

T41 Teensy 4.1 + Raspberry Pi 5 Integration Procedure

Step-by-step build procedure for adding a Raspberry Pi 5 (2GB) + Raspberry Pi Touch Display 2 (7") to the T41 transceiver, powered from the existing 12V distribution block. See [wiring_diagram.svg](#) for the visual layout and [teensy-pi-uart-protocol.md](#) (in the project) for the protocol/design rationale behind these choices.

Updated 2026-10-01 for Mustang 2 V1.0.25. Sections 1-8 are the original build procedure. **Section 9** covers everything confirmed and added once the hardware was running: the Pi 5's `/dev/ttyAMA0` UART device, the toolbar dropdowns, the web and phone remotes, live RX audio, the desktop icons, and the newer calibration/BIT screens and menu items.

[wiring_diagram.svg](#) has been redrawn to match this revision (Pi 5 + Touch Display 2, 5A converter, DSI + GPIO power cable, no SPI/I2C).

1. Bill of Materials

Part	Mouser P/N	Mfr P/N	Notes
USB-C PD power module (recommended for this build)	Not a Mouser item	Geekworm RPI5-5V5A-PD, terminal-block variant	From geekworm.com (\$15.50) or an authorized reseller. 9-24VDC in via a bare-wire 3.81mm terminal block (covers 12V with margin), genuine USB-C PD negotiation out (onboard PD protocol chip + CC signaling) at 5V/5A (6A peak). The Pi 5 sees this as a real PD source and runs at full performance with no config.txt workaround and no loss of its onboard overvoltage protection - see Section 2 for why that matters on the Pi 5. This is a different power path, not an add-on to the converters below - it taps 12V directly and does its own conversion, bypassing the GPIO-power wiring entirely
12V->5V DC-DC converter, 5A (alternate, GPIO-powered)	709-SD25A-5	Mean Well SD-25A-5	Verified on Mouser: 9.2-18VDC in (covers 12V with margin), 5V/5A (25W) out, isolated, chassis-mount. Matches the Pi 5's full-performance spec (5V/5A/25W), but powers it via GPIO rather than USB-C PD - see Section 2 for the tradeoffs vs. the PD module above
12V->5V DC-DC converter, 3A (alternate, power-limited GPIO mode)	709-SD15A-05	Mean Well SD-15A-05	Same SD-A series/footprint as the 5A unit. Only use this if you deliberately want the Pi 5 running in its reduced-peripheral-power mode; see the power note in Section 2. Stock one converter, not both Mean Well units
Inline ATO fuse holder	576-FHAC0003ZXJA	Littelfuse FHAC0003ZXJA	Wire-lead inline holder for standard ATO blade fuses. Needed regardless of which power path above you pick - each one still taps the 12V distribution block through its own fused branch
ATO blade fuse, 5A (pair with the PD module)	576-0ATO005.V	Littelfuse 0ATO005.V	The PD module's datasheet calls for >30W available at its 12V input (~2.5A+); sized above that with margin, below the wire gauge's rated ampacity. Also pairs with the 3A Mean Well converter alternate if you go that route instead
ATO blade fuse, 7.5A (pair with the 5A Mean Well converter only)	576-0ATO07.5V	Littelfuse 0ATO07.5V	The 25W converter can draw up to ~2.5A from the 12V rail at full output; sized above that, below the wire gauge's rated ampacity
Bulk capacitor, 1000uF 16V, low-ESR, radial	search Mouser for Nichicon UPW1C102MPD	Nichicon UPW1C102MPD	Confirmed as a genuine low-ESR 1000uF/16V radial part via Nichicon/DigiKey/TME listings; I could not confirm today's exact Mouser stock URL/SKU in this session - search this manufacturer part number directly on mouser.com before ordering. Only relevant to the Mean Well/GPIO-power path - the PD module's own output filtering makes this unnecessary if you go that route
Raspberry Pi 5 (2GB)	-	-	Not a Mouser item - source from a Pi distributor/retailer
Raspberry Pi Touch Display 2, 7"	-	-	From raspberrypi.com or an authorized reseller (Adafruit, PiShop, The Pi Hut, Vilros, etc.), not Mouser. Confirm the kit includes the 22-way-to-15-way FFC cable needed for the Pi 5's newer DSI connector - Raspberry Pi's own documentation says this is included in the box for the 5"/7" models, but double-check the listing before ordering
USB mouse	-	-	Not a Mouser product. Any standard wired USB HID mouse works; see Section 5 - unlike the earlier RA8875 design, the mouse is now useful for normal operation, not just debugging
Hookup wire, 16-18AWG stranded	-	-	For the 5V/GND run from the converter to the shared Pi/display power point; sized for the 5A converter's full output over a short run (see Section 2). Use 16AWG if the run exceeds ~12in to keep voltage drop low at 5A

Dupont/jumper wires (M-F, M-M)	-	-	Only needed now for the UART link from the Pi header to the Teensy (Section 4) - the display no longer needs SPI/I2C jumpering, it connects via its own DSI ribbon and GPIO power cable instead
--------------------------------	---	---	---

Confirmed-on-Mouser parts are marked with an exact Mouser P/N; the capacitor is given as a manufacturer part number to search for, since I could not verify a live Mouser SKU for it in this session - don't take that one as confirmed the way the converter and fuse are.

Three power paths are listed as alternates, not additions - pick one:

- **USB-C PD module (recommended):** genuine PD negotiation, full Pi 5 performance, no config.txt workaround, onboard overvoltage protection intact. Different wiring than the other two - see Section 2.
- **5A Mean Well converter, GPIO-powered:** full Pi 5 performance, but needs the `usb_max_current_enable=1` config.txt override and loses the Pi's onboard overvoltage protection, since GPIO power has no PD negotiation.
- **3A Mean Well converter, GPIO-powered:** simplest wiring, but the Pi 5 runs in its reduced-peripheral-power mode (600mA USB limit).

2. Power: 12V -> 5V conversion

One thing is genuinely different from the Pi 3B here, and it's worth reading before wiring. The Pi 3B could be powered from GPIO 5V/GND with no real downside. The Pi 5 is designed around USB-C power delivery negotiation: when it can't negotiate with a smart supply (which includes being GPIO-fed), it defaults to a conservative 600mA USB-peripheral limit, and GPIO powering also bypasses the overvoltage protection that sits in the USB-C path. This build still uses GPIO powering, to stay consistent with the existing 12V-distribution-block design and avoid adding a second power path - but it means being deliberate about wire gauge and one config.txt setting, rather than treating it as a drop-in repeat of the Pi 3B wiring. If you'd rather have the Pi 5's full official protections, the alternative is a 12V-input USB-C PD trigger/decoy board feeding the Pi's USB-C port instead of GPIO - not covered here, but worth knowing it exists.

1. Identify a spare 12V tap point on the existing distribution block (per the T41 Main Board schematic's "Power section" / heatsink area, or any other 12V node on the block - this is a new independent branch, not drawn from the Teensy board's own 5V regulator).
2. Wire: **12V tap -> fuse holder -> 7.5A fuse -> Mean Well SD-25A-5 pin 1 (+Vin) -> converter pin 2 (-Vin/GND)** back to the distribution block's ground. (If you're using the 3A alternate instead, substitute the SD-15A-05 and the 5A fuse.)
3. Converter output: **pin 3 (+Vout, 5V) and pin 4 (-Vout/GND).**
4. Solder/crimp the 1000uF low-ESR capacitor across +5V/GND at the junction described in step 5 below (not at the converter) - this is what damps the current-transient sag the Pi is sensitive to.
5. **The Pi and the display share one power feed, not two.** The Touch Display 2 ships with its own 3-pin GPIO power cable whose red lead is +5V (goes to Pi physical pin 2) and black lead is GND (goes to Pi physical pin 6) - per Raspberry Pi's documentation, connecting that cable is how the display gets power. Physical pins 2 and 6 sit on the same 5V/GND rail that powers the whole board, so splice the converter's +5V/GND output (with the bulk capacitor across it) into that cable's red and black leads, right where they'd otherwise meet the Pi header, rather than running a second, separate feed directly to the Pi. This single connection powers the Pi and the display together. (The display cable has a third conductor - check the display's quick-start guide for what it does, likely a display-detect/ID line rather than power; leave it connected as designed, just don't treat it as a second power lead.)
6. **Do not** connect this +5V rail to the Teensy main board's own 5V regulator output anywhere - common ground only, per the design doc.
7. Before connecting the Pi: power the converter alone and verify 5.0V (+/-0.25V) at its output with a multimeter, with no load and then with a ~2A test load if you have one, before trusting it with the Pi + display.
8. If you're using the 5A converter and want the Pi to actually use that available current instead of self-limiting to 600mA on its USB ports, see the `usb_max_current_enable=1` config.txt setting in Section 6 - without it, the Pi assumes the conservative GPIO-power default regardless of what the converter can deliver.

3. Display wiring

The Touch Display 2 replaces the RA8875/FT5316 SPI+I2C wiring from the original Pi 3B design entirely. There's no SPI, I2C, or logic-level jumper to deal with - it's a standard DSI panel that Raspberry Pi OS auto-detects.

1. **Connect the DSI ribbon cable.** Pi 5 side: the 22-pin end into either of the Pi 5's two DISP/CSI-DSI connectors (either port works). Display side: the 15-pin end into the Touch Display 2's connector. This is the 22-way-to-15-way adapter cable called out in the BOM - confirm you have the right one, since it differs from the plain 15-to-15 cable used on Pi 4 and earlier.
2. **Connect the GPIO power cable** as described in Section 2, step 5 (red to physical pin 2, black to physical pin 6, spliced to the converter output rather than run separately to the Pi).

3. **Before first boot**, make sure your Raspberry Pi OS image and EEPROM bootloader are current (Section 6) - Raspberry Pi's documentation notes the display needs both to be auto-detected correctly.

Software side of this is done, not just flagged. `radio_ui.py` has been ported to a windowed Tkinter UI (`display.py` replaces `ra8875.py`; `ft5316_touch.py` is gone entirely, since touch now arrives as ordinary button clicks through the desktop). See `README.md` in the PiDisplay folder for the current file list. It hasn't been run against real Pi 5 + Touch Display 2 hardware in this session, so budget time in Section 6's manual test step for working out any rough edges - the waterfall's scrolling technique in particular is a standard Tkinter approach but unverified here.

4. UART link to the Teensy

Unchanged from the Pi 3B design - the Pi 5 exposes the same GPIO14/15 UART pins at the same physical header positions, configured the same way through `raspi-config`, even though the Pi 5 routes I/O through its RP1 chip internally.

1. Wire **Pi physical pin 8 (GPIO14, UART TXD) -> Teensy pin 0 (RX1)**.
2. Wire **Pi physical pin 10 (GPIO15, UART RXD) -> Teensy pin 1 (TX1)**.

(Same physical pins and BCM numbering as every other 40-pin Raspberry Pi board - the GPIO header layout is identical across the Pi 5, 3B, and Zero boards that were considered earlier in this project, so nothing about the wiring plan itself changed when the board choice did.)
3. Tie Pi GND to Teensy GND (in addition to the shared ground from the power section above - a dedicated short ground wire between the two boards keeps the UART signal reference solid).
4. **Confirmed against the T41 Main Board schematic: pins 0/1 are free - conditional on one power-circuit choice, documented here so it doesn't get re-litigated later.** The schematic's Power section offers two mutually exclusive options ("Choose between 'Shutdown routine activation' and 'Voltage reversal prevention' sections... Populate only one"). This build uses **Voltage reversal prevention** (the MOSFET-based option, no MCU involved) - under that choice, pins 0/1 genuinely aren't used anywhere else on the board, so PiLink.h's assumption (`#define PI_LINK_SERIAL Serial1`) is correct as-is, no change needed.

If a build instead populates the other option - **Shutdown routine activation**, which adds an ATtiny85 (U4) coordinating a clean Teensy power-down - that circuit's `BEGIN_TEENSY_SHUTDOWN/TEENSY_SHUTDOWN_COMPLETE` handshake is hardwired to these same two pins (0/1), and would conflict with PiLink. Checked at the time: **every other Teensy 4.1 hardware UART pin-pair is also already claimed elsewhere on this board** (Serial2: encoder; Serial3/4: front panel INT/I2C; Serial5: RF Control clocks; Serial6: a second I2C bus; Serial7: band relay selects; Serial8: CW/Xmit Mode lines) - so there's no clean alternate hardware UART to fall back on if that option is ever chosen instead. That would need its own resolution at the time, not a change made preemptively here.

For reference, pins 11 (MOSI) / 12 (MISO) / 13 (SCLK) are a separate, unrelated SPI bus already in use for the Teensy's own onboard front-panel display - that's a completely different physical link from this UART and nothing here touches it.

5. On the Pi, disable the Linux serial console on the UART (it defaults to a login shell on `/dev/serial0`, which will collide with PiLink - note that on the Pi 5 the app itself must open `/dev/ttyAMA0`, not `/dev/serial0`; see Section 9.1): `sudo raspi-config` -> Interface Options -> Serial Port -> "login shell over serial?" **No**, "serial port hardware enabled?" **Yes**. Reboot.
-

5. Mouse and touch - what changed from the Pi 3B design

The mouse is now part of normal operation, not just a debugging tool. With the RA8875 panel, `radio_ui.py` drew directly to the display with no window manager, so there was no cursor for a mouse to move and touch went straight to the Teensy as virtual button presses - a mouse only mattered for setting up the Pi itself over HDMI. The Touch Display 2 changes that: it's a normal framebuffer + 5-point capacitive touch input device, so now that `radio_ui.py` runs as a windowed Tkinter app (Section 3), a standard desktop runs on it directly, and mouse and touch both work the ordinary way - clicking dropdowns, dragging sliders, scrolling - with no custom input handling needed on your end. Plug a USB mouse into any of the Pi 5's USB-A ports and it works immediately, same as any desktop Linux mouse.

WSJT-X is no longer blocked by the display choice. The earlier procedure flagged that the RA8875, as a register-level SPI panel with no mainline framebuffer driver, couldn't host a Qt application like WSJT-X. The Touch Display 2 doesn't have that limitation - it's a standard framebuffer surface, so WSJT-X can run on it like any other Raspberry Pi desktop app. One practical caveat worth knowing before that phase: WSJT-X's default window layout is wider than it is tall, while this display is 720x1280 in portrait orientation - you may want to rotate the display to landscape (Section 6, step 7) or plan WSJT-X's window placement around the portrait aspect ratio when you get there.

6. Raspberry Pi OS setup and software upload

1. **Flash the OS.** On another computer, use Raspberry Pi Imager (raspberrypi.com/software) to write "Raspberry Pi OS (64-bit) with desktop" to a microSD card - not the Lite image, since this build now runs a windowed desktop on the Touch Display 2. In the Imager's advanced options (gear icon / Ctrl+Shift+X) before writing: set hostname, enable SSH, set a username/password, and pre-configure WiFi if you'll use it - this gets you a headless-capable Pi for initial setup even though it'll run a desktop once booted.
2. **First boot.** Insert the card, power the Pi, and SSH in: `ssh <username>@<hostname>.local`
3. **Update the EEPROM bootloader**, required for the Touch Display 2 to be auto-detected reliably:

```
sudo apt update && sudo apt full-upgrade -y
sudo rpi-eeeprom-update -a
sudo reboot
```

After reboot, confirm with `sudo rpi-eeeprom-update` that it reports up to date.

4. **Enable the UART** for the Teensy link (the display needs no SPI/I2C enablement - it's auto-detected via DSI):

```
sudo raspi-config
```

- Interface Options -> Serial Port -> login shell: No, hardware: Yes
- Reboot when prompted.

5. **If using the 5A converter/GPIO-power path from Section 2**, edit `/boot/firmware/config.txt` (this path applies to current Raspberry Pi OS releases based on Debian Bookworm; older images used `/boot/config.txt`
 - check which your image uses) and add:

```
usb_max_current_enable=1
```

This tells the Pi to stop self-limiting USB peripheral current to 600mA, which it otherwise assumes by default any time it's GPIO-powered rather than negotiating with a USB-C PD supply. Only set this if you've verified the converter and wiring can actually deliver the current (Section 2, step 7) - it removes a safety margin the Pi would otherwise apply on its own.

6. **Install dependencies**, including `python3-tk` - Tkinter isn't a pip package and the windowed UI needs it:

```
sudo apt install -y python3-pip python3-tk git
```

7. **(Optional) Rotate the display to landscape.** The Touch Display 2 is natively 720x1280 (portrait); `config.py`'s default `DISPLAY_WIDTH/ DISPLAY_HEIGHT` (1280x720) assume you've rotated it to landscape to match the wide status-bar-plus-waterfall layout.

raspi-config no longer has this option. Current Raspberry Pi OS (Bookworm and Trixie) runs the desktop on Wayland (the `labwc` or `wayfire` compositor) instead of X11, and display rotation moved out of `raspi-config` into the desktop's own screen-configuration tool. Use whichever of these matches your setup:

- **Desktop (recommended, since this build runs a windowed UI anyway):** From the taskbar, open **Preferences -> Screen Configuration** (labeled **Screens** in the Control Centre on newer Trixie-based images - same tool, newer menu wording). Right-click the display rectangle (labeled `DSI-1`), choose **Orientation**, pick **Left** or **Right** (either gives you landscape - pick whichever matches how the panel is physically mounted in your chassis), then **Apply**. This also rotates touch input to match, and the setting persists across reboots.
- **Headless/console fallback**, if you're rotating before a desktop session is available: edit `/boot/firmware/cmdline.txt` and append (space-separated, same line, no newline):

```
video=DSI-1:720x1280@60,rotate=90
```

Use `90` or `270` for landscape, depending on mounting orientation. This only rotates the console/framebuffer output, not touch input alignment - if touch ends up mismatched after this, the touch overlay in `/boot/firmware/config.txt` supports `swapxy`, `invx`, `invy` parameters (e.g. `dtoverlay=vc4-kms-dsi-ili9881-7inch,invx,invy`) to correct it. Prefer the desktop method above when you have the option - it handles both video and touch rotation together.

Or skip rotation entirely and instead set `config.py` to 720x1280 to keep the UI in portrait - either works, just keep the two in sync.

8. **Copy the PiDisplay code onto the Pi.** From your Mac (with this PiDisplay folder), either:

- `scp -r PiDisplay <username>@<hostname>.local:/home/<username>/PiDisplay`, or
- put it in the same git repo as PhoenixSketch and `git clone/git pull` on the Pi.

9. **Install Python packages on the Pi.** Current Raspberry Pi OS (Bookworm and Trixie) blocks a plain `pip3 install` with an `externally-managed-environment` error (PEP 668) - it wants packages installed via `apt` or inside a virtual environment instead. The only dependency here is `pyserial`, and Debian already packages it, so install it that way rather than fighting the protection with `--break-system-packages`:

```
sudo apt install -y python3-serial
```

This satisfies everything in `requirements.txt` and needs no virtual environment - the `pilink-display.service` unit (Section 11) calls the system `/usr/bin/python3` directly, so there's nothing else to point at a `venv`.

10. **Adjust `config.py`** if you skipped the rotation step above, and adjust `radio_ui.py`'s `BUTTON_ID_BAND_UP/BAND_DOWN` placeholders to the real `BUTTON_*` values from `SDT.h` once you know which controls you're wiring up.
11. **Test manually first**, before installing the service, so you can see errors directly - this needs to run inside the desktop session (not over a plain SSH shell with no display) so Tkinter has somewhere to open its window:

```
python3 radio_ui.py
```

With the Teensy powered and PiLink initialized, you should see a fullscreen window open with the status bar and waterfall, and start showing frequency/S-meter/spectrum within a couple seconds (once `PiLinkIsConnected()` sees the Pi's heartbeats). Press Escape to drop out of fullscreen if you need to get back to the desktop.

12. **Install as a user `systemd` service** so it starts automatically at boot. This is a real change from the Pi 3B version, not just a path update: the old RA8875 app drew directly over SPI with no window system, so a plain system service worked. This app opens a Tkinter window on the desktop's X11 session, so it has to run as a **user** service tied to that graphical session - see the full explanation and commands in the header comment of `pilink-display.service` itself. Briefly:

```
mkdir -p ~/.config/systemd/user
cp pilink-display.service ~/.config/systemd/user/
systemctl --user daemon-reload
systemctl --user enable --now pilink-display.service
loginctl enable-linger $(whoami)
journalctl --user -u pilink-display -f      # watch logs
```

Also enable Desktop Autologin (`raspi-config` -> System Options -> Boot / Auto Login) so the X11 session this service expects is actually running at boot. Edit the `WorkingDirectory=` line in the service file first if your install path differs from `/home/pi/PiDisplay`.

13. **Enable the read-only overlay filesystem - do this LAST, once everything above is installed, tested, and working.** The T41's power is switched on/off externally and abruptly (see the "Power sequencing" note in Section 2) - there's no lead time for the Pi to shut down cleanly, so instead this makes an abrupt cutoff harmless by making sure nothing is being written to the SD card during normal operation:

```
sudo raspi-config
```

- Performance Options -> Overlay File System -> enable ("Would you like the overlay file system to be read-only?" - **Yes**)
- Reboot when prompted.

Once enabled, the root filesystem (including `~/PiDisplay`) is read-only; any writes during normal operation go to a RAM overlay and are discarded on reboot - there is nothing left for an abrupt power loss to corrupt. **This also means further code/config changes need the overlay disabled first** (same `raspi-config` menu, same option, set back to read-write, reboot, make your changes, re-enable, reboot again)

- so do this only after `radio_ui.py` is confirmed working and you're done iterating for the session. Note this is the simple, built-in overlay: it needs a reboot to toggle write access on and off, it does not support a live "unlock, write, relock" without rebooting - see Section 8 for why that matters if this project later wants the Pi to save its own settings on demand.

7. Verification checklist

- ☐ Converter outputs 5.00-5.25V unloaded, stays within that range under the Pi + display's actual current draw
- ☐ EEPROM bootloader reports up to date (`sudo rpi-eeeprom-update`)
- ☐ Pi boots to the desktop directly on the Touch Display 2, correct orientation, no auto-detection errors in `dmesg`
- ☐ Pi is also reachable over SSH

- ☐ Touching the screen registers normally in the desktop (drag an icon, open a menu) - confirms the touch layer end-to-end before layering the radio UI on top
- ☐ USB mouse works normally in the desktop, including opening dropdown menus - this is the capability this revision was built for
- ☐ `raspi-config` shows serial hardware enabled, login shell over serial disabled
- ☐ If using the 5A converter: `usb_max_current_enable=1` is set in `config.txt` and USB peripherals aren't unexpectedly power-limited
- ☐ `python3 radio_ui.py` opens a fullscreen Tkinter window with no Python exceptions
- ☐ BAND +/- buttons in the status bar respond to both mouse click and touch
- ☐ Waterfall scrolls cleanly once spectrum data is flowing (this is the one piece of `display.py` flagged as unverified against real hardware - watch for tearing/artifacts here first)
- ☐ Frequency/S-meter text updates once the Teensy is running with `InitializePiLink()` wired in (from the earlier Teensy-side work)
- ☐ Clicking or touching a BAND +/- button changes something visible on the Teensy's own front-panel/state (confirms the `VIRTUAL_BUTTON` path end-to-end)
- ☐ `systemctl --user status pilink-display` shows active, and the window reappears automatically after `sudo reboot` (requires Desktop Autologin + `loginctl enable-linger` from Section 6)
- ☐ After enabling the overlay filesystem (Section 6, step 13): `pilink-display.service` still starts and runs normally, and `sudo raspi-config` confirms the overlay is active - do this check last, once you're actually done testing, since re-disabling it to make further changes means another two reboots
- ☐ **Added in Section 9:** `config.py` `UART_PORT` is `/dev/ttyAMA0` and telemetry arrives (Section 9.1)
- ☐ Teensy version string reads Mustang 2 V1.0.15 or later; Menu -> Display -> Spectrum floor from the Pi lands on its adjustment box, not Home (Section 9.2)
- ☐ Web remote loads at `http://<pi-hostname>:8765/`, asks for the new passkey, and tuning/dropdowns work (Section 9.3)
- ☐ With the Teensy USB cable connected: web remote plays RX audio and Mute works (Section 9.4)
- ☐ Start Radio UI and Shutdown Pi desktop icons both work (Section 9.5)
- ☐ Phone page loads at `http://<pi-hostname>:8765/mobile` and the four sliders turn their knobs (Section 9.7)
- ☐ Settings pane shows Mustang 2 V1.0.25; Menu > Diagnostics > I2C BIT test and Menu > Calibration > Receive IQ open on the Pi and the remotes; greyed-out items can't be picked (Section 9.8)

8. Future items / roadmap

Status as of V1.0.15 (2026-09-26): the base integration has now run end to end on real hardware (item 9). Items 1-5 are done - dropdown Menu and toolbar, settings changes sent back via virtual button presses and the new `VIRTUAL_ENCODER` message (rather than separate "set parameter" messages), all Home panes wired, band/mode names and real button ids. The Teensy USB audio path (see the FT8 / WSJT-X procedure) now also feeds live audio to the web remote. See Section 9.

Everything in Sections 1-7 gets the two displays structurally mirrored: same pane names/layout, real data wherever PiLink already carries it. The items below are deliberately **not** part of that first pass - either because they were explicitly deferred, or because they're worth planning for even though nothing here depends on them. None of this is required to consider the base integration done.

1. **Touch/mouse-friendly Pi redesign (dropdown menus, etc.) - deferred by design.** The Pi's Home-screen mirror (`display.py`) is a 1:1 structural copy of the Teensy's `DrawHome()` panes. It does **not** yet cover the Teensy's separate Menu system (`MainBoard_DisplayMenus.cpp`) - 9 categories (RF Options, CW Options, Microphone, Audio Options, Display, EEPROM, Calibration, Diagnostics, Sample Rate), each with its own list of settings, reachable only via the front-panel Menu button today. Once the Home mirror has been run on real hardware and trusted, the next phase is a genuine touch UI for this - dropdowns/lists driven by touch or mouse instead of the physical encoder, living in their own Pi-side view rather than squeezed into the Home layout.
2. **Two-way settings sync for the Menu system.** A touch-driven Menu on the Pi (item 1) needs a way to *write* settings back to the Teensy, not just display them - the existing uplink only has `VIRTUAL_BUTTON` (mimics a single front-panel button

press) and HEARTBEAT. Wiring real dropdowns will need new uplink message types (e.g. "set parameter X to value Y") that call into the same IncrementVariable()/DecrementVariable()-style logic the physical Menu already uses, rather than trying to reuse VIRTUAL_BUTTON for things it wasn't designed for.

3. **Wire the remaining unwired Home panes.** `swr`, `audio_spectrum`, and `sam_offset` are still bordered placeholders on the Pi (see `display.py`'s `_UNWIRED_PANES`) because `PiLinkStatusMsg` has no fields for them yet. SWR/forward power in particular already exists on the Teensy (`ReadSWR()`/`ReadForwardPower()`, used by `DrawSWRPane()`) and would just need a new field or message type, same pattern as the power/filter/WPM fields added in this revision.
4. **Populate the band/mode name lookup table.** `display.py`'s `_BAND_MODE_NAMES` is currently empty, so the Pi shows raw numeric ids ("Band 3 / Mode 11") instead of names. `PiLink`'s `STATUS` message only sends the numeric `band/modeState` bytes (matching `bands[]` in the Teensy firmware and `ModeSm_StateId_*` in `ModeSm.h`) - either mirror those tables into `_BAND_MODE_NAMES` by hand, or add a one-time/on-change "definitions" message so the Pi doesn't have to be manually kept in sync with the Teensy's band table.
5. **Real BUTTON_ids.*** `radio_ui.py`'s `BUTTON_ID_BAND_UP` / `BUTTON_ID_BAND_DOWN` are placeholder values, not the real `BUTTON_*` constants from `SDT.h`. Replace these (and add more as the Pi grows more controls) once you've confirmed the actual ids the front panel uses.
6. **Ethernet header (IDC 2x3, native Teensy 4.1 100 Mbps PHY) - a future bandwidth upgrade path, not needed today.** The current UART link uses under 10% of its ~92 KB/s capacity at today's telemetry rates (status 10 Hz, S-meter 20 Hz, 512-byte spectrum ~15 fps), so there's no pressure to move. Ethernet would matter if the project later wants things UART can't comfortably do - full-resolution real-time waterfall instead of a stretched/quantized one, bulk transfers (logs, firmware push), or genuinely networked features (remote CAT control, SSH from the LAN). It needs the [PIRC Ethernet kit](#) magjack module on that header plus the QNEthernet library on the Teensy side, and a plain cable to the Pi 5's own Ethernet port (no switch needed for a direct point-to-point link). This is a bigger change than it sounds - it replaces the whole telemetry transport, not just adds a wire - so treat it as a project of its own, not an incremental step.
7. **USB Host header (IDC 2x3) - not a fit for the Pi link itself.** The Pi 5 can run in USB gadget/device mode over its USB-C port, but that port is already committed as the Pi's main power input in this design (Section 2), and the Teensy's USB Host port can't source anywhere near the amount of current a Pi 5 needs anyway. This header is more useful for something unrelated to the Pi - a USB keyboard, wireless dongle, or flash drive plugged directly into the radio - if that's ever wanted.
8. **Graceful shutdown for the Pi - resolved for this revision.** The T41's power comes from an external source switched on/off abruptly (see the power-sequencing note in Section 2/6) - there's no lead time available anywhere in the design for the Pi to shut down cleanly before power actually disappears, since the switch sits upstream of both boards. Rather than add holdup hardware (a supercap or UPS HAT, sensed via GPIO, to buy the Pi a few seconds' warning - a real option if ever needed, but its own small project), this revision takes the simpler path: **Section 6, step 13 enables the Pi's read-only overlay filesystem**, so nothing is written to the SD card during normal operation and an abrupt cutoff has nothing to corrupt. This is the built-in `raspi-config` overlay, which needs a reboot to toggle write access - fine for a "set it and forget it" field radio, since the Pi has no settings that change routinely today.
 - **Follow-on for when the Pi gets its own settings** (once the touch/dropdown Menu redesign in items 1-2 above lands): the Teensy already has a natural "commit now" trigger in its own Menu - EEPROM -> "Save data to storage" (`SaveDataToStorage()` in `Storage.cpp`) - this is the same action already in active use today to save Teensy settings to its SD card, confirmed as the current workflow. The same action could also tell the Pi to commit its settings, via a new `PILINK_MSG_COMMIT_SETTINGS` downlink message sent from inside `SaveDataToStorage()` (so it fires regardless of what triggered the save). The simple `raspi-config` overlay can't support this, though - it has no live "unlock, write, relock" without a reboot. That would need switching to a manually-configured overlay (root exposed read-only at a separate mountpoint, e.g. `/ro`, supporting `mount -o remount,rw /ro / remount,ro` on demand) and a small `persist_settings()` handler in `radio_ui.py`. Worth building once there's Pi-side state worth saving, not before.
9. **Full hardware verification pass.** Everything in this document has been built and reasoned through against the real Teensy firmware source and Raspberry Pi's own documentation, but has not been run end-to-end on physical Pi 5 + Touch Display 2 + Teensy hardware in this session - see Section 7's checklist for what to confirm first once you have hardware in hand, in particular the waterfall's scroll behavior.

9. Updates confirmed on real hardware (V1.0.3 - V1.0.25)

Sections 1-8 were written before the first power-up. Everything below was added once the Pi 5, Touch Display 2 and Teensy were running together on the bench, and reflects what's actually flashed and running as of **Mustang 2 V1.0.25 (2026-10-01)**. See `../CHANGELOG.md` for the full per-version history.

9.1 Pi 5 UART device: use `/dev/ttyAMA0`, not `/dev/serial0`

Found on the first power-up: on every earlier Pi, `/dev/serial0` is an alias for the GPIO-header UART (pins 8/10). **On the Pi 5 it isn't** - the RP1 chip routes `/dev/serial0` to the separate 3-pin debug-connector UART instead. That's a real, valid device, so opening it succeeds with no error; the app just receives nothing, which looks exactly like a wiring fault.

- `config.py` already has `UART_PORT = "/dev/ttyAMA0"` - leave it that way.
- The `raspi-config` serial step in Section 4, step 5 / Section 6, step 4 is still needed (hardware enabled, login shell off); only the device name the app opens is different.

- Quick check if telemetry never arrives: `ls -l /dev/ttyAMA0 /dev/serial0`, and confirm `config.py` points at `ttyAMA0`.

Where the UART wires land on this build: on the T41 main board, the PiLink wires plug into the empty ATtiny85 socket in the unused "Shutdown activation" section (only valid because this board uses the Voltage reversal prevention option - see Section 4, step 4); on the Pi 5, GPIO header pins 8 (TXD), 10 (RXD) and a ground pin.

9.2 Toolbar dropdowns and mouse wheel (V1.0.8 - V1.0.15)

The BAND +/- buttons in Sections 6-7 were replaced by a two-row toolbar of dropdowns, each built from the real firmware tables: **Menu** (cascading flyouts), **Vol** (Vol/AGC/Mic/STn), **Band**, **Mode**, **Demod**, **Course Inc**, **Fine Inc**, **Zoom**, **BW** (Hi Cut/Lo Cut), **Noise**, **Decode** and **Home**. The mouse wheel adjusts a Menu setting while its red "Label: value" box is showing (tap "Select >" to confirm), or the knob last picked from Vol/BW; the "Wheel:" indicator shows which, and clicking it hands the wheel back to tuning.

New uplink message: `PILINK_MSG_VIRTUAL_ENCODER` (knob + direction) for the Filter, Volume, Tune and Fine-tune encoders.

V1.0.15 adds the Menu's real position (menuPrimaryIndex/menuSecondaryIndex) to every STATUS frame - this is what makes remote Menu selection reliable; before it, the Pi had to guess the Menu position and could land back on Home. Reflash the Teensy and confirm the version string reads `Mustang 2 V1.0.15`.

9.3 Web remote on the home network (web_relay.py, V1.0.14)

`radio_ui.py` starts a small WebSocket server (`web_relay.py`) that mirrors the Pi's screen to any browser on the same network and forwards clicks to the same code a touch on the Pi uses.

1. Install the dependency listed in `requirements.txt` (`websockets` 12.0 or newer). If `apt policy python3-websockets` shows 12 or newer, use `sudo apt install -y python3-websockets`; otherwise `pip install --break-system-packages websockets`.
2. In `config.py`, **change** `WEB_PASSKEY` from its default. Leave `WEB_RELAY_PORT = 8765` unless it clashes with something else.
3. Restart `radio_ui.py`, then open `http://<pi-hostname>:8765/` from any browser on the network and enter the passkey. Phone browsers work too (dropdowns and Menu).
4. Tuning: mouse wheel over the frequency/spectrum area (Shift = fine), or a USB rotary encoder set up to send ArrowUp/ArrowDown (or PageUp/PageDown) key presses - Shift for fine.

LAN only. It's plain `ws://` with no encryption. Do **not** forward port 8765 through your router - anyone who can reach it can key the transmitter. Remote access from outside the home needs TLS and a stronger passkey, or a VPN.

9.4 Live RX audio to the web remote (audio_relay.py, V1.0.14)

Streams the radio's receive audio to every logged-in browser, with a per-browser **Mute** button. It uses the Teensy's own USB audio - the same path as the FT8 / WSJT-X procedure (`FT8_WSJTX_INTEGRATION.md`) - not the PiLink UART.

1. Build the sketch with Arduino IDE **Tools > USB Type > Serial + MIDI + Audio** and flash it.
2. Connect the Teensy's own USB port to a Pi 5 USB-A port. **Before you do, check whether the Teensy's VUSB-VIN link is cut** - the T41 powers the Teensy from its own 5 V, and an uncut link lets the Pi back-feed 5 V into the main board.
3. On the Pi: `sudo apt install -y libportaudio2`, then `pip install --break-system-packages sounddevice`.
4. Check what the Teensy enumerates as: `python3 -c "import sounddevice as sd; print(sd.query_devices())"`, and set `config.AUDIO_DEVICE_NAME_HINT` to match (default "Teensy"). Other settings: `AUDIO_ENABLED`, `AUDIO_SAMPLE_RATE` (44100), `AUDIO_CHANNELS` (1), `AUDIO_CHUNK_MS` (40).
5. Restart `radio_ui.py` and reload the web page.
6. To check the cable on its own, run `python3 test_audio_cable.py` on the Pi: it lists the audio devices, records a short sample from the Teensy, reports its level and saves a `.wav` you can listen to.

If the cable, `libportaudio2` or `sounddevice` is missing, audio just turns itself off with one log line; the rest of the app runs normally. Mute only affects that browser tab. If you still need the Teensy's USB port for reflashing from a PC, use a manual USB sharing switch, not a Y-cable.

9.5 Desktop icons: Start Radio UI / Shutdown Pi (V1.0.14)

Two launchers in `desktop/` - see `desktop/README.md` for the full steps:

1. `chmod +x ~/PiDisplay/desktop/*.sh`, copy `desktop/*.desktop` to `~/Desktop/`, `chmod +x` them, then right-click each icon once and choose **Trust this executable**.
2. For Shutdown Pi, add a sudoers rule limited to the shutdown command only: `sudo visudo -f /etc/sudoers.d/010_bob-shutdown` with the line `bob ALL=(ALL) NOPASSWD: /sbin/shutdown` (use your own username).

Start Radio UI refuses to start a second copy (the UART can only be opened once). Shutdown Pi asks for confirmation first.

9.6 Firmware source vs. what's flashed - resolved

Earlier (V1.0.14) the flashed firmware sent a 53-byte SETTINGS message while the Mac's copy of PiLink.h/PiLink.cpp only described 51 bytes. **Checked 2026-10-01 against Mustang 2 V1.0.25: resolved.** Every PiLink message struct in PiLink.h (STATUS 44 bytes, SETTINGS 58, PARAMETER 37, RX IQ cal 3, BIT results 7) matches pilink_client.py's unpack formats field for field, the message and encoder ids match, and SendSettings() fills in audioVolume and secondScreenActive. A build from the current source is safe to flash and publish. If the Settings pane ever stops updating after a reflash, pilink_client.py logs the SETTINGS frame size it received vs. expected - rebuild and re-sync both sides together.

9.7 Phone page: Mustang Mobile (/mobile)

The web remote has a separate phone page at `http://<pi-hostname>:8765/mobile` (the laptop page stays at /). Same passkey and the same controls, laid out for a phone: a four-column grid of toolbar buttons that open the phone's own pickers, and four touch sliders at the bottom - **Tune**, **Fine Tune**, **Volume** and **Filter** - in place of the mouse wheel. Drag a slider away from center to turn that knob (further = faster); it springs back when released. Its own version label (for example "Mustang Mobile V1.0.33") is shown on the page. Nothing extra to install: web_relay.py serves both pages.

9.8 New screens and menu items (V1.0.16 - V1.0.25)

Flash the Teensy and copy PiDisplay/ to the Pi together for every version from here on - several of these add fields to PiLink messages, and if the two sides don't match, pilink_client.py logs a frame-size mismatch and the Pi's Settings pane/readout strip stop updating.

- **Receive IQ calibration (V1.0.23-V1.0.24):** Menu > Calibration > Receive IQ opens on the Pi and both web remotes with full control (amplitude/phase adjust, step size, auto-tune, band cycling, Save & Exit). The Pi screen also has **Stop** (ends an Auto sweep early - a full sweep can take a couple of minutes) and **Exit** (leave without saving).
- **I2C BIT test (V1.0.25):** Menu > Diagnostics > I2C BIT test shows PASS/FAIL for the front panel, BPF, RF, RF SI5351 and K9HZ LPF boards on the Pi and remotes, with a Home button. The Pi hides its toolbar while this screen or the Receive IQ screen is open.
- **Greyed-out menu items (V1.0.25):** items with no Pi/web screen yet are shown but not selectable from the Pi and remotes: Calibration > S meter level, Frequency, Transmit IQ and Power; Diagnostics > Restore receive EQ and Buffer print; EEPROM > Print data to Serial. They still work from the front panel. (_MENU_DISABLED_ITEMS in display.py - remove an entry once its screen is ported.)
- **RX DSP Gain (V1.0.17):** Menu > RF Options > RX DSP Gain, -5.0 to +20.0 dB in 0.5 dB steps.
- **Spectrum fill on / off (V1.0.21):** Menu > Display. The filled trace could stall the audio on a strong or wide signal; turn the fill off if audio ever gets choppy.
- **Pi-only Display items: Brightness** (25/50/75/100% backlight for the Touch Display 2) and **PI Spectrum Fill** (On/Off for the Pi's screen and both remotes, separate from the T41's own setting). If Brightness does nothing and the terminal shows a permission error, add the user that runs radio_ui.py to the video group (sudo usermod -aG video <user>) and log out and back in.
- **Readout strip:** Vol, AGC, Scale, Floor, Zoom, TI, FT and Noise now sit in a strip above the spectrum on the Pi and remotes; the Pi also shows its LAN address next to the clock, so the web-remote address is easy to find.