DigiRadio

Open-Source Hi-Fi DAB+/FM Receiver

Technical Manual

Michele Bigi

Firmware 0.8.4 • 2026

https://github.com/manvalan/DigiRadio

Prototype PCB fabrication and assembly supported by **PCBWay**.

Contents

CHAPTER 1

Introduction

DigiRadio is an open-source, high-fidelity digital radio receiver. It receives DAB+ and FM broadcasts, processes the audio through a dedicated signal processor, and streams the result over Bluetooth using a high-resolution codec. Firmware 0.8.4 on `main` provides encrypted storage, a tabbed configuration web UI, and the full REST API documented in Chapter 8. The whole project — hardware and firmware — is released as open source for the maker and audio community to study, build, and improve.

1.1 What DigiRadio is

Most DAB+/FM hobby projects stop at basic reception. DigiRadio adds two things that lift it into hi-fi territory: a real audio DSP stage for equalisation and mixing, and a premium-codec Bluetooth output. The result is a genuine wireless audio source rather than a bench demo, and, being fully documented, a practical reference for anyone learning multi-chip audio design or RF-aware PCB layout.

At a glance

A four-chip design — Si4684 tuner, ADAU1701 DSP, ESP32-S3 host, and an FSC-BT1035 Bluetooth transmitter — on a six-layer board, powered from USB-C, configured through an elegant web interface.

1.2 Who this manual is for

This manual documents the system for someone building, flashing, or extending DigiRadio: the hardware at a block level (Chapter 2), the firmware architecture (Chapter 4), dedicated companion-chip guides for the Si4684 tuner (Chapter 5) and ADAU1701 DSP (Chapter 6), FSC-BT1035 Bluetooth (Chapter 7), the HTTP JSON API

exposed by the web UI (Chapter 8), the per-class design reference that grows with the code (Chapter 9), and how to build and flash (Chapter 10). Exact C++ signatures are generated by Doxygen from the source; this manual explains design, behaviour, and the reasoning behind it.

1.3 Licensing summary

The hardware is licensed under CERN-OHL-S v2 and the firmware under Apache-2.0; full details are in Chapter 11. The project repository is at `https://github.com/manvalan/DigiRadio`.

CHAPTER 2

Hardware Overview

This chapter describes the board in detail: the design choices behind it, each integrated circuit and how it is controlled, the bus architecture, and the inter-chip connections. The complete hardware design — schematics, PCB, Gerbers, and bill of materials — is published in the repository under the CERN-OHL-S v2 licence.

2.1 Design philosophy

Every major choice on the board serves audio quality and reproducibility.

The four-chip split

Rather than a single all-in-one radio IC, DigiRadio separates concerns: a dedicated tuner for reception, a dedicated DSP for audio conditioning, a dedicated Bluetooth module for the wireless codec, and a general-purpose host to coordinate them. Each stage can be understood, measured, and improved on its own.

The consequences of that split run through the rest of the board: a control host that speaks several buses, a clean power tree that keeps the tuner's analogue supplies away from digital noise, and careful RF layout so the two 2.4 GHz radios do not interfere.

2.2 System block diagram

Four devices are coordinated by the ESP32-S3. The tuner produces the audio stream, the DSP conditions and mixes it, and the Bluetooth module transmits it; the host manages all three and provides the network interface.

Device	Role	Control bus	Firmware
Si4684	DAB+/FM tuner	SPI	host-loaded image
ADAU1701	Audio DSP (EQ + mixer)	I^2C	host-loaded RAM
ESP32-S3-WROOM-1	Host, Wi-Fi/BLE	—	application FW
FSC-BT1035	Bluetooth 5.2 (aptX Ad.)	UART (AT)	vendor FW + config

Table 2.1: Principal integrated circuits, their control bus, and how each receives its firmware or configuration.

2.3 The tuner: Si4684

The Si4684 is a single-chip DAB+/FM receiver front-end. It delivers the decoded audio stream to the DSP and reports signal-quality metrics to the host.

2.3.1 Control interface

The device is controlled through a command/response protocol over its host bus. The host issues a command, then polls a clear-to-send (CTS) status bit before reading the response; the driver validates that status byte before trusting any returned payload.

SPI for the tuner

The tuner needs a large firmware image loaded at every power-up (Section 2.3.2). SPI offers substantially higher throughput than I^2C, so on DigiRadio the tuner uses **SPI**, keeping boot time short, while the low-rate DSP control uses a separate I^2C bus.

2.3.2 Firmware image upload

The Si4684 holds no application firmware of its own at reset. At power-up the host loads a bootloader/patch and then the firmware image for the desired mode (FM or DAB). These images are large, so the driver streams them to the chip in bounded chunks read from the ESP32's flash, rather than buffering a whole image in RAM.

Datasheet

The exact power-up, patch, image-load, and boot command sequence follows the Si468x programming guide (AN649). No command byte is issued without a documented source, and each step in the driver cites its section.

2.4 The audio DSP: ADAU1701

The ADAU1701 is a SigmaDSP audio processor. In DigiRadio it performs equalisation and mixes the tuner audio with the ESP32 audio source before the signal reaches the Bluetooth stage.

2.4.1 Control and program loading

The DSP is controlled over I^2C. Its signal-processing program is designed in SigmaStudio and exported as a sequence of register writes (control settings, program RAM, and parameter RAM). The full SigmaStudio schematic, signal chain, and runtime parameter map are documented in Chapter 3. Driver boot, safeload API, `AudioService`, and HTTP integration are in Chapter 6.

RAM boot, no EEPROM

The board deliberately omits the ADAU1701 self-boot EEPROM. Instead, the ESP32 writes the exported program into the DSP's RAM at *every* boot. This keeps the audio processing under host control and field-updatable — a new EQ or mixer configuration is a firmware update, not a chip re-programming.

2.4.2 Click-free parameter updates

Runtime changes — adjusting an EQ band, changing a mix level — are applied through the ADAU1701 safeload mechanism, which updates parameter cells atomically between audio samples. Writing parameter cells directly while audio plays would produce audible clicks and is not done.

2.5 The Bluetooth module: FSC-BT1035

The FSC-BT1035 (Qualcomm QCC3056) is a Bluetooth 5.2 audio transmitter with aptX, aptX HD, and aptX Adaptive support — the high-resolution codecs that make the wireless output hi-fi rather than merely convenient.

2.5.1 Control interface

The module is controlled with AT commands over UART, with hardware flow control (RTS/CTS). The host sends commands and parses the module's responses explicitly, treating timeouts as errors. Some settings are held in the module's own non-volatile memory.

Line-In mode

The initialisation sequence must enable Line-In mode (`AT+AUXCFG=1`) so the module accepts the wired audio coming from the DSP. Omitting it silently breaks the audio path. Driver boot flow and AT subset are documented in Chapter 7.

2.6 The host: ESP32-S3

The ESP32-S3-WROOM-1 coordinates the whole system. It loads the tuner and DSP firmware at boot, controls all three companion chips over their respective buses, serves the configuration web interface, and provides Wi-Fi connectivity. Its dual-core processor and generous RAM comfortably handle the firmware images and the network stack alongside the control tasks.

2.7 Bus architecture

The host speaks several interfaces, each chosen for its traffic:

Bus	Devices	Purpose
SPI	Si4684	fast firmware-image upload, tuner control
I^2C	ADAU1701, 24AA025E48 EEPROM	DSP control, device identity
UART	FSC-BT1035	AT command / response (with RTS/CTS)
I^2S	Si4684, ESP32, ADAU1701, BT1035	audio (ADAU is master)

Table 2.2: Interface assignment and rationale.

2.8 Power

The board is powered from USB-C. An AP63203 buck converter generates the main rail from the USB input, feeding the digital and analogue sections. The tuner's sensitive supplies are derived and filtered separately to keep RF performance clean.

2.9 RF and antennas

Two 2.4 GHz radios are present — the ESP32-S3 and the Bluetooth module. Their antennas are placed on opposite diagonal corners of the board to maximise separation and minimise mutual desense. Each module keeps its antenna keep-out clear of copper on all layers. The PCB is a six-layer stack-up with controlled impedance on the USB and RF sections.

2.10 Inter-chip connections (GPIO map)

This section is the authoritative wiring reference between the ESP32-S3 and the companion chips. The firmware pin definitions (`board_pins.hpp`) must match these tables exactly.

2.10.1 System and control lines

Signal	ESP32-S3 GPIO	Notes
USB D−	GPIO19	native USB
USB D+	GPIO20	native USB
Reset / boot button	GPIO0	strapping pin

Table 2.3: System and control lines.

2.10.2 Si4684 tuner (SPI)

2.10.3 ADAU1701 DSP (I^2C)

Signal	Si4684 pin	ESP32-S3 GPIO	Notes
SCLK	pin 30	GPIO13	
MOSI	pin 31	GPIO12	
MISO	pin 32	GPIO9	
SSB (CS)	pin 29	GPIO8	chip select, active-low
RSTB#	pin 4	GPIO38	active-low, ext. pull-down
INTB#	pin 3	GPIO39	active-low, ext. pull-up

Table 2.4: Si4684 control and SPI lines.

Signal	ADAU1701 pin	ESP32-S3 GPIO
SDA	—	GPIO4
SCL	—	GPIO5
RESET#	pin 5	GPIO47 (active-low, ext. pull-up)

Table 2.5: ADAU1701 control (I^2C) and reset. The 7-bit I^2C address is 0x34 (ADDR0 and ADDR1 tied to GND).

2.10.4 Device-identity EEPROM (24AA025E48)

A 24AA025E48 EEPROM shares the DSP's I^2C bus. Besides a small general-purpose memory, it carries a factory-programmed, globally unique EUI-48 identifier, which the firmware can read to give each unit a stable identity — for example a unique Bluetooth name or a device serial number. Its 7-bit address is 0x52 (A0 to GND, A1 to 3V3).

Device	7-bit address	Set by
ADAU1701	0x34	ADDR0, ADDR1 = GND
24AA025E48 EEPROM	0x52	A0 = GND, A1 = 3V3 (no A2 pin)

Table 2.6: I^2C address map (shared bus).

2.10.5 FSC-BT1035 Bluetooth

2.10.6 Audio bus (I^2S)

The audio path is a chip-to-chip I^2S bus with the ADAU1701 acting as master. Table 2.8 lists the chip-to-chip routing; Table 2.9 lists the ESP32's own I^2S source lines that feed the mixer.

Signal	BT1035 pin	ESP32-S3 GPIO	Direction
SYS_CTL	pin 34	GPIO15	ESP32 → BT
BT_CTS	pin 15	GPIO21	flow control
BT_RTS	pin 16	GPIO14	flow control
BT_RX	pin 14	GPIO40	ESP32 TX → BT RX
BT_TX	pin 13	GPIO41	BT TX → ESP32 RX
RESET	pin 8	GPIO17	ESP32 → BT

Table 2.7: FSC-BT1035 UART, control, and reset lines.

Signal	Path	ADAU1701 / BT1035 pins
SDATA_IN0	Si4684 → ADAU	MP0 (pin 11)
SDATA_IN1	ESP32 → ADAU	MP1 (pin 10)
SDATA_OUT0	ADAU → BT1035	MP6 (pin 15) → pin 5
LRCLK	ADAU → BT1035	MP4 (pin 8) + MP10 (pin 16) → pin 7
BCLK	ADAU → BT1035	MP5 (pin 9) + MP11 (pin 19) → pin 4

Table 2.8: I^2S audio routing (chip-to-chip). The MP4/MP10 and MP5/MP11 pairs are electrically connected on the board.

Signal	ESP32-S3 GPIO	Direction
BCLK	GPIO6	in from ADAU (master)
LRCLK	GPIO7	in from ADAU (master)
SDATA_IN1	GPIO16	out → ADAU MP1

Table 2.9: ESP32 I^2S source lines feeding the mixer. The ESP32 is an I^2S slave (clocks in), transmitting audio data. SDATA_IN1 lands on ADAU MP1 (pin 10); GPIO16 is the ESP32 module's physical pin 9.

2.10.7 Completeness

Map complete

The GPIO and I^2C maps are complete and match `board_pins.hpp`. No values remain open.

This section is the authoritative wiring reference for firmware bring-up: the pin definitions in the source code must match it exactly.

2.11 Design validation and critical checks

This section records the design-review checks performed against the component datasheets before fabrication, chip by chip. Items marked *critical* can cause permanent damage or a non-functional board if violated, and were verified explicitly.

2.11.1 Si4684 tuner

Critical — RSTB and power sequencing

The Si4684 datasheet requires that RSTB be held low during any power-supply transition and remain asserted for 10 µs after all supplies are stable; failing to do so *may permanently damage the device*. On DigiRadio, RSTB (GPIO38) has an external pull-down, so the chip powers up held in reset until the ESP32 releases it. **Verified.** The firmware must not drive RSTB high before the Si4684 supplies are stable.

Firmware bring-up

RSTB is driven only in `Si4684Driver::boot()` (Chapter 5): GPIO38 is configured as output with `GPIO_PULLDOWN_ENABLE` so the net cannot glitch high while the pin mode changes, then held low for 5 ms and released. The internal pull-down is *not* redundant with the external resistor: the external pull-down holds RSTB during board power-on before `app_main`; the internal one closes the brief window between `gpio_config` and `gpio_set_level(0)` where an output register default could otherwise violate the datasheet sequencing rule. No other code touches GPIO38. Release occurs only after all Si4684 supplies (VA, VIO, VCORE, VMEM)

are stable, satisfying the +10 µs requirement in practice.

Check	Requirement (datasheet)	Status
RSTB low at power-up	held low until supplies stable +10 µs	verified (pull-down)
RSTB release (firmware)	not before supplies stable; boot pulse only	verified (driver)
RSTB glitch guard (firmware)	`GPIO_PULLDOWN_ENABLE` in `gpio_config`	verified (driver)
Core supplies at 1.8 V	VA = VCORE = VMEM = 1.8 V	verified
VIO level	1.62–3.6 V (3.3 V to talk to ESP32)	verified
Bypass caps	2.2 nF + 1 µF per VIO/VMEM/VCORE rail	verified
SPI mode	Mode 0 or 3, up to 10 MHz, SMODE = GND	verified
SSB framing (firmware)	SSB held low for the whole transaction	driver rule
INTB	interrupt line, external pull-up (GPIO39)	verified

Table 2.10: Si4684 design checks. References: Si4684-A10 datasheet (power sequencing, SPI, supply levels) and AN851 (bypass and layout).

Datasheet

Power sequencing and SPI framing: Si4684-A10 data sheet. The user must not pulse SSB high between bytes; SSB frames the whole command/reply. Bypass-capacitor values and placement: AN851, *Si468x Schematic and Layout Guide*. Boot and command protocol: AN649.

The core, memory, and analogue rails run at 1.8 V; only VIO is at 3.3 V to interface the ESP32. The 2.2 nF and 1 µF bypass pairs are placed close to VIO (pin 34), VMEM (pin 35), and VCORE (pin 37) as AN851 requires. The VHF antenna input follows the AN851 layout guidance (short, narrow microstrip; ground-fill relief; ferrite beads near the connector).

2.11.2 ADAU1701 DSP

Master-clock loopback is mandatory, not optional

For an ADAU1701 operating as I^2S master while receiving data on its serial inputs, the datasheet requires OUTPUT_LRCLK (MP10) and OUTPUT_BCLK (MP11) to be set to master mode and *connected externally* to INPUT_LRCLK (MP4) and INPUT_BCLK (MP5). DigiRadio wires exactly this loopback (MP10→MP4, MP11→MP5). Without it the DSP locks up when input data arrives.

Check	Requirement (datasheet)	Status
Clock loopback	MP10→MP4, MP11→MP5 wired	verified
Output master	serial output port in Master Mode	verified
MCLK ratio	exactly $256 \times f_S$	verified
MCLK value	12.288 MHz oscillator (256×48 kHz)	verified
I^2C pull-ups	2.2 kΩ on SDA and SCL	to verify
I^2C address	0x34 (ADDR0 = ADDR1 = GND)	verified
Self-boot	disabled (host RAM load)	verified

Table 2.11: ADAU1701 design checks. Reference: ADAU1701 data sheet (Rev. C), serial-port master/slave section and I^2C port.

Datasheet

Master-clock loopback and serial-port modes: ADAU1701 data sheet, Rev. C, Table 63 and the serial-port section. The master clock must be exactly $256 \times f_S$. I^2C lines require 2.2 kΩ pull-ups.

Open item — I^2C pull-ups

The ADAU1701 datasheet requires 2.2 kΩ pull-up resistors on SDA and SCL. Confirm these are present on the board (shared bus with the EEPROM); this is the one ADAU check still to close.

2.11.3 FSC-BT1035 (QCC3056)

Check	Requirement	Status
UART with flow control	TX/RX + RTS/CTS wired	verified
Line-In enable (firmware)	`AT+AUXCFG=1` in init	driver rule
I^2S input	receives clocks from ADAU master (slave)	verified
Reset line	RESET (GPIO17) driven by ESP32	verified

Table 2.12: FSC-BT1035 design checks. The module is an I^2S slave fed by the ADAU master; audio arrives on its I^2S input pins.

2.11.4 System-level summary

The only open hardware check is the presence of the 2.2 kΩ I^2C pull-ups on the ADAU/EEPROM bus. All damage-class items (Si4684 power sequencing and supply levels) are verified.

Area	Item	Status
Si4684	RSTB pull-down, 1.8 V rails, bypass caps, SPI	verified
Si4684	RSTB release only in `Si4684Driver::boot()`	verified
ADAU1701	clock loopback, master mode, $256\times f_S$	verified
ADAU1701	I^2C 2.2 kΩ pull-ups	to verify
FSC-BT1035	UART + flow control, I^2S slave	verified
Clocking	single 48 kHz domain, ADAU master	verified

Table 2.13: Consolidated pre-fabrication validation status.

CHAPTER 3

DSP Configuration (SigmaStudio)

This chapter documents how the ADAU1701 is configured in SigmaStudio: the hardware settings that must match the board, the audio signal-processing chain, and the parameters exposed for runtime control by the ESP32. The resulting SigmaStudio export is what the host writes into the DSP's RAM at every boot (Section 2.4). For the firmware driver, safeload API, persistence, and HTTP usage, see Chapter 6.

3.1 Overview

The DSP program is built once in SigmaStudio and exported as a sequence of register writes. Because the board carries no self-boot EEPROM, the ESP32 replays that sequence over I^2C (address 0x34) on every power-up. SigmaStudio is therefore used purely to *design and export* the program, not to drive the chip in operation.

No hardware required to export

The whole program can be built and exported offline, without a physical ADAU1701 or a USBi programmer connected. A USBi block is placed in the Hardware Configuration only because SigmaStudio requires a communication channel to compile; it is never used for a real download in this project.

3.2 Hardware Configuration

The Hardware Configuration tab must mirror the board exactly. The key settings are collected in Table 3.1.

Setting	Value
Control port (I^2C address)	0x34 (ADDR0/ADDR1 = GND)
Self-boot	Disabled (host RAM load)
System sample rate	48 kHz
MCLK	12.288 MHz active oscillator
Serial Input format	I^2S, 24-bit
Serial Output	**Master Mode** (ADAU generates clocks)
Word length	24 bits
RAM Modulo / Program Length	1x (1024 instructions)

Table 3.1: ADAU1701 hardware settings in SigmaStudio.

3.2.1 Multipurpose pin (MP) assignment

The ADAU1701 multipurpose pins are assigned to the serial audio signals as wired on the board. Table 3.2 lists the assignment; it matches the board GPIO map (Section 2.10).

MP pin	Direction / function	Signal
MP0	Input Sdata_in0	audio from Si4684 (radio)
MP1	Input Sdata_in1	audio from ESP32
MP4	Input Lrclk_in	LRCLK (returned via board link)
MP5	Input Bclk_in	BCLK (returned via board link)
MP6	Output Sdata_out0	audio to FSC-BT1035
MP10	Lrclk_out	LRCLK generated (master)
MP11	Bclk_out	BCLK generated (master)

Table 3.2: Multipurpose pin assignment. MP2, MP3, MP7–MP9 are unused (GPIO input, default).

Master clock routing

The ADAU1701 is the I^2S master: it generates LRCLK and BCLK on MP10/MP11. On the board these are wired to MP4/MP5, so the same master clocks time the serial inputs. This shared-clock scheme means the serial inputs are configured as *clock inputs* even though the ADAU itself is master — the clocks originate at MP10/MP11 and return through the board link. All companion chips (Si4684, ESP32, BT1035) are I^2S slaves.

3.3 Signal-processing chain

The audio flows through a fully stereo chain, built in the Schematic tab (Figure 3.1). Two stereo sources are level-trimmed, mixed, equalised, volume-controlled, and peak-limited before reaching the two digital outputs that feed the Bluetooth module.

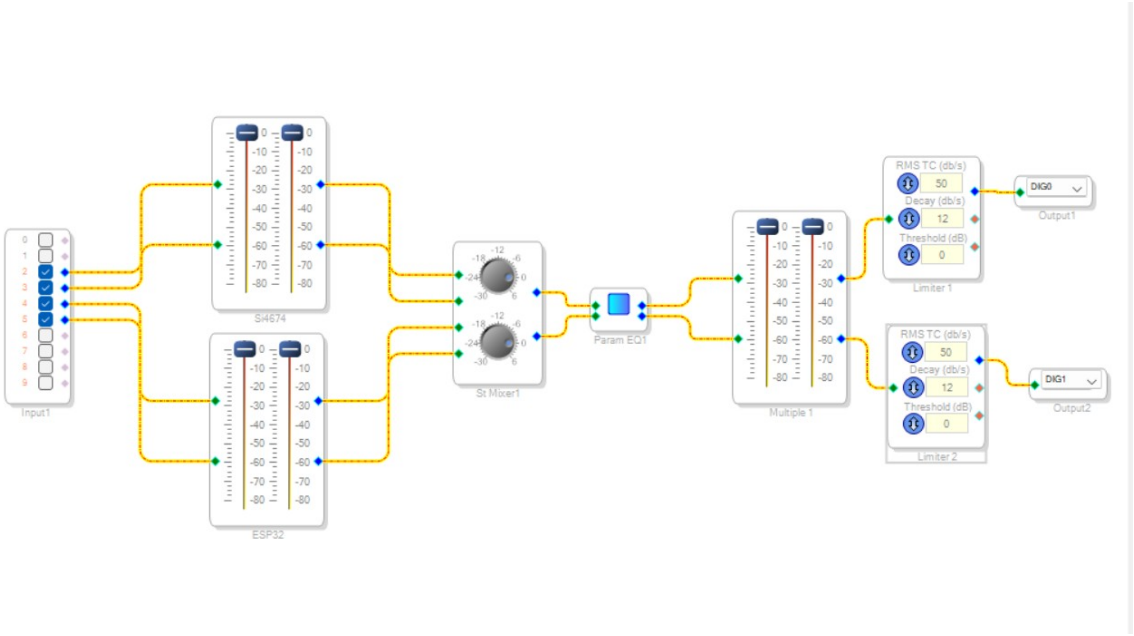

Figure 3.1: The complete stereo signal-processing chain in SigmaStudio: Input → source faders → stereo mixer → parametric EQ → master volume → per-channel limiters → outputs.

3.3.1 Block-by-block

Stage	Block	Role
Input	Input (ch. 2–5)	radio L/R, ESP32 L/R
Source trim	Si4674, ESP32 faders	per-source stereo level
Mix	Stereo Mixer	combine sources, keep L/R
EQ	Parametric EQ (2-ch)	high-pass + 5 peaking bands
Volume	Multiple volume (2-ch)	master volume
Protect	Limiter 1 / Limiter 2	per-channel peak limiting
Output	Output (DIG0, DIG1)	L/R to FSC-BT1035

Table 3.3: Signal-processing blocks and their roles. The chain is stereo end to end; L and R never collapse to mono.

3.3.2 Signal flow

Signal flow (stereo)

```
Input ch2/3 (radio L/R)  -> Si4674 fader  --.
                                            |-> Stereo Mixer
   -> Param EQ
Input ch4/5 (ESP32 L/R)  -> ESP32  fader  --'
   -> Master Volume -> Limiter L / Limiter R -> Output DIG0 /
    DIG1 -> BT1035
```

3.4 Equaliser

The parametric EQ is a two-channel (stereo) block; a single set of bands applies identically to L and R. It is configured with one high-pass and five peaking bands, all starting flat (0 dB) so the default response is neutral and any tone shaping is an explicit choice made at runtime by the firmware.

Band	Type	Frequency	Gain	Q
1	Butterworth High (high-pass)	20 Hz	—	1.41
2	Peaking	100 Hz	0 dB	1.0
3	Peaking	400 Hz	0 dB	1.0
4	Peaking	1 kHz	0 dB	1.0
5	Peaking	3 kHz	0 dB	1.0
6	Peaking	8 kHz	0 dB	1.0

Table 3.4: Parametric EQ bands. Band 1 removes DC and subsonic content; bands 2–6 are the tone controls the firmware adjusts.

Double precision for the high-pass

The high-pass sits at 20 Hz, very low relative to the 48 kHz sample rate, so it uses a double-precision second-order block to avoid coefficient-quantisation noise in the bass. The peaking bands can use single precision.

3.5 Master volume and limiters

A two-channel volume block provides the master level (0 dB default). Two single-channel limiters — one per channel — protect the FSC-BT1035 input from clipping on peaks. Suggested limiter settings are in Table 3.5.

Parameter	Value
RMS TC	50 dB/s
Decay	12 dB/s
Threshold	−1 dB (protective)

Table 3.5: Per-channel limiter settings. The limiter is left fixed (not runtime-adjustable).

3.6 Runtime-controllable parameters

The firmware controls a subset of the DSP at runtime via safeload writes. These are the parameter cells whose addresses must be recorded from the SigmaStudio export (Section 3.8).

Control	Block	Runtime-adjustable
Source levels (2)	Si4674 / ESP32 faders	yes
EQ bands 2–6	Parametric EQ	yes (gain/freq/Q)
Master volume	Multiple volume	yes
High-pass (band 1)	Parametric EQ	fixed
Limiters	Limiter 1 / 2	fixed

Table 3.6: Parameters exposed to the firmware. Fixed blocks are set once in the program and never written at runtime.

3.7 Virtual enhancements (stereo depth and bass boost)

The SigmaStudio export does not include dedicated stereo widener or bass boost blocks. Firmware maps enhancement levels (0–100) onto the existing Param EQ1 bands at runtime:

- **Bass enhance** — peaking boost at 100 Hz (+9 dB max) and 400 Hz (+3 dB max).
- **Stereo enhance** — slight 1 kHz cut, plus 3 kHz and 8 kHz lift for a wider, more present image. This is a psychoacoustic curve, not mid/side processing.

Enhancement levels are stored in `AudioProfile::enhancements` and applied by `core::applyEnhancementsToEq()` before safeload. Base EQ band settings in NVS are preserved; overlays replace affected bands only while the corresponding level is greater than zero.

Use safeload for live changes

All runtime updates to the volume, source levels, and EQ bands must go through the ADAU1701 safeload mechanism. Writing parameter cells directly while audio is playing produces audible clicks.

3.8 Exporting the program

With the schematic complete, the program is exported for the firmware:

1. Enable *Action* → *Enable Short Parameter Name for Export* so the exported cell names are concise.
2. *Action* → *Export System Files*, and choose an output folder.

The export produces the program data (control registers, program RAM, and parameter RAM) as byte sequences, plus the symbolic parameter addresses. The firmware's ADAU1701 driver replays the program data over I^2C at boot, and uses the parameter addresses for safeload updates.

No download needed

Since there is no ADAU1701 attached during development, do *not* use *Link Compile Download* (it would fail trying to reach the chip). *Export System Files* compiles the schematic and writes the files directly.

CHAPTER 4

Firmware Architecture

The DigiRadio firmware runs on the ESP32-S3 and orchestrates three companion chips into a single high-fidelity audio path: the Si4684 DAB+/FM tuner, the ADAU1701 SigmaDSP, and the FSC-BT1035 Bluetooth transmitter. This section describes how the firmware is organised, how audio flows through the system, and how each chip is controlled. The day-to-day coding conventions (style, documentation, review rules) are intentionally kept out of this manual and live in the repository's `CONTRIBUTING.md` and `AGENTS.md`.

4.1 Design principles

Two ideas shape the whole codebase:

- **Strong typing.** Domain quantities (frequency, gain, station identifier, band) are dedicated types rather than raw integers or floats. Invalid states are made unrepresentable: input from the network, UART, or flash is validated once at the boundary into a domain type, and trusted thereafter.
- **Functional core, imperative shell.** All hardware-free logic — coefficient math, boot-image framing, configuration parsing, station-list operations — lives in a pure *core* that compiles and is unit-tested on the host, with no dependency on ESP-IDF. Every access to the hardware (I^2C, SPI, UART, flash) lives in a thin *shell* around that core. This keeps the logic testable without a board attached.

4.2 Layered structure

The firmware is organised in three layers, with dependencies pointing inward only (Figure 4.1). The outer shell may depend on the services and the core; the pure core

depends on nothing outside itself.

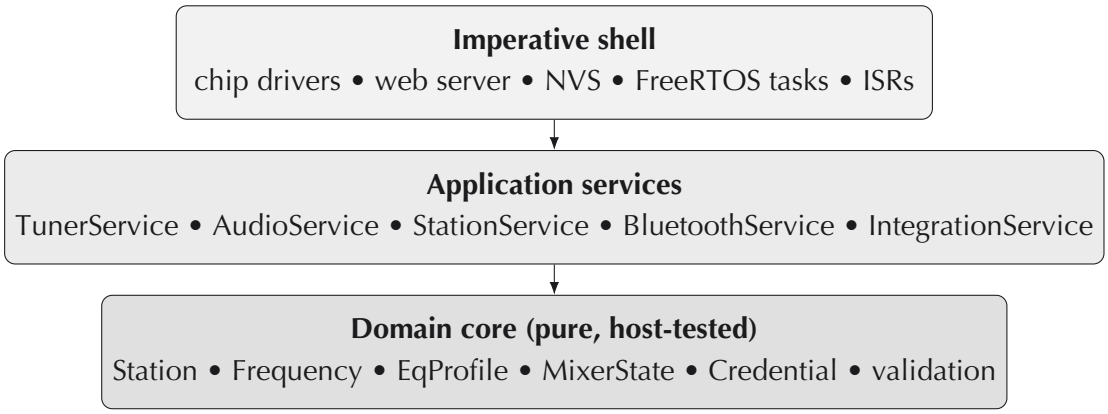

Figure 4.1: Firmware layering. Dependencies point inward; the pure core has no hardware dependencies.

Application services take driver *interfaces* (`ITuner`, `IDsp`, `IBtModule`, `ISecureStore`) injected at construction. Because the services depend on abstractions rather than concrete hardware, the same logic can be exercised against fakes in host tests and against real silicon on the device.

4.3 Audio signal path

The audio chain is built around sound quality. The Si4684 produces the DAB+/FM audio stream; the ADAU1701 applies equalisation and mixes it with the ESP32 audio source; the FSC-BT1035 streams the result over Bluetooth with aptX Adaptive (Figure 4.2).

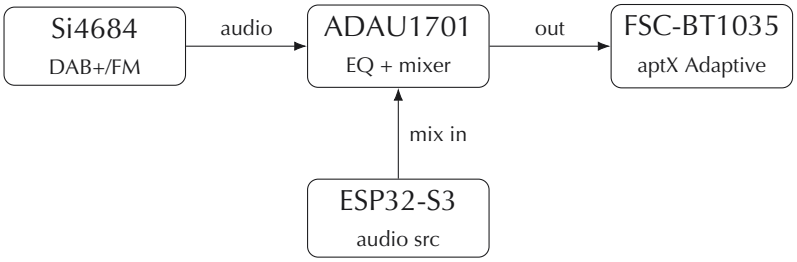

Figure 4.2: Audio signal path. The ADAU1701 input mixer selects/blends the Si4684 and ESP32 sources; equalisation is applied before the Bluetooth output stage.

Equalisation and input mixing are exposed to the rest of the firmware as typed operations — setting an EQ band from a gain, centre frequency, and Q, or setting the mix

level of a given source — never as raw DSP cell addresses. The biquad coefficient math runs in the pure core and is verified against reference values in host tests.

4.4 Chip control and boot

Si4684 (tuner). At power-up the driver follows the documented boot sequence: power on, load the patch/bootloader, load the firmware image (FM or DAB), then boot. Firmware images are large and are streamed to the chip in bounded chunks from flash rather than buffered whole in RAM. See Chapter 5 for blob sources, uGreen extraction, and the full `Si4684Driver` API (tuning, RSQ, DAB service list).

ADAU1701 (DSP). The board carries no self-boot EEPROM. Instead, the ESP32 writes the SigmaStudio-exported program into the DSP's RAM at *every* boot. Runtime changes to equalisation and mixing use the ADAU1701 safeload mechanism, so parameter updates are click-free while audio is playing. Which blocks are runtime-controllable (source faders, EQ bands 2–6, master volume) versus fixed at export (high-pass, limiters) is listed in Chapter 3, Section 3.6. The full driver stack, parameter map, `AudioService`, and HTTP routes are documented in Chapter 6.

Safeload

Writing DSP parameter cells directly while audio is running produces audible clicks. All runtime EQ and mixer changes must go through the safeload registers.

FSC-BT1035 (Bluetooth). The module is controlled by AT commands over UART with hardware flow control. The initialisation sequence includes enabling Line-In mode (`AT+AUXCFG=1`), which is required for the wired audio path from the DSP; command responses are parsed explicitly, with timeouts treated as errors. Full driver API, boot flow, and error codes are in Chapter 7.

4.5 Configuration, storage, and user interface

Network provisioning is implemented as an explicit state machine (`net::NetState`): on first boot (or when STA join fails) the device opens a setup SoftAP (`DigiRadio-setup`) and serves a minimal gzipped web UI from flash. After the user submits

credentials via `POST /api/wifi`, values are validated in the pure core, persisted through `core::ISecureStore`, and the device reboots into STA mode on the next boot. The HTTP endpoints and JSON schemas are documented in Chapter 8.

Implemented (fw 0.8.3).

- **Network** — `GET /api/health`, `POST /api/wifi`, SoftAP/STA state machine (`NetState`), tabbed gzipped web UI.
- **Tuner** — `/api/tuner/*` (FM/DAB tune, seek, services, play); RDS and DAB dynamic labels in status JSON.
- **Audio** — `/api/audio/*` (profile, reset, stereo/bass enhance); six-band EQ via `AudioService`.
- **Bluetooth** — `/api/bluetooth/*` (pair, stop, disconnect, status).
- **Presets** — `/api/stations/*` including reorder and integrated recall via `IntegrationService`.
- **Storage** — `NvsSecureStore`, `NvsAudioProfileStore`, encrypted NVS + flash encryption (development mode); keys in `nvs_keys` partition; init via `initEncryptedStorage()`.

Legacy note (Slices 1–2). The first slices introduced health, Wi-Fi provisioning, and the secure-store boundary; the bullets above supersede the original slice-scoped list.

Sensitive data uses `core::Secret` where applicable so values cannot be logged or implicitly converted to a string; buffers are cleared on destruction. Firmware 0.8.3 enables NVS and flash encryption at rest via `secure_store::initEncryptedStorage()` (see `docs/security-flash-nvs.md` and Chapter 8.4).

4.6 Companion-chip boot at power-up

The ESP32-S3 loads both audio companion chips from assets under `Software/Firmware/` during `app_main`, before network bring-up:

1. **Si4684** (SPI): ROM patch `rom_patch_016.bin`, then either DAB image `dab_firmware.bin` or FM image `fm_firmware.bin`, following AN649 (Chapter 5). DAB blob default from PE5PVB; FM from eval pack or uGreen `radio_cli`. Blobs are local-only (`.gitignore`); embedded via ESP-IDF `EMBED_FILES`.

2. **ADAU1701** (I^2C): SigmaStudio export (`DigiRadio_IC_1.h`) replayed through `SIGMA_WRITE_REGISTER` after hardware reset. The DSP program lives in RAM only; download runs on every boot (Chapter 6, Section 6.3).

3. **Audio profile**: `audio::AudioService::loadAndApply()` restores the saved `core::AudioProfile` from NVS (or factory defaults) via ADAU1701 safeload before network bring-up.

4. **FSC-BT1035** (UART): `bt1035::Bt1035Driver::boot()` enables Line-In (`AT+AUXCFG=1`) after the DSP path is configured (Chapter 7).

If either driver returns an error, the firmware logs the failure and stops before starting Wi-Fi (fail-closed bring-up).

4.7 Error-handling model

Every operation that can fail returns a typed result (`std::expected<T, Error>`); nothing fails silently and every timeout is an explicit error value. Errors propagate to the layer that can act on them — a driver reports, a service decides (retry, degrade, or surface to the UI), and the top level logs. C++ exceptions are disabled.

4.8 Toolchain

Item	Choice
Framework	ESP-IDF v5.5.x (native)
Language	C++23 (`-std=gnu++23`)
Error model	`std::expected`; exceptions off
Hardware licence	CERN-OHL-S v2
Firmware licence	Apache-2.0

Table 4.1: Firmware toolchain and licensing summary.

The firmware source, build instructions, and development conventions are maintained in the `Software/` directory of the project repository.

CHAPTER 5

Si4684 DAB+/FM Tuner

The Skyworks Si4684 is the primary RF front-end of DigiRadio: it demodulates DAB/DAB+ and FM, decodes RDS, and delivers PCM over I^2S to the ADAU1701. This chapter documents how firmware images are obtained and kept local, how the chip is booted on the ESP32-S3, and how `Si4684Driver` exposes every day-to-day operation without leaking register-level details to application code.

5.1 Role in the audio chain

After boot the Si4684 runs as an I^2S master at 44.1 kHz stereo. The ESP32 loads proprietary application firmware into chip RAM at every cold start (there is no self-boot EEPROM on DigiRadio). Tuning, service selection, and signal-quality reads all go through `si4684::Si4684Driver`; higher layers (`TunerService`, web UI) will depend on that class rather than issuing SPI transactions directly.

Programming guide

Command opcodes, property IDs, and reply layouts follow Skyworks application note **AN649** (Si468x programming guide). The DigiRadio driver aligns with the PE5PVB/SI4684-DAB-Receiver reference for DAB and with community FM images for the FM application blob.

5.2 Firmware images

Three binary blobs live under `Firmware/Si4684-Firmware/`. They are **not** committed to the public Git repository (proprietary Skyworks / uGreen licence); `.gitignore` keeps them on the developer machine only.

Local file	Typ. size	Source
`rom_patch_016.bin`	5796 B	ROM patch (`rom00_patch.016.bin`); refreshed from PE5PVB or uGreen `radio_cli`.
`dab_firmware.bin`	~517–497 KB	DAB application image; default from PE5PVB header extract; optional uGreen `dab_radio_6_0_6.bin`.
`fm_firmware.bin`	~530 KB	FM/HD application; eval CD, dabpi `si46xx_firmware/`, or uGreen `fmhd_radio_5_1_3.bin`.

Table 5.1: Si4684 firmware blobs (local only).

5.2.1 Where images come from

Community projects name the same Skyworks files differently; none of them redistribute the binaries in git:

- **PE5PVB/SI4684-DAB-Receiver** — DAB image and ROM patch as C arrays (`firmware.h`, `Si468xROM.h`). Safe to automate.
- **uGreen DABBoard** (`https://ugreen.eu/downloads/`) — `Files_v16.zip` embeds FM, DAB, and patch inside `radio_cli`; use `tools/extract_ugreen_radio_cli.py`.
- **hitech95/si468x_dab_receiver** — Linux driver only; device tree references `si468x/fmhd_radio_5_1_(` but does not ship the files (closed source).
- **PhilBladen/Radio** — STM32 reference using *external* SPI flash (`FLASH_LOAD`); different hardware path from DigiRadio's `HOST_LOAD` flow.

Licence

Do not push `*.bin` firmware to GitHub. Obtain images from your own eval board, uGreen customer download, or dabpi folder; keep them in `Firmware/Si4684-Firmware/` locally.

5.2.2 Refreshing blobs

```
# DAB + patch from PE5PVB (always safe to re-run):
python3 tools/fetch_si4684_firmware.py --dab-only
```

```
# FM from uGreen radio_cli (Files_v16.zip):
python3 tools/fetch_si4684_firmware.py \
    --from-ugreen-radio-cli ~/Downloads/Files_v16.zip

# All three blobs from uGreen (overrides PE5PVB DAB):
python3 tools/fetch_si4684_firmware.py \
    --from-ugreen-radio-cli ~/Downloads/Files_v16.zip --ugreen-all

# FM from dabpi / Skyworks eval folder:
python3 tools/fetch_si4684_firmware.py --si46xx-dir /path/to/si46xx_firmware
```

Low-level helpers: `tools/extract_si4684_blob.py` (C header or copy), `tools/extract_ugreen_radi` (ELF symbol scrape).

5.3 Boot sequence (HOST_LOAD)

DigiRadio follows AN649 “cold start with HOST_LOAD” (same as PE5PVB), not the `FLASH_LOAD` path used by boards with a pre-programmed SST25V.

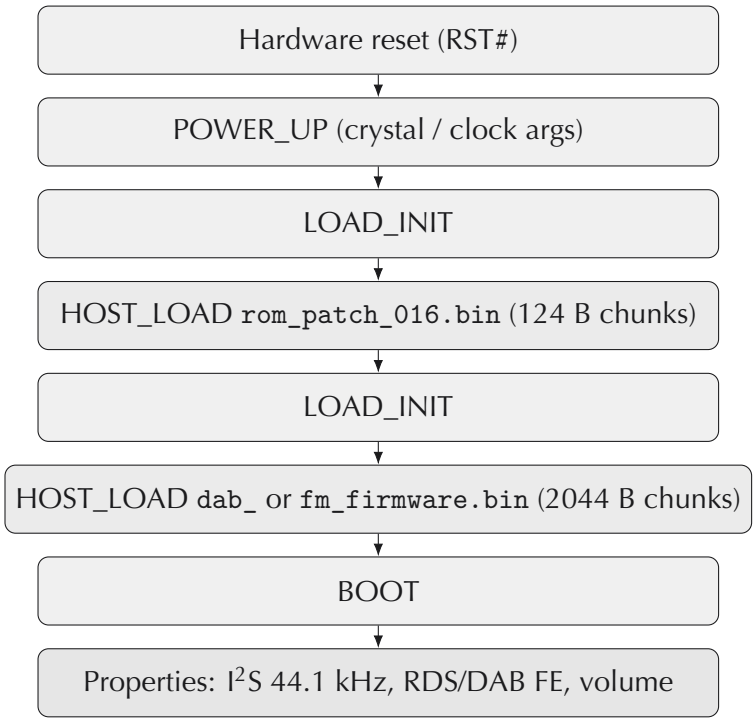

Figure 5.1: Si4684 boot flow on DigiRadio (AN649 / PE5PVB).

Images are embedded in the ESP32 flash partition via ESP-IDF `EMBED_FILES` and streamed through `IFirmwareBlobReader` so the full ~1 MB of patch + DAB + FM never has to fit in RAM at once. Switching band (`Si4684Band::Fm` ↔ `Dab`) performs reset and a complete reload of patch + the other application image.

RSTB before SPI

The first step in `Si4684Driver::boot()` is a hardware reset pulse on RSTB# (GPIO38). `gpio_config` sets `GPIO_PULLDOWN_ENABLE` while enabling the output driver so RSTB cannot glitch high before `gpio_set_level(0)`; this complements the board's external pull-down (Section 2.11.1), which holds reset during power-on before the ESP32 runs. After 5 ms low the line is released, then SPI and the AN649 command sequence follow.

5.4 Si4684Driver API

`Si4684Driver` owns SPI, implements boot, and exposes intent-level methods grouped below. All return `std::expected<T, Si4684Error>`; FM calls fail with `WrongBand` if the FM image is not loaded, and vice versa for DAB.

5.4.1 Bring-up and diagnostics

Method	Purpose
`boot(Si4684Band)`	Cold start + default properties
`isBooted()`, `loadedBand()`	State query
`getPartInfo()`	Chip ID + firmware version
`getSysState()`	Running application type
`setProperty(id, value)`	Generic SET_PROPERTY
`setVolume(0--63)`	Audio attenuator

Table 5.2: Si4684 bring-up and property access.

5.4.2 FM operations

Requires `boot(Si4684Band::Fm)`.

- `tuneFm(frequencyKhz)` — FM_TUNE_FREQ, waits for STC.

- `seekFm(up, wrap)` — FM_SEEK_START; returns tuned kHz.
- `readFmRsq()` — RSSI, SNR, stereo flag, validity.
- `readFmRds()` — last RDS group blocks A–D (FIFO depth).
- `readDabServiceData(statusOnly, ack)` — DAB data-service payload (DLS when `DATA_SRC=2`).

5.4.3 DAB operations

Requires `boot(Si4684Band::Dab)`. The default Band III frequency plan (38 channels, Table 5.3) is installed automatically after boot.

- `installDefaultDabFrequencyPlan()` — DAB_SET_FREQ_LIST (also called from boot).
- `tuneDab(freqIndex)` — tune ensemble by index 0–37.
- `readDabDigRadStatus()` — FIC quality, CNR, lock.
- `readDabEventStatus()` — service-list-ready flag.
- `fetchDabServiceList()` — parsed services + labels.
- `startDabService(serviceId, componentId)` — begin audio decode (START_DIGITAL_SERVICE).

Index	Centre frequency (kHz)
0–3	174928 – 180064 (5A–5D)
4–7	181936 – 187072 (6A–6D)
8–11	188928 – 194064 (7A–7D)
12–15	195936 – 201072 (8A–8D)
16–19	202928 – 208064 (9A–9D)
20–23	209936 – 215072 (10A–10D)
24–27	216928 – 222064 (11A–11D)
28–31	223936 – 229072 (12A–12D)
32–35	230784 – 235776 (13A–13D)
36–37	237488 – 239200 (13E–13F)

Table 5.3: DAB Band III plan (`kDefaultDabFrequencyKhz`).

5.4.4 Typical DAB session

1. `boot(Dab)` at power-up (already done in `HardwareBootstrap`).
2. `tuneDab(index)` for the desired ensemble (Band III index 0–37, Table 5.3).
3. Poll `readDabDigRadStatus()` until FIC quality > 0.
4. `fetchDabServiceList()` when `readDabEventStatus().serviceListReady`.
5. `startDabService(serviceId, componentId)` for the chosen programme; PCM appears on I²S to the ADAU1701.

DAB, not DVB

DigiRadio receives **DAB/DAB+** (Digital *Audio* Broadcasting), not DVB-T/T2 video. Tuning is by ensemble frequency index and digital service/component IDs — there is no transport-stream PAT/PMT scan like DVB-T.

5.4.5 Typical FM session

FM requires the FM application image loaded at boot (`boot(Si4684Band::Fm)`). DigiRadio defaults to DAB at power-up; switching to FM reloads patch + `fm_firmware.bin` (full HOST_LOAD).

1. `boot(Fm)` — resets the chip and loads the FM image.
2. `tuneFm(frequencyKhz)` — e.g. 101500 for 101.5 MHz; waits for seek/tune complete (STC).
3. `readFmRsq()` — RSSI, SNR, stereo flag, validity.
4. Optional: `seekFm(up, wrap)` — scan to next station.
5. Optional: `readFmRds()` — last RDS group (PI, PS, RT).

FM audio is emitted on the same I²S pins as DAB once the FM image is running.

5.4.6 Choosing DAB or FM

Only one application image runs at a time. `Si4684Driver::boot(band)` selects the blob:

Band	Image loaded	Primary API
`Si4684Band::Dab`	`dab_firmware.bin`	`tuneDab`, service list, `startDabService`
`Si4684Band::Fm`	`fm_firmware.bin`	`tuneFm`, `seekFm`, RSQ, RDS

Table 5.4: Band-specific firmware and driver entry points.

Calling an FM method while the DAB image is loaded (or vice versa) returns `Si4684Error::WrongBand`. Switching band performs reset and a complete HOST_LOAD of patch + the other image (∼1 s).

5.4.7 HTTP API mapping (web UI)

The setup UI and REST clients use `tuner::TunerService`, which wraps `Si4684Tuner` (Chapter 8). Summary:

Endpoint	Band	Action
`POST /api/tuner/tune`	DAB	`{"band":"dab","freq_index":0..37}`
`POST /api/tuner/tune`	FM	`{"band":"fm","frequency_khz":64000..108000}`
`GET /api/tuner/services`	DAB only	Programme list for ensemble
`POST /api/tuner/play`	DAB only	Start `service_id/component_id`
`POST /api/tuner/seek`	FM only	Seek up/down, return new kHz
`GET /api/tuner/status`	both	Locked state, RSQ or DIGRAD

Table 5.5: Tuner HTTP routes vs band (full schemas in Chapter 8).

FM tune via HTTP today

`POST /api/tuner/tune` with `band:"fm"` calls `TunerService::tuneFm`, which may trigger a band reload if the device booted in DAB. Plan station presets and automatic band selection in a later slice.

5.5 Integration at power-up

`main/hardware_bootstrap.cpp` constructs a static `Si4684EmbeddedImages`, `Si4684Driver`, and `Si4684Tuner`, calls `boot(Si4684Band::Dab)` before Wi-Fi, then boots the ADAU1701. Failure is fail-closed (logged, no network start). `main/main.cpp` wraps the tuner adapter in `tuner::TunerService` and passes it to `NetBootstrap::start()` so HTTP routes under `/api/tuner/*` can tune and play at runtime.

5.6 Further reading

- Skyworks AN649 — command and property reference.
- `Firmware/Si4684-Firmware/README.md` — blob refresh cheatsheet.
- PE5PVB `si4684.cpp` — DAB service list and MOT (not yet ported).
- uGreen `radio_cli_RELEASE_NOTES.md` — embedded firmware versions.

CHAPTER 6

ADAU1701 Audio DSP

The Analog Devices ADAU1701 is the central audio processor of DigiRadio: it mixes the Si4684 radio stream with the ESP32 I²S source, applies parametric equalisation and master volume, and feeds the FSC-BT1035 over I²S. This chapter documents how the SigmaStudio program is stored and replayed at boot, how `adau1701::Adau1701Driver` controls the chip over I²C, and how application code (`audio::AudioService`, HTTP, web UI) uses the driver without touching parameter RAM addresses directly.

Companion chapter

The SigmaStudio *design* — hardware settings, signal chain, EQ bands, and export procedure — is documented in Chapter 3. This chapter covers the *firmware side*: boot, safeload, driver API, domain types, persistence, and HTTP integration.

6.1 Role in the audio chain

After boot the ADAU1701 runs as the I²S **master** at 48 kHz, 24-bit stereo. It receives PCM from the Si4684 on serial input 0 and from the ESP32 on serial input 1, mixes and equalises the result, and outputs to the Bluetooth module on `SDATA_OUT0`. The ESP32 never processes audio samples in software; all tone shaping and mixing happen inside the DSP program exported from SigmaStudio.

Every runtime adjustment — input level, EQ band, master volume, stereo/bass enhancement — flows through typed domain objects in `components/core` and reaches the hardware only via `Adau1701Driver` safeload writes. Board wiring and I²C address are in Section 2.4 (Chapter 2).

6.2 Program image (SigmaStudio export)

Unlike the Si4684 blobs (local-only, not in git), the ADAU1701 export is **committed** under `Firmware/ADAU1701-Firmware/`. It is the authoritative DSP program for firmware builds.

File	Role	Used by firmware
`DigiRadio_IC_1.h`	Program + parameter RAM init data	`default_download_IC_1()` re played at boot
`DigiRadio_IC_1_PARAM.h`	Parameter cell indices	`ADDR_*` macros; safeload targets
`DigiRadio_IC_1_REG.h`	Control register map	Safeload trigger registers
`DigiRadio_NetList.xml`	Schematic netlist	Reference when re-exporting
`DigiRadio.params`	Parameter metadata	SigmaStudio round-trip
`SigmaStudioFW.h`	Safeload / I2C glue header	`sigma_safeload_*` API

Table 6.1: ADAU1701 SigmaStudio export files (in git).

The driver component wraps the generated C in three layers: `adau1701_program.c` calls `default_download_IC_1()`; `SigmaStudioFW.c` implements `SIGMA_WRITE_REGISTER_BLOCK` and safeload on ESP-IDF I^2C; `Adau1701Driver.cpp` adds reset, boot state, and typed runtime control.

6.2.1 Refreshing after a SigmaStudio change

When the schematic changes in SigmaStudio (new block, different EQ layout, renamed cells):

1. Edit the project offline (Chapter 3, Section 3.8).
2. *Action* → *Export System Files* into `Firmware/ADAU1701-Firmware/`, overwriting the generated headers.
3. Copy or reconcile any new `ADDR_*` symbols into `Adau1701ParamMap.hpp` if block names moved (fader/EQ addresses must match `DigiRadio_IC_1_PARAM.h`).
4. Rebuild firmware; the ESP32 will replay the new download on next boot.
5. Update Chapter 3 tables if band frequencies or the signal chain changed.

No USBi download on DigiRadio

Do *not* rely on SigmaStudio *Link Compile Download* on this hardware path. Export-only workflow is sufficient: the ESP32 is the programmer at every power-

up.

6.3 Boot sequence (I²C RAM load)

DigiRadio has no ADAU1701 self-boot EEPROM. The ESP32 holds the only copy of the DSP program and writes it into RAM after each reset.

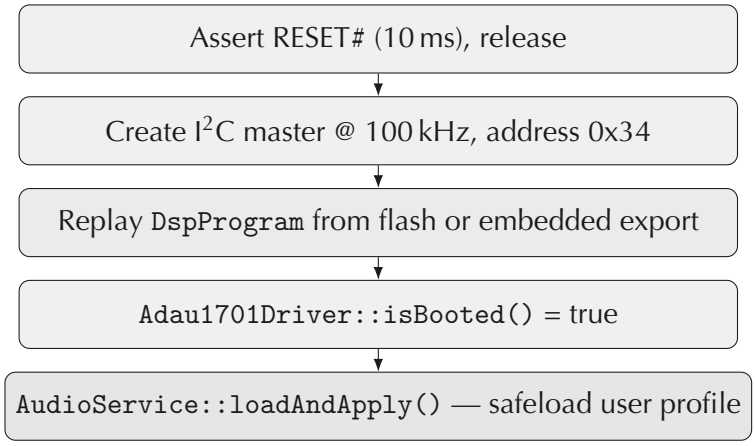

Figure 6.1: ADAU1701 boot flow on DigiRadio (host RAM load).

`SIGMA_WRITE_REGISTER_BLOCK` streams each write in 64-byte I²C chunks (payloads may exceed 255 bytes). The active script comes from the `dsp` data partition when a valid `DRAD` blob is present; otherwise the embedded SigmaStudio export is used (Section 8.2.10). After download, `loadAndApply()` restores the saved `core::AudioProfile` from NVS (or factory defaults) so user settings survive power cycles without re-writing the whole program.

6.4 Software architecture

Application code never includes `DigiRadio_IC_1_PARAM.h`. The stack separates concerns as follows:

`AudioService` merges `AudioProfile::enhancements` into the effective EQ via `core::applyEnhancemer` before calling `IDsp::applyEq()` or `applyProfile()`. Base EQ settings in NVS stay unchanged; enhancement bands are overlays (Section 3.7).

Layer	Type	Responsibility
HTTP / UI	`SetupWebServer`, web UI	JSON → `AudioService`
Service	`audio::AudioService`	Profile in RAM, NVS, enhancement overlay
Domain	`core::AudioProfile`, `IDsp`	Host-testable types, no ESP-IDF
Adapter	`adau1701::Adau1701Dsp`	Maps `DspError` ↔ `Adau1701Error`
Driver	`adau1701::Adau1701Driver`	I²C, boot, safeload
Generated + glue	`DigiRadio_IC_1.h`, `SigmaStudioFW.c`	SigmaStudio export, register writes

Table 6.2: ADAU1701 control stack (Slice 5).

6.5 Adau1701Driver API

`Adau1701Driver` owns the I²C bus (RAII), drives RESET#, and exposes intent-level methods. All return `std::expected<T, Adau1701Error>`. Runtime methods require a prior successful `boot()`.

6.5.1 Bring-up and state

Method	Purpose
`boot()`	Reset chip, init I²C, run `default_download_IC_1()`
`isBooted()`	`true` after successful download

Table 6.3: ADAU1701 bring-up.

6.5.2 Full profile and partial updates

Method	Purpose
`applyProfile(profile)`	Safeload mixer + EQ + master in one call
`applyMixer(mixer)`	Si4684/ESP32 faders + St Mixer1 L/R
`applyEq(eq)`	All six PEQ bands (band 0 skipped, see below)
`setInputVolume(source, L, R)`	One source path (`MixSource::Si4684` or `Esp32`)
`setMasterVolume(L, R)`	Multiple 1 master output
`setEqBand(band, gain, center, q)`	Design biquad in core, safeload five coefficients

Table 6.4: Runtime safeload entry points.

High-pass band fixed

`setEqBand` and `applyEq` *skip* band index 0 (SigmaStudio band 1, 20 Hz Butterworth high-pass). Calls with index 0 return `Adau1701Error::InvalidParameter`. Only peaking bands 1–5 are runtime-adjustable, matching Table 3.6.

6.5.3 Gain and EQ coefficient path

Volume cells (faders, master) store linear gain as ADAU **8.23 fixpoint**. `core::gainDbToLinearFixpoint(` converts `GainDb` before `sigma_safeload_param()`.

PEQ bands use `core::designPeakingEq()` at 48 kHz sample rate to produce five biquad coefficients (`B0..B2, A0..A1`), converted to fixpoint and written atomically with `sigma_safeload_block()` (up to five parameters per transfer — one band per call).

Host tests in `components/core/test/biquad_design_test.cpp` lock the coefficient math; `enhancements_design_test.cpp` locks enhancement curves.

6.6 Parameter address map

SigmaStudio assigns small integer indices to parameter RAM cells. The export header `DigiRadio_IC_1_PARAM.h` defines `ADDR_*` symbols; `Adau1701ParamMap.hpp` maps domain concepts to those addresses:

Domain	`ADDR_*`	SigmaStudio block
Si4684 L/R	`SI4674`, `SI4674_1`	Si4674 fader
ESP32 L/R	`ESP32`, `ESP32_1`	ESP32 fader
Mix L/R	`STMIXER1_ST0/1_VOLUME`	St Mixer1
EQ band n	`PARAMEQ1_ST`n`_B0` + 0..4	Param EQ1 coeffs
Master L/R	`MULTIPLE1`, `MULTIPLE1_1`	Multiple 1

Table 6.5: Runtime parameter map (current export). Re-verify after SigmaStudio re-export.

If a schematic rename changes cell names, update `Adau1701ParamMap.hpp` and this table together with `DigiRadio_IC_1_PARAM.h`.

6.7 Safeload mechanism

Direct parameter RAM writes during playback cause audible clicks. The ADAU1701 provides *safeload* registers: the host stages up to five (address, value) pairs, then triggers a single atomic update between samples.

DigiRadio implements this in `SigmaStudioFW.c`:

1. Write each fixpoint value to `SAFELOAD_DATA_0..4`.
2. Write each target parameter index to `SAFELOAD_ADDR_0..4`.
3. Pulse `CORE_CONTROL` with `IST_TRIGGER`.

`sigma_safeload_param()` handles one cell (volume fader). `sigma_safeload_block()` handles up to five (full PEQ band). `Adau1701Driver` never writes parameter data registers directly for runtime control.

Safeload only for live changes

Boot-time programming uses `SIGMA_WRITE_REGISTER_BLOCK` to fill program and parameter RAM from the export. After audio is running, mixer and EQ updates must use safeload only.

6.8 Domain model and persistence

`core::AudioProfile` is the unit of persistence and HTTP serialisation:

- `MixerState` — six `GainDb` fields (Si4684 L/R, ESP32 L/R, St Mixer1 L/R).
- `EqProfile` — six bands; index 0 fixed high-pass, 1–5 peaking (100 Hz–8 kHz defaults per Chapter 3.4).
- `masterLeft`, `masterRight` — Multiple 1.
- `AudioEnhancements` — `stereo_level` and `bass_level` (0–100), mapped to PEQ overlays at apply time.

JSON parse/serialise lives in `core::AudioProfileJson` (pure core, host tested). NVS key `audio_profile_json` in namespace `digiradio` via `secure_store::NvsAudioProfileStore`.

`AudioProfile::factoryDefault()` is flat EQ, 0 dB gains, enhancements off. `POST`

`/api/audio/reset` restores and persists this snapshot.

6.9 audio::AudioService

`AudioService` is the only entry point HTTP and future UI code should use for audio path control:

Method	Effect
`loadAndApply()`	Load NVS (or default), safeload full profile
`applyProfile(p, persist)`	Replace profile, safeload, optional NVS
`setInputVolume(source, L, R, persist)`	One input path
`setMasterVolume(L, R, persist)`	Master fader
`setEqBand(band, ...)`	Update base EQ + effective overlay
`setStereoEnhance(level, persist)`	PEQ overlay bands 3–5
`setBassEnhance(level, persist)`	PEQ overlay bands 1–2
`currentProfile()`	Snapshot for `GET /api/audio/profile`

Table 6.6: `AudioService` intent-level API.

On boot, `HardwareBootstrap::boot()` runs Si4684 boot, then `Adau1701Driver::boot()`, then `loadAndApply()`. ADAU boot failure is fail-closed (no network start); profile apply failure is logged as warning but boot continues.

6.10 HTTP and web UI

REST routes (Chapter 8, Section 8.2.8) delegate to `AudioService`:

- `GET/PUT /api/audio/profile` — full JSON profile.
- `POST /api/audio/reset` — factory flat profile.
- `POST /api/audio/stereo-enhance` — body `{"level":0..100}`.
- `POST /api/audio/bass-enhance` — same schema.

The gzipped setup page (`components/net/www/index.html`) is a tabbed SPA: a *Now playing* view (RDS/DLS metadata), tuner and preset panels, master/path levels, six PEQ band gain sliders, enhancement overlays, Bluetooth controls, and Wi-Fi provisioning. Saving sends `PUT /api/audio/profile`; enhancement sliders call the dedicated POST

routes for immediate safeload + persist. Regenerate `index.html.gz` with `tools/gzip-www.sh` after editing the HTML source.

6.11 Typical usage patterns

6.11.1 C++ firmware (direct service access)

1. After `HardwareBootstrap::boot()`, obtain `HardwareBootstrap::audioService()`.
2. Adjust levels: `setMasterVolume(GainDb::tryFromDb(-3), ..., true)`.
3. Tune one EQ band (index 2 = 400 Hz peaking in default map): `setEqBand(band, gain, center, q, true)`.
4. Read back: `currentProfile()` for logging or display.

6.11.2 HTTP client

```
# Read current snapshot:
curl http://digiradio.local/api/audio/profile

# Boost bass enhance to 60%:
curl -X POST http://digiradio.local/api/audio/bass-enhance \
  -H "Content-Type: application/json" -d '{"level":60}'

# Restore factory flat:
curl -X POST http://digiradio.local/api/audio/reset
```

6.11.3 After SigmaStudio EQ layout change

Re-export, rebuild firmware, flash ESP32. Existing NVS profiles remain valid only if band indices and centre frequencies still match; otherwise reset via `POST /api/audio/reset` or migrate JSON manually.

6.12 Error handling

`Adau1701Error` values and typical causes:

Error	Typical cause
`ResetFailed`	GPIO reset line configuration
`I2cInitFailed`	Bus or device add on ESP32
`DownloadFailed`	I²C failure during boot download
`NotBooted`	Runtime call before `boot()`
`SafeloadFailed`	I²C or safeload trigger failure
`InvalidParameter`	EQ band 0 or out-of-range domain input

Table 6.7: ADAU1701 driver errors.

`Adau1701Dsp` maps these to `core::DspError` for `AudioService`. HTTP handlers map store/DSP failures to `{"status":"error","reason":"store_failed"}` with HTTP 500.

6.13 Integration at power-up

Static instances in `main/hardware_bootstrap.cpp`:

```
Adau1701Driver  -> Adau1701Dsp -> AudioService(NvsAudioProfileStore)
```

Boot order: Si4684 `boot(Dab)` → ADAU1701 `boot()` → `loadAndApply()` → network stack. `main/main.cpp` passes `audioService()` to `NetBootstrap::start()` for HTTP registration.

6.14 Further reading

- Chapter 3 — schematic, EQ bands, export workflow.
- Chapter 8 — JSON schemas for `/api/audio/*`.
- `Firmware/ADAU1701-Firmware/README.md` — export file cheatsheet.
- Analog Devices ADAU1701 datasheet — safeload and I²C protocol.
- `components/core/test/` — biquad and profile JSON host tests.

CHAPTER 7

FSC-BT1035 Bluetooth Transmitter

The Feasycom FSC-BT1035 (Qualcomm QCC3056) is the wireless output stage of DigiRadio: it receives PCM from the ADAU1701 over I²S and streams Bluetooth audio with aptX, aptX HD, and aptX Adaptive. This chapter documents how the ESP32-S3 controls the module over UART (AT commands with RTS/CTS), why Line-In mode is mandatory, and how `bt1035::Bt1035Driver` implements the bring-up sequence.

Hardware context

Board wiring (UART pins, I²S to the module, flow control) is in Chapter 2, Section 2.5. The ADAU1701 master clock and limiter settings that feed the module are in Chapter 6 and Chapter 3.

7.1 Role in the audio chain

The BT1035 is an I²S **slave** at 48 kHz: LRCLK and BCLK come from the ADAU1701; PCM arrives on `SDATA_OUT0` (ADAU MP6 → module pin 5). The ESP32 does not process audio samples for Bluetooth; it only configures the module so the wired path is accepted and encoded for transmission.

Without firmware init the module may stay in a default mode that ignores the Line-In from the DSP. The mandatory `AT+AUXCFG=1` command selects auxiliary/Line-In input — omitting it silently breaks the entire wireless output (Section 7.3).

7.2 Control interface

7.2.1 UART parameters

Setting	Value
Port	UART2 (not the console UART)
Baud rate	115200
Data format	8N1
Flow control	Hardware RTS/CTS (required)
Reset	GPIO active-low pulse at boot
SYS_CTL	Held active to enable the module

Table 7.1: FSC-BT1035 UART and control lines (`board_pins.hpp`).

7.2.2 Response handling

Every command expects a module reply containing `OK` or `ERROR`. The driver:

- builds lines in the pure core via `core::buildBt1035AtLine()`;
- classifies replies with `core::parseBt1035AtResponse()`;
- treats timeouts and unexpected payloads as errors (never ignored).

Host tests in `components/core/test/bt1035_at_test.cpp` lock the init sequence (including `AT+AUXCFG=1`) and the parser.

7.3 Mandatory Line-In mode

AT+AUXCFG=1 is not optional

The documented init sequence must include `AT+AUXCFG=1` after a successful `AT` ping. This tells the QCC3056 firmware to take audio from the wired I²S/Line-In port (the ADAU1701 output) rather than an internal source. AGENTS.md and the hardware manual both treat skipping this step as a production bug.

7.4 Boot sequence

At power-up `HardwareBootstrap::boot()` runs the Si4684 and ADAU1701 first, applies the saved audio profile, then initialises the BT1035 so the Line-In path is ready before Wi-Fi starts.

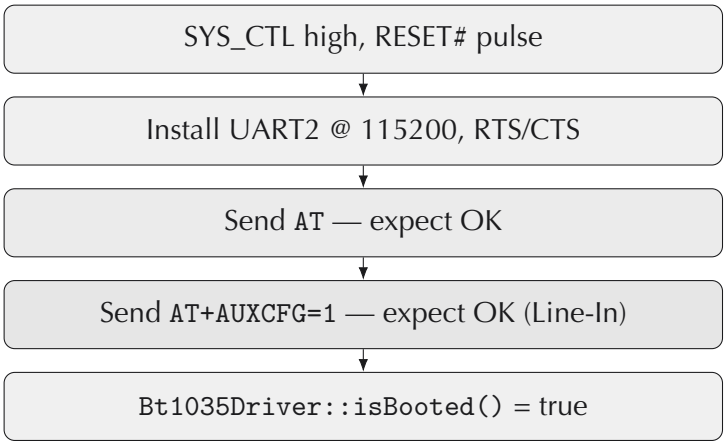

Figure 7.1: FSC-BT1035 AT init on DigiRadio.

Pairing, codec selection, and volume over Bluetooth are handled by the module's own firmware and NVS; DigiRadio firmware currently implements only the Line-In bring-up required for the wired audio path.

7.5 Software architecture

Layer	Type	Responsibility
Bootstrap	`HardwareBootstrap`	Constructs driver, calls `boot()` after ADAU1701
Driver	`bt1035::Bt1035Driver`	UART, reset, AT init sequence
Pure core	`core::Bt1035AtCommand`, parser	Host-testable command strings and OK/ERROR classification

Table 7.2: BT1035 control stack (Slice 6–7). Pairing and A2DP status are exposed on `/api/bluetooth/*` via `BluetoothService`.

7.6 Supported AT command subset

The firmware enumerates every command it sends. Extending the subset requires updating `core::Bt1035AtCommand`, the manual, and a host test.

7.7 Bt1035Driver API

Enum	Line sent	Purpose
`Ping`	`AT`	Verify UART link
`AuxLineIn`	`AT+AUXCFG=1`	Enable Line-In from ADAU
`PairDiscoverable`	`AT+PAIR=1`	Enter discoverable mode
`PairHidden`	`AT+PAIR=0`	Leave discoverable mode
`A2dpStat`	`AT+A2DPSTAT`	Read link state
`A2dpDisconnect`	`AT+A2DPDISC`	Release A2DP session
`QueryName`	`AT+NAME`	Read module friendly name
`QueryAutoConn`	`AT+AUTOCONN`	Read auto-reconnect count
`QueryPairedList`	`AT+PLIST`	List paired remotes

Table 7.3: Enumerated AT commands (`core::Bt1035AtCommand`). Boot uses Ping + AuxLineIn only; pairing commands are runtime.

`Bt1035Driver` owns UART and GPIO reset. All methods return `std::expected<T, Bt1035Error>`.

Method	Purpose
`boot()`	Reset, UART init, run `bootInitSequence()`
`isBooted()`	`true` after Line-In init succeeded
`sendCommand(cmd)`	Send one typed command, expect OK
`enterPairingMode()`	`AT+PAIR=1`
`leavePairingMode()`	`AT+PAIR=0`
`queryA2dpState()`	`AT+A2DPSTAT`, parse `+A2DPSTAT=`
`disconnectA2dp()`	`AT+A2DPDISC`
`queryDeviceName()`	`AT+NAME`, parse `+NAME=`
`queryAutoReconnect()`	`AT+AUTOCONN`, parse `+AUTOCONN=`
`setAutoReconnect(times)`	`AT+AUTOCONN=n` (0–15)
`queryPairedList()`	`AT+PLIST`, parse `+PLIST=` lines

Table 7.4: Public driver API.

7.7.1 Error codes

7.8 Integration at power-up

`main/hardware_bootstrap.cpp` constructs a static `Bt1035Driver` and calls `boot()` after `AudioService::loadAndApply()`. Failure returns `HardwareBootError::Bt1035BootFailed` and `app_main` halts before network bring-up (fail-closed, same as Si4684/ADAU1701).

Boot order:

Bt1035Error	Typical cause
ResetFailed	GPIO configuration failure
UartInitFailed	uart_driver_install / pins
NotBooted	sendCommand before boot()
AtTimeout	No OK/ERROR within 2 s
AtError	Module returned ERROR
UnexpectedResponse	Unrecognised payload (future use)

Table 7.5: Driver error enumeration.

1. Si4684 boot(Dab) — tuner image in RAM.
2. ADAU1701 boot() — SigmaStudio program in RAM.
3. AudioService::loadAndApply() — user mixer/EQ profile.
4. BT1035 boot() — Line-In enabled for wireless output.

7.9 Typical usage (firmware developer)

After a successful HardwareBootstrap::boot(), the module is ready; no further calls are required for basic listening. To re-send Line-In config after a module reset:

```
bt1035::Bt1035Driver& bt = ...;
if (auto r = bt.sendCommand(core::Bt1035AtCommand::AuxLineIn); !r) {
  // handle Bt1035Error
}
```

7.10 Further reading

- Feasycom FSC-BT1035 AT command manual (vendor) — full command set; firmware wraps name, paired list, and auto-reconnect for the Web UI.
- Chapter 2 — pin map and I²S routing.
- Chapter 6 — DSP output that feeds the module.
- components/core/test/bt1035_at_test.cpp — init sequence test.

CHAPTER 8

HTTP API

The setup web interface is a thin client over a typed JSON REST API implemented in `SetupWebServer`. Request bodies are parsed into domain types in the pure core (`components/core`) before any persistence or driver call. Exact C++ signatures live in the generated Doxygen output under `docs/api/`; this chapter documents the wire protocol and behaviour as shipped in firmware 0.8.4.

8.1 Transport and reachability

- **Port:** TCP 80 on the ESP32-S3 HTTP server.
- **Setup mode (`NetState::SoftApSetup`):** join the SoftAP `DigiRadio-setup` (open network), then open `http://192.168.4.1/`. The root path serves a gzipped HTML page embedded from flash.
- **STA mode (`NetState::StaConnected`):** after successful provisioning and reboot, the same API is available at the device's DHCP address on the configured LAN.

All responses use `Content-Type: application/json` except `GET /`, which returns gzipped HTML with `Content-Encoding: gzip`.

8.2 Endpoints

8.2.1 `GET /api/health`

Returns a health-check DTO serialised by `core::serializeHealthStatusJson()` from a `core::HealthStatus` value.

Response schema

JSON

```
{"status":"ok","fw":"0.8.4","serialNumber":"0004A3123456",
 "chips":{"si4684":true,"adau1701":true,"bt1035":true}}
```

- `status` — coarse indicator; `ok` when the firmware is running normally.
- `fw` — firmware release string (`core::FirmwareVersion`).
- `serialNumber` — canonical EEPROM serial, or `unknown` when the 24AA025E48 read fails.
- `chips` — companion-chip boot flags after `HardwareBootstrap::boot()` (Slice 8 integration).

HTTP status: **200 OK** on success.

8.2.2 POST /api/wifi

Accepts Wi-Fi credentials for station (STA) join. The body is parsed by `core::parseWifiProvisionJson()` into a `core::WifiCredentials` value (SSID as `core::WifiSsid`, password as `core::Secret`). On success the credentials are persisted through `core::ISecureStore` (device implementation: `secure_store::NvsSecureStore`) and the device schedules a reboot so the next boot can enter STA mode.

Request schema

JSON

```
{"ssid":"MyNetwork","password":"secret1234"}
```

- `ssid` — required, 1–32 characters (802.11 SSID limits).
- `password` — optional for open networks; for WPA-PSK, 8–63 characters. Validated by `core::WifiCredentials::isPasswordValid()`.

Success response

JSON

```
{"status":"saved","reboot_in_sec":3}
```

Serialised by `core::serializeWifiProvisionSavedJson()`. The device reboots after the indicated delay.

Error response

JSON

```
{"status":"error","reason":"invalid_ssid"}
```

Serialised by `core::serializeWifiProvisionErrorJson()`. Reason tokens (never include secrets): `invalid_json`, `missing_field`, `invalid_ssid`, `invalid_password`, `store_failed`.

HTTP status: **200 OK** on save success; **400 Bad Request** for parse/validation failures; **500 Internal Server Error** when NVS persistence fails.

8.2.3 GET /api/tuner/status

Returns a tuner snapshot serialised by `core::serializeTunerStatusJson()` from `core::TunerStatus`. The handler calls `tuner::TunerService::refreshStatus()`. DAB and FM tuning workflows are in Chapter 5, Sections 5.4.4 and 5.4.5.

Response schema (DAB example)

JSON

```
{"booted":true,"band":"dab","locked":true,"volume":63,
 "dab":{"freq_index":12,"fic_quality":80,"cnr_db":25,
 "playing_service_id":42,"playing_component_id":7,
 "dynamic_label":"Live show"},"fm":null}
```

For FM, `fm` carries `frequency_khz`, `rssi_dbuv`, `snr_db`, `stereo`, optional `station_name` (RDS PS), and `radiotext` (RDS RT); `dab` is `null`.

HTTP status: **200 OK**; **500** with `{"status":"error","reason":...}` on driver

failure; **503** when the tuner service is unavailable. FM responses may include `station_name` and `radiotext`; DAB responses may include `dynamic_label` when a programme is playing.

8.2.4 GET /api/tuner/services

Lists DAB programmes for the current ensemble via `TunerService::listDabServices()`.

Response schema

JSON

```
{"services":[{"service_id":12345,"component_id":1,"label":
   "BBC R1"}]}
```

HTTP status: **200 OK**; **409 Conflict** when the list cannot be fetched (wrong band, empty ensemble, hardware error).

8.2.5 POST /api/tuner/tune

Tunes to a DAB ensemble or FM frequency. Body parsed by `core::parseTunerTuneJson()`.

Request schema

JSON

```
{"band":"dab","freq_index":12}
```

or

JSON

```
{"band":"fm","frequency_khz":101500}
```

DAB `freq_index` is 0–37; FM `frequency_khz` is 64 000–108 000.

On success returns the updated status JSON (same shape as `GET /api/tuner/status`). HTTP status: **200 OK**; **400** for invalid JSON; **409** for tune failures.

8.2.6 POST /api/tuner/play

Starts a DAB audio service. Body parsed by `core::parseTunerPlayJson()`.

Request schema

JSON

```
{"service_id":12345,"component_id":1}
```

Success response: `{"status":"playing"}`. HTTP status: **200 OK**; **409** when playback cannot start.

8.2.7 POST /api/tuner/seek

Seeks FM in the requested direction. Optional JSON body parsed by `core::parseTunerSeekJson()`; an empty body defaults to upward seek. Returns `{"frequency_khz":...}` on success. HTTP status: **200 OK**; **400** for invalid `direction`; **409** on seek failure.

Request schema

JSON

```
{"direction":"up"}
```

Use `"down"` for downward seek. Omit the body or send an empty JSON object (`{}`) for upward seek (backward compatible).

8.2.8 GET /api/audio/profile

Returns the current ADAU1701 audio snapshot serialised by `core::serializeAudioProfileJson()` from `core::AudioProfile`. The handler reads `audio::AudioService::currentProfile()`. Driver behaviour, domain types, and persistence are documented in Chapter 6.

Response schema (excerpt)

JSON

```
{"mixer":{"si4684_left_db":0,"si4684_right_db":0,"
   esp32_left_db":0,
 "esp32_right_db":0,"mix_left_db":0,"mix_right_db":0},
 "master":{"left_db":0,"right_db":0},
 "eq":[{"gain_db":0,"center_hz":40,"q":1.414}, ...],
 "enhancements":{"stereo_level":0,"bass_level":0}}
```

Six EQ bands are always present (`eq` array length 6). Enhancement levels are 0–100; at 0 the base EQ band settings apply.

HTTP status: **200 OK**; **503** when the audio service is unavailable.

8.2.9 PUT /api/audio/profile

Applies a full audio profile. Body parsed by `core::parseAudioProfileJson()`; on success `audio::AudioService::applyProfile(..., persist=true)` safeloads the ADAU1701 and writes NVS key `audio_profile_json`.

Success response

JSON

```
{"status":"saved"}
```

HTTP status: **200 OK**; **400** for parse/validation failures; **500** when safeload or NVS persistence fails.

8.2.10 POST /api/dsp/program

Uploads a framed ADAU1701 RAM download blob (`DRAD` v1, see `core::parseDspProgramBlob()`). The body is raw bytes (`application/octet-stream` or any content type). On success the blob is written to the `dsp` flash partition and the device reboots; the next boot replays the new program before `loadAndApply()`.

Success response

JSON

```
{"status":"stored","reboot_sec":3}
```

HTTP status: **200 OK**; **400** with `{"error":"..."}` for invalid/truncated/CRC failures; **413** when the body exceeds 200 KiB; **500** on flash write failure. Pack blobs with `tools/pack_dsp_program.py` or `core::serializeDspProgramBlob()`.

8.2.11 POST /api/system/ota

Streams a raw ESP-IDF application `.bin` into the inactive OTA slot (`ota_0`/`ota_1`). The body is raw bytes; the server validates the embedded `esp_app_desc_t` at offset 0x20 (magic and `project_name == "digiradio"`) while streaming. On success the new slot is selected and the device reboots; the first healthy boot after Wi-Fi and HTTP are up calls `ota::OtaService::confirmBoot()` to cancel rollback.

Success response

JSON

```
{"status":"stored","reboot_sec":3}
```

HTTP status: **200 OK**; **400** with `{"error":"..."}` for invalid project/magic or flash write failures; **413** when the body exceeds 1.7 MiB (`0x1B0000`); **500** on `esp_ota_end`/set-boot failures. Push with `curl --data-binary @build/digiradio.bin`.

8.2.12 POST /api/audio/reset

Restores `AudioProfile::factoryDefault()` (flat EQ, 0 dB gains), applies it to the DSP, and persists to NVS. Success response: `{"status":"saved"}`. HTTP status: **200 OK**; **500** on apply/persist failure.

8.2.13 POST /api/audio/stereo-enhance

Adjusts stereo depth via a psychoacoustic PEQ overlay on bands 3–5 (1 kHz / 3 kHz / 8 kHz). This is *not* true M/S widening—there is no dedicated SigmaStudio widener block; see Section 3.7.

Request body

JSON

```
{"level":50}
```

`level` is an integer 0–100 (0 = off, 100 = maximum).

Success response: `{"status":"saved"}`. HTTP status: **200 OK**; **400** for invalid JSON or level; **500** when safeload or NVS persistence fails.

8.2.14 POST /api/audio/bass-enhance

Adjusts bass emphasis via a PEQ overlay on bands 1–2 (100 Hz / 400 Hz). Request and response schema match `POST /api/audio/stereo-enhance` (Section 8.2.13).

8.2.15 GET /api/bluetooth/status

Returns BT1035 boot flag, whether discoverable mode was requested, the last read A2DP state, module friendly name, and auto-reconnect count. Serialised by `core::serializeBluetooth`

Response schema

JSON

```
{"booted":true,"pairing":false,"a2dp":"standby",
 "device_name":"DigiRadio-A1B2","auto_reconnect":3}
```

8.2.16 POST /api/bluetooth/pair

Enters discoverable mode (`AT+PAIR=1`). Success: `{"status":"pairing"}`.

8.2.17 POST /api/bluetooth/pair/stop

Leaves discoverable mode (`AT+PAIR=0`). Success: `{"status":"idle"}`.

8.2.18 POST /api/bluetooth/disconnect

Releases the current A2DP session (`AT+A2DPDISC`).

8.2.19 GET /api/bluetooth/paired

Returns paired remotes from `AT+PLIST`, serialised by `core::serializeBluetoothPairedJson()`.

Response schema

JSON

```
{"devices":[{"index":1,"mac":"001122334455","name":"Phone"
   }]}
```

8.2.20 POST /api/bluetooth/auto-reconnect

Sets the module auto-reconnect retry count (`AT+AUTOCONN=0..15`). Body parsed by `core::parseBluetoothAutoReconnectJson()`.

Request schema

JSON

```
{"times":3}
```

Success: `{"status":"saved"}`.

8.2.21 GET /api/stations

Returns presets serialised by `core::serializeStationListJson()`.

8.2.22 POST /api/stations

Adds one preset (`core::parseStationJson()`); persists via `ISecureStore::saveStationListJson()`.

8.2.23 POST /api/stations/remove

Removes a preset by list index (`{"index":0}`).

8.2.24 POST /api/stations/reorder

Reorders presets using `StationList::move()`. Body: `{"from":1,"to":0}`. Success: `{"status":"reordered"}`.

8.2.25 POST /api/stations/tune

Recalls a preset via `integration::IntegrationService::recallPreset()` (tune, re-apply saved audio profile, persist last index).

8.3 Boot and network state machine

At boot, `integration::IntegrationService::startup()` loads presets and best-effort recalls the last station. Then `net::NetBootstrap::start(store, tuner, audio, bluetooth, stations, integration)` consults `ISecureStore::hasWifiCredentials()`:

1. **Credentials present** — create STA netif, connect via `net::StaClient` (30 s timeout). On success: `NetState::StaConnected` and HTTP on the LAN address.
2. **No credentials, or STA join fails** — fall back to SoftAP setup mode (`NetState::SoftApSetup`).

This explicit `enum class NetState` replaces ad-hoc flags; see `net/NetState.hpp` in the source tree.

8.4 Credential storage (Slice 2)

Wi-Fi credentials are stored in NVS namespace `digiradio`, keys `wifi_ssid` and `wifi_pwd`. Audio profiles (non-secret) use the same namespace, key `audio_profile_json`, via `secure_store::NvsAudioProfileStore`. Station presets use key `station_list` (JSON blob). The last recalled preset index is stored as `last_preset` (u8). Passwords are wrapped in `core::Secret` in RAM and are never logged or returned by the API.

Encryption at rest

Firmware 0.8.3+ enables NVS encryption (`CONFIG_NVS_ENCRYPTION`) and flash encryption in development mode (`sdkconfig.defaults`). Keys live in the `nvs_keys` partition; Wi-Fi passwords remain wrapped in `core::Secret` in RAM and are never logged. First upgrade from plain NVS requires `idf.py erase-flash`. HIL checklist: `Software/docs/security-flash-nvs.md`.

CHAPTER 9

Class Reference

This chapter documents the firmware's public classes at the design level: what each class is responsible for, which collaborators it depends on, its key invariants, and how it fits the system. It complements — and does not duplicate — the generated API documentation, which carries the exact method signatures. The HTTP JSON endpoints are documented separately in Chapter 8.

How this chapter grows

Per the development rules, every public, architecturally-significant class (the drivers, the application services, and the public domain-core types) gets a section here, tagged `\label{cls:ClassName}`, added in the same change that introduces the class. A tooling check keeps this chapter in step with the code, so it is always current.

The class reference tracks firmware 0.8.4 on `main`. Public classes are grouped by layer: domain core, application services, and hardware drivers.

9.1 FirmwareVersion

Strong type wrapping the firmware release identifier (e.g. `0.8.3`). Used by `HealthStatus` and the `/api/health` endpoint so version strings are never passed as bare `char*` across module boundaries. Invariant: non-empty at construction.

9.2 FrequencyKHz

Validated FM centre frequency in kilohertz (64 000–108 000). Parsed once from JSON at the HTTP boundary via `FrequencyKHz::tryFromKhz()`; trusted downstream by `ITuner`, `Si4684Driver`, and status DTOs.

9.3 HealthStatus

Immutable health-check DTO returned by `GET /api/health`. Built via `HealthStatus::ok(FirmwareVe` `CompanionChipStatus, serialNumber)` in the pure core; JSON serialisation lives in `serializeHealthStatusJson()`. The `serialNumber` field carries the EEPROM-derived serial or `unknown` when the EUI-48 read fails. No ESP-IDF headers.

9.4 Eui48

Strong type for the factory-programmed six-byte EUI-48 read from the 24AA025E48 at word address `0xFA` (Microchip DS20001191). Exposes `serialNumber()` (12 uppercase hex digits) and `shortSuffix()` (last three bytes) for SSID and hostname derivation.

9.5 DeviceIdentity

Per-board identity derived from `Eui48`: SoftAP SSID `DigiRadio-<suffix>`, Bluetooth name, STA hostname `digiradio-<suffix>`, and canonical serial. `unknown()` yields documented fallbacks (`DigiRadio-setup`, serial `unknown`).

9.6 IDeviceIdentitySource

Port for reading `DeviceIdentity` from board storage. Implemented in the shell by `eeprom24aa::Eeprom24aa` on the shared I2C bus.

9.7 SoftApConfig

Immutable value type holding SoftAP parameters (SSID, channel, station limit). `forSsid()` builds setup parameters from `DeviceIdentity::softApSsid()`; `setupDefault()` keeps the `DigiRadio-setup` fallback when the EEPROM read fails.

9.8 SoftApHost

RAII wrapper around the ESP-IDF Wi-Fi stack that starts and stops the configured SoftAP. Imperative shell; no business logic.

9.9 SetupWebServer

Minimal HTTP server: gzipped setup UI, `GET /api/health`, `POST /api/wifi`, tuner routes (`/api/tuner/*`), audio routes (`/api/audio/*`), Bluetooth (`/api/bluetooth/*`), and station presets (`/api/stations/*`). JSON parsing and serialisation delegate to the pure core; credentials persist via `ISecureStore`; tuner via `tuner::TunerService`; audio via `audio::AudioService`; Bluetooth via `bluetooth::BluetoothService`; presets via `station::StationService`.

9.10 NetBootstrap

Owns network resources for setup or STA mode. Initialises encrypted NVS via `secure_store::initEncryptedStorage()`, then Wi-Fi and the HTTP server. Wires tuner, audio, Bluetooth, station, and integration services into REST handlers. Must outlive `app_main` for the process lifetime.

9.11 Secret

Opaque wrapper for sensitive strings. Buffer is zeroised on destruction; no `operator<<` and no plaintext serialisation. Shell code uses `usePlaintext()` transiently for driver APIs only.

9.12 WifiSsid

Validated Wi-Fi SSID (1–32 bytes). Parsed once from untrusted JSON at the HTTP boundary before persistence or driver calls.

9.13 WifiCredentials

Domain value pairing `WifiSsid` with a `Secret` password. Open networks use an empty password; WPA-PSK requires 8–63 characters.

9.14 ISecureStore

Abstract persistence boundary for credentials and preset data at rest. Wi-Fi credentials, station list JSON, and last-preset index use `NvsSecureStore`; audio profiles use `NvsAudioProfileStore`. Host tests use fakes; the shell uses NVS backends after `initEncryptedStorage()`.

9.15 StaClient

RAII STA join helper with an explicit connect timeout. Assumes `esp_wifi_init()` was already called by `NetBootstrap`.

9.16 NvsSecureStore

`ISecureStore` implementation backed by an NVS namespace. Passwords are stored as NVS strings and never logged. NVS encryption and flash encryption are enabled in `sdkconfig.defaults`; initialisation runs in `secure_store::initEncryptedStorage()` before network bring-up.

9.17 IFirmwareBlobReader

Abstract streaming reader for firmware blobs embedded in flash. The Si4684 driver uses it to HOST_LOAD patch and application images in bounded SPI chunks without allocating the full image on the heap. Host tests use the same interface over in-memory buffers.

9.18 EmbeddedBlobReader

Non-owning `IFirmwareBlobReader` over a contiguous byte range (linker symbols from `EMBED_FILES` or test data). Implements offset-based `read()` with truncation at end-of-blob.

9.19 Si4684EmbeddedImages

Owns `EmbeddedBlobReader` views over `rom_patch_016.bin`, `dab_firmware.bin`, and `fm_firmware.bin`. Constructed once; passed to `Si4684Driver`. `applicationImage(Si4684Band)` selects DAB or FM.

9.20 Si4684Driver

Full SPI driver for the Si4684 DAB+/FM tuner (Chapter 5).

`boot(Si4684Band)` runs AN649 HOST_LOAD, configures I²S and band-specific properties, and records `loadedBand()`. After boot the driver exposes FM tuning/seek/RSQ/RDS, DAB ensemble tuning, digital service list fetch, and `startDabService`. Property access (`setProperty`, `setVolume`) and diagnostics (`getPartInfo`, `getSysState`) are included. Wrong-band calls return `Si4684Error::WrongBand`. Register-level opcodes remain private; see `Si4684Types.hpp` for status DTOs. Full DAB/FM tuning guide: Chapter 5.

9.21 Bt1035PairedDevice

Plain DTO for one `+PLIST=` line from the FSC-BT1035: paired-record index (1–8), 12-digit MAC, and optional friendly name. Parsed by `core::parseBt1035PairedListResponse()` and serialised on `GET /api/bluetooth/paired`.

9.22 Bt1035Driver

UART driver for the FSC-BT1035 (Chapter 7). `boot()` pulses RESET#, opens UART2 with RTS/CTS, and runs `core::bootInitSequence()` (Ping + `AT+AUXCFG=1`). Returns `Bt1035Error` on timeout, ERROR response, or UART failure.

9.23 Adau1701Driver

RAII I²C driver for the ADAU1701 SigmaDSP (Chapter 6). `boot()` asserts RESET#, initialises the shared I2C bus, and replays the SigmaStudio export from `Firmware/ADAU1701-Firmware/` on every power-up (no EEPROM self-boot on DigiRadio). Exposes `i2cBusHandle()` after boot for the shared 24AA025E48 EEPROM. Runtime mixer, EQ, and master volume updates use the ADAU1701 safeload mechanism via `applyProfile()` and related methods (Section 6.5).

9.24 Eeprom24aa

I²C shell driver for the 24AA025E48 identity EEPROM (Chapter 2, address `0x52`). Implements `IDeviceIdentitySource`: reads six bytes at word address `0xFA` per Microchip DS20001191 and builds `DeviceIdentity`. Called from `HardwareBootstrap` after `Adau1701Driver::boot()`.

9.25 Adau1701Dsp

`core::IDsp` adapter over `Adau1701Driver`. Maps domain-level audio control to safeload I²C transactions without exposing parameter RAM addresses to services (Chapter 6, Section 6.4).

9.26 RegisterWrite

One SigmaStudio `SIGMA_WRITE_REGISTER_BLOCK` step: 16-bit address plus payload bytes. Aggregated by `DspProgram`.

9.27 DspProgram

Ordered ADAU1701 RAM download script (pure domain). Parsed from a framed `DRAD` flash blob or built by `EmbeddedDspProgramSource`.

9.28 IDspProgramSource

Port supplying `DspProgram` to `Adau1701Driver::boot()`. Implementations: flash partition, embedded export, and fallback composite.

9.29 EmbeddedDspProgramSource

Builds the factory `DspProgram` from `DigiRadio_IC_1.h` (five writes matching `default_download_IC_1()`

9.30 FlashDspProgramSource

Reads and parses the `dsp` flash partition; `storeBlob()` erases and writes a validated blob from `POST /api/dsp/program`.

9.31 FallbackDspProgramSource

Tries `FlashDspProgramSource` first, then `EmbeddedDspProgramSource` when the partition is empty or invalid.

9.32 GainDb

Validated decibel gain/attenuation (-96 dB to $+12$ dB). Used for input volumes, master level, and EQ band gain. Converted to ADAU 8.23 fixpoint via `gainDbToLinearFixpoint()` before safeload.

9.33 FrequencyHz

Validated PEQ centre frequency (20–20 000 Hz) for the 48 kHz SigmaStudio project. Parsed at the HTTP boundary before coefficient design.

9.34 EqBandIndex

Strong index into the six-band `Param EQ1` module (0–5). Maps to parameter RAM via `adau1701::paramAddrEqBandBase()`.

9.35 EqProfile

Six-band parametric EQ snapshot aligned with the SigmaStudio map (Chapter 3): band index 0 is the fixed 20 Hz high-pass; bands 1–5 are peaking at 100 Hz–8 kHz. Designed in the pure core; peaking bands are safeloaded by `Adau1701Driver::applyEq()` (band 0 is skipped).

9.36 IDsp

Abstract ADAU1701 control boundary. Defines `applyProfile()`, `setInputVolume()`, `setEqBand()`, and related intent-level operations without ESP-IDF types.

9.37 IAudioProfileStore

Persistence boundary for `AudioProfile` (mixer, EQ, master). Device implementation: `NvsAudioProfileStore`; host tests use fakes.

9.38 ITuner

Abstract tuner boundary in the pure core. Defines boot, band selection, status readout, FM/DAB tuning, service list, playback, and volume without ESP-IDF types. `si4684::Si4684Tuner` implements it on device; host tests can use fakes.

9.39 TunerService

Application service exposing intent-level tuner operations to HTTP and future UI. Holds a reference to `core::ITuner`, tracks last tune target and volume, and maps driver failures to `core::TunerError`.

9.40 Si4684Tuner

`ITuner` adapter over `Si4684Driver`. Translates domain calls into SPI commands and maps `Si4684Error` to `TunerError`. Constructed once in `HardwareBootstrap` alongside the driver.

9.41 AudioService

Application service for ADAU1701 mixer, EQ, and master volume (Chapter 6, Section 6.9). Holds a reference to `core::IDsp`, tracks the in-memory `AudioProfile`, loads from `IAudioProfileStore` after boot, and applies changes via safeload. Stereo depth and bass enhance levels are merged into the effective EQ via `core::applyEnhancementsToEq()`. Exposed on `/api/audio/*` and the web UI Audio section.

9.42 EnhanceLevel

Strong type for 0–100 enhancement intensity (stereo depth and bass boost). Validated at the HTTP boundary by `core::parseEnhanceLevelJson()`.

9.43 NvsAudioProfileStore

`IAudioProfileStore` implementation storing serialised `AudioProfile` JSON in NVS namespace `digiradio`.

9.44 StationName

Strong type for a preset label (1–32 bytes). Parsed at the HTTP boundary via `StationName::tryFrom()`.

9.45 BroadcastLabel

Trimmed on-air label from RDS or DAB chip byte buffers (up to 128 bytes). Built by `BroadcastLabel::tryFromChipBytes()` in the pure core.

9.46 RdsMetadataAccumulator

Stateful RDS decoder assembling program service name and radiotext from raw block A–D snapshots returned by the Si4684 driver.

9.47 DabDynamicLabelAccumulator

Reassembles segmented DAB DLS payloads into one dynamic label for `TunerStatus`.

9.48 PresetSlot

Optional hardware preset button index (1–20). Validated by `PresetSlot::tryFrom()`.

9.49 Station

Immutable preset value: name, band (DAB/FM), tune coordinates, optional preset slot. FM presets carry `FrequencyKHz`; DAB presets carry ensemble index and optional service/component ids for playback.

9.50 StationList

Pure-domain ordered collection (max 20 entries) with add/remove/move and duplicate tune-target rejection. Persisted as JSON through `ISecureStore`.

9.51 StationService

Application service loading/saving presets from NVS and recalling them via `TunerService`. Exposed on `/api/stations/*` and the Presets web UI section.

9.52 IntegrationService

End-to-end orchestration at boot and on preset recall: loads the station list, reapplies the saved `AudioProfile` after tune, and persists the last preset index (NVS key `last_preset`). Used by `app_main` and `POST /api/stations/tune`.

9.53 BluetoothService

Application service for BT1035 pairing and A2DP status (Chapter 7). Delegates to `Bt1035Driver`; tracks whether discoverable mode was requested. Exposed on `/api/bluetooth/*`.

9.54 OtaService

Application service wrapping `esp_ota_ops`: streams firmware into the inactive OTA slot, validates the app descriptor via `core::validateOtaAppDescriptor()`, and exposes `confirmBoot()` for rollback cancellation after a healthy network boot.

CHAPTER 10

Building and Flashing

The firmware is built with ESP-IDF; the hardware-free domain core also builds and is unit-tested on the host machine.

10.1 Toolchain

Item	Choice
Framework	ESP-IDF v5.5.x (native)
Language	C++23 (`-std=gnu++23`)
Error model	`std::expected`; exceptions off
Security	NVS + flash encryption (dev mode); see `docs/security-flash-nvs.md`
Documentation	Doxygen (build must pass)

Table 10.1: Firmware toolchain.

10.2 Si4684 firmware blobs (local only)

Proprietary Si4684 images are **not** in git. Before the first device build:

Populate blobs (from Software/)

```
python3 tools/fetch_si4684_firmware.py --dab-only
python3 tools/fetch_si4684_firmware.py --si46xx-dir /path/to/
    si46xx_firmware
python3 tools/check_si4684_blobs.py
```

See `Firmware/Si4684-Firmware/README.md` for procurement options.

10.3 Device build

First flash after enabling encryption (fw 0.8.3+) requires a one-time erase:

Build, flash, monitor

```
idf.py set-target esp32s3
idf.py build
idf.py erase-flash flash monitor
```

Production flash-encryption release mode uses `sdkconfig.defaults.production` as an overlay — irreversible on the chip; see the security doc before use.

10.4 Host unit tests

The pure core is tested on the host, with no board attached. On macOS the tests need a C++23 standard library — use a Homebrew LLVM (>= 18) or GCC 14 rather than the system Apple Clang.

Host tests

```
cmake -S components/core/test -B build-host \
      -DCMAKE_CXX_COMPILER="$(brew --prefix llvm)/bin/clang++
   "
cmake --build build-host
ctest --test-dir build-host --output-on-failure
```

Thirteen test executables cover JSON parsing, audio design, station list, integration service, and broadcast metadata accumulators.

10.5 Documentation

Documentation has enforced checks, run from the `Software/` directory:

1. **Doxygen** — C++ API reference from source doc blocks. An undocumented public class, method, or parameter fails the build.

2. **Manual sync** — every public class under an `include/` tree must have a matching `\label{cls:ClassName}` section in `docs/manual/ch-classes.tex`. Design-level HTTP API documentation lives in Chapter 8.

3. **Si4684 blob policy** — `tools/check_si4684_blobs.py` ensures no proprietary `.bin` is tracked in git.

Docs (from Software/)

```
doxygen Doxyfile
python3 tools/check-manual-sync.py
python3 tools/check_si4684_blobs.py
python3 tools/gzip-www.sh    # after editing components/net/
    www/index.html
```

10.6 Continuous integration

Every push and pull request to `main` runs `.github/workflows/ci.yml` at the repository root (four parallel jobs, all from the `Software/` directory):

1. **Host tests** — `cmake + ctest` on `components/core/test` (C++23, `g++-14` on Ubuntu).

2. **Doxygen** — must exit 0 with an empty `docs/api/doxygen-warnings.log`.

3. **Manual sync** — `tools/check-manual-sync.py`.

4. **Si4684 blobs** — `tools/check_si4684_blobs.py`.

To rebuild the PDF manual (requires a LaTeX installation):

Manual PDF

```
cd docs/manual
latexmk -lualatex manual.tex
```

Keep it green

All checks are part of the definition of done. A firmware change that adds or modifies a public class, a REST endpoint, or its behaviour must update the

> Doxygen doc blocks, `ch-classes.tex` (for classes), and `ch-api.tex` (for HTTP) in the same change.

10.7 Hardware validation

Automated CI does not attach to a board. When the PCB is available, run the checklist in `docs/security-flash-nvs.md` and the HIL items in `docs/TODO.md` (section P4).

CHAPTER 11

Licensing

DigiRadio is dual-licensed, with hardware and firmware under separate but compatible open-source licences.

11.1 Hardware

The hardware — schematics, PCB layout, Gerbers, and bill of materials — is licensed under the **CERN Open Hardware Licence Version 2, Strongly Reciprocal (CERN-OHL-S v2)**. Anyone who modifies and distributes the hardware must release their changes under the same terms.

11.2 Firmware

The firmware is licensed under the **Apache License 2.0**. It grants patent rights explicitly and pairs naturally with the ESP-IDF ecosystem, which is Apache-licensed as well. Every source file carries the Apache header with an SPDX identifier.

11.3 Documentation and project pages

Where a hosting platform does not offer the CERN-OHL-S licence in its selector, the closest share-alike option is used for that page, with the authoritative licence stated as CERN-OHL-S v2. The definitive licensing is always the one declared in the project repository.

11.4 Third-party components

The design integrates third-party silicon (Silicon Labs, Analog Devices, Espressif, Feasycom/Qualcomm) whose datasheets, firmware images, and DSP tools remain subject to their respective vendors' terms. Those terms are unaffected by the DigiRadio licences and must be observed when redistributing vendor-supplied binaries.

Manufacturing partner

Prototype PCB fabrication and assembly are generously supported by **PCBWay**. See the acknowledgements on the back cover of this manual and the project repository for our thanks to their team.

Acknowledgements

This open-source hardware project was made possible with generous support from **PCBWay**, who sponsored prototype PCB fabrication and assembly for DigiRadio. Our sincere thanks and deep gratitude go to the PCBWay team for believing in the project and helping bring the first boards to life.

PCBWay

https://www.pcbway.com