THE URF REFLECTOR REFERENCE

A written, sourced guide to the URF (urfd) universal digital-voice reflector — what it is, where it came from, how it works from the packet upward, how you connect to it, and how you build, configure, operate, secure, and update one.

Compiled 8 August 2026 · sourced from primary documentation and live-operator practice

Edition 1.1 — expanded with capacity and FEC math, testing and troubleshooting procedures, and operations detail

Preface

Anyone who’s spent time with URF reflectors knows the information is scattered — a piece in the source code, a piece on a forum, a piece on someone’s dashboard, and some of it out of date. This document pulls it together.

It’s compiled from numerous searches using many different sources — the reflector’s source code, the vendor and project documentation, and the pages of operators running live systems — and assembled into one reference, so the answer’s in a single place the first time you look. It’s built to save you the hunt: get oriented, find the exact command, port, or talkgroup you need, and see where sources agree and where they conflict.

It’s a starting point, not the last word — a springboard for your own research; since reflectors and software change, confirm anything critical against your own running system.

Free to use, copy, and share, and searchable — jump straight to what you need.

Contents

A written, sourced guide to the URF (`urfd`) universal digital-voice reflector — what it is, where it came from, how it works from the packet upward, how you connect to it, and how you build, configure, operate, secure, and update one.

Compiled 8 August 2026. This guide is built from primary sources — chiefly the `urfd` and `tcd` source code and documentation read directly — and corroborated by independent operators running live URF reflectors and by the relevant standards and reference material. Sources are named in the text as each point is made and collected at the end. Where the sources genuinely disagree, the disagreement is described rather than smoothed over; where something could not be established from a reliable source, that is said plainly rather than guessed. Figures that drift with time — prices, patent-expiry estimates, and default ports — are marked *"as of"* where they appear. The legal chapter concerns United States law and is a factual summary, not legal advice.

Quickstart — the whole guide in brief

If you read nothing else, this is URF in a nutshell. A **URF reflector** is a server running the open-source `urfd` software that joins amateur digital-voice radios into shared rooms and, uniquely, speaks every major mode at once — D-STAR, DMR, Fusion, P25, NXDN, and M17 — bridging between them so that users of different modes can actually talk to one another. It organizes traffic into up to twenty-six independent **modules** lettered A–Z, of which up to three can be **transcoded** (converted between the modes' incompatible voice codecs) with the help of a separate transcoder program, `tcd`, while the rest carry single-mode traffic. Transcoding needs a DVSI AMBE dongle for D-STAR, but M17, P25, and — with a software library — the DMR and Fusion codecs can run with no hardware at all.

To **stand one up**: on a Debian or Ubuntu server with a public address, install the dependencies, clone `urfd` (and `tcd` beside it if you are transcoding), copy and edit the configuration templates, run `make`, validate with `inicheck`, and install with the `radmin` script or `make install`; then copy the dashboard into your web

server. Verify each mode end to end with a software client before you announce it (Chapter 14). To **use one without a radio**, connect with a software client like DroidStar or mvoice. The reflector runs as an ordinary systemd service, and if audio ever goes silent the usual cause is a user on the wrong module or a stopped transcoder (Chapter 15 has a decision tree).

Everything after this — the mechanics, the internals down to the packet and the FEC math, capacity math, operations and troubleshooting, the law, the trade-offs, the costs, and the wider ecosystem of bridges and codecs URF lives in — is that same story told in full.

Chapter 1 — What a URF Reflector Actually Is

To understand a URF reflector you first have to understand a reflector. In amateur digital voice, a reflector is a server on the internet whose job is to take whatever one connected station transmits and repeat it out to every other station connected to the same place. It is the digital-voice equivalent of a conference bridge on a telephone system: a dozen operators around the world can all link their radios or hotspots to one reflector, and when any of them keys up, the rest hear it. The reflector holds no transmitter of its own; it moves audio between people over IP. This is how a handheld radio in California can be heard, in real time, by an operator in Germany.

A **URF reflector** is a particular, ambitious kind of reflector: a *universal* one. The software that runs it, called `urfd`, describes itself in a single line that is worth reading carefully — “A Universal Reflector with M17, a superset of XLX.” The fuller name in its own documentation is “The URF Multi-protocol Gateway Reflector Server,” and it is released under the GNU General Public License. The word that matters most in that description is *multi-protocol*. Amateur digital voice is not one technology but a half-dozen mutually incompatible ones — D-STAR, DMR, Yaesu System Fusion, P25, NXDN, and the newer open M17 — each with its own way of framing audio and, crucially, its own voice codec. Most reflectors speak exactly one of these. A URF reflector speaks all of them at once, on one server, and — with the right helper — can translate between them so that a DMR user and an M17 user on the same reflector can actually talk to each other. That translation is the hard part, and most of this guide is ultimately about how it is done.

The phrase “superset of XLX” tells you where URF sits in the ecosystem. XLX was an earlier and very successful multi-protocol reflector; URF does everything XLX did and adds native support for M17 along with integrated P25, NXDN, and USRP/AllStar audio. For the sake of fitting into the existing infrastructure — dashboards, host lists, registries that were all built around XLX — a URF reflector even **reports itself as an XLX reflector** in the status file it generates (the mechanism is in Chapter 7). But this backward compatibility has a hard limit that its own documentation states flatly: **“You can’t interlink urfd with xlxd.”** The two are cousins, not the same thing, and a URF reflector links to other URF reflectors, not to the older XLX ones — a boundary that returns wherever interlinking and migration come up (Chapters 10 and 25). One live operator, HRCC Link, describes their URF in operator’s terms as “a multi-protocol Digital Voice (DV) Server” that “supports all protocols of its predecessors, as well as both M17 protocols, voice-only and voice+data” — which is a good plain-language summary of the whole idea.

Chapter 2 — Where URF Came From

URF did not appear out of nowhere. It is the current end of a chain of reflector software that stretches back roughly twenty years, and knowing that chain explains a great deal about why URF looks and behaves the way it does — including some of its quirks, like the XLX-shaped status file mentioned above.

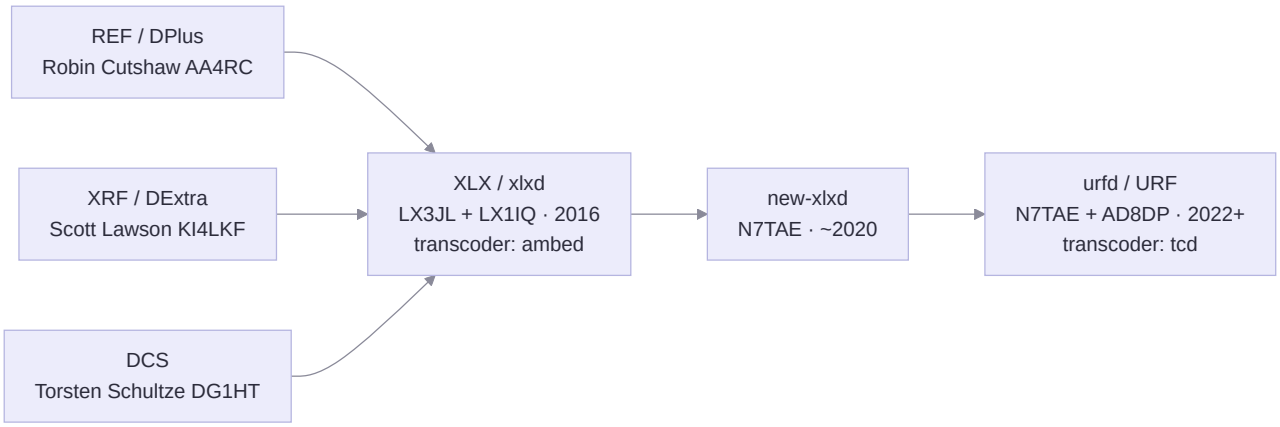
The story begins with D-STAR, the first mainstream amateur digital-voice mode. In D-STAR's early days three different reflector systems grew up, each speaking its own linking protocol. The first was **REF**, built on a protocol called DPlus and written by Robin Cutshaw, AA4RC; it was tied to a central registration system known as US-Trust and offered a handful of rooms lettered A through E. Then came **XRF**, using an open protocol called DExtra written by Scott Lawson, KI4LKF — notable as the first *open-source* reflector protocol, a deliberate open replacement for the proprietary DPlus. Lawson stopped active work around 2013. Alongside these grew **DCS**, a centrally-run system that originated in Germany, created by Torsten Schultze, DG1HT. The independent writer F4FXL has a good concise account of these three and their differences, and the amateurradionotes reference collection documents their authorship and history.

The breakthrough came in 2015 and 2016 with **XLX**, written by Jean-Luc Deltombe (LX3JL) and Luc Engelmann (LX1IQ) as part of a Radio Amateurs du Luxembourg working group. XLX's insight was to speak all three of the older D-STAR protocols — REF, XRF, and DCS — at once, plus its own protocol for linking XLX servers to each other, and then to reach *beyond* D-STAR entirely into DMR and Yaesu Fusion, transcoding audio between them. A reproduction of the XLX team's own timeline (hosted by Lyons Computer) dates the project's conception to October 2015, its first beta to 7 November 2015, and its adoption of the GPL to January 2016, which is why the source carries a 2016 copyright. XLX's transcoding was handled by a separate component called `ambed`. It was, as one how-to puts it, the first multi-protocol reflector with transcoding — the direct ancestor of everything URF does.

Around 2020, Thomas A. Early (N7TAE) produced `new-xlxd`, described as "an improved XLX reflector." This was a modernization rather than an expansion: the code was rewritten to use modern C++ containers, futures instead of raw threads, smart pointers, and a clean shutdown path. It added no new protocols. That groundwork set the stage for the real leap. In 2022, Early — now joined by Doug McLain (AD8DP) — released `urfd`, the Universal Reflector. This is where M17 arrived, along with integrated P25 (using a software IMBE vocoder), NXDN, and USRP, and where the old `ambed` transcoder was replaced by a new hybrid transcoder called `tcd`. The copyright headers in the source record the whole lineage in three lines: 2016 to Deltombe and Engelmann, 2022 to McLain and Early, and 2024 to Early alone. The same author, N7TAE, also maintains a sibling project, `mrefd`, which is an M17-only reflector — useful to know because URF and mrefd share a common code ancestry and a common way of thinking about "modules."

One practical wrinkle: there are two active copies of the URF source on GitHub, `n7tae/urfd` and `nostar/urfd` (nostar being Doug McLain's handle). They are nearly identical, and for the fine details of ports and configuration they can be treated as one source. The most meaningful difference between the two, as we will see in Chapter 9, lives not in the reflector but in their respective copies of the `tcd` transcoder.

The chain, in one picture:



Chapter 3 — The Module Model

The single most important structural idea in a URF reflector is the **module**. A URF reflector is not one room but as many as twenty-six of them, lettered A through Z, and each module is a completely independent conference. People linked to module A hear each other; people on module B hear each other; the two do not mix unless the operator has deliberately linked them. This lets one reflector host, say, a busy daily net on one module, a quiet technical rag-chew on another, and a mode-bridging experiment on a third, all at once and without interference.

The design is flexible in ways worth knowing. A reflector can run as few as one module or as many as twenty-six, and the modules need not be contiguous — a reflector can offer A, B, C, and E while skipping D, according to the documentation. The reason an operator would deliberately *not* use every module comes down to transcoding, which is limited. The documentation frames the capacity split precisely: a URF reflector can support “up to three transcoded modules and up to twenty-three more untranscoded modules.” A transcoded module is one where the reflector converts between codecs so that different modes can talk; an untranscoded module simply passes audio straight through, which means, in the documentation’s own blunt words, “clients will only hear other clients using the same codec.” Because of that limitation, operators commonly dedicate specific modules to specific modes and tell their users exactly which module to select — and, as a matter of convention noted in the docs, name a module after the mode it is meant to carry. The sibling M17 reflector `mrefd` uses this same twenty-six-channel model, which is why module-thinking carries across the whole family of software.

Chapter 4 — The Modes It Carries

A single running `urfd` presents itself to the world across a remarkably wide set of protocols simultaneously. Reading from its documentation, it supports the D-STAR family (the DPlus, DCS, and DExtra linking protocols, plus Icom’s G3 terminal and access-point mode), the DMR world (via MMDVMHost, the DMR+ dongle protocol, and NXDN), native M17, Yaesu System Fusion, P25 using a software IMBE vocoder, an integrated NXDN reflector, and USRP audio for bridging to AllStar. In effect, whatever kind of digital-voice radio or hotspot an operator owns, a URF reflector can probably talk to it.

A quick word on counting, because it trips people up: this guide speaks of **six voice modes** — D-STAR, DMR, Fusion, P25, NXDN, and M17. USRP/AllStar is carried too, but it is a *bridge transport* that moves raw PCM audio to and from the analog/AllStar world, not a seventh over-the-air voice mode with its own codec. So the honest shorthand is "six modes plus a USRP bridge."

Two points about this are easy to get wrong and worth stating clearly. First, the Fusion side is served as a **YSF Master**, which the documentation is emphatic about: the URF server "acts as an YSF Master, which provides 26 wires-x rooms. It has nothing to do with the regular YSFReflector network, hence you don't need to register your URF at ysfreflector.de!" In other words, the Fusion rooms on a URF reflector are self-contained; they are its own twenty-six modules, not entries in the separate worldwide YSFReflector directory. (There is a subtlety here — the configuration file does contain YSF registration fields — which is taken up in Chapter 32, on disagreements.) Second, the USRP/AllStar connection is unusual in that, as the documentation notes, an AllStar node "doesn't support any connect or disconnect protocol"; the reflector simply creates the client from configured settings at startup, and that client must point at a transcoded module to be useful.

Before going deeper, here is the operator's-eye summary of what the reflector carries — the codec each mode uses, its default listening port, whether it can participate in a transcoded (mode-bridging) module, and where to look for the connect procedure. The codec and modulation detail is expanded in Chapter 19, the ports in Appendix A, and the connect steps in Appendix B.

Mode	Voice codec	Default UDP port	Transcode-capable?	How users connect
D-STAR	AMBE	DPlus 20001, DExtra 30001, DCS 30051, G3 40000	Yes — needs DVSI hardware	URCALL linking, G3 terminal, or DTMF (Ch 8)
DMR	AMBE+2	MMDVM 62030, DMR+ 8880	Yes — hardware or software	Private-call sequence via DMRGateway (Ch 8)
Fusion (YSF)	AMBE+2	42000	Yes — hardware or software	YSF Master → Wires-X room or DG-ID (Ch 8)
P25 Phase 1	IMBE	41000	Yes — software	Numeric reflector ID → auto-link module
NXDN	AMBE+2	41400	Yes — hardware or software	Numeric reflector ID → auto-link module
M17	Codec2	17000	Yes — software (native Codec2)	Pick host, then module letter
USRP / AllStar	raw PCM (bridge)	32000 (client Rx 34000)	Bridge transport, not a voice mode	Fixed in reflector/bridge config

Chapter 5 — Transcoding: The Heart of the Whole Thing

Everything interesting about a multi-mode reflector comes down to one hard problem, and it is worth slowing down to understand it, because it explains the hardware, the software architecture, and most of the operational headaches that follow.

Each digital-voice mode encodes speech with a different **vocoder** — the algorithm that turns a human voice into a low-bitrate stream of data and back again. D-STAR uses a proprietary codec called AMBE. DMR, Fusion, and NXDN use a related proprietary codec, AMBE+2. P25 uses yet another, IMBE. M17, being the open-source newcomer, deliberately uses an open, patent-free codec called Codec2. The consequence is simple and brutal: a DMR radio and an M17 radio produce completely different, mutually unintelligible data for the same spoken words. Put both users on the same reflector module and, unless something actively converts between their codecs in real time, they will sit in silence — each hearing nothing but static from the other. The documentation states this plainly: without transcoding, "clients will only hear other clients using the same codec."

The thing that does the conversion is a separate program from the reflector itself, called `tcd`, the transcoder. It is described in its own documentation as "a new kind of hybrid transcoder that uses AMBE DVSI-based hardware for vocoding digital voice streams used in DStar/DMR/YSF and David Rowe's open-source Codec2 used in M17 as well as the open-source P25 vocoder, IMBE." The word *hybrid* is the key. Some of these codecs are proprietary and only exist, legally and practically, in hardware chips you buy from a company called DVSI; others are open source and run in software. `tcd` bridges both worlds in one place, using a DVSI hardware chip for the AMBE and AMBE+2 conversions that require it, and running Codec2 (for M17) and IMBE (for P25) in software. It can even do the AMBE+2 conversion in software on an ARM processor if you accept that tradeoff. The upshot is a table that is worth memorizing: D-STAR's AMBE needs the DVSI hardware; DMR, Fusion, and NXDN's AMBE+2 can use either the DVSI hardware or an ARM software library; P25's IMBE runs in software; and M17's Codec2 runs in software.

The way the reflector and the transcoder talk to each other changed meaningfully from the XLX era, and the change matters operationally. In XLX, the `ambed` transcoder communicated over UDP. In URF, the documentation explains, "communication between urfd and tcd is now handled via TCP, with urfd acting as the server and tcd is the client," and "a two-way TCP channel will be opened for each transcoded module." Choosing TCP was deliberate — as the transcoder's documentation puts it, TCP "guarantees that packets moving between the reflector and transcoder are never lost and arrive at their destination in order." A useful side effect is that the transcoder can run on the same machine as the reflector or on a completely different one, because the reflector is the server and the transcoder dials in; there is no need to open any ports on the transcoder's host. The internals of exactly how a frame makes this round trip are described in Chapter 9, on the transcoder's guts.

Chapter 6 — The Vocoder Hardware

Because D-STAR's AMBE codec cannot legally or practically be done in open software, anyone running a transcoding URF reflector that carries D-STAR needs real DVSI hardware, and it helps to understand what that hardware is. DVSI's AMBE chips come in fixed channel counts, and those counts set hard limits on how many

modules you can transcode. The single-channel chip is the AMBE-3000, which you will find inside devices sold as the USB-3000, the DVMEGA DVstick-30, and the NW Digital Radio ThumbDV. The three-channel chip is the AMBE-3003, sold as the USB-3003 and the DVstick-33 among others. DVSI's own product description confirms the difference: the AMBE-3003 "offers three separate encoder/decoder operations to provide the capability of up to three full duplex channels," where the 3000 provides one. There are larger devices too — a six-channel USB-3006 and a twelve-channel USB-3012 that packs three AMBE-3003 chips onto one interface — documented in the XLX device tables. The transcoder's own code enforces these limits: it will refuse to run more than one transcoded module on a lone 3000, or more than three on a 3003, printing "Too many transcoded modules for the devices" if you ask for too much.

There is one hardware quirk that trips up nearly every first-time builder, and it is worth flagging loudly. The DVSI USB devices use FTDI chips, and they need FTDI's own D2XX driver — but that driver, as the transcoder's documentation warns, "will only work if the normal Linux kernel `ftdi_sio` and `usbserial` drivers are removed." If those standard kernel modules grab the device first, the transcoder can't. The good news is that this is handled for you automatically: the systemd service file that starts `tcd` unloads those two kernel modules before launching. The trap is only sprung if you try to run the transcoder by hand outside of systemd, in which case you have to unbind the kernel drivers yourself — a technique one operator, Florian Wolters, documents in detail with a udev rule targeting the USB-3003 by name.

This also explains the "three transcoded modules" ceiling that keeps coming up. An AMBE-3003 has three hardware channels. Because AMBE+2, IMBE, and Codec2 can all be pushed into software, the precious hardware channels only have to serve D-STAR's AMBE, so three modules can be transcoded from one 3003. An operator running a real reflector, K2AS, adds a useful clarification about how the counting actually works: their transcoder allows "up to four channels to be transcoded at once. That would be two modules allowing D-Star to DMR or Fusion communication at one time." The point is that transcoding is counted in channels, roughly two per active module, so "three modules" is the software's ceiling and what you actually achieve depends on your hardware. (There is a genuine, well-documented disagreement about whether a lone AMBE-3003 yields two or three usable streams; it is taken up in Chapter 32, and the short version is that the all-hardware `ambed` code used two while the hybrid `tcd` reaches three.)

Chapter 7 — Inside the Reflector

It is worth opening the reflector up and seeing how it actually moves a voice stream, because the design is elegant and it makes the later operational advice make sense. What follows was read directly from the `urfd` source code.

At the center sits a single object, `CReflector`, that owns everything: a list of recently-heard users, a list of connected clients, a list of linked peer reflectors, and a set of protocol handlers — one for each mode it speaks. It also owns, for each module, a small piece of machinery: a stream object and a dedicated **router thread**. Each protocol handler runs its own loop, listening on its own UDP sockets for its own mode's traffic.

When someone keys up, the sequence is precise. The relevant protocol handler receives a voice header, validates that it came from a known client who isn't already transmitting, and hands it to the reflector's `OpenStream`

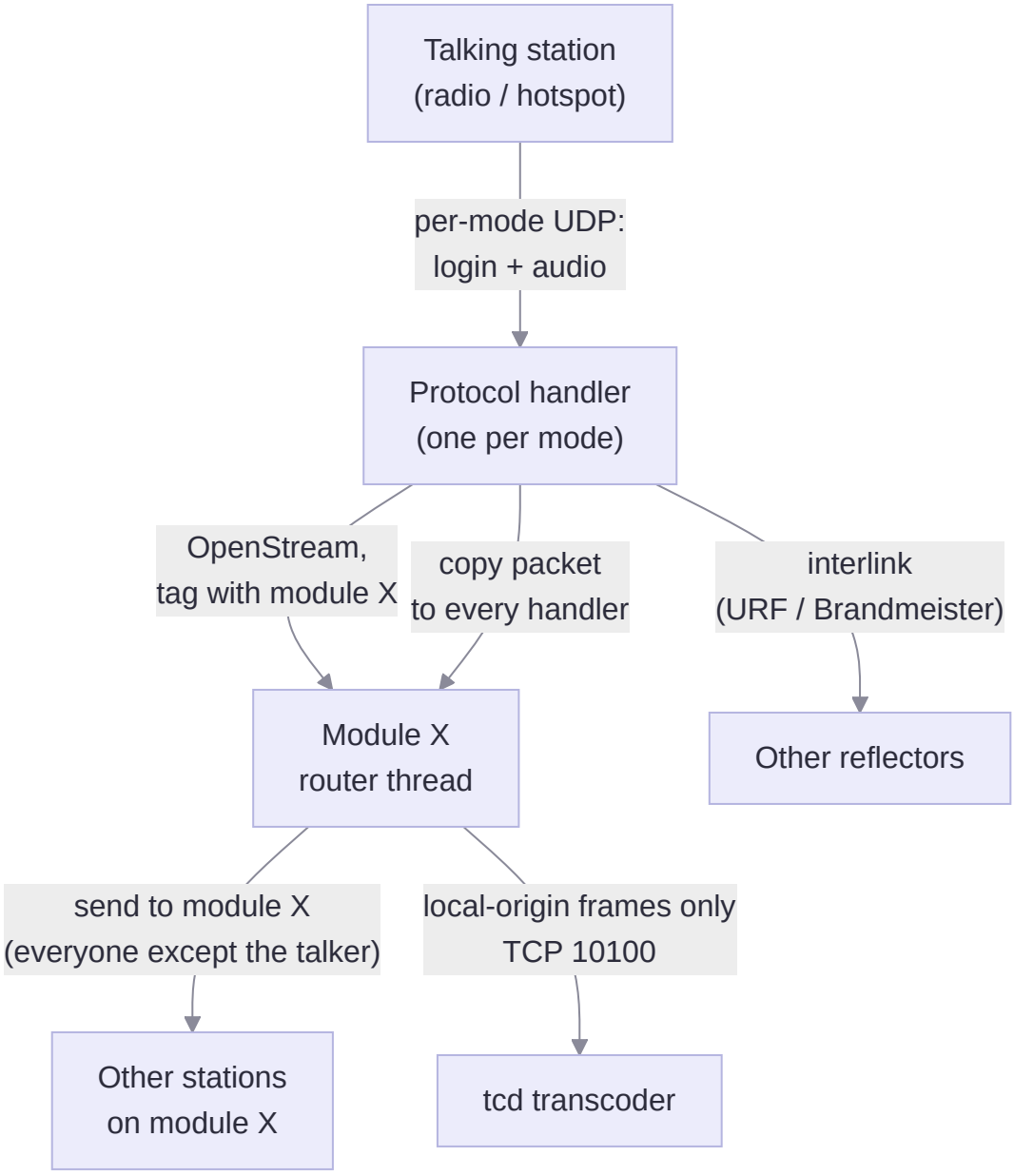
routine. That routine does something clever for loop prevention: it rejects any stream whose ID is zero or that duplicates a stream already open on another module, which is how the reflector avoids echoing traffic in circles. It then tags the packet with the module the client is linked to, opens that module's stream, and marks the client as the current "master" of the module — the one whose audio everyone else will hear. Subsequent frames are matched not just by stream ID but by the sender's IP address; anything that doesn't match is logged as an "orphaned frame" and discarded. If the audio stops arriving, the stream times out after 1.6 seconds and closes.

The actual fan-out — getting the audio to everyone else — is the job of the per-module router thread. It pops each packet from the module's stream and copies it into *every* protocol handler's outbound queue, adjusting the destination callsign for each protocol as it goes. Each handler then sends its copy to the clients linked to that module, and here is the universal rule that makes a reflector a reflector: it sends to everyone on the module *except* the station currently transmitting. That single condition — don't send the talker their own audio back — appears in every protocol handler in the code.

Transcoding is woven into this flow with care. Only frames that originated locally are diverted to the transcoder; a frame that arrived already-transcoded from a linked peer reflector is left alone, so audio isn't needlessly transcoded twice. When a frame is sent to the transcoder, the reply that comes back is matched to the original by both stream ID and sequence number, timed for round-trip latency, and re-injected into the module's flow. There is even a small touch of protocol housekeeping visible in the code: a D-STAR sync pattern is force-inserted every twenty-first packet for streams that didn't originate as D-STAR.

Two more behaviors from the internals are worth knowing because they surface elsewhere in this guide. The reflector writes a status file every ten seconds in which it deliberately patches its own callsign to begin with "XLX" — this is the mechanism behind "reports itself as an XLX reflector," and it exists so the whole ecosystem of XLX dashboards and registries treats a URF reflector as one of their own. And the reflector's Ham-DHT presence — its entry in the distributed directory that lets other reflectors find it — runs on UDP port 17171 and publishes a signed record containing its callsign, addresses, module list, ports, and per-module descriptions.

In block-diagram form, a single voice stream moves through the reflector like this:



Chapter 8 — How Each Mode Connects, and the Security Behind It

Every mode has its own way of logging in and its own way of choosing a module, and while the full protocol-by-protocol detail is more than most readers need, the shape of it is genuinely useful — especially the parts an operator has to explain to their users.

DMR is the most involved, because DMR has no natural concept of “pick a room.” Connecting through the standard DMRGateway software (which is what Pi-Star and WPSD use), a user selects the reflector and module with private calls and then talks on a talkgroup. The convention, corroborated across many operator sites, is

captured in the table below: to reach module E on reflector 493 you make a private call to 68493, then 64005, then talk on TG 6. Under the hood the reflector's own code maps DMR talkgroups 4001 through 4026 to modules A through Z, with 4000 meaning "unlink" (some DMR+ setups drive linking through those talkgroups directly rather than the 64000-series private calls).

Action	What you send	Example (module E, reflector 493)
Connect to a reflector	Private call to `68` + reflector number	`68493`
Select a module	Private call to `64000` + module number (A=1 … Z=26)	`64005`
Talk	Transmit on talkgroup `6`	TG `6`
Disconnect	Private call to `64000`	`64000`
Query status	Private call to `65000`	`65000`
Module via talkgroup (DMR+)	TG `4001`–`4026` = module A–Z; TG `4000` = unlink	TG `4005` = module E

There is a notorious trap here that catches everyone at least once, flagged by the QuadNet operators: the `XLXHosts.txt` file that ships with DMRGateway "defaults to Module D, which is not the module the transcoder is active on," so a user who doesn't override it lands on the wrong, silent module and assumes the reflector is broken.

D-STAR selects a module the old-fashioned way, by putting a linking command in the radio's URCALL field: the reflector's callsign, then the module letter, then an "L" to link or "U" to unlink, all in the eighth character position — so `XLX301CL` links to module C. A question that took real digging to answer definitively is whether a URF reflector answers to a literal `URFnnn` URCALL or only to the `XLXnnn` form: the answer, confirmed on a live URF reflector by HRCC Link, is that it answers to both — HRCC explicitly instructs terminal-mode users to use "UR: /URF621A," and because the reflector also reports as XLX, the `XLXnnn` form keeps working too. For Icom's G3 terminal and access-point mode, the operator VA6MO documents the full procedure: register on a G3 (not G2) gateway, assign a module letter avoiding G and S, use Icom's RS-MS3A or RS-MS3W software with that letter in the eighth character of your access-point callsign, point it at the reflector's address, punch a hole for UDP port 40000 if it isn't forwarded, and press the PTT.

There is a third D-STAR path worth spelling out, because it is the one most radios can do without a computer: **DTMF**. Where URCALL editing is fiddly on a handheld, many D-STAR radios and hotspots let you key a linking command as DTMF tones instead — sending the module letter's position and an `L` / `U` equivalent as a short DTMF string (for example a sequence like `D03` to link module C, or `D#` / `D0` to unlink), which the gateway translates into the same reflector-plus-module link that the URCALL field would set. The exact DTMF digits are set by the gateway or hotspot software rather than by the reflector, so — as with DG-ID on Fusion — you follow your hotspot's DTMF table rather than a single universal code. It is the same underlying link; only the way you enter it differs.

Fusion, P25, and NXDN each have their own logic. On Fusion, you connect your hotspot to the URF YSF Master and then choose a module either with the radio's Wires-X button (the twenty-six rooms) or by DG-ID, where the reflector maps DG-IDs 10 through 35 to modules A through Z — though the exact DG-ID-to-module mapping is set per reflector, so you read each reflector's own instructions. P25 and NXDN are more limited from the user's side: you set a numeric reflector ID as your startup host and you land on whatever module the reflector operator has configured as its auto-link module. The end user generally cannot freely pick a module A through Z from a P25 or NXDN radio, because `AutoLinkModule` is a setting on the reflector, not on the hotspot.

Behind all of this sits a security model that is worth understanding because it is simpler than people assume. The reflector authorizes both linking and transmitting against a **blacklist** and a **whitelist** of callsigns, read from files that support a limited wildcard. The logic, from the code, is that an empty whitelist lets everyone through; a non-empty one requires a match; and in either case a callsign that matches the blacklist is refused. The same check governs both linking and transmitting for the RF-facing protocols — there is, per a "to-do" comment left in the source, no separate transmit-level authorization implemented. Peer reflectors are authorized more strictly: a linking reflector's callsign, its IP address, and its requested set of shared modules must all match an entry in the interlink map, and the shared modules must match on both sides. A whitelist, then, is the tool that turns a public reflector into a private, invitation-only one.

Chapter 9 — Inside the Transcoder

The transcoder deserves its own look, because it is where the real signal-processing happens and where the hardware quirks live. This, too, was read from source.

`tcd` is the client in the TCP relationship; the reflector is the server. When it starts, it opens one TCP socket per transcoded module, all pointed at the reflector's address and transcoder port, and the very first thing it sends on each connection is a single byte identifying which module that socket carries. If the reflector isn't up yet, it retries every tenth of a second, printing "Connection refused! Restart the reflector." roughly every ten seconds — which is the origin of that famous nag message operators see in the logs. Every few seconds it sends a keepalive ping so the reflector knows it's alive. The unit of exchange is a fixed data structure that carries the sequence number, the module letter, the stream ID, which codec the audio arrived as, and a slot for each codec's version of the frame. The reflector fills in the incoming codec's slot and sends it over; the transcoder fills in all the others and sends it back.

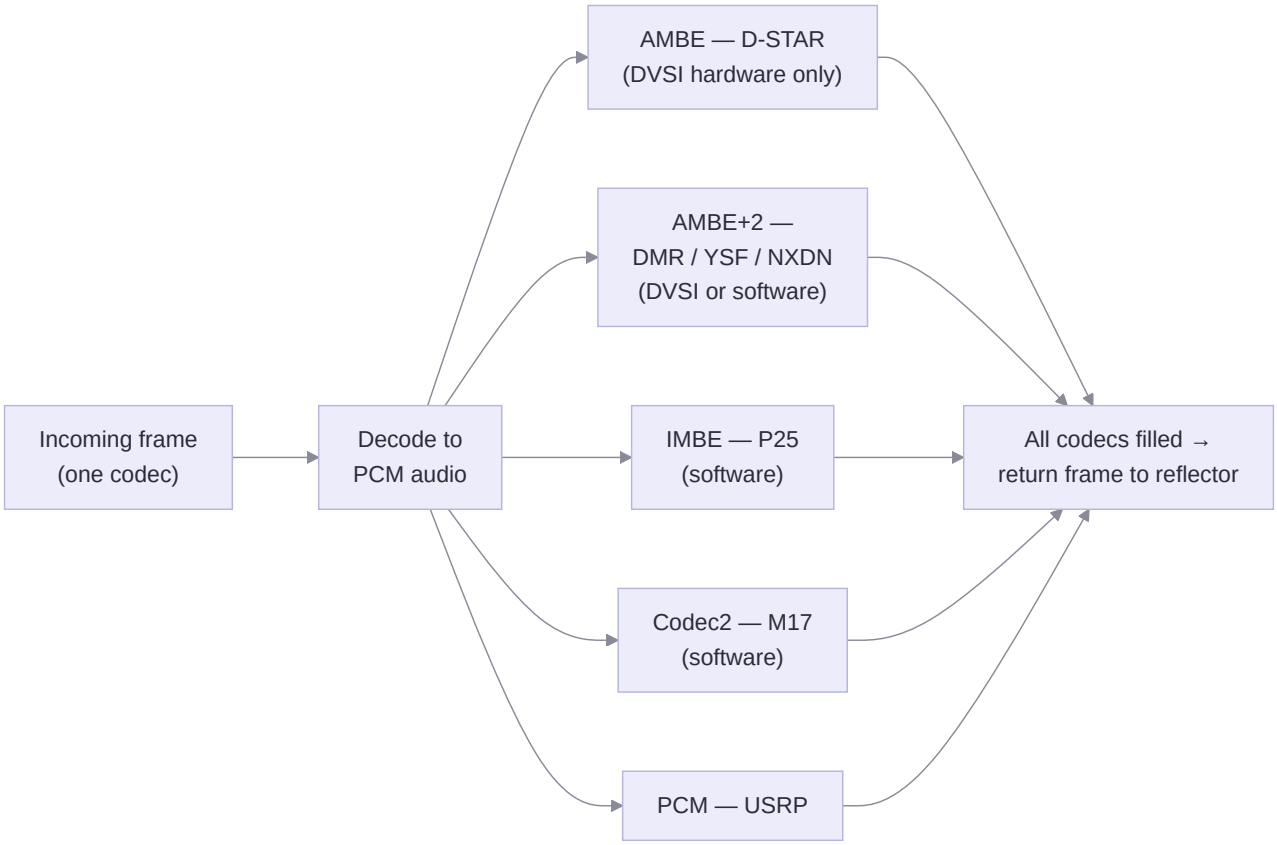
The processing pipeline inside `tcd` is the elegant part. When a frame arrives, the transcoder decodes it from whatever codec it came in as into plain PCM audio — ordinary digitized sound — and then re-encodes that audio into every other codec: AMBE for D-STAR, AMBE+2 for DMR and Fusion, IMBE for P25, Codec2 for M17, and raw PCM for USRP. Only when all of those are filled in does it send the fully-transcoded frame back to the reflector, which then hands each connected client the version in the codec that client understands. Some of these conversions run on the DVSI hardware and some in software, on their own threads, which is why the documentation notes that "each configured module has a dedicated encoding and decoding instance running on a different thread."

The hardware handling is meticulous. The transcoder discovers DVSI devices through the FTDI driver, and by default it insists on finding exactly two — one to handle D-STAR's AMBE and one for the AMBE+2 modes — unless it has been built to do AMBE+2 in software, in which case one device suffices. It opens each device by serial number, sets the correct baud rate (460800 for a 3000, 921600 for a 3003), and configures each vocoder channel with the specific DVSI parameter words for the codec it will run. If its input queue ever backs up past two hundred frames, it shuts itself down deliberately rather than fall behind.

This is also where the most important difference between the two source repositories lives. The `nostar` copy of `tcd` adds a software-only build mode that the upstream copy does not document as prominently. With it, a reflector can transcode M17, P25, and USRP — and, with an additional option, even AMBE+2 for DMR and Fusion — entirely in software, with no DVSI hardware at all. The one thing software cannot do is D-STAR's AMBE. So the practical rule is this: if you don't need D-STAR, you can run a transcoding multi-mode reflector on the `nostar` software stack without buying any hardware; the moment you want D-STAR in the mix, you need a real DVSI dongle.

There is a small but important configuration coupling to remember: the transcoder's own configuration file lists which modules it transcodes, and that list must exactly match the transcoded-modules line in the reflector's configuration. If the two disagree, the transcoder and reflector won't line up their channels correctly.

The pipeline inside the transcoder, where one frame becomes every codec:



Chapter 10 — Interlinking and the Wider Network

Reflectors gain their reach by linking to each other, and URF does this through a file called `urfd.interlink`. Each line names another reflector to peer with — its callsign, its IP address, the modules to share, and optionally a port that defaults to 10017. If both ends are participating in the Ham-DHT directory, the IP can be omitted and the line reduces to just a callsign and a module list, because the address is looked up automatically. The same file also handles Brandmeister links, written with three parameters and no port.

Three rules govern all of this. Interlinks are mutual: both reflectors must list each other, or the link never forms — a point the documentation states explicitly. Linking is module-to-module of the same letter; the M17 sibling reflector's documentation spells out that "cross module linking is not supported," so module A links to module A, never A to B. And the hard boundary already mentioned holds firm: a URF reflector cannot interlink with an old XLX reflector, only with other URF reflectors and with Brandmeister. As a historical footnote, the XLX interlink protocol actually came to support two forms over its life — the native XLX peer that shares a whole list of modules, and a later addition that links a single local module to a single remote one over the DExtra protocol — but URF inherits the native form for its reflector-to-reflector links.

Chapter 11 — M17 in Particular

M17 deserves a short chapter of its own because it is the newest mode, the reason URF exists as a distinct project, and the one with genuinely different capabilities. M17 is an open protocol, and its openness shows up in features the older modes lack. The dedicated M17 reflector `mrefd` — same author as URF, built on the same XLX-derived code — handles "up to 26 channels" and works in both of M17's modes, continuous Stream mode for voice and Packet mode for short data bursts.

Two things about M17 are distinctive. First, its interlinking is version-aware in a way that can bite you: there are three ways to write an interlink entry (a two-parameter form for DHT-enabled links, a full IP-and-port form, and a legacy form marked with a trailing "L"), and the documentation warns that if you link to a legacy reflector without the "L," "traffic will not pass in either direction, even if it successfully links." Second, M17 supports encryption in a way no other amateur mode really does, and this has legal weight covered in Chapter 17. The M17 specification defines three options — no encryption, a simple scrambler, and true AES in counter mode with a 96-bit nonce. Encryption in M17 is end-to-end, carried in the payload and the link-setup frame; the reflector itself doesn't decrypt anything, it simply relays encrypted streams, and only on a module configured to permit them. A newer listen-only connection type in `mrefd` powers browser-based listening, marketed as "M17web."

Chapter 12 — How a Reflector Is Found and Registered

For radios and hotspots to connect to a reflector, its existence and address have to be published somewhere, and URF sits across three overlapping systems for this. The oldest is "calling home": the dashboard can be configured

to periodically report the reflector to a central XLX API server at `xlxapi.rlx.lu`, which keeps a live list of who is up and generates the host files that hotspots download. This feature is off by default and comes with a stern warning discussed below. The newer system is Ham-DHT, a distributed directory built on the OpenDHT library, in which each reflector publishes a record — its configuration, its connected peers, its clients, and its recently-heard users — keyed by its callsign, and other systems look it up directly with no central server. A URF reflector bootstraps into this network through a node whose address is set in its configuration (the default is `xrf757.openquad.net`), and a set of command-line tools can query and map it. Layered on top of both is DVRef, a web registry with user accounts where an operator registers a reflector and is assigned a five-digit number; DVRef absorbed the older YSFReflector registry. The host files that ultimately populate the reflector dropdown menus in Pi-Star and WPSD are mirrored and refreshed hourly.

The warning that comes with calling home is important enough to repeat in full, because getting it wrong steps on someone else's reflector. A URF callsign has the form `URF` followed by three characters, and URF, XLX, and XRF all share a single numbering space — URF307, XLX307, and XRF307 are the same "307." So the documentation says, in capital letters: "DO NOT enable the 'calling home' feature unless you are sure that you will not be infringing on an existing XLX or XRF reflector with the same callsign suffix." The suffix is your identity across the whole ecosystem — in calling home, in the DHT, and in DVRef — so you pick an unused one and, ideally, register it before you turn calling home on.

Chapter 13 — Connecting Without a Radio

You do not actually need a radio or a hotspot to use a URF reflector; several software clients let a computer or phone connect directly, which is invaluable for testing your own reflector (a full per-mode test procedure is in Chapter 14). The most versatile is **DroidStar** (by nostar), which connects to M17, Fusion, DMR, P25, NXDN, and D-STAR reflectors as well as AllStar nodes, and runs on Linux, Windows, macOS, Android, and iOS. To reach a URF reflector you pick the mode, choose the `URFxxx` host, and — for D-STAR and M17 — select the module. For the M17 side specifically, N7TAE's **mvoice** is the natural choice: it is an M17-only desktop client that uses Codec2 in software and needs no AMBE dongle at all; you type the target callsign (six characters for a URF reflector, seven for an M17-named one) and pick a module. PA7LIM's long-standing **BlueDV** connects on the D-STAR, DMR, and Fusion sides (it does not document M17 support), though for those modes it needs a DVSI dongle to do the vocoding. And for M17 over the air, there are standard MMDVM hotspots running M17 or the M17 Project's dedicated Module17 board.

Chapter 14 — Building and Operating One

Bringing a URF reflector up follows a well-worn path, and the design of the project makes it cleaner than it first appears. The software runs only on systemd-based Linux, with Debian or Ubuntu recommended. You start on a machine with a permanent internet connection and a public address, install the dependencies (git, Apache, PHP, the build tools, the JSON and cURL libraries, and optionally the OpenDHT library for directory support), and clone the `urfd` repository — and, if you intend to transcode, the `tcd` repository alongside it in the same parent

directory, because the transcoder's source contains symbolic links into the reflector's source and the two must sit side by side.

The configuration step is where the project's cleanest design decision shows. You never edit files inside the repository itself; instead you copy the templates out (`cp ../config/* .` inside the reflector directory), which creates the seven configuration files, and you edit those copies. The documentation explains the reasoning directly: the repository "is designed so that you don't have to modify any file in the repository when you build your system," which is what makes updating painless later — you just pull the new code and your configuration copies are untouched. After editing, you compile with `make`, and — this is a step people skip at their peril — you validate the configuration with the included `inicheck` tool, which uses the very same parsing code the reflector uses. Anything it prints as an "ERROR" means the reflector will refuse to start.

Installation itself takes one of two forms. If you are running a local transcoder, you use the interactive `radmin` script and choose the install option, which also builds and installs the transcoder. If your transcoder is remote or you have none, you simply run `sudo make install`, which installs the systemd service, drops the binary in place, and starts it. Finally you copy the dashboard into your web server's directory and edit its one configuration file to set your email, country, and comment — leaving calling home off unless you have confirmed your callsign suffix is free.

Day to day, the reflector and transcoder are ordinary systemd services, managed with the familiar `systemctl start`, `stop`, `restart`, and `status`, with logs going to the journal rather than to a flat file — you watch them live with `journalctl -u urfd -f`. The `radmin` script wraps the common actions (restarting either service, following either log) behind single keystrokes. One operational behavior is worth internalizing because it looks alarming and has a simple fix: because the reflector is the TCP server and the transcoder the client, if the link between them drops, the transcoder cannot re-establish it on its own — it blocks and prints a message "every 10 seconds suggesting that the reflector needs to be restarted." The fix is exactly that: restart the reflector, which re-opens the per-module channels the transcoder is waiting on.

Verifying it works: a per-mode test procedure

A reflector that compiles and starts is not yet a reflector that works — the failure that matters most, callsigns-appear-but-no-audio, only shows up when real traffic flows. Before you announce a new reflector or trust a config change, walk this checklist using the software clients from Chapter 13; none of it needs a radio. The principle is to test each module the way its users will actually reach it, and to prove transcoding in both directions rather than just one.

#	Check	How	Pass criteria
1	Config parses	`./inicheck -q urfd.ini`; confirm `tcd` module list matches `[Transcoder] Modules`	No `ERROR` lines; lists identical
2	Services up	`systemctl status urfd tcd`	Both active; no “Restart the reflector” loop in `journalctl -u tcd`
3	Transcoder linked	Watch `journalctl -u urfd -f` at startup	One TCP channel opens per transcoded module
4	Single-mode audio	Two DroidStar instances on the same untranscoded module, same mode	Each hears the other; dashboard shows both
5	Transcode A→B	DroidStar in mode A + a client in mode B on the transcoded module; key up A	B hears A clearly
6	Transcode B→A	Reverse the above; key up B	A hears B (proves both directions)
7	M17 path	mvoice to the M17 module (no dongle)	Heard on the transcoded module by other modes
8	Dashboard truth	Compare dashboard “last heard” to who actually keyed up	Callsigns, module, and timestamps correct
9	Discovery	Confirm DNS resolves, host file lists you, DHT record present (`ham-dht-tools`)	A hotspot elsewhere can find and link
10	Failure drill	Stop `tcd`, key up, then restart the reflector	You reproduce the silent-audio symptom and recover — so you recognize it live

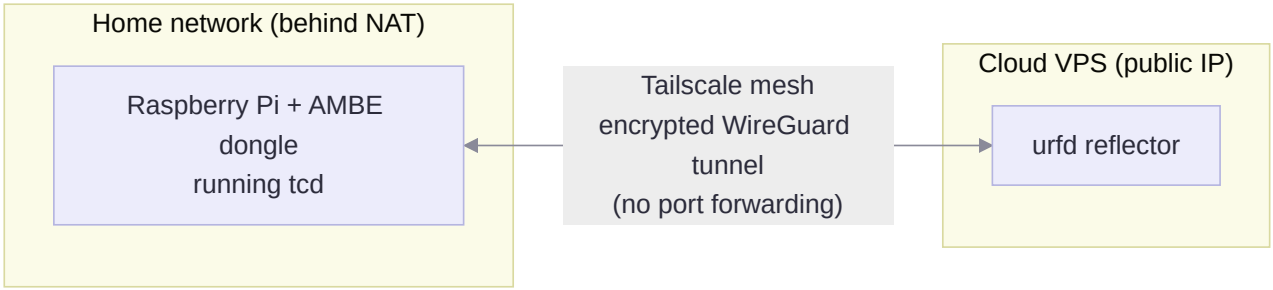
Connecting the distributed pieces with Tailscale

A URF deployment is often not a single machine. The reflector commonly lives on a public cloud server while the AMBE transcoder, a gateway, or a bank of hotspots live at the operator’s home, behind a NAT router with no public address of their own. Getting those pieces to reach each other securely, without opening holes in a home firewall, is exactly the problem Tailscale solves — and it fits URF’s architecture unusually well.

Tailscale is a mesh VPN built on the WireGuard protocol. Instead of funnelling everything through one central VPN server, it gives every device you enrol a stable private address on a virtual private network of your own — a “tailnet” — and lets those devices reach each other directly through encrypted tunnels, handling the NAT traversal automatically so you never forward a port. Membership is tied to an identity login, so only your own machines join the network.

For a URF reflector the natural application is the **remote transcoder**. Recall that the transcoder is the TCP *client* — it dials into the reflector. That means you can run `tcd`, with the AMBE dongle physically plugged in, on a

small machine at home and have it connect across the tailnet to the cloud reflector using the reflector's private Tailscale address, with nothing at all exposed to the public internet on the transcoder's side.



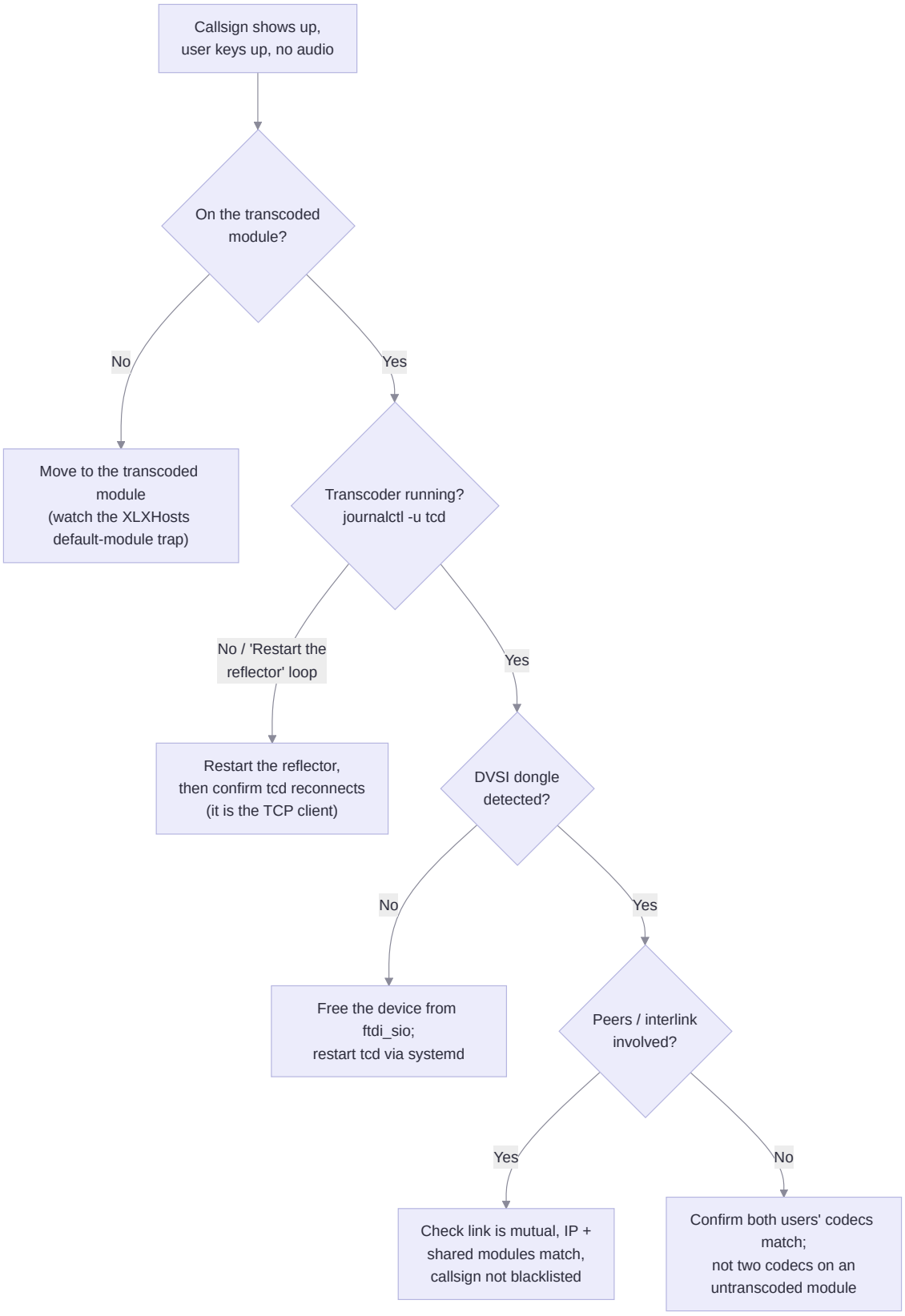
The same pattern connects home gateways and hotspots to a cloud reflector, and it gives you private SSH access to the server for administration without leaving SSH open to the whole internet. It is worth being clear that this is ordinary good networking practice rather than a URF feature — `urfd` knows nothing about Tailscale, and would work identically over any private network or a plain public IP. But the reflector's clean split between a public-facing server and a dial-in transcoder makes Tailscale a natural fit, which is why so many distributed digital-voice setups lean on it.

Chapter 15 — When Things Go Wrong

The failure modes of a URF reflector are, thankfully, a short and knowable list, and almost all of them present as the same symptom: someone connects, their callsign shows up, they key up — and no one hears them. The trick is knowing the handful of causes.

The most serious is **transcoder failure**. There is a well-documented pathology, recorded in the XLX issue tracker that URF inherits its architecture from, in which a dead or degraded transcoder causes "only the D-Star headers" to be sent "while the actual DV stream is missing" — the connection works, the callsign appears, but there is silence. The fix is to restart the transcoder and confirm the DVSI device is present. Closely related is the simpler case of a **user on the wrong module**: if two users are on an untranscoded module using different codecs, or a user has landed on a module that isn't the one carrying the transcoded bridge, they will hear nothing, exactly as the documentation warns. This is most often the "XLXHosts default-module trap" described in Chapter 8 — the host file quietly points at the wrong module — and the fix is to select the transcoded module explicitly. Then there is the **hardware conflict**: if the DVSI dongle isn't found, the culprit is usually the `ftdi_sio` kernel driver holding the device, which the systemd service normally unbinds for you. Peers that refuse to link are almost always a case of the link not being mutual, or an IP or module mismatch, or a blacklisted callsign. And a reflector that no one can find is a discovery problem — check that its DNS name is published, that it appears in the host files, and that its DHT bootstrap node is set.

The same logic as a decision tree, for the silent-audio symptom:


Callsign shows up, user keys up, no audio
On the transcoded module?
No
Yes
Move to the transcoded module (watch the XLXHosts default-module trap)
Transcoder running? journalctl -u tcd
No / 'Restart the reflector' loop
Yes
Restart the reflector, then confirm tcd reconnects (it is the TCP client)
DVSI dongle detected?
No
Yes
Free the device from ftdi_sio; restart tcd via systemd
Peers / interlink involved?
Yes
No
Check link is mutual, IP + shared modules match, callsign not blacklisted
Confirm both users' codecs match; not two codecs on an untranscoded module

Chapter 16 — Performance, Sizing, and Hardening

Honesty compels a plain statement here: there are no published, quantified performance figures for `urfd`. Neither the documentation nor any operator account I could find gives a specific CPU, memory, or bandwidth number. What the documentation does say is that a reflector "requires a 24/7 internet connection which can support up to three transcoded modules and up to 23 more untranscoded modules," and it warns that the G3 (Icom terminal) protocol in particular "requires significantly higher connection resources than any other mode," to the point that it can be left out entirely. Whether a Raspberry Pi is sufficient is genuinely unsettled — the clearest public data point is an operator asking, in a forum, whether it would run on a Pi 3B+, with no answer — while at the other end of the spectrum one operator, W0CHP, runs his on a proper Debian server behind a pfSense firewall on a gigabit fiber connection. The transcoder's software-vocoder path is explicitly built for ARM, which is at least a strong hint that a Pi is viable for the software side.

Estimating bandwidth and CPU from first principles

The sources decline to give a number, but you do not have to fly blind: the bitrates in Chapter 19 are enough to bound the problem, and the arithmetic is worth doing so capacity planning is estimation rather than guesswork. Treat these as engineering estimates, not measurements.

A digital-voice stream on the wire is small. The codecs run around 3,600–9,600 bits per second of payload, but what actually leaves the server is the mode's network frame plus IP/UDP overhead. A useful rule of thumb is to budget on the order of **~25 kbit/s per active stream per direction** once packet headers (roughly 28–40 bytes of IP+UDP on a 20 ms frame) are counted — call it ~3 kB/s. The key multiplier is that a reflector *fans out*: one talker keying a module with N listeners sends the audio out N times. So a single busy module with, say, 30 connected clients costs on the order of 30 × 25 kbit/s ≈ **0.75 Mbit/s** outbound while someone is talking, and only one person talks at a time per module. Even a reflector with several active modules and a few hundred total clients lands in the low single-digit megabits per second — which is why a modest VPS on a 1 Gbit/s port is never the bottleneck. What *does* scale badly is a large fan-out of many simultaneously-active modules, and the G3 terminal protocol, which the documentation singles out as unusually heavy; if bandwidth ever matters, those are the two dials.

CPU splits cleanly in two. The reflector's own job — receiving, matching, and copying packets — is light integer work that a single modern core handles for hundreds of clients; memory likewise sits in the tens of megabytes. The cost that actually matters is **transcoding**, and there the hardware/software split from Chapters 6 and 9 is the whole story: AMBE/AMBE+2 on a DVSI chip costs the host almost nothing (the chip does the DSP), whereas software Codec2, IMBE, and software AMBE+2 each run a vocoder instance per direction per transcoded module on the CPU. Since only three modules can be transcoded at once, the ceiling is bounded: a handful of software vocoder instances, comfortably within a small VPS's two cores and squarely in the range an ARM board was designed for. The practical sizing conclusion matches operator experience: one or two vCPUs and a gigabyte or two of RAM run a transcoding multi-mode reflector, the network is rarely the limit, and the only genuine unknowns — because no one has published them — are the extremes of very high client counts and the G3 protocol.

Security and hardening

On security, it is important to be clear about what the project does and does not provide. What it provides is the application-level gatekeeper described in Chapter 8 — the callsign blacklist and whitelist, with a whitelist being the way to make a reflector private. What it does not provide, and what therefore falls to the operator as ordinary Linux server hygiene, is everything else: a firewall configured to deny by default and open only the specific protocol ports actually in use plus SSH, key-only SSH login, and a tool like fail2ban to blunt brute-force attempts. One caveat deserves emphasis so operators don't over-trust their defenses: fail2ban and similar tools protect SSH and the web dashboard, but they do nothing for the open UDP reflector ports, which — because UDP has no handshake — are inherently exposed to spoofed-source floods.

Because fail2ban can't help there, it is worth naming what actually can. The single most effective and URF-specific measure is simply **not opening ports for modes you don't run** — the documentation itself notes you needn't open ports for G3, USRP, or Brandmeister if they are disabled, and every closed port is one fewer spoofable target. Beyond that, the mitigations are the ordinary ones for exposed UDP services: rate-limit per-source packet rates at the firewall (an `nftables` or `iptables` `limit` / `hashlimit` rule on each mode's port caps how fast a single source can flood you), drop obviously bogus source addresses with reverse-path filtering, and — if you are ever the target of a real volumetric flood — lean on upstream protection, since a home or small-VPS uplink cannot absorb one no matter how the host is configured. This is why several operators put the reflector behind a proper firewall appliance (W0CHP's pfSense setup is the documented example) or a provider that offers DDoS scrubbing. The honest summary is that URF's built-in security is a callsign gate, not a network defense, and the network defense is yours to build.

Time, logs, and disk

A few pieces of routine hygiene keep a 24/7 reflector healthy and are easy to forget because the software never complains about them. Keep the clock disciplined with **NTP** (`systemd-timesyncd` or `chrony`): the status file, the DHT record, and every dashboard "last heard" timestamp are stamped in UTC from the host clock, and TDMA-style modes are happier on a machine whose time is right. Mind the **logs and the status file on disk**: `urfd` logs to the systemd journal, so set a sane cap with `SystemMaxUse=` in `journald.conf` (or `journalctl --vacuum-time=`) rather than letting the journal grow unbounded on a small VPS, and remember that the XML status file is rewritten *every ten seconds* — it does not grow, but it is written constantly, so put it on normal storage and don't point `XmlPath` at anything with limited write endurance or a tiny `tmpfs`. If you enable the JSON report (Chapter 21) it behaves the same way.

IPv6, monitoring, and updates

`urfd` is dual-stack. If your host has IPv6 you set both an `IPv4Binding` and an `IPv6Binding` in the `[IP Addresses]` section (use `0.0.0.0` and `::` to listen on all addresses), publish both an A and an AAAA record (Chapter 22), and confirm your firewall's v6 rules mirror its v4 rules — a common mistake is hardening IPv4 thoroughly while leaving IPv6 wide open. For **monitoring**, the cleanest signals are already there: watch that the `urfd` and `tcd` services stay active (a `systemd` service with `Restart=on-failure`, or a simple external check that alerts if either drops), and — because the killer failure is a silent transcoder while the reflector stays up — treat the XML or JSON status file as your health probe. A tiny cron or systemd-timer script that reads the status

file, checks it was written in the last minute, and alerts you if it goes stale or if `tcd` is down catches the exact "callsigns but no audio" condition before your users report it; Chapter 21 shows how to parse that file.

Updating is, by design, painless: because your configuration lives in copies rather than in tracked files, you pull the new code, rebuild with `make`, and reinstall. **Back up** the small set of files you edited — the reflector's `.ini`, its blacklist, whitelist, interlink, terminal, and service files, the transcoder's configuration (keeping its module list in sync with the reflector's), and the dashboard's configuration — and preserve the side-by-side directory layout that the transcoder's symbolic links depend on. Because `urfd` has no tagged releases, record the **git commit hash** you built from alongside those files (`git rev-parse HEAD`); it is the only reliable way to answer "which version am I running," and it is what makes a restore reproducible. A restore, then, is: clone at the recorded commit, drop your saved config copies back in the side-by-side layout, `make`, `inicheck`, install — and run the Chapter 14 test procedure before trusting it.

Chapter 17 — The Law (United States)

Running a reflector puts you squarely inside the amateur regulations, and in the United States those are FCC Part 97. This is a factual summary, not legal advice, and operators elsewhere are under their own countries' rules. Two provisions matter most. The first is station identification, under §97.119, which requires each station to transmit its assigned callsign "at the end of each communication, and at least every 10 minutes during a communication." In the reflector context, this duty falls on each transmitting radio station — the hotspot, repeater, or user's radio — rather than on the reflector as an internet relay. The second, and the one that shapes M17 in particular, is the prohibition on encryption under §97.113(a)(4), which bars "messages encoded for the purpose of obscuring their meaning." The FCC has affirmed this prohibition, dismissing a petition that sought to relax it.

This is exactly why M17's two security features have such different legal standing. M17's AES option is genuine encryption — a block cipher that obscures the meaning of the transmission — and so, on a US station, it runs afoul of the encryption prohibition. But M17's other feature, ECDSA signing, does something legally different: as Bruce Perens has argued, a cryptographic signature "leaves the message text in the clear, and adds authenticating data" — it proves who sent a transmission without hiding what was said, and so it is defensible under a rule that only forbids obscuring meaning. The practical takeaway for a US reflector operator is to treat M17 AES as off-limits and M17 signing as the compliant way to get authentication.

A closing note on jurisdiction: everything above is United States law, and while its broad strokes are widely shared, the details are not universal. The prohibition on obscuring-encryption is in fact a treaty-level principle — ITU Radio Regulation 25.2A provides that transmissions between amateur stations of different countries "shall not be encoded for the purpose of obscuring their meaning," with the same satellite-control exception the US rule carries — and national licences transpose it. The United Kingdom's Ofcom licence, for example, forbids transmissions "encrypted for the purpose of obscuring their meaning" and requires that "the call sign is transmitted as frequently as is practicable" in a form "consistent with the mode of operation." So the M17-AES-versus-signing distinction and the identification duty hold in spirit almost everywhere, but the exact wording, intervals, and any local exceptions belong to your own national regulator, and an operator outside the US should read their own licence terms rather than assume the American rules apply unchanged.

Chapter 18 — Choosing What to Run

A few decisions recur for anyone standing up a reflector, and the research points to reasonably clear answers. On the choice between the two source repositories, `n7tae/urfd` and `nostar/urfd` are so close as to be interchangeable at the reflector level; the meaningful difference is in their transcoders, where the `nostar` copy adds the software-only mode that lets you transcode everything except D-STAR without buying a DVSI dongle. So the rule is: choose the `nostar` stack if you want multi-mode transcoding on the cheap and don’t need D-STAR, and choose the `n7tae` upstream for the reference implementation with full hardware AMBE quality. On the broader choice between URF, XLX, and mrefd: URF is the answer when you want many modes including M17; classic XLX still suits an established D-STAR-and-DMR network with no M17 ambitions; and mrefd is the lean choice for a pure-M17 network, needing no transcoder hardware at all since M17 is natively Codec2. And on hardware, a single three-channel AMBE-3003 is more efficient than several single-channel 3000s for a busy reflector, while the hardware-versus-software question comes down to that one immovable fact: software can transcode M17, P25, and (with the right option) AMBE+2, but D-STAR’s AMBE always needs the real chip.

Chapter 19 — The Six Modes and Their Codecs, Compared

The guide has explained each mode where it came up, but a reference ought to lay them side by side, because the differences between them are exactly what makes transcoding necessary and hard. The six modes a URF reflector carries divide along a few axes: how they modulate the radio signal, which codec they use and at what bitrate, how they gate channel access, how stations are identified, and whether the standard is open or proprietary.

Mode	Modulation / bandwidth	Voice codec (bitrate)	Access code	Station ID	Standard
D-STAR	GMSK, ~6.25 kHz (4800 bit/s on air)	AMBE, 3600 bit/s (2400 voice + 1200 FEC)	none (callsign routing)	callsign + US-Trust registration	JARL — open protocol, closed codec
DMR	4FSK, 12.5 kHz two-slot TDMA (9600 bit/s)	AMBE+2, ~2450 bit/s voice	Color Code (0–15)	DMR ID via RadioID	ETSI — open standard, closed codec
Fusion (C4FM)	C4FM/4FSK, 12.5 kHz (9600 bit/s)	AMBE+2	DG-ID (00–99)	callsign (no registry)	Yaesu — open spec, closed codec
P25 Phase 1	C4FM, 12.5 kHz (9600 bit/s)	IMBE, 7200 bit/s (4400 voice + 2800 FEC)	NAC (3 hex digits)	operator-assigned	TIA-102 — open standard, closed codec
NXDN	4FSK, 12.5 / 6.25 kHz (4800 / 2400 bit/s)	AMBE+2	RAN (0–63)	operator-assigned	Icom + Kenwood — open standard, closed codec
M17	4FSK (h=1/3), 9 kHz (9600 bit/s)	Codec2, 3200 bit/s	CAN (0–15)	callsign carried in the signal	fully open (GPL)

The striking thing about that table is how much the modes have in common at the radio layer — four of the six are a flavour of 4FSK running at 9600 bits per second in a 12.5 kHz channel — and how completely they diverge at the codec layer, which is the layer that actually matters for interoperability. That divergence is the whole reason a reflector needs a transcoder.

The FEC math, briefly

Two of the bitrates in that table are split into “voice + FEC,” and it is worth a paragraph on what that means, because it explains why digital voice fails the abrupt way it does and why the on-air rate is higher than the codec rate. **Forward error correction** adds redundant bits so a receiver can *correct* errors without asking for a retransmission — essential on a radio link where there is no time to ask again. The two classic codes named in the sources are the ones these modes lean on. A **Hamming code** adds a few parity bits computed across the data bits so that any single-bit error can be located and flipped back (a Hamming(7,4) code, for instance, protects 4 data bits with 3 parity bits). A **Golay code** is stronger: the binary Golay(23,12) — often extended to (24,12), i.e. exactly half data, half protection — encodes 12 data bits into 23 or 24 and can correct up to three bit-errors in that block, which is why it guards the most important bits of a voice frame (the codec’s coarse parameters, where an error is most audible).

The arithmetic then falls out of the table. D-STAR’s AMBE frame is 3,600 bit/s total: 2,400 bit/s of actual codec payload wrapped in 1,200 bit/s of FEC and framing — a 2:1 protection ratio. P25’s IMBE is 7,200 bit/s: 4,400 of voice under 2,800 of error correction. The general shape is that roughly a quarter to a third of what a digital-voice mode transmits is not voice at all but the redundancy that keeps the voice intelligible right up to the “digital cliff”

— the point where errors finally outrun the code's ability to correct them and the audio drops to silence rather than fading gently like analog. This also quietly reinforces the transcoding story: FEC protects a codec's *bits* in transit, but it does nothing to bridge two *different* codecs, which is why a transcoder still has to decode all the way to PCM audio and re-encode — the subject of the quality cost in Chapter 28.

The four codecs themselves are worth their own comparison, because the pattern in them is the story of amateur digital voice: three proprietary codecs from one company, and one open challenger.

Codec	Owner	Used by	Bitrate	Patent status
IMBE	DVSI	P25 Phase 1	7200 bit/s (4400 + 2800 FEC)	**expired**
AMBE	DVSI	D-STAR, early Fusion	2–9.6 kbit/s (3600 for D-STAR)	1997 patent **expired** (~2017 for the D-STAR variant)
AMBE+2	DVSI	DMR, Fusion, NXDN, P25 Phase 2	~3600 bit/s (2450 voice)	still under DVSI license (a remaining patent is estimated to run to ~2028 — *as of mid-2026*)
Codec2	open, David Rowe VK5DGR	M17, FreeDV	3200 / 1600 (and lower modes)	**patent-free, LGPL**

Codec2 is the outlier and the point: Wikipedia notes it "uses half the bandwidth of Advanced Multi-Band Excitation to encode speech with similar quality," and it is patent-free and open. That single fact — an open codec that is good enough — is what made M17 possible, and it is why a URF reflector can do M17's transcoding entirely in software while D-STAR's still needs a chip. The patent history behind all of this is important enough to get its own chapter (Chapter 24).

Chapter 20 — Bridging Beyond URF: DVSwitch and the Wider Network

A URF reflector's built-in transcoder solves one problem beautifully — letting users of different modes talk to each other within one reflector — but it does not, and is not meant to, connect that reflector to the rest of the digital-voice universe. It cannot follow a specific Brandmeister talkgroup, it cannot link to an XLX reflector, and it does not reach EchoLink or arbitrary AllStar hubs. For all of that, operators reach for a separate and older toolkit called **DVSwitch**, and understanding it is essential to understanding how real multi-mode systems — including complex ones that bridge many networks — are actually built.

DVSwitch is a suite of small programs that together plumb one network to another. The two central pieces are **Analog_Bridge** and **MMDVM_Bridge**. MMDVM_Bridge is the network client: it logs into a digital network — a Brandmeister or TGIF talkgroup, an XLX reflector, a YSF room — exactly as a hotspot would, and it

“decomposes each network protocol and creating an agnostic version of the stream,” a mode-neutral form called TLV. Analog_Bridge is the codec engine and the analog leg: it takes that stream and transcodes the encoded audio to and from plain PCM, moving the PCM over a simple UDP protocol called **USRP** — “8KHZ signed 16 bit PCM samples,” in the documentation’s words — which is the same USRP that AllStar and URF itself speak. Analog_Bridge does its vocoding with an AMBE dongle, a networked dongle served by PA7LIM’s **AMBEServer**, or a software vocoder.

Put a pair of these together and you have a cross-mode bridge. A canonical chain runs: network A ↔ MMDVM_Bridge ↔ (TLV) ↔ Analog_Bridge ↔ (USRP PCM) ↔ Analog_Bridge ↔ (TLV) ↔ MMDVM_Bridge ↔ network B, with the mode-neutral PCM in the middle as the meeting point. When the two ends use the same codec the encoded audio can be preserved untouched; when they differ, the two Analog_Bridge instances decode to PCM and re-encode — which, exactly as with a reflector’s transcoder, is lossy. Where a system needs to fan one audio feed out to many destinations, a dedicated USRP relay (DVSwitch’s Analog_Reflector, or URF’s own integrated USRP reflector) does the one-to-many job that a single Analog_Bridge cannot.

The practical division of labour is clean once you see it. Use URF’s own transcoder to let a D-STAR user and a DMR user *on the same reflector* hear each other — it is integrated, per-module, and reliable. Use DVSwitch to connect that reflector to *something outside it*: to inject a specific Brandmeister talkgroup into a module, to bridge to an XLX reflector that URF cannot interlink with, or to reach EchoLink and analog through AllStar. A large multi-network system is typically a URF reflector at the center with a small fleet of MMDVM_Bridge-and-Analog_Bridge pairs hanging off its modules, each pair reaching out to one external network — which is precisely the architecture of the elaborate six-mode bridges some operators build.

The bridging fabric: USRP is the lingua franca

Everything below rides one simple idea: the analog and VoIP world speaks **USRP** — “8 kHz signed 16-bit PCM samples” over UDP — and DVSwitch’s **Analog_Bridge** is the universal adapter between that plain-PCM world and the coded digital modes. URF has a real advantage here, because it carries an *integrated* USRP client of its own (Chapter 4): a URF module can source and sink USRP directly, so it can point straight at an AllStar node or an Analog_Bridge. Recall the one quirk from Chapter 4 — a USRP link has no connect/disconnect protocol, so it is fixed in the configuration at startup and must point at a transcoded module to be useful. With that in mind, here is how each of the common analog and VoIP networks is reached.

Reaching AllStar

AllStarLink is a VoIP network of amateur nodes built on the Asterisk PBX and its `app_rpt` application; every node has a number, and users link nodes with DTMF — `*3<node>` to connect two-way (transceive), `*2<node>` to connect monitor-only, `*1<node>` to drop one node, `*76` to drop all, and `*70` / `*75` for status. Because AllStar and URF both speak USRP, they join naturally. The usual arrangement is a dedicated **private AllStar node** (operators often number it 1999) whose channel is the USRP driver — in `rpt.conf`, `rxchannel = USRP/127.0.0.1:34001:32001` — with the URF USRP client (or a DVSwitch Analog_Bridge) mirroring those ports on the other end (`[USRP] rxPort = 34001`, `txPort = 32001`). Once the link is up, AllStar users reach

the digital side by connecting that private node, and digital users hear AllStar because its audio lands on the transcoded module.

Reaching EchoLink

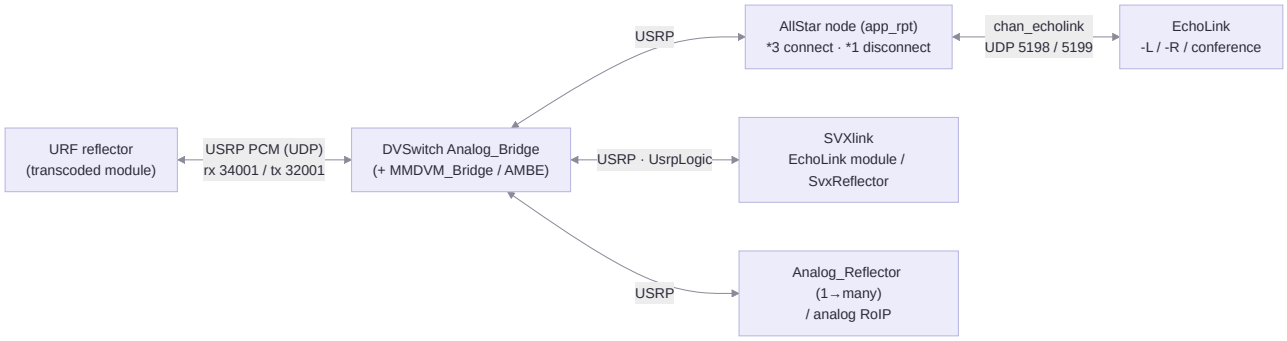
EchoLink (Jonathan Taylor, K1RFD) is the long-established system that links licensed amateurs' computers, phones, and RF nodes over the internet. It uses UDP `5198` (RTP audio) and `5199` (RTCP control), with a central directory (commonly cited on TCP `5200`) that **validates each user's callsign** before first use; nodes come as plain users, `-L` simplex links, `-R` repeaters, and conference servers (marked with a leading `*`). No reflector in this family speaks EchoLink directly — the standard route is *through AllStar*, which has a built-in EchoLink channel driver (`chan_echolink`, set up in `echolink.conf` with your call, EchoLink password, node number, and the `astnode` it attaches to). AllStar presents EchoLink nodes as if they were AllStar nodes using a node-number offset: it pads the EchoLink number to six digits and prefixes a `3`, so EchoLink node 1234 becomes `3001234` and you connect with `*3 3001234`. The full chain to put EchoLink into a URF module is therefore *URF ↔ USRP ↔ Analog_Bridge ↔ AllStar (chan_echolink) ↔ EchoLink*. In practice you run (or borrow) one AllStar node that carries both the USRP bridge to your reflector and an EchoLink registration, and it becomes the meeting point.

Reaching SVXlink and SvxReflector

SVXlink (Tobias Blomberg, SM0SVX) is an open-source Linux repeater controller widely used to run repeaters and simplex links. It includes an **EchoLink module** (so an SVXlink node can itself be an EchoLink node) and **SvxReflector**, its own reflector network for linking SVXlink systems together. Bridging it to digital voice uses the same USRP fabric through SVXlink's **UsrpLogic**, which connects SVXlink to a DVSwitch Analog_Bridge over USRP; Analog_Bridge (with MMDVM_Bridge, and `md380_emu` or a dongle for AMBE where a proprietary codec is on the far side) carries it onward to the reflector. A commonly-used community build (F5VMR's SVXlink-with-UsrpLogic) packages this, and because SVXlink has no built-in way to pick digital destinations, operators drive the connection from the DVSwitch Mobile app or a fixed configuration. Through that one UsrpLogic link you can join an SVXlink node — and, via its EchoLink module or a whole SvxReflector network — to your URF reflector.

Analog, RoIP, and the general pattern

The through-line is that anything which can source or sink USRP PCM can be joined to a URF reflector: an AllStar node, a plain analog radio interface, SVXlink via UsrpLogic, or a bank of them. Where one feed must fan out to many destinations, DVSwitch's **Analog_Reflector** — or URF's own integrated USRP reflector — does the one-to-many job a single Analog_Bridge cannot. The cost is always the same and worth keeping in mind: every hop that crosses a codec boundary is a transcode, with the intelligibility and latency penalty from Chapter 28, so bridge with purpose and keep the chains shallow.



Chapter 21 — The Reflector’s Status File, and Building Your Own Tools

Anyone who wants to monitor a reflector, build a custom dashboard, or feed its activity into some other system needs to know what data the reflector actually exposes, and `urfd` exposes a surprising amount — this was read directly from its source.

The primary output is an **XML status file**. Its location is set by the `XmlPath` key in the configuration (the shipped default is the legacy path `/var/log/xlxd.xml`), it is mandatory — the reflector will not start without it — and it is completely rewritten every ten seconds. Its structure is straightforward. After a `<Version>` element, it contains three sections whose tag names are built from the reflector’s callsign patched to begin with “XLX”: a linked-peers section, a linked-nodes section, and a heard-users section. Each connected peer reflector appears as a `<PEER>` block carrying its callsign, IP address, the modules it shares, its protocol, and ISO-8601 UTC connect and last-heard timestamps. Each linked node — a repeater or hotspot — appears as a `<NODE>` block with its callsign, IP, single linked module, protocol, and the same timestamps. And each recently-heard station appears as a `<STATION>` block with the user’s callsign, the node they came in through, the module they were heard on, the peer they arrived via, and a last-heard time. This is the file the built-in PHP dashboard parses, and it is the natural thing for any external monitoring to read.

There is a second, richer output that most operators never turn on: a **JSON report**. Its path is set by the `JsonPath` key, but that key ships commented out and labelled “for future development,” so by default no JSON is written. Uncomment it and the reflector emits, every ten seconds, a compact JSON document with four top-level keys — a `Configure` object containing the running configuration, and `Peers`, `Clients`, and `Users` arrays. It is worth knowing that the JSON is not a one-to-one twin of the XML: its `Clients` array includes every client rather than only the repeater-style nodes the XML lists, so for machine consumption the JSON is often the better source once enabled.

A worked example: reading the status file

Because the guide promises tools and not just formats, here is a complete, dependency-free reader. It parses whichever source you have — the always-on XML or the richer JSON — prints a one-line “who’s connected and who was last heard” summary, and, crucially, exits non-zero if the file is stale (older than 60 seconds) so it can

double as the health probe described in Chapter 16. Drop it in a cron job or a systemd timer and you have monitoring that catches the silent-transcoder failure before your users do.

```
#!/usr/bin/env python3
# urfstat.py — summarise a urfd status file; exit 2 if stale.
# Usage: urfstat.py /var/log/xlxd.xml   or   urfstat.py /path/urfd.json
import sys, os, time, json
import xml.etree.ElementTree as ET

path = sys.argv[1]
age  = time.time() - os.path.getmtime(path)          # file is rewritten every 10s
peers = nodes = users = 0
last  = []

if path.endswith(".json"):
    d = json.load(open(path))
    peers = len(d.get("Peers", []))
    nodes = len(d.get("Clients", []))            # JSON lists every client
    for u in d.get("Users", [])[:5]:
        last.append(f"{u.get('Callsign','?')}@{u.get('Module','?')}")
else:
    root = ET.parse(path).getroot()
    peers = len(root.findall(".//PEER"))
    nodes = len(root.findall(".//NODE"))
    for s in root.findall(".//STATION")[:5]:
        cs  = s.findtext("Callsign", "?")
        mod = s.findtext("Module", "?")
        last.append(f"{cs}@{mod}")

summary = f"peers={peers} nodes={nodes} last_heard={', '.join(last) or '(none)'}"
print(f"{summary} age={age:.0f}s")
if age > 60:
    print("STALE: status not updated in 60s — check urfd/tcd",
          file=sys.stderr)
    sys.exit(2)
```

The same two files are all the built-in dashboard itself consumes, so anything the dashboard shows you can compute yourself — a Grafana feed, a Discord notifier, a network-wide “who’s talking” board. And for a multi-reflector view without touching any single server, the third source below is cleaner still.

The third and most modern way to read a reflector’s state is over **Ham-DHT** itself. A DHT-enabled reflector publishes a signed configuration record — its callsign, addresses, module and transcoded-module lists, every per-protocol port, its auto-link modules and reflector IDs, its YSF frequencies, and per-module descriptions — which the `ham-dht-tools` utilities can fetch from anywhere without touching the reflector directly. For building a network-wide map or a multi-reflector monitor, that DHT record is the cleanest data source of all.

Chapter 22 — Deployment: DNS, HTTPS, and Watching the Traffic

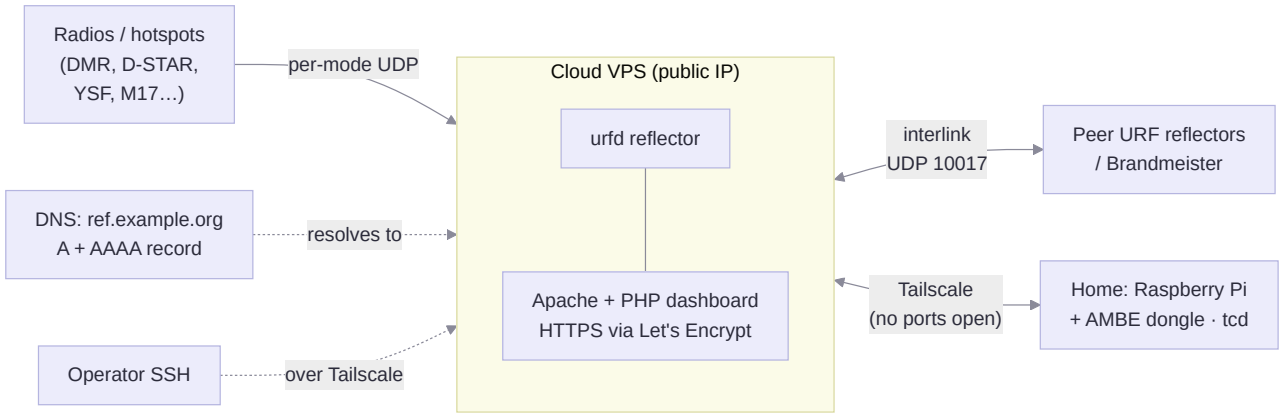
Turning a working reflector into a properly deployed public service takes a few steps the software itself does not perform, and a complete reference should cover them even though some are ordinary web-hosting practice rather than anything URF-specific.

It starts with **DNS**. The reflector's documentation notes that "the public IP addresses should have a DNS record which must be published in the common host files," which in practice means creating an A record — and, if the host has IPv6, an AAAA record — that maps a stable hostname like `ref.example.org` to the server's public address. That hostname, not a bare IP, is what belongs in the host files and the dashboard URL, so that the address can change without breaking every user's configuration.

Next is **HTTPS for the dashboard**. The dashboard is a PHP site served by Apache, and while the reflector's documentation only mentions the optional HTTPS port, serving it securely is standard practice. The usual tool is Let's Encrypt via Certbot: on an Apache host, running `certbot --apache` will, in Certbot's own description, "get a certificate and have Certbot edit your apache configuration automatically to serve it, turning on HTTPS access in a single step," and it renews automatically thereafter. For a more elaborate setup — or to keep the dashboard on a different machine from the reflector daemon, which is entirely possible since the dashboard only needs to read the reflector's generated files — a reverse proxy in front of the dashboard (Apache or nginx) terminates TLS and adds caching and security headers. This last part is general architecture guidance rather than anything the URF documentation prescribes.

Finally, many operators like to **see their traffic** — how many people visit the dashboard, from where — without loading a heavy commercial tracker onto what is a hobby community page. A good fit is a lightweight, privacy-respecting, self-hostable analytics package such as GoatCounter, which describes itself as "easy web analytics, no tracking of personal data," adds only a few kilobytes to a page, uses no cookies, and needs no GDPR consent banner. It is exactly the kind of small, self-hosted addition that suits a reflector dashboard, and it is entirely optional.

A complete deployment, with the pieces of the last several chapters in one place:



Chapter 23 — The Dashboard: Setup, Use, and Access

The dashboard is the reflector's public face — the web page hams open to see who is connected, who is talking, and which module carries which mode. For most users it is the *only* part of a reflector they ever touch directly, and it is the page they leave open and glance at all day, so it deserves its own chapter separate from the plumbing that feeds it (Chapter 21) and the deployment steps that serve it (Chapter 22).

What it is, and how it is set up

The dashboard ships with `urfd` as a small PHP web application, and it is deliberately simple: it does not talk to the reflector directly or hold any state of its own. It only *reads* the files the reflector regenerates every ten seconds — chiefly the XML status file described in Chapter 21 — and renders them as HTML. That design has a pleasant consequence: because the dashboard is just a reader of generated files, it can run on the same server as the reflector or on a completely separate web host that has access to those files, and restarting or rebuilding the reflector never disturbs it.

Setup is the last step of a build (Chapter 14). You copy the dashboard directory into your web server's document root — Apache with PHP is the documented stack — and edit its single configuration file to set the handful of things the reflector itself doesn't know: your sysop **email**, **country**, a free-text **comment** or reflector description, the public **dashboard URL**, and the **calling-home** toggle (which you leave off until you have confirmed a collision-free callsign suffix, per Chapter 12). You then serve it over HTTPS — `certbot --apache` is the one-command route (Chapter 22) — because it is a public web page and there is no reason to serve it in the clear. Nothing about the dashboard needs a database; its "database" is the reflector's own status file.

What it shows, and how operators and users read it

A URF dashboard presents the reflector's live state as a few stacked panels, all built from the status file's three sections. At the top is an identity banner — the reflector's callsign, its description, and often an online indicator. Below that, the panels are:

A **module / room list**, showing each active module A–Z with the operator's description of what it carries ("Multi-mode," "DMR," "M17-only"), so a newcomer can tell at a glance which module to select — this is the antidote to the wrong-module confusion that causes most "no audio" reports. A **linked-nodes list** (repeaters and hotspots) and a **peer-reflectors list**, each showing callsign, the module it is on, its protocol, a masked IP, and connect/last-heard times. And the panel everyone actually watches, the **"Last Heard" list**: the most recent stations to key up, with callsign, the module and mode they came in on, and a timestamp.

The Last Heard list is the reason hams keep the page open. After you transmit, your callsign appearing there — on the module and mode you expect — is the quickest possible confirmation that the reflector heard you and put your audio on the right module. It is also the first diagnostic when something is wrong: if your callsign shows up in Last Heard but no one hears you, you have the classic transcoder-or-wrong-module symptom from Chapter 15, and you have already ruled out "the reflector never received me." Operators use the same panels to watch activity, confirm a peer link actually formed, and spot a hotspot that has landed on the wrong module.

How hams access it

Access is deliberately frictionless: the dashboard is an ordinary web page at the reflector's hostname — for example `https://ref.example.org` — opened in any browser on a computer or phone, with **no login required** to view. That URL is what an operator publishes as the reflector's `DashboardUrl`, what appears next to the reflector in the DVRef registry and host-file listings, and what club members bookmark. Viewing is read-only for the public; there is no web-based administration, so an operator changes the reflector's configuration on the server itself (Chapter 14), not through the dashboard. Because the address that matters is the DNS hostname rather than a bare IP (Chapter 22), the server can move without breaking anyone's bookmark.

One clarification worth making, since it is a common point of confusion: the dashboard is a *view*, not the reflector. It is not how radios connect — hams connect with the per-mode procedures in Chapter 8 and Appendix B — and if the dashboard is down but the reflector is up, traffic still flows; you have simply lost the window into it. Conversely, if you want a different view — a custom page, a phone widget, a network-wide board across several reflectors — you are not limited to the shipped dashboard: it reads the same public files any tool can read, which is exactly what Chapter 21 shows how to do.

A representative URF dashboard layout (illustrative — a live dashboard's styling varies):

URF123 — Multi-mode Reflector ● online · updated 4s ago

Modules: A — Multi-mode | B — DMR | C — M17

LAST HEARD

Callsign	Module	Mode	Last heard (UTC)
W1ABC	A	D-STAR	2026-08-08 17:42:10 ◀ transmitting
K2AS	A	DMR	2026-08-08 17:41:58
VA6MO	C	M17	2026-08-08 17:39:12
G4XYZ	B	DMR	2026-08-08 17:35:47

CONNECTED NODES

Callsign	Module	Protocol	IP	Since (UTC)
W1ABC B	A	DExtra	73.x.x.x	16:02
VA6MO	C	M17	142.x.x.x	15:47

LINKED PEERS

Callsign	Shared modules	Protocol	Since (UTC)
URF587	A	URF interlink	Aug 07 09:14

Chapter 24 — The AMBE Patents and the Open-Codec Movement

To really understand why URF is built the way it is — why it needs a hybrid transcoder, why D-STAR is the awkward one, why M17 exists at all — you have to understand the patent history of the codecs, because that history is the hidden force shaping the whole field.

For most of amateur digital voice's life, the codecs were a closed door. The Multi-Band Excitation family — IMBE, then AMBE, then AMBE+2 — was developed and patented by a single company, Digital Voice Systems, Inc., and using them meant buying DVSI's hardware or licensing their software. This is why D-STAR, DMR, Fusion, P25, and NXDN all depend on hardware vocoder chips, and it is the reason a transcoding reflector historically needed real DVSI dongles for everything. It also sat awkwardly with amateur radio's culture of building and understanding your own equipment: you could not legally write your own software to encode or decode these modes.

That door has been slowly opening as the patents age out. The original 1997 Multi-Band Excitation patent has expired, the IMBE patent used by P25 has expired, and — the point most relevant to reflectors — the specific patents on the AMBE variant used by D-STAR expired around 2017, a fact widely attributed to Bruce Perens's analysis and repeated in the community. That expiry is what made open software decoders like `mbelib` and the MD-380-firmware-based vocoders legally defensible for those older modes, and it is why a URF transcoder can now do some AMBE work in software at all. The important caveat is that AMBE+2 — the newer codec used by DMR, Fusion, and NXDN — is *not* yet free; it remains under DVSI license, with a remaining patent estimated by community analysts to run until roughly 2028 (*as of mid-2026*, and worth re-checking against current filings before relying on it). So the picture today is genuinely mixed: some of the proprietary codecs have fallen into the public domain, and one central one has not.

Into that landscape came M17, and its motivation makes sense only against this backdrop. The M17 Project states its purpose plainly: its voice mode "uses the free and open Codec 2 voice encoder," which means "there are no patents, no royalties, and no licensing or legal barriers to scratch-building your own radio or modifying one you already own," and it frames this as restoring something that "has been missing from the commercially available digital voice modes." M17 is, in other words, a deliberate answer to the closed-codec problem — an attempt to give amateur radio a fully open digital voice mode, codec and protocol alike. URF's role in this story is as the bridge between the two worlds: its hybrid transcoder is precisely the thing that lets the open future (M17, Codec2) talk to the proprietary present (D-STAR, DMR, and the AMBE family), which is why it exists in the shape it does.

Chapter 25 — A Worked Example, and Migrating from XLX

To make all of this concrete, consider a realistic build: a three-module URF reflector called URF123 on a small cloud server, carrying a transcoded multi-mode module plus a couple of single-mode ones, with the AMBE hardware living on a Raspberry Pi at home and reaching the reflector privately over Tailscale.

The heart of it is the reflector's configuration. In `urfd.ini` you would set the identity and modules, and point the transcoder section at the remote Pi:

```
[Names]
Callsign = URF123
Bootstrap = xrf757.openquad.net
DashboardUrl = https://ref.example.org
SysopEmail = you@example.org
Country = US

[IP Addresses]
IPv4Binding = 0.0.0.0
IPv6Binding = ::              ; dual-stack; publish an AAAA record too

[Modules]
Modules = ABC
DescriptionA = Multi-mode (transcoded)
DescriptionB = DMR only
DescriptionC = M17 only

[Transcoder]
Port = 10100
BindingAddress = 0.0.0.0     ; listen for the remote transcoder
Modules = A                  ; module A is the transcoded one

[YSF]
Port = 42000
AutoLinkModule = A           ; Fusion users land on the transcoded module
```

On the Raspberry Pi at home, with the AMBE dongle plugged in, the transcoder's `tcd.ini` simply dials into the reflector across the Tailscale network, and its module list must match the reflector's exactly:

```
Port = 10100
ServerAddress = 100.x.y.z    ; the reflector's private Tailscale address
Modules = A                  ; identical to urfd.ini [Transcoder] Modules
```

With that, module A transcodes among all its modes, modules B and C carry single-mode traffic with no transcoding overhead, nothing but the reflector's own protocol ports is exposed to the public internet, and the AMBE hardware sits safely at home. Validate the reflector config with `./inicheck -q urfd.ini`, install, point a DNS record and an HTTPS certificate at the dashboard, walk the Chapter 14 test procedure, and it is a complete, working, properly-deployed reflector.

For the many operators who already run an XLX reflector, moving to URF deserves its own note, because the relationship is close but not seamless. URF is a genuine superset of XLX, so everything an XLX reflector did, a URF reflector does, plus M17, P25, NXDN, and USRP — nothing is lost by moving. And because URF reports itself as an XLX reflector, it slots into the same dashboards, host files, and registries. The catch is the boundary

from Chapter 1: a URF reflector cannot interlink with an XLX reflector, so you cannot run the two as peers of one network during a gradual transition — they are separate islands. The realistic migration path is therefore to stand up the URF reflector alongside the existing XLX one, ideally reusing the same callsign suffix if the XLX is being retired, move your users and any interlinks over to it, and then decommission the old XLX once traffic has shifted. The configuration itself is different in form — URF uses the clean `urfd.ini` file described in this guide, where older XLX builds compiled settings into headers — so the move is a genuine reconfiguration rather than a drop-in, but the concepts (modules, interlink files, transcoding) carry across directly.

Chapter 26 — The Case For URF: Strengths and Benefits

Having taken the machine apart, it is worth stepping back and asking plainly what URF is *good* at, because a reference guide that only explains mechanics leaves the most important question — is this the right tool? — unanswered.

Its central strength is the obvious one: URF is genuinely universal. A single server speaks every major amateur digital-voice mode at once — D-STAR, DMR, Fusion, P25, NXDN, and M17 — and, with a transcoder attached, lets users of those otherwise-incompatible modes actually converse. For a club whose members are scattered across three or four different radio ecosystems, that is transformative; instead of running separate infrastructure per mode and keeping those communities apart, one reflector holds them together. Layered on top of that is the twenty-six-module structure, which gives an operator real flexibility to run many independent conversations, dedicate rooms to modes or purposes, and grow into the space as activity develops.

URF's second great strength is that it is forward-looking without abandoning the past. It is the software that brought native M17 — the open, patent-free, community-developed mode — into the multi-protocol reflector world, and it bridges M17 to the older proprietary modes rather than walling it off. Because it is a genuine superset of XLX, nothing is lost relative to its predecessor, and it plugs straight into the existing ecosystem of XLX dashboards, host files, and registries. Its hybrid transcoder is a real engineering achievement: it uses proprietary DVSI hardware only where the closed codecs truly require it and runs the open codecs in software, and in the `nostar` software-only configuration a multi-mode reflector can run with no vocoder hardware at all as long as it doesn't need D-STAR. The transcoder can even live on a separate machine, over a reliable TCP link.

Finally, there are the virtues that come from how it is built and licensed. URF is open source under the GPL — unlike Brandmeister's unpublished core server or the licensed, closed IPSC2 — so an operator can read exactly what it does, which is the ethos this whole guide is written in. It is actively maintained by N7TAE on a modern C++ codebase. Its discovery can run over the decentralized Ham-DHT, removing dependence on any single central server. It is dual-stack IPv4/IPv6. Its update model is deliberately clean, keeping your edits in copies so a code update never clobbers your configuration. And it is inexpensive to run: the software path needs no special hardware and a small virtual server is enough, a point developed with real figures in Chapter 29.

Chapter 27 — The Case Against: Limitations and Trade-offs

An honest reference has to be as clear about the weaknesses as the strengths, and URF has real ones — some inherent to what it is trying to do, some particular to how it is built.

The most fundamental limitation is that **transcoding does not scale**. Only three modules can be transcoded at once, a ceiling set by the vocoder hardware, and every transcoded conversation costs channels. A reflector can have twenty-six modules, but only three of them can bridge modes; the rest are single-codec pass-throughs. Bound up with this is a limitation URF cannot solve because no one can: D-STAR's AMBE codec is proprietary and, in practice, requires buying DVSI hardware. You cannot run a fully open, hardware-free reflector that includes D-STAR. The open software path covers M17, P25, and — with a software library — the AMBE+2 modes, but D-STAR always needs the chip.

The transcoder is also a **single point of failure**, and it fails ungracefully. As documented in the inherited XLX issue tracker, when the transcoder dies the reflector keeps forwarding the signalling headers while the actual audio vanishes — so a user connects, sees callsigns appear, keys up, and is met with silence, with no obvious indication of what is wrong. Worse, because the reflector is the TCP server and the transcoder the client, the transcoder cannot recover the link on its own; the operator has to restart the reflector. There is no automatic fallback to same-codec relay when transcoding is unavailable.

Then there is the quality question, which Chapter 28 treats in full but belongs on the ledger here: cross-mode transcoding is *tandem vocoding*, decoding one lossy codec to audio and re-encoding it into another, and this measurably degrades intelligibility. It is the unavoidable price of talking across modes, and it is worse when the software AMBE path is used instead of hardware. Each transcode stage also adds latency.

The remaining limitations are more prosaic but real. URF cannot interlink with the older XLX reflectors, which splits it from a large installed base. Cross-module linking is not supported — you link like letters only. It runs only on systemd Linux, is compiled from source (there is no packaged binary short of a community Docker image), and at least one experienced operator called it "a little more involved" to build than XLX or DVSwitch. Its built-in security is minimal — a callsign blacklist and whitelist and nothing more — so firewalling, rate-limiting, and DDoS resilience are entirely the operator's problem, and its open UDP ports are inherently exposed. It has no tagged releases and no published performance figures, so capacity planning is guesswork (Chapter 16 estimates from first principles, but that is not the same as a benchmark), and because it is a niche project the documentation has gaps and the support is scattered rather than centralized. It leans on external services — the XLX API and RadioID databases — for its ID lookups. P25 and NXDN users cannot freely choose a module from their radios, and the per-reflector DG-ID-to-module mapping on Fusion is a recurring source of confusion. And on the legal front, M17's AES encryption — a headline feature of the mode — is not usable on a US station at all.

None of this makes URF a poor choice; it makes it a *tool with a shape*. For bridging modes it is close to unmatched; for someone who wants a single-mode reflector with minimal fuss and no hardware, a simpler purpose-built server may serve better.

Chapter 28 — Voice Quality and Latency

The quality cost of transcoding is real, measurable, and worth understanding honestly, because it is the one downside a URF operator cannot engineer away — it is inherent in bridging codecs.

Every cross-mode conversation on a URF reflector is an instance of what the literature calls *tandem vocoding*: the audio is decoded from the sender's codec into plain PCM and then re-encoded into the receiver's codec. Each of those steps is lossy. DVSI's own transcoding white paper — self-interested, since DVSI sells the hardware, but corroborated by an independent NTIA study on tandem vocoder configurations — states the mechanism directly: "By their nature, vocoders introduce loss… During the process of transcoding from one vocoder to another, additional degradation occurs," and "in some cascading situations, multiple transcoding stages occur in tandem, which can result in severe loss of voice quality and intelligibility." The white paper puts a number on it from a Modified Rhyme Test: transcoding produced "an average loss of more than 12% in intelligibility," meaning roughly twelve percent fewer words understood. That figure should be read as indicative rather than gospel, but the direction is not in doubt: a transcoded QSO sounds worse than a same-mode one, and chaining multiple transcodes compounds it.

The choice between hardware and software vocoding also affects quality, though less dramatically. KB9MWR's notes observe that software D-STAR AMBE decoding "still sounds like it could use work when compared against the audio processed by the chip," so the DVSI hardware remains the quality benchmark for the AMBE modes. On the open side, Codec2's own author, David Rowe, describes the difference between AMBE+2 and Codec2 as small on typical radio audio — "AMBE is a little better, Codec 2 is a bit clicky or impulsive" — so the M17 path is not a large quality sacrifice. What could not be found anywhere is a rigorous, controlled comparison of the specific `md380` software AMBE+2 vocoder against a DVSI dongle; operator opinion exists but no measured result, so this is left as an honest gap.

Latency is the other cost. No one has published an end-to-end figure for a URF transcoded round trip, but — as with bandwidth — first principles bound it usefully. The pieces are known: each codec works on **20 ms frames**, so framing alone contributes tens of milliseconds at each encode/decode; a transcode adds one decode-to-PCM and one re-encode on top of that; the reflector adds a **jitter buffer** (typically tens to a couple hundred milliseconds, since it must hold packets to smooth out network arrival jitter); and then there is real **network round-trip time** between the talker, the reflector, the transcoder (which may be a separate machine, as in the Tailscale example), and the listener. Adding those up, a same-mode QSO across a reflector is often in the ballpark of 100–200 ms mouth-to-ear, and a transcoded cross-mode contact — with the extra decode/re-encode and, frequently, an extra network hop to a remote transcoder — realistically lands a few hundred milliseconds higher, roughly **~300–500 ms** in a typical cloud-reflector-plus-home-transcoder setup. Treat that as an order-of-magnitude estimate, not a measurement; the DVSI white paper confirms only the direction, noting that using one vocoder end to end "reduces delay." The practical mitigations follow from the mechanism: keep heavily-used single-mode traffic on untranscoded modules where it belongs, reserve the scarce transcoded modules for genuine cross-mode bridging, keep the transcoder near the reflector on the network, and minimize the number of hops a signal makes.

Chapter 29 — Who Should Run One, and What It Costs

The clearest way to decide whether to run a URF reflector is to look at why people actually do. The recurring rationale, stated plainly by the Silvercreek club (W8WKY), is that the fragmentation of digital voice into incompatible standards has held all of them back, and a transcoding reflector fixes that by letting “D-STAR modules, YSF rooms, and DMR talkgroups all talk to each other” with a transmission on one mode received in real time by users of the others. The second rationale, and the reason URF exists as distinct from XLX, is to bridge the open modes to the legacy ones — to bring M17 and P25 into the same room as D-STAR, DMR, and Fusion, which is exactly what the hybrid transcoder was built to do. The third is the regional or community network: QuadNet is the flagship example, anchored by two interconnected URF reflectors, URF307 in Wyoming and URF587 in Alabama, that let operators across D-STAR, DMR, Fusion, and M17 share common channels regardless of their equipment. And the fourth, more modest, is simple consolidation — a club standing up one multi-mode server instead of separate infrastructure for each mode its members happen to own.

From that, the audience sorts itself out. URF is an excellent fit for a club or regional group that spans multiple modes, for anyone who wants to bring M17 into contact with the older modes, and for an operator who enjoys running infrastructure and wants to understand it. It is a poorer fit for someone who only cares about a single mode and wants the least possible fuss, or who is unwilling to buy DVSI hardware but needs D-STAR, or who needs a supported, packaged appliance rather than a compile-it-yourself project.

The costs are modest and worth stating concretely, with the caveat that prices change (*figures below are as of mid-2026*). On the hardware side, a single-channel AMBE dongle — a DVMEGA DVstick-30 or an NW Digital Radio ThumbDV — runs around one hundred US dollars (roughly ninety-four pounds) and transcodes one stream; the three-channel AMBE-3003 device needed for real multi-mode transcoding, sold as the DVstick-33, is more, on the order of three hundred-plus Canadian dollars. If you take the `nostar` software-only path and don’t need D-STAR, you can skip the dongle entirely. On the hosting side, a URF reflector is light enough to run on a small virtual server: entry plans with one virtual CPU and a gigabyte or two of memory run about five to six US dollars a month from providers like Linode, Vultr, or DigitalOcean, and cheaper still — around three to five euros — from Hetzner, according to a 2026 provider comparison. So the realistic entry cost is a few dollars a month for a software-only multi-mode reflector, or that plus a one-time hardware purchase of roughly one hundred to three hundred-plus dollars if you want hardware AMBE and D-STAR.

Chapter 30 — Frequently Asked Questions

Do I need a radio to use a URF reflector? No. Software clients — DroidStar for most modes, mvoice for M17, BlueDV for the AMBE modes — let a computer or phone connect directly, which is also the easiest way to test your own reflector.

Do I need a DVSI dongle to run one? Only if you want D-STAR or the highest-quality AMBE transcoding. The software path (the `nostar` transcoder) covers M17, P25, and USRP with no hardware, and AMBE+2 for DMR and Fusion with a software library; D-STAR’s AMBE always needs the chip.

Is software transcoding actually possible, then? Yes. The `nostar` transcoder's software-only mode runs a full multi-mode reflector with no DVSI hardware at all — M17, P25, and USRP in software, and AMBE+2 for DMR, Fusion, and NXDN with an additional option. The single exception is D-STAR's AMBE, which cannot be done in open software and always requires a dongle.

How many modules can be transcoded? With hardware, one module on a single-channel AMBE-3000 dongle and up to three on a three-channel AMBE-3003; the reflector is built to expect either one or three transcoded modules. In the software-only mode there is no DVSI channel to run out of, but the project is still designed and validated around that same one-or-three convention, so the practical, documented ceiling is three unless you verify a higher number on your own build.

How much bandwidth and CPU does it need? No one has published a benchmark, but the arithmetic in Chapter 16 puts a busy single module at well under a megabit per second and the whole server, even with several active modules, in the low single-digit megabits — the network is rarely the limit. CPU is light except for software transcoding, which is bounded because only three modules can transcode at once; one or two vCPUs and a gigabyte or two of RAM is the realistic sizing.

How do I know it's actually working? Compiling and starting is not the same as working. Walk the per-mode test procedure in Chapter 14 with software clients — prove single-mode audio, then transcoding in both directions, then discovery — and deliberately run the failure drill so you recognize the silent-transcoder symptom live.

Why do some setups run separate YSF, NXDN, and P25 reflectors alongside the URF reflector? Not because URF requires it — this is a common misconception. A URF reflector already contains an integrated P25 reflector, an integrated NXDN reflector, and its own YSF Master offering twenty-six rooms, so it does not need separate single-mode reflectors for those. When a deployment does run separate standalone reflectors, it is a deliberate architectural choice: a standalone reflector gets its own numeric ID listed in the public host files that hotspots browse, it can offer its own independent set of rooms, and it can be bridged into URF in a particular way. The reason is discoverability or bridging flexibility, never a dependency of URF itself.

What is Tailscale, and why do reflector operators use it? Tailscale is a mesh VPN (built on WireGuard) that gives your machines private, stable addresses on a network of your own and lets them reach each other through encrypted tunnels with no port-forwarding. Its natural use with URF is running the transcoder — or a home gateway or hotspots — on a machine behind your home router and connecting it privately to a cloud-hosted reflector, exposing nothing to the public internet. It is explained in full in Chapter 14.

Can a URF reflector link to an XLX reflector? No. URF links to other URF reflectors and to Brandmeister, but the documentation is explicit that it cannot interlink with xlxd.

Will it run on a Raspberry Pi? This is genuinely unsettled — there is no published benchmark and one operator's Pi question went unanswered — but the software vocoder path is explicitly built for ARM, which strongly suggests a Pi is viable for light, software-only use. A busy transcoding reflector is better hosted on a small VPS.

I connected but no one can hear me — why? Almost always either you are on the wrong or an untranscoded module (the shipped host file's default module is a classic trap), or the transcoder is down (which produces callsigns-but-no-audio). Select the correct transcoded module; if that isn't it, restart the transcoder and then the reflector. Chapter 15 has a decision tree.

Is M17 encryption legal? In the United States, no for AES — it obscures meaning, which Part 97 forbids — but yes for M17's ECDSA signing, which authenticates without hiding the message. Elsewhere, check your own regulations.

Should I build the n7tae or the nostar version? They are nearly identical at the reflector level; choose `nostar` if you want the software-only transcoder to avoid buying hardware, and `n7tae` for the reference implementation with full hardware AMBE quality.

How many users can one reflector handle? There is no published figure; capacity depends on modes, transcoding, and host. Plan conservatively (Chapter 16 has the first-principles math) and watch your logs.

How do people find my reflector? Through the Ham-DHT directory (set your bootstrap node), by listing on DVRef, and — cautiously, only with a collision-free callsign suffix — by enabling calling home.

Chapter 31 — Getting Help

There is no single official support forum for URF, which is worth knowing so you look in the right places. The primary channel is GitHub itself: the `n7tae/urfd` and `nostar/urfd` repositories and the matching `tcd` transcoder repositories carry the source and their issue trackers, and filing a well-described issue there reaches the people who write the software. For the M17 side, the M17 Project runs an active real-time community on both Matrix and Discord, along with an M17-Users mailing list on groups.io. For transcoding and bridging questions — including the perennial hardware-versus-software vocoder debates — the DVSwitch groups.io lists are the gathering place. Broader digital-voice and mode-specific questions get answered on the RadioReference forums, and hotspot-side configuration for connecting to a URF reflector is Pi-Star forum territory. One caution worth stating because the name is so misleading: the "digitalvoice" Google Group is not a URF support channel at all — it is centered on FreeDV and Codec2 on HF, useful for codec questions but not for reflector operations.

Chapter 32 — Where the Sources Disagree

In keeping with the principle of not smoothing over genuine conflicts, a handful deserve to be named. The sharpest is the AMBE-3003 "two versus three streams" question: the chip has three channels, DVSI and the DVstick documentation say three, and the hybrid `tcd` reaches three, but the older all-hardware `ambed` code allocated only two usable streams from a lone 3003, a discrepancy confirmed in the XLX issue tracker. The two figures are both true in their own context. There is an apparent internal contradiction in the Fusion documentation — it says you needn't register your URF's YSF side, yet the configuration file has registration fields — which resolves once you see that the master functions without registration and the fields merely let it appear in directories if the operator wants. There is a repeatedly-confusing point that `AutoLinkModule` is a reflector-side setting and not something a hotspot user sets. On DMR+ reflector control, one operator describes linking by kerchunking a talkgroup while SharkRF's documentation describes a private call to the reflector ID — both agree on talkgroup 9 for talking and 4000 for disconnecting. Brandmeister is often called open source, but its core server is not actually published, so that claim should be avoided. And there are trivial but real documentation

typos in the URF README — an `inicheck` example that names the wrong file and an install command missing a word — worth knowing only so they don't trip you up.

Chapter 33 — What Only a Live Operator Can Confirm

Finally, in the same spirit of honesty, a few things in this guide rest on lighter evidence than the rest, and a working operator could firm them up. Several registry and dashboard sites — DVRef's own documentation, the live XLX API, PA7LIM's pages, a couple of self-hosted URF dashboards — could not be reached directly during research and are therefore described from secondary sources. No quantified performance benchmark for `urfd` exists anywhere I could find; the bandwidth, CPU, and latency figures in Chapters 16 and 27 are first-principles estimates and are explicitly labelled as such, not measurements, and a live operator with monitoring could replace them with real numbers. The exact current behavior of the reflector when its transcoder fails should be re-confirmed against a running build rather than trusted from the older XLX-era report. A couple of the finer legal provisions were not verified against the primary CFR text. And because `urfd` has no tagged releases, the honest way to record "which version am I running" is to note your built commit hash. None of these gaps undermines the picture; they are simply the edges where documentation runs out and a practitioner's fifteen minutes would be worth more than any amount of further reading.

Glossary

AllStar (AllStarLink) — a VoIP network of amateur repeaters and nodes built on the Asterisk PBX; URF bridges to it via USRP.
AMBE — the proprietary DVSI voice codec used by D-STAR (and early Fusion).
AMBE+2 — the newer proprietary DVSI codec used by DMR, Fusion, and NXDN.
Analog_Bridge — a DVSwitch component that transcodes AMBE/IMBE to and from PCM and bridges digital audio to analog and AllStar over USRP.
BrandMeister — one of the two large worldwide DMR networks; URF can link to it through the interlink file.
Calling home — the optional feature that reports a reflector to the central XLX directory at xlxapi.rlx.lu.
CAN (Channel Access Number) — M17's four-bit channel-access code (0–15).
C4FM — Continuous 4-level FM, the 4FSK modulation used by Yaesu System Fusion and P25 Phase 1.
Codec — hardware or software that encodes and decodes a data stream; here, the voice coder.
Codec2 — David Rowe's open, patent-free voice codec used by M17.
Color code — a DMR channel-access code (0–15), analogous to a CTCSS tone.
DCS — one of the three legacy D-STAR reflector protocols; originated in Germany.
DExtra — the open D-STAR reflector protocol behind XRF reflectors.
DG-ID (Digital Group ID) — a Fusion group-access number; on URF it can also select a room/module.
DHT (distributed hash table) — a decentralized, serverless key/value directory; the basis of Ham-DHT.
Digital cliff — the abrupt failure of a digital signal below a threshold, versus analog's gradual fade.
DMO (Direct Mode) — simplex, radio-to-radio DMR operation with no repeater.

DMR — Digital Mobile Radio, the ETSI two-slot TDMA standard.
DMR ID — a station's unique numeric DMR identifier, issued by RadioID.net.
DMRGateway — the software that lets one hotspot connect to several DMR networks at once.
DMR+ (IPSC2) — the other large DMR network, using licensed IPSC2 server software.
DPlus — the original D-STAR reflector protocol, behind REF reflectors.
DroidStar — a multi-mode software client that connects to reflectors from a computer or phone.
D-STAR — the JARL digital-voice protocol; the oldest mainstream amateur digital mode.
DTMF — dual-tone multi-frequency (touch-tone) signalling; on D-STAR, an alternative to URCALL for entering a link command.
DVRef — the web registry/directory where reflectors are listed.
DVSI (Digital Voice Systems, Inc.) — the company that owns the AMBE, AMBE+2, and IMBE codecs.
DVstick-30 / DV3000 / ThumbDV — single-channel AMBE dongles (built on the AMBE-3000 chip).
DVstick-33 / DV3003 — three-channel AMBE transcoding dongles (built on the AMBE-3003 chip).
DVSwitch — a suite of tools for bridging digital-voice modes and networks to each other and to analog.
EchoLink — a long-established VoIP system linking licensed amateurs' computers, phones, and RF nodes; reached from URF through an AllStar node running the EchoLink channel driver.
FCS — a centralized Fusion reflector service with numbered rooms.
FEC (Forward Error Correction) — redundant data that lets a receiver correct errors without retransmission.
FTDI D2XX — the USB driver the DVSI AMBE dongles require; it conflicts with the kernel `ftdi_sio` driver.
G3 — Icom's D-STAR terminal and access-point gateway mode.
Gateway — hardware or software that passes traffic between two different networks.
Golay code — a strong block FEC code (binary Golay(23,12), often extended to (24,12)) able to correct up to three bit-errors per block; guards a voice frame's most important bits.
GPL (GNU General Public License) — the open-source license under which `urfd` is released.
Hamming code — a block FEC code that adds parity bits so a single-bit error can be located and corrected.
Ham-DHT — the amateur distributed-hash-table directory (built on OpenDHT) that URF uses for discovery.
Host file — the address list hotspots download to populate their reflector menus.
Hotspot — a small personal device that links a radio to internet digital-voice networks.
IMBE — the DVSI codec used by P25 Phase 1; its patents have expired.
IMRS — an internet-linking protocol built into newer Yaesu repeaters.
Interlink — a mutual, same-letter link between two reflectors.
ircDDB — a distributed database that shares D-STAR routing information (not audio) between systems.
Jitter buffer — a small buffer that holds arriving packets briefly to smooth out network timing variation, at the cost of a little added latency.
Kademlia — the distributed-hash-table algorithm that OpenDHT, and thus Ham-DHT, is based on.
LSF (Link Setup Frame) — the first frame of an M17 transmission, carrying addressing and metadata.
M17 — the open, patent-free digital-voice protocol built on Codec2.
md380 vocoder — an ARM software AMBE+2 vocoder derived from MD-380 radio firmware, used to avoid a dongle.
MMDVM — the Multi-Mode Digital Voice Modem firmware for hotspots.
MMDVMHost — the host program that drives an MMDVM modem (used inside Pi-Star and WPSD).
Module — one of a reflector's up-to-26 independent rooms, lettered A–Z.

mrefd — N7TAE's M17-only reflector, sibling to `urfd`.

mvoice — N7TAE's M17-only desktop software client, needing no dongle.

NAC (Network Access Code) — P25's access code, a three-hex-digit value.

new-xlxd — N7TAE's modernization of the XLX code, the step between XLX and `urfd`.

NXDN — an open FDMA narrowband digital standard from Icom and Kenwood.

OpenDHT — the C++ distributed-hash-table library behind Ham-DHT.

openSPOT — SharkRF's standalone hardware hotspot.

P25 — Project 25, the public-safety land-mobile-radio standard; Phase 1 uses IMBE.

Pi-Star — a popular pre-built hotspot software image.

RadioID.net — the global registry that issues DMR and NXDN IDs.

RAN (Radio Access Number) — NXDN's access code (0–63).

Reflector — a server that relays a mode's traffic among all connected stations; a conference bridge.

Repeater — an RF relay that receives and retransmits a signal to extend range.

Simplex / duplex — one-way versus two-way operation; half-duplex alternates, full-duplex is simultaneous.

SVXlink — an open-source Linux repeater controller (SM0SVX) with an EchoLink module and its own SvxReflector network; bridged to digital voice over USRP via its UsrpLogic.

SvxReflector — SVXlink's reflector network for linking SVXlink systems together.

systemd — the Linux service manager under which `urfd` and `tcd` run.

Talkgroup — a logical group of users on a networked radio system who all hear each other.

Tandem vocoding — decoding one codec to audio and re-encoding it into another; the source of transcoding's quality loss.

tcd — URF's hybrid transcoder, converting between AMBE, AMBE+2, IMBE, and Codec2.

TDMA (time-division multiple access) — sharing one channel among users by time slots; DMR uses two.

ThumbDV — see DVstick-30.

Transcoding — converting audio from one codec to another in real time so that different modes can talk.

URCALL — the D-STAR field where a linking command (reflector + module + L/U) or a target callsign is entered.

urfd / URF — the universal reflector software this guide is about.

USRP — a simple UDP protocol carrying PCM audio ("8 kHz signed 16-bit") between DVSwitch/AllStar components and URF; the lingua franca of analog/VoIP bridging.

UsrpLogic — the SVXlink component that connects it to a DVSwitch Analog_Bridge over USRP.

US-Trust — the original central registration/trust-server system for D-STAR gateways.

Vocoder — the algorithm that turns speech into a low-bitrate data stream and back; the reason transcoding is needed.

VPS (virtual private server) — a rented virtual machine, the usual host for a reflector.

Watchdog — an automatic check that restarts or alerts on a stalled service; here, a status-file staleness probe.

Wires-X — Yaesu's proprietary internet repeater-linking system.

WPSD — a modern successor to Pi-Star for managing hotspots.

XLX (xlxd) — the multi-protocol reflector that URF is a superset of; URF cannot interlink with it.

YCS — a multiprotocol C4FM reflector server handling YSF, FCS, and IMRS together.

YSF (Yaesu System Fusion) — Yaesu's C4FM digital-voice system; on URF it is served as a self-contained YSF

Master.
YSFReflector — the open, decentralized Fusion reflector network, separate from URF's YSF Master.

Appendix A — Network Ports

All ports are UDP except HTTP/HTTPS and the transcoder link, which are TCP. You do not need to open ports for modes you have disabled. *Values are the shipped defaults as of the 2026 source; a given reflector may override any of them in its configuration.*

Service	Port	Transport	Notes
Dashboard	80 / 443	TCP	web
DMR+ DMO	8880	UDP	
BrandMeister	10002	UDP	
URF interlink	10017	UDP	reflector-to-reflector
Transcoder	10100	TCP	only if the transcoder is remote
G3 presence / request	12345–12346	UDP	Icom terminal
M17	17000	UDP	
Ham-DHT	17171	UDP	omitted from the README firewall list — open it if using DHT
DPlus	20001	UDP	
DExtra	30001	UDP	
DCS	30051	UDP	
USRP	32000 (client Rx 34000)	UDP	
G3 terminal	40000	UDP	
P25	41000	UDP	
NXDN	41400	UDP	
YSF	42000	UDP	
MMDVM	62030	UDP	DMR

Appendix B — How to Connect, by Mode (quick reference)

Mode	How the user selects a module
DMR	Private call `68` + reflector number, then `64000` + module number (A=1…Z=26); talk on **TG 6**; `64000` to disconnect; `65000` for status. On DMR+, TG `4001`–`4026` selects module A–Z directly and TG `4000` unlinks.
D-STAR	URCALL = reflector callsign + module letter + `L` / `U` (e.g. `URF621A L`, or the `XLX…` form); or DTMF (digits set by your hotspot); or the dashboard
D-STAR G3 (terminal)	Register on a G3 gateway, assign a module letter, RS-MS3A/W, reflector IP, forward UDP 40000
YSF	Connect to the YSF Master, then Wires-X room, or DG-ID 10–35 → module A–Z (mapping is per-reflector)
M17	Pick the `URFxxx` / `M17-xxx` host, then the module letter, then connect
P25 / NXDN	Set the numeric reflector ID as startup host; you land on the reflector's configured auto-link module
USRP / AllStar	Fixed in the reflector/bridge configuration

Appendix C — urfd.ini at a Glance

The seven config files are `urfd.mk` (compile options), `urfd.ini` (runtime), `urfd.blacklist` / `urfd.whitelist` (callsign deny/allow, wildcard `*`), `urfd.interlink` (URF + Brandmeister links), `urfd.terminal` (G3), and `urfd.service` (systemd). The essential `urfd.ini` keys: `[Names]` `Callsign=URF???` and `Bootstrap=` and `DashboardUrl=`; `[IP Addresses]` `IPv4Binding=` and `IPv6Binding=`; `[Modules]` `Modules=`; `[Transcoder]` `Port=` (0 = none), `BindingAddress=` (`127.0.0.1` local / `0.0.0.0` remote), and `Modules=` (must equal the transcoder's own `Modules`); a port section per protocol; `AutoLinkModule=` for P25/NXDN/YSF; and three database sections (`Mode=http|file|both`, `URL=`, `FilePath=`, `RefreshMin=`). Always validate with `./inicheck -q urfd.ini` before starting — an `ERROR` means it won't run.

Appendix D — A Timeline of the Reflector Lineage

mid-2000s — REF reflectors (DPlus protocol), Robin Cutshaw AA4RC; tied to the US-Trust system.
late 2000s–2013 — XRF reflectors (DExtra), Scott Lawson KI4LKF; the first open reflector protocol; work

ceased in 2013.
~2012 — DCS reflectors, originating in Germany with Torsten Schultze DG1HT.
October 2015 — XLX conceived by the Radio Amateurs du Luxembourg working group (LX3JL, LX1IQ).
7 November 2015 — first XLX beta released.
January 2016 — XLX adopts the GPL; the "© 2016" copyright dates from here.
2016 onward — XLX adds DMR and Fusion with AMBE (`ambed`) transcoding.
~2017 — the patents on the D-STAR AMBE codec variant expire, making software vocoders legally defensible.
2019 onward — the M17 Project develops a fully open protocol built on the patent-free Codec2.
~2020 — N7TAE releases `new-xlxd`, a modernization of the XLX codebase.
2022 — N7TAE and Doug McLain AD8DP release `urfd` (URF), adding M17, P25, NXDN, and USRP, and replacing `ambed` with the hybrid `tcd`.
2024 — the copyright shows N7TAE as sole listed maintainer; development continues on GitHub with no tagged releases.

Index

Because `urfd` has no fixed pagination and this guide is read as much on screen as on paper, this index points to *chapters and appendices* rather than page numbers. Bold entries are where a topic is defined or treated in depth.

Topic	Where
AMBE / AMBE+2 codecs	Ch 5, **6**, 19, **24**; Glossary
AutoLinkModule	Ch 8, 32; App C
Backup and restore	**Ch 16**
Bandwidth / capacity math	**Ch 16**, 30
Blacklist / whitelist	**Ch 8**, 16, 27
Brandmeister	Ch 10, 20, 26, 32; App A
Building / installing	**Ch 14**, 18, 25
Calling home	**Ch 12**, 14, 30
Codec2 / M17 codec	Ch 5, **11**, 19, 24, 28
Commit hash / versioning	Ch 16, 33
Costs	**Ch 29**
Dashboard	**Ch 23**, 14, 21, 22
Decision tree (no audio)	**Ch 15**
DMR connection / talkgroups	**Ch 8**, 32; App B
DNS / HTTPS deployment	**Ch 22**, 23
DTMF linking	**Ch 8**; App B; Glossary
DVSI hardware / dongles	**Ch 6**, 9, 18, 29
DVSwitch bridging	**Ch 20**
FEC (Golay, Hamming)	**Ch 19**; Glossary
Ham-DHT / discovery	Ch 7, **12**, 21
Interlinking	**Ch 10**, 11, 25, 27
IPv6 / dual-stack	Ch 16, 22, 25, 26; App C
Last Heard list	**Ch 23**, 21
Latency	**Ch 28**, 27
Law / FCC Part 97	**Ch 17**, 11, 27

Topic	Where
M17 encryption / signing	Ch 11, **17**, 30
Migrating from XLX	**Ch 25**, 1, 27
Modules	**Ch 3**, 4, 8, 10
Monitoring / watchdog	**Ch 16**, 21, 23
Ports	Ch 7, 16; **App A**
Security / hardening	Ch 8, **16**, 27
Software-only transcoding	Ch 9, 18, 26, 30
Status file (XML / JSON)	Ch 7, **21**, 16, 23
Tailscale	**Ch 14**, 22, 25, 30
Testing / verification	**Ch 14**, 13, 23, 30
Transcoding (concept)	**Ch 5**, 9, 27, 28
Troubleshooting	**Ch 15**, 14, 23, 30
USRP / AllStar	Ch 4, 20; App A; Glossary
Voice quality	**Ch 28**, 27
YSF / Fusion as YSF Master	**Ch 4**, 8, 32

Sources

The backbone of this guide is the primary source code and documentation of the software itself: the `urfd` reflector and its configuration templates and C++ source (`github.com/n7tae/urfd` and the parallel `github.com/nostar/urfd`); the `tcd` transcoder and its source (`github.com/n7tae/tcd` , `github.com/nostar/tcd`); the ancestral `xlxd` (`github.com/LX3JL/xlxd`) and `new-xlxd` (`github.com/n7tae/new-xlxd`); the sibling `mrefd` (`github.com/n7tae/mrefd`); the discovery tooling (`github.com/n7tae/ham-dht-tools`); the vocoder libraries (`nostar/md380_vocoder` , `nostar/imbe_vocoder` , `drowe67/codec2`); the container build (`github.com/mfiscus/urfd-docker`); the host-file mirror (`github.com/airphel/WPSD-HostFiles`); and DMRGateway (`github.com/g4klx/DMRGateway`).

Independent operators running live URF or XLX reflectors provided real-world corroboration: QuadNet (`openquad.net`), HRCC Link (`urf.hrcc.link`), New England Digital Radio (`newenglanddigitalradio.com`), W0CHP (`w0chp.radio`), K2AS via XARC (`xarc.us`), WhoCaresRadio

(`wiki.whocaresradio.com`), VA6MO (`va6mo.ca`), the gajewski.net reflector-codes reference, and the WPSD manual (`manual.wpsd.radio`).

Standards, specifications, and reference material grounded the technical and historical chapters: the M17 Protocol Specification (`m17-protocol-specification.readthedocs.io`); the FEC and codec references (Wikipedia on the Binary Golay code, Hamming code, Multi-Band Excitation, Codec 2, Project 25, NXDN, and M17; ok-dmrlib; Daniel Estévez on AMBE FEC; UrgentComm on P25 error-control coding; gr-ysf); F4FXL, KB9MWR, and amateurradionotes on reflector history; Lyons Computer for the XLX timeline; DVSI's AMBE-3000/3003 product page; and the FTDI D2XX driver documentation. The legal chapter draws on 47 CFR §97.119 and §97.113 (Cornell Legal Information Institute), the ARRL's report on the FCC encryption petition, and Bruce Perens on cryptographic signing. Client software is documented from the DroidStar, mvoice, and BlueDV project pages and the M17 Project's Module17. The comparison chapter additionally used g4klx/YSFClients, W5JER on YSF versus FCS, the BrandMeister internals paper and GitHub organization, FreeSTAR on IPSC2, and the DVSwitch organization.

The evaluative chapters drew on additional sources. Costs came from GigaParts and Ham Radio (UK) for the DVstick-30, NW Digital Radio for the ThumbDV, dxcanada for the DVstick-33, DVSI's product page, and a 2026 VPS-price comparison (apicalculators.com). The voice-quality material came from DVSI's transcoding white paper and the corroborating NTIA tandem-vocoder study, KB9MWR's codec notes, and David Rowe (rowetel.com) on Codec2 versus AMBE. Use cases came from W8WKY (Silvercreek ARA), QuadNet, and the N5AMD and AD6DM multi-mode how-tos. The community and support channels are the n7tae / nostar GitHub repositories and issue trackers, the M17 Project's Matrix, Discord, and M17-Users list, the DVSwitch groups.io lists, the RadioReference and Pi-Star forums, and — noted only to warn against it as a URF venue — the FreeDV-centric "digitalvoice" Google Group.

The reference and deep-dive chapters added further sources. The mode and codec comparison came from the Wikipedia articles on D-STAR, DMR, Project 25, NXDN, M17, System Fusion, Multi-Band Excitation, and Codec 2, the M17 specification, and RepeaterBook, SigidWiki, and BuyTwoWayRadios for the access-code details. The DVSwitch bridging material came from the Analog_Bridge and MMDVM_Bridge repositories and their configuration files, the DVSwitch wiki, KB9MWR's MMDVM_Bridge and Allstar-digital-bridge write-ups, the NW Digital Radio AMBEServer documentation, and operator accounts at we8chz.org and erunix.wordpress.com. The analog and VoIP linking material (Chapter 20) drew on the AllStarLink manual and DTMF command references, the AllStarLink EchoLink channel-driver documentation, and the SVXlink project (SM0SVX) together with F5VMR's and G4NAB's SVXlink-over-USRP write-ups. The status-file chapter was read directly from `urfd`'s `Reflector.cpp`, `Peer.cpp`, `Client.cpp`, `User.cpp`, and `dht-values.h`. The patent history drew on the Multi-Band Excitation and Codec 2 Wikipedia articles, KB9MWR's codec notes (which attribute the ~2017 D-STAR AMBE expiry to Bruce Perens), DVSI's own AMBE+2 page, and the M17 Project's about page. The regulation-outside-the-US material came from the Ofcom UK amateur licence terms and the ITU Radio Regulation 25.2A text as quoted by Radio Amateurs of Canada. The deployment chapter used the official Certbot Apache instructions and the GoatCounter project.

The sizing, testing, and operations material added in this edition — the bandwidth and CPU estimates (Chapter 16), the latency estimate (Chapter 28), the FEC arithmetic (Chapter 19), the per-mode test procedure (Chapter 14), the status-file reader (Chapter 21), and the hardening, time/log, IPv6, and monitoring guidance (Chapter 16)

— is derived from the codec bitrates and protocol behavior already cited above combined with standard Linux server and networking practice; where a figure is an estimate rather than a measurement, the text says so, and Chapter 33 records what a live operator could confirm.

This is the written edition — the same researched, sourced material as the reference versions, set out as prose to be read rather than scanned. Where a claim rests on a single source or could not be independently confirmed, the text says so.