

T41 Teensy 4.1 <-> Raspberry Pi 5 -- UART Protocol Design

T41 Teensy 4.1 ↔ Raspberry Pi 5 — UART Protocol Design

Context

Hybrid architecture: Teensy 4.1 keeps all real-time audio/DSP/RF control exactly as-is. A Raspberry Pi 5 (2GB) is added to drive the display/UI, with a future goal of running WSJT-X/FT8 and other digital modes. Link between the two boards is a simple UART.

Board history: originally scoped for a Raspberry Pi 5 (full replacement) → revised to hybrid Teensy+Pi 5 → Pi Zero 2 W (unavailable) → Pi Zero W (works for display+touch only, but no CPU/RAM headroom for digital modes) → Pi 3 Model B (chosen once digital-mode software was named as a future goal, for its CPU/RAM headroom) → **Pi 5 (current, final choice)** — the CPU/RAM/USB headroom the digital-modes goal needs, plus native DSI support for the official Raspberry Pi Touch Display 2.

Display history: originally a 7" buydisplay.com ER-TFTM070-5V4 SPI/I2C panel (RA8875 controller + FT5316 capacitive touch, the same RA8875 chip already used on the Teensy's own main-board display) → **replaced by the official Raspberry Pi Touch Display 2, 7" (current, final choice)** once the board choice moved to the Pi 5. The Touch Display 2 is a standard DSI panel that Raspberry Pi OS auto-detects and drives as a normal framebuffer + 5-point capacitive touch input device — no SPI/I2C wiring, no register-level display driver, and (unlike the RA8875) a mainline framebuffer means Qt apps like WSJT-X can run on it directly.

Design principle

The Pi is an enhancement, not a dependency. If it's unplugged or crashes, the radio must keep working off its existing physical front panel (buttons/encoder/keys/PTT), which stay wired directly to the Teensy exactly as today — no protocol messages needed for those.

Touch/mouse taps on the Pi are treated as a second input source into the same button-event queue the physical panel already uses (`FrontPanel.h: GetButton() / SetButton()` , `BUTTON_*` ID space). With the Touch Display 2, touch and mouse both arrive as ordinary desktop UI events (Tkinter button clicks) rather than raw touch-controller coordinates — the Pi-side app does its own hit-testing and only sends semantic button IDs across.

Downlink: Teensy → Pi

Message	Rate	Payload	Notes
STATUS	~10Hz + on change	freq (uint32 Hz), band id (uint8), ModeSm state id (uint8), PTT (bool), power level, filter bandwidth (uint16), keyer WPM, fault flags	Fixed-size struct (<code>PiLinkStatusMsg</code>)
SMETER	~10-20Hz	int16, 0.1dB units	Reuses <code>GetSMeterDBM()</code> (new getter added to <code>MainBoard_DisplayHome.cpp</code> , wraps <code>AudioToDBM(audioMaxSquaredAve)</code>)
SPECTRUM	~10-15fps	512 bytes: psdnew[512] quantized float32 dBm → uint8	Don't send raw float32 (2KB/frame); uint8 is enough resolution for waterfall/panadapter

Bandwidth: $512B \times 15fps \approx 7.7KB/s$ against a 921600 baud (~92KB/s) link — under 10% of capacity at today's rates, so there's headroom for more fields before this link is the bottleneck. See "Future items" below for when that would change.

Uplink: Pi → Teensy

Message	Payload	Notes
VIRTUAL_BUTTON	one BUTTON_* id	Maps directly onto existing SetButton() plumbing
HEARTBEAT	none	Lets Teensy detect the Pi went away; reaction is just “stop expecting display commands”

Framing

```
[0x7E][0xA5][MSG_TYPE][LEN_LO][LEN_HI][...payload...][CRC-8]
```

Resync on CRC failure: drop frame, scan forward for next 0x7E 0xA5. Non-blocking byte-at-a-time parser on the Teensy side to avoid blowing the 10ms main-loop budget. Incremental CRC-8 (poly 0x07) is used on both TX and RX so no payload-size buffer is needed for the 512-byte spectrum frame.

Power (for the Pi + display branch)

Tap the existing 12V distribution rail with its own branch — do not tie into the Teensy board’s existing 5V regulator output; common ground only. The Pi and the Touch Display 2 share **one** power feed (the display’s own 3-pin GPIO power cable, spliced to the converter’s output, going to Pi physical pins 2/6) rather than two separate feeds.

One thing is genuinely different from the Pi 3B here. The Pi 3B could be GPIO-powered 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 Pi 5’s onboard overvoltage protection that normally sits in the USB-C path.

Three power paths are documented as alternates (pick one — see `INTEGRATION_PROCEDURE.md` for full wiring):

- **Geekworm RPI5-5V5A-PD, terminal-block variant (recommended):** 9-24V DC in via a 3.81mm 2-pin terminal block, genuine USB-C PD negotiation out at 5V/5A — full Pi 5 performance, onboard overvoltage protection intact, no config.txt workaround needed. ~\$15.50 from geekworm.com, not a Mouser item.
- **Mean Well SD-25A-5, 5A (Mouser 709-SD25A-5), GPIO-powered:** full Pi 5 performance, but needs the `usb_max_current_enable=1` config.txt override and loses the onboard overvoltage protection.
- **Mean Well SD-15A-05, 3A (Mouser 709-SD15A-05), GPIO-powered:** simplest wiring, but the Pi 5 runs in its reduced-peripheral-power (600mA USB) mode.

A 1000µF low-ESR bulk capacitor across +5V/GND at the Pi/display junction damps current-transient sag (only needed for the two GPIO-powered alternatives — the PD module’s own output filtering makes it unnecessary). Independent fuse on this branch regardless of which path is chosen.

Display hardware notes — FINAL: Raspberry Pi Touch Display 2

- Official Raspberry Pi Touch Display 2, 7”, 720×1280 native (portrait), DSI-connected via a 22-way (Pi 5) to 15-way (display) FFC cable.
- Standard framebuffer + 5-point capacitive touch, auto-detected by Raspberry Pi OS — requires a current EEPROM bootloader. No SPI/I2C wiring, no register-level driver.
- Pi-side app (`radio_ui.py` + `display.py` in `PiDisplay/`) runs as a windowed Tkinter UI rather than drawing directly to the panel — this replaced the earlier bare-SPI RA8875 approach (`ra8875.py` / `ft5316_touch.py` , both removed).
- Mouse and touch both work as normal desktop input (clicking, dragging, scrolling) with no custom input handling — unlike the RA8875 design, where a mouse was only useful for initial Pi setup over HDMI, not runtime operation.
- WSJT-X is no longer blocked by the display choice (the RA8875 had no mainline framebuffer driver and couldn’t host a Qt app); the Touch Display 2’s standard framebuffer removes that limitation. Its native orientation is portrait, so WSJT-X’s default (wide) layout needs either a landscape display rotation or window placement planned around the portrait aspect ratio.

- The Pi-side UI mirrors the Teensy’s own onboard display panes (`MainBoard_DisplayHome.cpp` `WindowPanes`) by name and relative layout, scaled 1.6x/1.5x from the Teensy’s 800×480 to the Pi’s 1280×720 canvas — see `PiDisplay/README.md` for the full pane-mapping table and which panes have real wire-format data today vs. are still placeholders (`swr` , `audio_spectrum` , `sam_offset` have no `PiLinkStatusMsg` field yet).

UART pin assignment — confirmed against the schematic

`PiLink.h`’s `PI_LINK_SERIAL = Serial1` (Teensy pins 0/1) is correct **conditional on one power-circuit choice**: the T41 Main Board schematic’s Power section offers two mutually exclusive options (“Choose between ‘Shutdown routine activation’ and ‘Voltage reversal prevention’... Populate only one”). This build assumes **Voltage reversal prevention** (the MOSFET-based option, no MCU involved) — under that choice, pins 0/1 are genuinely free.

If a build instead populates **Shutdown routine activation** (adds an ATtiny85 coordinating a clean Teensy power-down), that circuit’s handshake is hardwired to the same two pins and would conflict with PiLink. Every other Teensy 4.1 hardware UART pin-pair is 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. Would need its own resolution at that time.

Pi side: physical pin 8 (GPIO14, TXD) → Teensy pin 0 (RX1); physical pin 10 (GPIO15, RXD) → Teensy pin 1 (TX1); dedicated Pi-GND-to-Teensy-GND wire in addition to the shared power ground. Same physical pins/BCM numbering as every other 40-pin Pi board considered in this project (Pi 5, 3B, both Zero boards) — the GPIO header layout didn’t change when the board choice did. On the Pi, the Linux serial console on `/dev/serial0` must be disabled via `raspi-config` (hardware: enabled, login shell over serial: disabled) so it doesn’t collide with PiLink.

Teensy-side implementation status

Done — see `PiLink.h/PiLink.cpp` in `PhoenixSketch`, wired into `SDT.h` (include), `Globals.cpp` (`InitializePiLink()` call), and `Loop.cpp` (`CheckForPiLinkEvents()` poll + `UpdatePiLinkTelemetry()` send, each loop iteration). This code is unaffected by the board/display choices above — it only cares about the UART link. All three TODOs from the original design are resolved:

- `PI_LINK_SERIAL = Serial1` — confirmed free (see “UART pin assignment” above).
- `SendStatus()`’s `powerLevel` / `filterBandwidthHz` / `keyerWpm` fields are wired to real fields (`ED.powerOutCW/SSB[]` , `ED.FHiCut_Hz` / `FLoCut_Hz` , `ED.currentWPM`).
- `SendSMeter()` calls the new `GetSMeterDBM()` getter (declared in `MainBoard_Display.h`, defined in `MainBoard_DisplayHome.cpp`), which wraps the home screen’s own `AudioToDBM(audioMaxSquaredAve)`.

Pi-side implementation status

Done, structurally; not yet run against real hardware. `PiDisplay/` contains: `pilink_client.py` (PiLink protocol client, mirrors `PiLink.h/.cpp` exactly), `display.py` (Tkinter UI, replaces the removed `ra8875.py`), `radio_ui.py` (main app — routes PiLink telemetry to panes, button clicks to `VIRTUAL_BUTTON` sends, using a thread-safe queue since PiLink telemetry arrives on a background thread but Tkinter isn’t thread-safe), `config.py` (UART port/ baud + display geometry only — no SPI/I2C/GPIO config needed), `pilink-display.service` (user-level systemd unit, since the app needs a graphical/X11 session to open its window). Full step-by-step build/wiring/OS-setup instructions are in `INTEGRATION_PROCEDURE.md`.

Open items / next steps

(See `INTEGRATION_PROCEDURE.md` Section 8 “Future items/roadmap” for the fuller writeup.)

- Full hardware verification pass — nothing here has run on physical Pi 5 + Touch Display 2
 - Teensy hardware yet; the waterfall’s scrolling technique in `display.py` in particular is flagged as unverified.
- Touch/mouse-friendly Pi redesign (dropdown menus etc.) for the Teensy’s separate Menu system (`MainBoard_DisplayMenus.cpp` , 9 categories) — deferred until the Home-screen mirror is trusted on real hardware.
- Two-way settings sync (new uplink message type(s) beyond `VIRTUAL_BUTTON` / `HEARTBEAT`) — needed once

the Pi can write settings back, not just display them.

- Wire the remaining unwired Home panes (`swr` , `audio_spectrum` , `sam_offset`) — needs new `PiLinkStatusMsg` fields; SWR/forward power already exist on the Teensy (`ReadSWR()` / `ReadForwardPower()`).
- Populate the band/mode name lookup table (`display.py` 's `_BAND_MODE_NAMES` is currently empty, so the Pi shows raw numeric ids).
- Real `BUTTON_*` ids in `radio_ui.py` (currently placeholders).
- Ethernet upgrade path (PJRC Ethernet kit + QNEthernet) — not needed at today's UART utilization (~8%), but documented as an option if the link ever needs to carry full-resolution waterfall, bulk transfers, or networked features.
- Graceful shutdown: resolved for this revision via the Pi's read-only overlay filesystem (no clean-shutdown lead time exists in this design, so the fix is making an abrupt power cut harmless rather than detecting it) — see `INTEGRATION_PROCEDURE.md` Section 8 item 8 for the follow-on plan (`PILINK_MSG_COMMIT_SETTINGS`) if the Pi ever gains its own persisted settings.