed, we could implement basic auth or an API token, but given this is a private monitoring tool, it may not be necessary.

**UI Pages/Endpoints**: - **Dashboard page** (served at “/”): An HTML page with a simple JavaScript that periodically fetches new data. The page will include controls: a dropdown or buttons for bands (e.g. 80m, 40m, 30m, 20m, etc.) and possibly checkboxes or toggles for “Decode FT8” and “Decode JS8”. When the user changes the band selection, an AJAX request (e.g. GET /setBand?band=40m) is sent to the server. The server handler then calls the RF module to re-tune and updates internal state (current band = 40m). Similarly, if toggling modes, we could have /setMode?ft8=on&js8=on for example, which the server uses to enable/disable JS8 decoding in the software (this might set a flag the decode thread checks to decide whether to run the JS8 path, saving CPU if not needed). The page will also show status like “Current Band: 40m, Mode: FT8+JS8” etc.  
- **Message feed**: We’ll have an endpoint like /messages that returns recent decodes in JSON (or even as an SSE stream). For example, a GET request might return a JSON array of the last N messages with fields (time, mode, snr, text). The front-end JavaScript can poll this every few seconds or use a streaming approach to append new messages to a table. We could also implement this as a WebSocket that pushes new decodes to connected clients as soon as they arrive (the decode thread can notify the server code on each cycle completion). Given the 15 s cycle, a simple poll every 5 or 10 seconds is adequate and easier to implement.  
- **Decode status**: The UI should indicate when a decode cycle is in progress. Since FT8/JS8 are time-slot based, we know that e.g. at second 0,15,30,45 of each minute, the system is receiving signals, and a couple seconds after each interval it will be decoding. We can provide a visual cue like a spinner or a “Decoding…” text that appears during the decoding computation. Implementation-wise, the decode thread can set a flag or call a small function in the server module when it starts and finishes a decode pass. The client could hit an endpoint like /status returning {"decoding": true} or false. But an even simpler method: because the cycles are fixed, the client can infer decoding is happening during e.g. second 15–18 of each minute. However, to be robust to any slight delays, a status endpoint updated by the decoder is fine. We will likely implement a small shared state variable for decoder status that the web server can query. This way, if the decoder gets bogged down, the UI will show “Decoding…” continuously (alerting the user that the Pi is struggling or currently busy).

**GUI Design**: The page will list messages in reverse chronological order (newest on top). Each entry shows time (possibly truncated to HH:MM:SS), mode, SNR, and message text. We might color-code modes (FT8 vs JS8) or have an icon. For JS8 messages that are long or multipart, we’ll just show them as received. We will implement basic CSS to make it readable (e.g. a monospaced font or table for alignment). The interface should be kept minimal to avoid heavy lifting by the Pi – no huge frameworks; simple vanilla JS or a lightweight library like Alpine.js is fine.

**Concurrent Connections**: Typically only one or a few users (maybe the operator on their PC/phone) will access the page, so load is negligible. Even if left open, the polling every few seconds and a few KB of JSON is fine. The server will handle requests in a separate thread (or threads) and must be thread-safe with respect to the rest of the system. If we embed the server in C++, we will use locking when reading from the SQLite or shared data. We should ensure that the server doesn’t block the decoder thread for long – SQLite reads are fast, but to be safe, we could maintain an in-memory ring buffer of last N messages (updated by decode thread), which the server simply reads and outputs as JSON. This avoids hitting the DB for every UI update and decouples the UI from the DB slightly. We can decide on this optimization if needed.

In essence, the web UI provides a **real-time view** of what the decoder is hearing on the band. It replicates the basic display one would see in WSJT-X (a list of decodes with time and SNR)[[20]](https://github.com/rtmrtmrtmrtm/ft8mon#:~:text=You%20should%20see%20output%20like,this), but via a browser. It also gives a convenient way to switch bands remotely (rather than needing SSH or physical access to the Pi).

If a GUI on the Pi itself were desired (the user did say “GUI preferable, but CPU usage dictates which”), one could run a small Qt or curses UI. However, given CPU concerns and the “over LAN” requirement, the web GUI is the best choice. It offloads rendering to the client device and keeps the Pi’s job to serving text data. Should CPU become an issue, the UI polling interval can be lowered or the UI closed when not needed, ensuring the decoder has maximum resources.

## Performance Considerations and Optimizations

We recognize that real-time DSP and decoding on an RPi3 is non-trivial, but by leveraging efficient languages and libraries, it is achievable. Here are key optimizations in our design:

* **Compiled, native code**: The heavy-lifting parts (SDR sampling, FFT, decoding algorithms) are in C/C++ or Rust, which compile to native ARM code. Python is only used minimally (if at all) for glue; the critical path avoids interpreted languages. This ensures we use the Pi’s CPU and NEON SIMD effectively.
* **NEON / SIMD usage**: Libraries like FFTW automatically use NEON optimizations on ARM for FFTs. We will compile with appropriate flags to enable this. For our own signal loops, if using C++, we can use compiler auto-vectorization or even intrinsics for things like vector multiply, etc., especially in the filter/decimation stage. The LDPC decoder can also benefit from vectorization (some implementations use bit-level SIMD for parity checks). We will see if available libraries already do this.
* **Parallel threads**: The Pi 3B+ has 4 cores. Our architecture uses concurrency to avoid any single thread doing all the work. For example, the *SDR input thread* runs continuously but is mostly I/O waiting (which can run on one core), the *decode thread* (or threads) use another core or two to perform math, and the *web server* can run on a third core when a request comes in. This way, reading samples, decoding, and serving UI don’t block each other. Particularly, offloading FFT and LDPC decoding tasks to multiple threads (or dividing candidates among threads) can shorten the decode time per cycle. We must be careful to not spawn too many threads and overhead; a thread pool of a few workers for FEC decoding could be ideal.
* **Efficient algorithms**: We will choose algorithms known to be efficient for this application. The K9AN/Williams FT8 decoder (used in WSJT-X) is highly optimized. We mimic its approach: coherent detection using the Costas sync (which reduces the search space significantly), and the use of fast FFT convolution for finding signals. The LDPC decoding will be the most iterative part – we will set a reasonable iteration limit (e.g. 50 iterations) to balance decoding success vs. CPU time. Most decodes converge well before that. Also, we might implement early cutoff: once the CRC checks out, stop iterating further.
* **Resource management**: We ensure the program doesn’t use excessive memory. 15 s of 12 kHz IQ samples is only ~3.6 million samples; at 16-bit that’s ~7.2 MB of data – manageable. FFTW will allocate some buffers, LDPC matrices are a few KB. So memory is fine on a 1 GB Pi. We will also free or reuse buffers when possible (e.g. reuse FFT plan across cycles).

In case of extremely busy band conditions (e.g. 50+ signals every interval), the CPU might approach its limits. The user can mitigate this by disabling JS8 decoding if not needed (halving the work), or by focusing on one band at a time (which is our design anyway). We will include a **configuration file or constants** where one can adjust things like maximum number of decodes to attempt per cycle or enable/disable features (like the subtract-after-decode or deep search) to trade off CPU vs completeness.

## Repository Structure and Components

We plan a clear project structure for maintainability. Below is a sketch of the repository layout and module organization:

hf-decoder/  
├── CMakeLists.txt # build script (assuming CMake for C++ project)  
├── src/  
│ ├── main.cpp # program entry, initialization  
│ ├── config.hpp/.cpp # configuration (band freq table, etc.)  
│ ├── sdr/  
│ │ ├── sdr\_reader.cpp # handles RTL-SDR input thread  
│ │ └── sdr\_reader.hpp  
│ ├── dsp/  
│ │ ├── decimator.cpp # filtering & downsampling  
│ │ ├── decoder\_ft8.cpp # FT8 decode routines (sync, FFT, LDPC, message unpack)  
│ │ ├── decoder\_js8.cpp # JS8 decode routines (could reuse FT8 lower-level functions)  
│ │ ├── ldpc\_decoder.cpp # LDPC implementation (or integrate from library)  
│ │ ├── fft\_util.cpp # wrappers for FFTW/kissFFT calls  
│ │ └── dsp\_common.hpp # common definitions (constants, lookup tables, etc.)  
│ ├── net/  
│ │ ├── web\_server.cpp # HTTP server and request handlers  
│ │ ├── web\_server.hpp  
│ │ └── static/ # static web content  
│ │ ├── index.html # Dashboard HTML  
│ │ ├── script.js # Client-side JS for AJAX and UI updates  
│ │ └── styles.css # Basic styling for UI  
│ ├── db/  
│ │ ├── database.cpp # SQLite wrapper (opening DB, insert/query functions)  
│ │ └── database.hpp  
│ └── util/  
│ ├── logging.cpp # simple logging utilities  
│ └── thread\_pool.hpp # (if implementing a small thread pool for decoding tasks)  
├── include/ # headers for external use (if any)  
├── README.md # usage instructions  
└── docs/  
 └── protocol\_spec.pdf # reference docs like FT8 protocol (for developer reference)

In this structure, we separate logically: sdr for hardware interface, dsp for signal processing and decoding logic, net for network/UI, db for database handling, and util for common utilities. The main program (main.cpp) would parse any command-line args (e.g. to run headless vs with some debug UI, or to select a different output), load config (like which device index, etc.), initialize the SQLite database (creating tables if not exists), spawn the SDR thread, start the web server thread, and then enter the decoding loop.

We will use a publisher-subscriber or observer pattern for communication between threads: e.g., the decoder can publish new message events, which the web server can either poll or be notified of. Possibly using condition variables or simple flags for synchronization. The database.cpp can provide an interface like logMessage(Message m) which handles the SQL insert. The web\_server.cpp might use the SQLite API to SELECT recent messages when serving /messages endpoint. (Alternatively, maintain an in-memory ring of the last, say, 100 messages for quick retrieval – which can be updated alongside the DB insert.)

**Dependencies**: We will document in README how to install needed libraries on Pi (e.g. sudo apt-get install librtlsdr-dev libfftw3-dev libsqlite3-dev). If we use Rust, then crates (rtlsdr, rustfft, etc.) will be in Cargo.toml. In C++, linking to these libs in CMake.

## Libraries and Tools Selection

Summarizing the key libraries used (all open-source compatible):

* **librtlsdr** – for RTL-SDR control and IQ sample streaming (GPL2).
* **FFTW3** – for fast FFT operations in signal detection (GPL2 or FFTW’s license). This provides optimized Fourier transforms[[8]](http://www.kk5jy.net/ft8modem/#:~:text=The%20ft8modem%20program%20takes%20two,of%20available%20options%20and%20sound) needed for the spectrum analysis. We’ll plan fallback to KissFFT (BSD) if FFTW is not available, but FFTW likely is fine on Pi.
* **Liquid-DSP** – for DSP utilities like filtering, resampling, and potentially demodulation helpers (MIT). Liquid-DSP can design FIR filters for decimation and may have an optimized resampler we can use[[23]](https://github.com/jgaeddert/liquid-dsp#:~:text=jgaeddert%2Fliquid,on%20your%20build%20platform). It also has some FSK demodulator functions we might examine.
* **LDPC decoder library or code** – since FT8’s LDPC(174,91) is unique, we can use existing code from WSJT-X (which we can include under GPL). The *ft8\_lib* project by YL3JG provides FT8 encode/decode routines including LDPC, which we can integrate[[24]](https://git.jerryxiao.cc/rpitx/ft8_lib#:~:text=A%20C%2B%2B%20implementation%20of%20FT8,for%20experimental%20use%20on%20microcontrollers)[[13]](https://git.jerryxiao.cc/rpitx/ft8_lib#:~:text=Thanks%20to%20Robert%20Morris%2C%20AB1HL%2C,various%20parts%20of%20the%20code). Alternatively, porting the FT8 decoder from WSJT (Fortran) to C++ has been done by others (e.g. in the **ft8modem** project by KK5JY[[25]](http://www.kk5jy.net/ft8modem/#:~:text=implemented%20by%20other%20tools,you%20want%20them%20to%20work)). We might use ft8modem as inspiration – it effectively wraps WSJT-X’s decoding in a CLI. For our internal integration, a pure C++ approach is preferred to avoid spinning up separate processes.
* **SQLite3** – for the database (public domain). We will link against the system’s SQLite library.
* **HTTP server library** – likely **CivetWeb** (which is MIT licensed, a fork of Mongoose). This single library can handle HTTP requests easily. Alternatively, we could use **Boost.Beast** (header-only via Boost) for a more C++ approach to HTTP, but that might be heavier. CivetWeb will let us set up routes for /setBand and so on quickly. If using Python Flask as an external server, then Python’s sqlite3 module and something like pySerial to communicate band changes could be used, but again, we’ll avoid multi-process for now.
* **Logging** – we might integrate a very simple logging library or just use stdout for console logs. On a Pi running headless, it’s useful to log events (like “Tuned to 14.074 MHz, decoding started, decoded 5 messages at 12:00:15,” etc.). This can help debugging and let advanced users tail the log.

The project being GPL-compatible means we can mix GPLv3 code (from WSJT-X, JS8Call) if needed. We will attribute and include their licenses accordingly.

## Additional Considerations

* **Time Sync**: We must ensure the Pi’s clock is accurate to within ~±1 s for best results (FT8 tolerates a couple seconds error). We will recommend the user to enable internet time sync (NTP) or even use a GPS time source if available. The decoder can measure any time offset of signals and display it (helpful to see if our clock is off, by seeing all signals have e.g. +0.5 s DT).
* **GPS or External Clock (Optional)**: In some setups, users use a GPSDO for time/freq. Our design doesn’t explicitly include that, but we could easily add a feature to accept an external 1 PPS for sync if needed. Not in scope unless user asks.
* **Error Handling**: The software should handle cases like no SDR present (error message), or SDR reading errors (perhaps try to re-init), and database errors (log and continue). Also, if the band is quiet and no messages decoded, the system should simply show nothing new – essentially idle. That’s expected behavior (no false decodes ideally due to CRC).
* **Testing and Calibration**: We will test the decoder against known signals. For example, use test audio recordings of FT8 (or even generate an artificial FT8 signal via known tools[[26]](https://www.hydrogen18.com/blog/calling-wsjtx-encoding-decoding-from-c.html#:~:text=Calling%20WSJT,The%20normal%20way)) to verify our decode pipeline. The reference FT8 QEX paper provides performance expectations – e.g. decode probability vs SNR curves – we won’t exactly measure that, but we aim qualitatively to decode at similar thresholds.
* **Extensibility**: The architecture is modular. In the future, support for other WSJT-X modes (JT65, FT4, WSPR, etc.) could be added by plugging into the DSP pipeline and adding new decoders. The web UI could also be expanded to show, say, a rudimentary waterfall graph or statistics (though plotting a waterfall in browser might be too heavy for Pi, so we skip that now). We focus on textual message logging as required.

## Conclusion

This design meets the requirements by combining proven DSP techniques with a lightweight server for user interaction. The **Raspberry Pi 3B+** will continuously monitor the HF band, using optimized C++/Rust code to decode FT8 and JS8 messages in real-time. Each decoded message (with its SNR) is stored in an on-device SQLite database and is accessible on a web-based dashboard. The user can easily change bands (covering 2.5–50 MHz, which includes all HF ham bands and 6 m) through the interface, and even enable or disable decoding of each mode as needed to manage CPU load. By leveraging open-source libraries (GPL and MIT licensed), we ensure the solution is robust and maintainable without reinventing complex algorithms from scratch. The system effectively acts as an **automated FT8/JS8 monitoring server**, suitable for applications like propagation monitoring, digital mode research, or just keeping logs of all messages heard on the air. The architecture emphasizes efficiency (critical on Pi hardware) and clarity (modular components), fulfilling the project’s goals of decoding message content and estimating signal SNR with high reliability.

**Sources:**

* Taylor et al., *“The FT4 and FT8 Communication Protocols,”* *QEX* Jul/Aug 2020 – describes FT8’s 77-bit messages, LDPC FEC, and synchronization method[[1]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=other%20digital%20modes%20pioneered%20in,specifi%20cally%20for%20this%20application)[[5]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=Tone%20patterns%20known%20as%20Costas,bit%20fi%20elds%20listed%20in).
* Signal Identification Wiki – *“JS8Call (JS8)”* – notes that JS8 is based on FT8 modulation, enabling longer messages for weak-signal chats[[3]](https://www.sigidwiki.com/wiki/JS8#:~:text=Location%20Worldwide%20Short%20Description%20JS8Call,Raw%20Recording%20%E2%80%94%20Audio%20Sample).
* Amateur Radio StackExchange – discussion of FT8 SNR definition – SNR is reported relative to 2500 Hz noise bandwidth for standardization[[7]](https://ham.stackexchange.com/questions/22477/why-is-snr-for-ft8-defined-using-the-noise-power-in-the-2500-hz-channel-containi#:~:text=But%20it%27s%20easy%20to%20change,based%20on%20625%20etc%20etc).
* *ft8mon* project (Robert Morris, AB1HL) – open-source C++ FT8 decoder, demonstrating multi-signal decode output with timestamp, SNR, frequency, message[[20]](https://github.com/rtmrtmrtmrtm/ft8mon#:~:text=You%20should%20see%20output%20like,this).
* *ft8\_lib* (rpitx) – open C++ implementation of FT8 encoding/decoding for experimentation[[24]](https://git.jerryxiao.cc/rpitx/ft8_lib#:~:text=A%20C%2B%2B%20implementation%20of%20FT8,for%20experimental%20use%20on%20microcontrollers)[[13]](https://git.jerryxiao.cc/rpitx/ft8_lib#:~:text=Thanks%20to%20Robert%20Morris%2C%20AB1HL%2C,various%20parts%20of%20the%20code). Contains LDPC and message routines that can be leveraged.
* *ft8modem* by KK5JY – an example of a command-line FT8 modem which decimates audio to 12 kHz for the WSJT-X jt9 decoder and uses simple text I/O[[27]](http://www.kk5jy.net/ft8modem/#:~:text=The%20ft8modem%20program%20takes%20two,available%20options%20and%20sound%20devices)[[9]](http://www.kk5jy.net/ft8modem/#:~:text=The%20decode%20,when%20each%20transmission%20actually%20starts), informing our approach to interfacing with decoding logic.

[[1]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=other%20digital%20modes%20pioneered%20in,specifi%20cally%20for%20this%20application) [[2]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=FT4%20and%20FT8%20transmissions%20always,targeted%20purposes%2C%20we%20allocate%20three) [[5]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=Tone%20patterns%20known%20as%20Costas,bit%20fi%20elds%20listed%20in) [[6]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=77,CRC%20word) [[10]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=FT8%20messages%20are%20transmitted%20using,only%20one%20bit%20position%2C%20thereby) [[11]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=of%20the%20information%20symbols%20by,%EF%80%BD%20%EF%81%BB%20%EF%81%BD%203%2C2%2C0%2C1) [[14]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=77,matrix%20used%20to%20compute%20the) [[15]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=FT4%20and%20FT8%20are%20digital,radio%20activity%20on%20the%20high) [[16]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=FT4%20and%20FT8%20are%20digital,popularity%2C%20by%20some%20measures%20soon) [[17]](https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf#:~:text=c28%20Standard%20callsign%2C%20CQ%2C%20DE%2C,h12%20Hashed%20callsign%2C%2012%20bits) Taylor.indd

<https://wsjt.sourceforge.io/FT4_FT8_QEX.pdf>

[[3]](https://www.sigidwiki.com/wiki/JS8#:~:text=Location%20Worldwide%20Short%20Description%20JS8Call,Raw%20Recording%20%E2%80%94%20Audio%20Sample) [[4]](https://www.sigidwiki.com/wiki/JS8#:~:text=FT8%20%20JS8Call%20160m%20,136%20%20All) JS8 - Signal Identification Wiki

<https://www.sigidwiki.com/wiki/JS8>

[[7]](https://ham.stackexchange.com/questions/22477/why-is-snr-for-ft8-defined-using-the-noise-power-in-the-2500-hz-channel-containi#:~:text=But%20it%27s%20easy%20to%20change,based%20on%20625%20etc%20etc) [[19]](https://ham.stackexchange.com/questions/22477/why-is-snr-for-ft8-defined-using-the-noise-power-in-the-2500-hz-channel-containi#:~:text=Matter%20of%20opinion%2C%20but%20it%27s,have%20a%20different%20occupied%20bandwidth) wsjt x - Why is SNR for FT8 defined using the noise power in the 2500 Hz channel containing the FT8 signal? - Amateur Radio Stack Exchange

<https://ham.stackexchange.com/questions/22477/why-is-snr-for-ft8-defined-using-the-noise-power-in-the-2500-hz-channel-containi>

[[8]](http://www.kk5jy.net/ft8modem/#:~:text=The%20ft8modem%20program%20takes%20two,of%20available%20options%20and%20sound) [[9]](http://www.kk5jy.net/ft8modem/#:~:text=The%20decode%20,when%20each%20transmission%20actually%20starts) [[25]](http://www.kk5jy.net/ft8modem/#:~:text=implemented%20by%20other%20tools,you%20want%20them%20to%20work) [[27]](http://www.kk5jy.net/ft8modem/#:~:text=The%20ft8modem%20program%20takes%20two,available%20options%20and%20sound%20devices) ft8modem - a command-line software modem for FT8

<http://www.kk5jy.net/ft8modem/>

[[12]](https://git.jerryxiao.cc/rpitx/ft8_lib#:~:text=Decoding%20is%20is%20still%20work,of%20RAM%20for%20that%20purpose) [[13]](https://git.jerryxiao.cc/rpitx/ft8_lib#:~:text=Thanks%20to%20Robert%20Morris%2C%20AB1HL%2C,various%20parts%20of%20the%20code) [[24]](https://git.jerryxiao.cc/rpitx/ft8_lib#:~:text=A%20C%2B%2B%20implementation%20of%20FT8,for%20experimental%20use%20on%20microcontrollers) rpitx/ft8\_lib - Forgejo: Hosted by JerryXiao

<https://git.jerryxiao.cc/rpitx/ft8_lib>

[[18]](https://news.ycombinator.com/item?id=22830504#:~:text=The%20indicated%20FT8%20signal%20to,1%20dB%20in%2050) The indicated FT8 signal to noise ratio is very misleading. The ...

<https://news.ycombinator.com/item?id=22830504>

[[20]](https://github.com/rtmrtmrtmrtm/ft8mon#:~:text=You%20should%20see%20output%20like,this) GitHub - rtmrtmrtmrtm/ft8mon: FT8 receiver in C++

<https://github.com/rtmrtmrtmrtm/ft8mon>

[[21]](https://www.hydrogen18.com/blog/calling-wsjtx-encoding-decoding-from-c.html#:~:text=Unfortunately%20for%20myself%2C%20decoding%20is,call%20a%20Fortran%20subroutine%20called) [[22]](https://www.hydrogen18.com/blog/calling-wsjtx-encoding-decoding-from-c.html#:~:text=call%20my_ft8,napwid%2Cmycall%2Chiscall) [[26]](https://www.hydrogen18.com/blog/calling-wsjtx-encoding-decoding-from-c.html#:~:text=Calling%20WSJT,The%20normal%20way) Calling WSJT-X encoding and decoding functions from C

<https://www.hydrogen18.com/blog/calling-wsjtx-encoding-decoding-from-c.html>

[[23]](https://github.com/jgaeddert/liquid-dsp#:~:text=jgaeddert%2Fliquid,on%20your%20build%20platform) jgaeddert/liquid-dsp: digital signal processing library for ... - GitHub

<https://github.com/jgaeddert/liquid-dsp>