Push-button power control for a Sorter V2 build

Concept for switching a Mean Well HRPG-150-24 on and off with a push button, with a safe Orange Pi shutdown and an RGB status LED. Includes the servo-flap approach and the full source code.

Document	Concept and design description
Version / date	Draft 0.1, 21 October 2026
Status	**Concept only. Not built, not simulated, not tested.**
Scope	Power-up/power-down control, status indication, servo flap connection (up to 16 layers), Orange Pi software hooks
Based on	Sorter V2 open-source project (docs and KiCad files), Mean Well HRPG-150 datasheet (2017-07-14)

Please read first. Component values are calculated, part numbers are suggestions and must be checked against the manufacturers' datasheets before anything is built. Mains wiring must only be done by a qualified electrician. The Sorter V2 documentation used here is marked by its authors as partly AI-generated first drafts that were not checked against a built machine, so every project detail should be confirmed in the official KiCad files and firmware. This concept is not affiliated with the Sorter V2 project.

1. Summary

- **Goal:** start the machine with one button and shut it down cleanly. The Orange Pi must finish its shutdown before the 24 V supply is switched off.
- **Approach:** a small power manager (ATtiny214) runs from the always-on 5 V auxiliary output of the HRPG-150-24 and drives the supply's remote-control input.
- **Key datasheet finding:** the HRPG remote input is **active-low-off**: open = ON, shorted = OFF. The concept therefore uses a normally-closed (1-Form-B) photo-MOS relay, which also makes the system fail-safe: if the power manager is dead, the supply stays off.
- **Isolation:** the auxiliary ground (AUXG) is isolated from the 24 V output. The power manager is therefore coupled to the Orange Pi by two optocouplers.
- **Status LED:** a plain RGB LED (common anode, no data protocol) on its own cable, separate from the button. It is supplied with about 4.6 V because green and blue LEDs do not light properly from 3.3 V.
- **Servo flaps:** no per-layer boards. The servos plug into the 3-pin ports of the PCA9685 servo-driver module, which gives up to 16 layers per module.
- **Automatic shutdown:** an optional Orange Pi service can switch the whole machine off when sorting has finished (no new pieces for a configurable time). It uses the Sorter V2 shutdown API and the same shutdown path as the button.
- **Open risks:** the total power budget with all steppers is not yet measured (the HRPG-150 has 156 W), the 6.5 V servo rail is limited to 1.5 A, and the exact photo-MOS part must be selected.

2. Starting point and scope

The original question was how to build and wire a sorting machine with a variable number of layers (1 to 20), each layer with a flap that releases the sorted part. A review of the Sorter V2 project showed that its control board, servo-driver arrangement and power distribution already solve most of this. The concept therefore re-uses the project and adds only what is missing.

Provided by the Sorter V2 project (re-used)	Added by this concept

"basically board v1.3" control board: Raspberry Pi Pico, five TMC2209 stepper drivers, 5 V / 12 V / 6.5 V (ServoV) rails, four LED ports, two switch inputs	Power manager at the HRPG-150-24 auxiliary output (button, remote on/off, shutdown handshake)
PCA9685 servo driver (16 channels, I²C 0x40), MG995 servos on 6.5 V	Direct servo connection to the module ports (no layer adapter boards, no ribbon cables)
24 V to 5 V USB-C converter for the Orange Pi 5, USB hub, three PSU jacks	RGB status LED and button as a separate control panel
Orange Pi 5 (8 GB) with SorterOS	Two small systemd services and a GPIO configuration on the Orange Pi, plus an optional automatic-shutdown service for the end of a sorting job

Out of scope: the mechanical design, the sorter software, the mains-side wiring inside the PSU box, and a PCB layout.

3. System architecture

3.1 Power paths

- Mains goes through the project's PSU box (fused IEC inlet with switch) into the **HRPG-150-24** (24 V, 6.5 A, 156 W).
- The 24 V output feeds three loads through separate leads: **PWR1** the control board, **PWR2** the USB hub, **PWR3** the 24 V to 5 V USB-C converter for the Orange Pi.
- The auxiliary output **AUX** (5 V, max. 0.3 A) stays live while the supply is switched off. It powers the power manager and the status LED.
- On the control board, 24 V is converted to 5 V, 12 V and 6.5 V (ServoV). ServoV feeds the PCA9685 module and the servos.

3.2 Control and signal paths

Signal	From to	Level / medium	Notes
RC+ / RC-	Power manager to HRPG CN100 pins 6/5	dry contact (photo-MOS)	open = supply ON, closed = OFF
REQ_N	Power manager to Orange Pi	optocoupler U5, 3.3 V on Pi side	low pulse (500 ms) = shut down now
PWROFF	Orange Pi to power manager	optocoupler U4	high for at least 500 ms = Pi is about to halt
USB	Orange Pi to USB hub to Pico	USB	project communication (steppers, servos, lamps)
I²C	Pico to PCA9685 module	3.3 V to 5 V via PCA9306	servo angles, 50 Hz PWM generated by the module
PWM_0 ... PWM_15	PCA9685 to servo ports	5 V PWM, 220 ohm per output	one port per layer

4. HRPG-150 datasheet findings

Source: Mean Well HRPG-150 series specification, 2017-07-14. The signal connector is **CN100** (mating housing HRS DF11-6DS).

CN100	Signal	From the datasheet
1	-S	Remote sense negative, not used
2	+S	Remote sense positive, not used
3	AUXG	Auxiliary ground, **isolated from the output terminals (+V and -V)**
4	AUX	4.75 to 5.25 V, max. 0.3 A, **not** controlled by remote on/off
5	RC-	Remote control ground
6	RC+	**4 to 10 V or open = power ON; 0 to 0.8 V or short = power OFF**

Further values used in the concept: output 21.6 to 28.8 V adjustable (24 V nominal), 6.5 A / 156 W, overload protection at 105 to 135 % with constant-current limiting and automatic recovery, inrush typically 35 A at 115 V and 70 A at 230 V, setup time about 3 s. The datasheet also states: *"The 5V standby must be connected with a minimum load of 0.03 A."*

Consequences for the design

Datasheet fact	Design consequence
RC open = ON, short = OFF	A normal (normally-open) relay would switch the supply on as soon as mains is applied. U2 is a **normally-closed photo-MOS (1-Form-B)**. Unpowered, the contact is closed and the supply stays off (fail-safe).
AUX minimum load 30 mA	Resistor **R14 (150 ohm, 0.25 W)** draws about 30 mA from the 4.6 V rail behind D4. Regulator, ATtiny, LED and U2 add to this and stay below 0.3 A.
AUXG isolated from -V	The standby ground is **not** tied to the 24 V ground. The Orange Pi interface uses **two optocouplers** (U4, U5). Earlier drafts that joined the grounds with a link have been dropped.
Output adjustable to 28.8 V	The 24 V to 5 V converter accepts up to 32 V, so the adjustment range is uncritical.
Constant-current limiting at 105 to 135 %	An overload does not trip a fuse. The total load must be kept clearly below 6.5 A (see section 9).

5. Power manager design

5.1 Circuit blocks

Block	Parts	Function
Supply	D2 (SS14), U1 (MCP1700-3302), C3/ C4 1 uF, C5 100 nF	3.3 V logic rail from AUX; the ATtiny runs whenever mains is present
Remote switch	U2 photo-MOS 1-Form-B (e.g. G3VM-61B1 or AQY412), R4 220 ohm	shorts RC+/RC- while its LED is off (supply OFF). ATtiny drives the LED to open the contact (supply ON)
Controller	U3 ATtiny214 (SOIC-14), UPDI header J2	state machine, debouncing, LED control
Pi interface	U4, U5 (PC817), R7, R9, R12, R13	isolated handshake: REQ_N to the Pi, PWROFF from the Pi
Minimum load	R14 150 ohm / 0.25 W	satisfies the 30 mA AUX minimum load
Control panel	SW1 push button, LED2 RGB LED (common anode), D4 (SS14), R3, R10, R11, R2, D3, C6	button input with ESD protection, status output, LED supply from AUX

5.2 Sequence

State	Event	Action
OFF	short button press	U2 LED on, remote input opens, supply starts (about 3 s setup), state ON
ON	short button press	REQ_N low for 500 ms (through U5); the Orange Pi runs `systemctl poweroff`
ON	PWROFF high for at least 500 ms (through U4)	state SHUTDOWN, latched even if the line drops again
SHUTDOWN	after 25 s	U2 LED off, remote input shorted, supply OFF
ON	button held for 8 s	emergency off immediately (for a hung Pi); LED red for 1.5 s
ON	sorting finished (optional watchdog on the Pi, section 5.6)	the Pi shuts itself down; PWROFF follows and the supply is switched off after 25 s. The LED goes straight to red slow blinking

A Pi **reboot** does not set PWROFF, so the supply stays on. After a mains failure the system stays off until the button is pressed (the constant `AUTO_START` can change this). The 25 s delay covers the file-system sync and the kernel halt

after the shutdown service has signalled PWROFF.

5.3 Failure behaviour

Situation	Result
Power manager unpowered or dead	U2 contact closed, supply stays OFF
Mains returns after a failure	supply stays OFF until the button is pressed
Orange Pi hangs	hold the button for 8 s (emergency off); no clean shutdown, so file-system recovery may be needed
Cable to the control panel cut	button reads open (no press); LED dark; the machine can still be shut down from the Pi
PWROFF glitch on the Pi pin during boot	ignored unless the signal stays high for 500 ms

5.4 Control panel: button and RGB LED

The push button SW1 has no LED. Status is shown by an external **RGB LED (LED2) with common anode** on its own 4-wire cable (+4.6 V, red, green, blue). Button and LED connectors are separate so that either part can be replaced independently.

Why 4.6 V and a common anode: green and blue LEDs have about 3 V forward voltage, which leaves almost no current from a 3.3 V pin. With the anode at 4.6 V (AUX through the Schottky diode D4) each colour carries about 5 mA. The ATtiny pins switch the cathodes: pin low = colour on, pin high (3.3 V) = off, because only about 1.3 V then remains across LED and resistor.

> **Important:** the three LED pins must never be left high-impedance. A floating cathode would rise towards 4.6 V and feed the 3.3 V rail through the ATtiny's protection diode. The firmware writes HIGH first and then makes the pins outputs. A MOSFET per colour (for example BSS138) would be the more robust, but larger, alternative. An addressable LED (WS2812B) would need only one data pin, but adds a library dependency and needs a reliable 3.3 V data level.

LED indication	Meaning
dark, short green blink every 3 s	supply off, power manager waiting
blue	starting, first 45 s after power-on (Orange Pi booting, time assumption)
green	running
yellow (red + green) fast blinking	shutdown requested, Pi shutting down
red slow blinking	Pi has signalled PWROFF, supply switches off in 25 s
red for 1.5 s	emergency off

5.5 Interface to the Orange Pi (isolated)

Signal	Path	Parts
REQ_N	ATtiny PA7 pulls the LED of U5 to ground; the transistor of U5 pulls REQ_N low on the Pi side	R13 470 ohm, U5 PC817, R9 10 kohm pull-up to the Pi's 3.3 V
PWROFF	the Pi's GPIO drives the LED of U4; the transistor of U4 puts PA6 on the 3.3 V standby rail	R12 680 ohm, U4 PC817, R7 100 kohm pull-down

The logic seen by the Orange Pi is unchanged: REQ_N low = shut down, PWROFF high = going down.

5.6 Automatic shutdown when sorting has finished

The power manager does not need to know about the end of a sorting job. Any Linux shutdown of the Orange Pi ends in the same path as the button: `powerctl-off.service` signals PWROFF, and the supply is switched off 25 s later. The machine can therefore switch itself off after sorting if software on the Pi starts a normal shutdown.

The Sorter V2 backend already provides `POST /api/system/shutdown` (powers the whole machine down, equal to `shutdown -h now`; the backend runs as root on the Pi). What it does not provide is a "finished" signal: the controller lifecycle has the states initializing, paused, running and ready, and the operator stops the sorter manually. A small

optional watchdog, **powerctl-auto**, therefore defines "finished" as "no new pieces for a while":

- It polls `GET /api/system/status` (hardware state, error) and `GET /api/pieces/overview` (piece counters) on the loopback interface every 30 s.
- It arms itself only after at least one piece has been distributed since it started, so a machine that was just switched on and is still idle is never shut down.
- If the counters stay unchanged for **IDLE_MIN** minutes (default 20), the hardware state is `ready` and no hardware error is active, a grace period of **GRACE_S** seconds (default 60) starts. Any new piece, state change or error during the grace period cancels it.
- After the grace period it calls `POST /api/system/shutdown` once. The Pi powers down, PWROFF is raised, and the power manager cuts the 24 V supply 25 s later.
- If the backend is unreachable, the idle timer restarts, so a crashed or restarting backend never causes a shutdown.

Setting (/etc/default/powerctl-auto)	Default	Meaning
AUTO_SHUTDOWN	0	1 = enabled. The feature is off by default
API_URL	http://127.0.0.1:8000	local Sorter V2 backend (loopback only)
IDLE_MIN	20	minutes without any new piece before the countdown starts
GRACE_S	60	seconds of grace; any activity cancels the shutdown
POLL_S	30	polling interval in seconds
DRY_RUN	1	1 = only log what would happen (use for the first tests)

Limits. A paused machine cannot be told apart from a finished one over the REST API (the controller lifecycle is only broadcast over the WebSocket), so IDLE_MIN must be generous. The API port (8000) and the endpoint names were taken from the current repository and may differ in other versions. A proper "run finished" event in SorterOS, for example when the bulk bucket is empty, would be a better trigger than inferring the end from silence.

The watchdog logic was tested against a fake backend (five scenarios: never sorted, sorted then idle, activity during the grace period, live shutdown call, hardware error). It was not tested against a real Sorter V2 backend.

6. Servo flaps

The Sorter V2 design connects one MG995 servo per layer. The signals come from a PCA9685 chip or module (16 outputs, 50 Hz PWM, I^2C address 0x40); in the project, ten outputs are routed on a ribbon cable to passive per-layer adapter boards. This concept drops the layer boards and plugs the servos directly into the module's 3-pin ports (brown GND, red V+, orange PWM).

- **Capacity:** 16 ports per module, so up to 16 layers instead of 10. A second module (for example address 0x41) would allow 20 layers in hardware, but the project firmware only addresses 0x40, so a firmware change would be needed.
- **ServoV rail:** the rail is rated 1.5 A (6.5 V, from the 12 V rail). A stalled MG995 draws about 1.2 A, two stalled servos overload it. The project firmware releases a servo after at most 3.5 s.
- **Cable length:** the servo lead is 300 mm. Higher layers need extension leads (3-core, 22 AWG or thicker). The 100 uF capacitor of the adapter board is missing at the servo, so add 100 to 220 uF across V+ and GND at the plug for long leads.

7. Orange Pi software

Two small systemd units run next to the Sorter software (SorterOS). The full source is in Appendix B.

- **powerctl-req.service** waits for a falling edge on REQ_N (`gpiomon`) and runs `systemctl poweroff`.
- **powerctl-off.service** runs only on poweroff/halt (not on reboot), before `poweroff.target`, and sets PWROFF high for 2 s with `gpioset`.
- **powerctl-auto.service** (optional, off by default) is the automatic-shutdown watchdog from section 5.6. Its settings are in `/etc/default/powerctl-auto`.
- **/etc/default/powerctl** holds the GPIO chip and line numbers. They depend on the pins chosen on the 40-pin header (check with `gpioinfo`).

The scripts use the libgpiod 1.x command-line syntax. With libgpiod 2.x the options must be adapted. The Orange Pi GPIOs are 3.3 V and not 5 V tolerant.

8. Bill of materials (power manager and control panel)

Part numbers are suggestions ("e.g."). Check ratings in the manufacturers' datasheets. Project parts (control board, converter M1, Orange Pi, hub, servos) are not listed again.

Ref	Part	Value / type
U1	LDO 3.3 V	MCP1700-3302
U2	Photo-MOS relay, 1-Form-B (normally closed)	e.g. Omron G3VM-61B1 or Panasonic AQY412; check LED current to open and off-state voltage
U3	Microcontroller	ATtiny214 (SOIC-14)
U4, U5	Optocouplers	PC817 (or one PC827 dual)
D2, D4	Schottky diodes	SS14
D3	ESD protection diode	3.3 V, SOD-323
C3, C4	Capacitors	1 uF
C5, C6	Capacitors	100 nF
R2	Resistor	1 kohm (button)
R3, R11	Resistors	330 ohm (LED green, LED blue)
R10	Resistor	470 ohm (LED red)
R4	Resistor	220 ohm (LED of U2)
R7 / R9	Resistors	100 kohm pull-down / 10 kohm pull-up
R12 / R13	Resistors	680 ohm / 470 ohm (optocoupler LEDs)
R14	Resistor	150 ohm, 0.25 W (AUX minimum load)
SW1	Push button, no LED	panel mount, normally open
LED2	RGB LED 5 mm, common anode, not addressable	panel or bezel mount
J2, J3, J4, J_PI	Headers	UPDI, button (2-pin), LED (4-pin), Orange Pi
CN100 plug	HRS DF11-6DS with DF11-**SC contacts	to the supply, pins 3 to 6 are used

9. Risks and open points

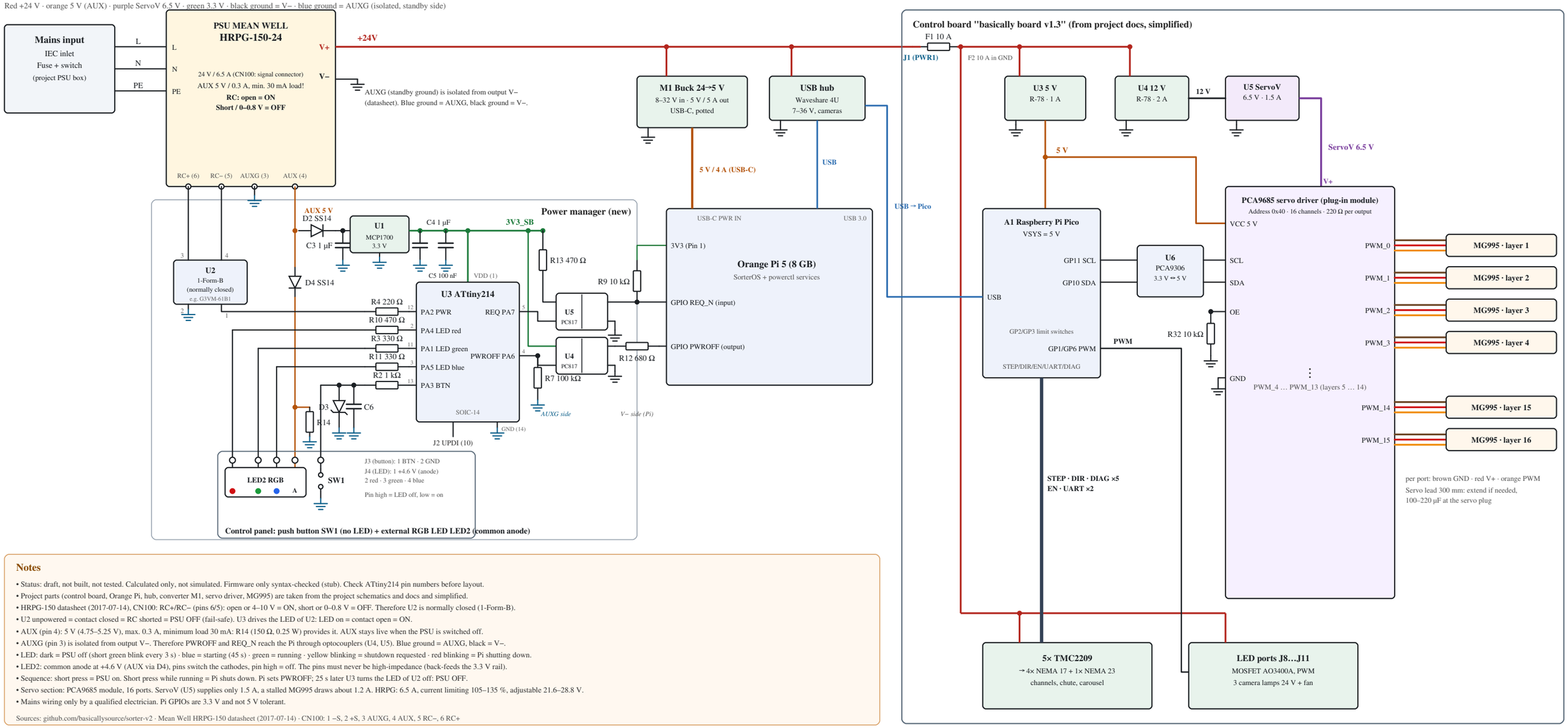
#	Open point	Priority	Next action
1	**Total power budget not verified.** The project sized its 350 W supply "as something known to work" and names a 150 W Mean Well as a candidate after measurement. A rough upper bound from the project's own ratings is about 120 to 130 W: Orange Pi branch up to 25 W, hub and cameras about 5 to 10 W (assumption), three lamps about 28 W, fan 1 W, four NEMA 17 steppers up to about 40 W at full holding current, the NEMA 23 about 5 W at the driver limit, servo rail up to about 10 W, plus converter losses.	High	Measure the real 24 V current in all machine states (homing, sorting, lamps at full brightness) before relying on the 156 W supply.
2	**Servo rail 1.5 A.** Several simultaneously stalled servos overload U5.	High	Limit concurrent moves in software, or size a stronger 6.5 V converter on the control board.
3	**U2 part selection.** The HRPG datasheet does not give the open-circuit voltage or short-circuit current of the RC pins.	High	Measure RC+/RC- on the actual supply; check the relay's LED current and voltage ratings.
4	**Supply protection.** The project's board fuses are 10 A, but the HRPG delivers only 6.5 A and relies on current limiting from 105 %.	Medium	Add a 6 to 8 A fuse in the 24 V feed, or accept limiting behaviour after the measurement in point 1.
5	**Pin numbers and firmware.** ATtiny214 pin numbers were not checked against the datasheet; the firmware was only syntax-checked against a stub.	Medium	Verify the pinout, compile with megaTinyCore, test on a bench setup with an LED and a dummy load.
6	**Project documentation status.** Several project pages are labelled AI-generated first drafts.	Medium	Confirm details against the official KiCad files, firmware and the published v1.3 board schematic.
7	**More than 16 layers.** Needs a second PCA9685 module and a firmware change.	Low	Decide whether 16 layers are sufficient.
8	**No PCB layout.**	Low	Create after the points above are closed. Keep the board low voltage only.
9	**Definition of "finished".** The backend has no finished state. The watchdog infers it from missing pieces and cannot tell a pause from the end of a job. API port and endpoints were read from the current repository.	Medium	Keep IDLE_MIN generous, start with DRY_RUN=1, confirm port and endpoints on the real machine, and consider a real run-finished event in SorterOS.

10. Suggested next steps

- Measure the 24 V load of a running machine, then decide on the supply and fuse (points 1 and 4).
- Bench-test the power manager with the real HRPG-150-24: measure RC+/RC-, AUX load, and the start/stop sequence with a dummy load.
- Compile and flash the firmware, check the LED colours and timing.
- Install the Orange Pi services, set the GPIO lines, and test the button shutdown and the PWROFF path end to end.
- Test the optional automatic shutdown with DRY_RUN=1 on a real machine, then enable it.
- Test servo ports with the longest planned leads and the added capacitors.

Overall schematic: sorting machine (Sorter V2) with HRPG-150-24 and power manager

Red +24 V · orange 5 V (AUX) · purple ServoV 6.5 V · green 3.3 V · black ground = V− · blue ground = AUXG (isolated, standby side)



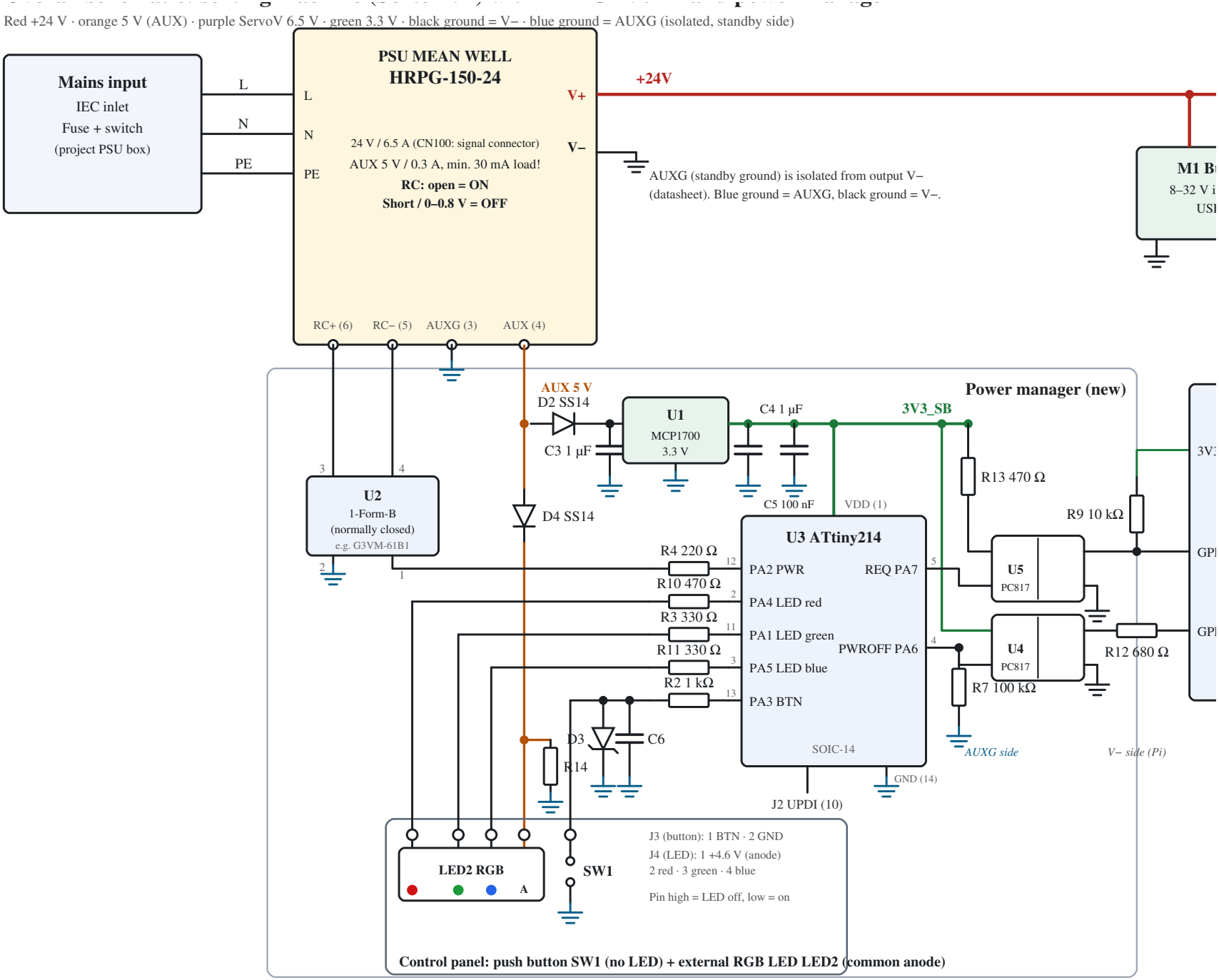
Notes

- Status: draft, not built, not tested. Calculated only, not simulated. Firmware only syntax-checked (stub). Check ATtiny214 pin numbers before layout.
- Project parts (control board, Orange Pi, hub, converter M1, servo driver, MG995) are taken from the project schematics and docs and simplified.
- HRPG-150 datasheet (2017-07-14), CN100: RC+/RC− (pins 6/5): open or 4–10 V = ON, short or 0–0.8 V = OFF. Therefore U2 is normally closed (1-Form-B).
- U2 unpowered = contact closed = RC shorted = PSU OFF (fail-safe). U3 drives the LED of U2: LED on = contact open = ON.
- AUX (pin 4): 5 V (4.75–5.25 V), max. 0.3 A, minimum load 30 mA: R14 (150 Ω, 0.25 W) provides it. AUX stays live when the PSU is switched off.
- AUXG (pin 3) is isolated from output V−. Therefore PWROFF and REQ_N reach the Pi through optocouplers (U4, U5). Blue ground = AUXG, black = V−.
- LED: dark = PSU off (short green blink every 3 s) · blue = starting (45 s) · green = running · yellow blinking = shutdown requested · red blinking = Pi shutting down.
- LED2: common anode at +4.6 V (AUX via D4), pins switch the cathodes, pin high = off. The pins must never be high-impedance (back-feeds the 3.3 V rail).
- Sequence: short press = PSU on. Short press while running = Pi shuts down. Pi sets PWROFF; 25 s later U3 turns the LED of U2 off: PSU OFF.
- Servo section: PCA9685 module, 16 ports. ServoV (U5) supplies only 1.5 A, a stalled MG995 draws about 1.2 A. HRPG: 6.5 A, current limiting 105–135 %, adjustable 21.6–28.8 V.
- Mains wiring only by a qualified electrician. Pi GPIOs are 3.3 V and not 5 V tolerant.

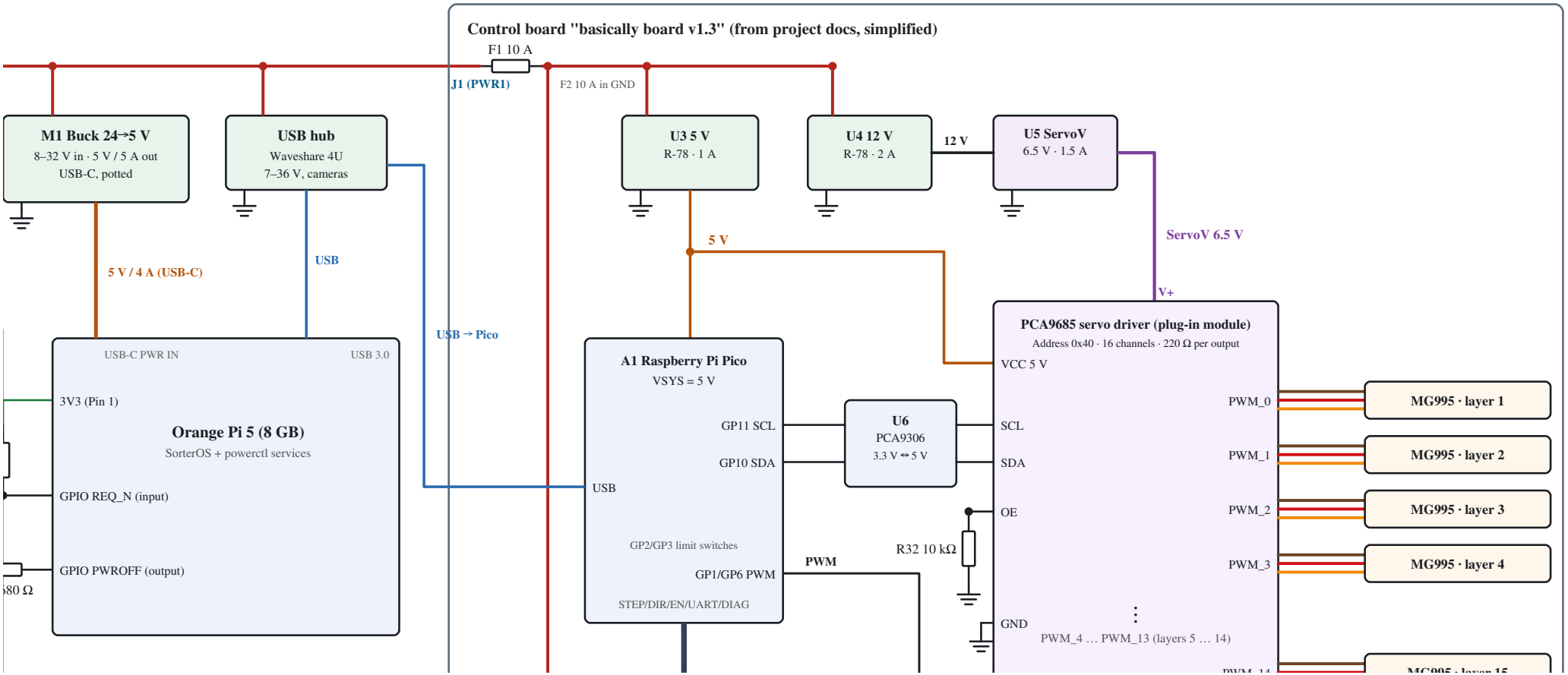
Sources: github.com/basicallysource/sorter-v2 · Mean Well HRPG-150 datasheet (2017-07-14) · CN100: 1 –S, 2 +S, 3 AUXG, 4 AUX, 5 RC–, 6 RC+

Red = +24 V, orange = 5 V (AUX), purple = ServoV 6.5 V, green = 3.3 V standby rail. Black grounds = 24 V side (V-), blue grounds = isolated auxiliary ground (AUXG). Project blocks are simplified from the Sorter V2 documentation. Pages A.2 to A.4 show details.

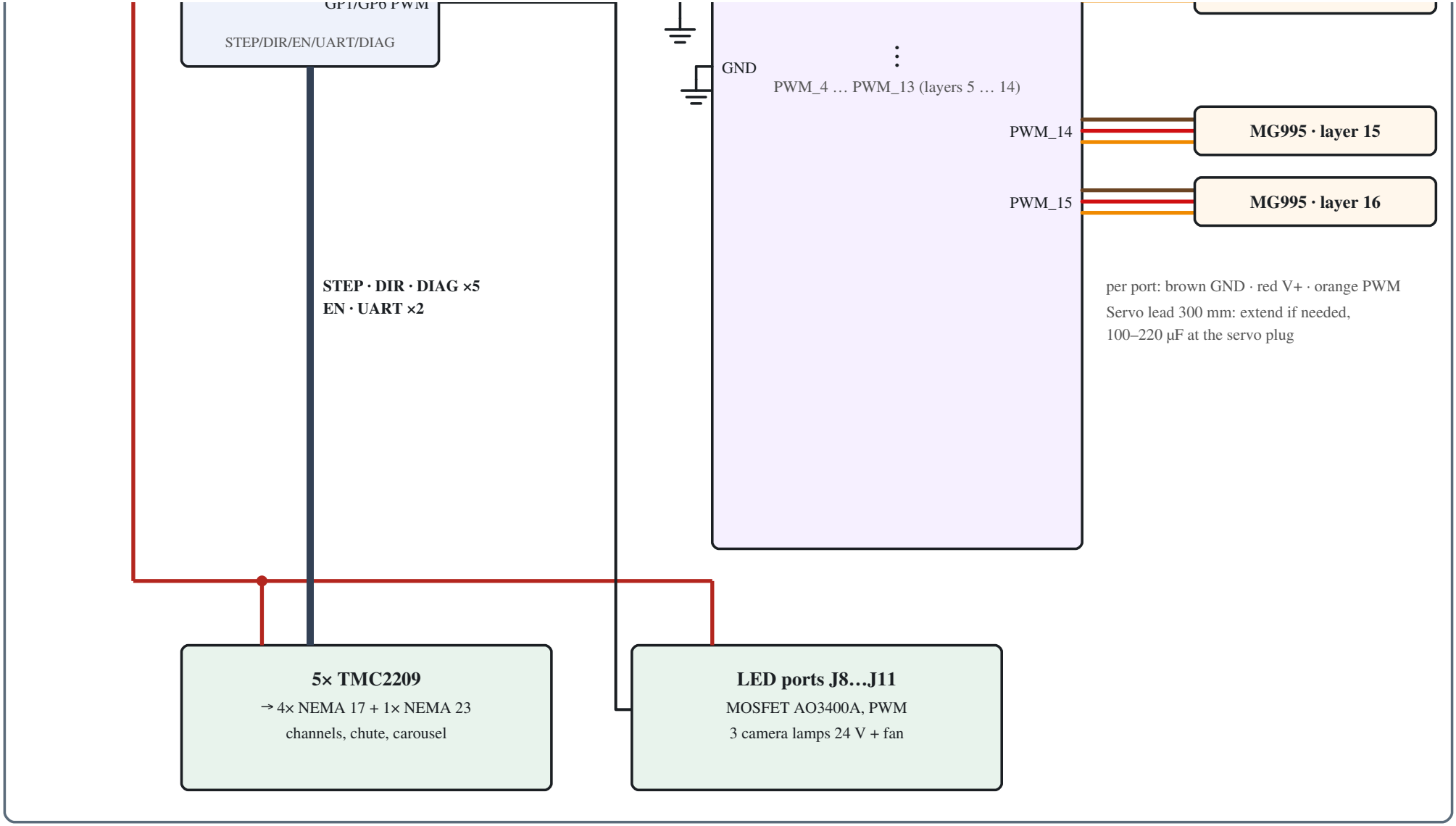
Appendix A.2 Detail: HRPG-150-24, power manager, control panel



Appendix A.3 Detail: Orange Pi, converter, hub, control board (upper part)



Appendix A.4 Detail: servo ports, stepper drivers, LED ports (lower part)



Appendix B. Source code

All files are shown in full. The firmware is for the ATtiny214 (megaTinyCore) and was only syntax-checked against a stub, not compiled for the target and not tested on hardware. The Orange Pi files are for Debian/Ubuntu with libgpiod 1.x. Install steps:

```
sudo apt install gpiod
sudo cp powerctl.conf /etc/default/powerctl      # adjust the GPIO chip/line values
sudo cp powerctl-req.sh powerctl-off.sh /usr/local/bin/
sudo chmod +x /usr/local/bin/powerctl-req.sh /usr/local/bin/powerctl-off.sh
sudo cp powerctl-req.service powerctl-off.service /etc/systemd/system/
sudo systemctl daemon-reload
sudo systemctl enable --now powerctl-req.service
sudo systemctl enable powerctl-off.service
# optional automatic shutdown after sorting (off by default)
sudo cp powerctl-auto.py /usr/local/bin/ && sudo chmod +x /usr/local/bin/powerctl-auto.py
sudo cp powerctl-auto.conf /etc/default/powerctl-auto      # AUTO_SHUTDOWN=0, DRY_RUN=1 by default
sudo cp powerctl-auto.service /etc/systemd/system/ && sudo systemctl daemon-reload
# to enable: set AUTO_SHUTDOWN=1 (and DRY_RUN=0 once tested), then
sudo systemctl enable --now powerctl-auto.service
```

B.1 power_manager.ino (ATtiny214 firmware)

```
/*
  Power manager for the Mean Well HRPG-150-24 + Orange Pi 5
  Target MCU: ATtiny214 (SOIC-14, megaTinyCore), runs on 3.3 V from the 5 V standby (AUX) output.
  Control panel: push button (no LED) + external RGB LED (common anode at +4.6 V).

  STATUS: not compiled against megaTinyCore, not tested on hardware (syntax-checked against a stub only).

  HRPG-150 remote control (datasheet): RC open = ON, RC shorted = OFF.
  U2 is therefore a normally-closed photo-MOS relay (1-Form-B). The firmware only has to drive PWR_ON
  HIGH to switch the supply on. The link to the Orange Pi uses optocouplers (AUXG is isolated from the output).

  Sequence
    OFF        short press         -> supply on (RC released), state ON
    ON         short press         -> REQ_N low for 500 ms (Pi shuts down)
    ON         PWROFF >= 500 ms high (from the Pi, just before it halts)
                                   -> state SHUTDOWN (latched)
    SHUTDOWN   after 25 s          -> supply off, state OFF
    ON         press >= 8 s        -> immediate emergency off (Pi hung)
  A Pi reboot does not set PWROFF and therefore does not switch the supply off.

  RGB LED (each colour on/off only)
    OFF                          dark, short green blink every 3 s (alive indicator)
    ON, first 45 s               blue          = starting, Pi booting (time assumption)
    ON                           green         = running
    ON, shutdown requested       yellow (red+green), fast blinking
    SHUTDOWN                     red, slow blinking
    after emergency off          red for 1.5 s

  LED drive: common anode at +4.6 V. The pins switch the cathodes.
    Pin LOW  = colour on  (about 5 mA through 470/330 ohm)
    Pin HIGH = colour off (only about 1.3 V across LED + resistor, does not light)
  IMPORTANT: the pins must never be high-impedance, otherwise the cathode rises towards 4.6 V and
  feeds the 3.3 V rail through the protection diode. So write HIGH first, then make the pin an OUTPUT.

  Pin assignment ATtiny214 (see overall schematic)
    PA4 LED_RED   (cathode via 470 ohm)                              pin 2
    PA1 LED_GREEN (cathode via 330 ohm)                              pin 11
    PA5 LED_BLUE  (cathode via 330 ohm)                              pin 3
    PA2 PWR_ON    (output, LED of U2 via 220 ohm)                    pin 12
                  HIGH = LED on = U2 (normally closed) open = RC open = supply ON
                  LOW/unpowered = U2 closed = RC shorted = supply OFF
    PA3 BTN       (input, internal pull-up, button to GND)           pin 13
    PA6 PWROFF    (input, from the Pi via optocoupler U4, 100 kohm pull-down)  pin 4
    PA7 REQ_N     (open-drain substitute: low = LED of optocoupler U5 on, otherwise high-impedance)  pin 5
    PA0 UPDI      (programming)                                      pin 10
  Check the pin numbers against the datasheet before layout.
*/

#include <Arduino.h>

const uint8_t PIN_LED_R  = PIN_PA4;
const uint8_t PIN_LED_G  = PIN_PA1;
const uint8_t PIN_LED_B  = PIN_PA5;
const uint8_t PIN_PWR    = PIN_PA2;
const uint8_t PIN_BTN    = PIN_PA3;
const uint8_t PIN_PWROFF = PIN_PA6;
const uint8_t PIN_REQ    = PIN_PA7;

const uint32_t DEBOUNCE_MS      = 40;
const uint32_t REQ_PULSE_MS     = 500;
const uint32_t PWROFF_VALID_MS  = 500;      // PWROFF must stay high this long
const uint32_t GRACE_MS         = 25000UL;  // delay until the supply is switched off
const uint32_t HARD_OFF_MS      = 8000UL;   // hold the button this long for emergency off
const uint32_t BOOT_SHOW_MS     = 45000UL;  // the LED shows "starting" (blue) for this long
const uint32_t HARD_OFF_LED_MS  = 1500UL;   // red light after an emergency off
const bool     AUTO_START       = false;    // true: switch on automatically after a power failure

```

```
enum State : uint8_t { ST_OFF, ST_ON, ST_SHUTDOWN };
State state = ST_OFF;

uint32_t tState = 0;       // time of the last state change
uint32_t tReqStart = 0;
bool     reqActive = false;
bool     shutdownRequested = false;  // button was pressed while running
uint32_t hardOffUntil = 0;           // LED red until this time

// Button debouncing
bool     btnStable = false;   // true = pressed
bool     btnRaw    = false;
uint32_t btnChange = 0;
uint32_t btnPressedAt = 0;
bool     waitRelease = false; // after an emergency off, wait for the button to be released

uint32_t pwroffHighSince = 0;

static inline void pwrSet(bool on) { digitalWrite(PIN_PWR, on ? HIGH : LOW); }

// Common anode: LOW = colour on
static inline void ledSet(bool r, bool g, bool b) {
  digitalWrite(PIN_LED_R, r ? LOW : HIGH);
  digitalWrite(PIN_LED_G, g ? LOW : HIGH);
  digitalWrite(PIN_LED_B, b ? LOW : HIGH);
}

static void reqAssert() {
  digitalWrite(PIN_REQ, LOW);
  pinMode(PIN_REQ, OUTPUT);
  reqActive = true;
  tReqStart = millis();
}

static void reqRelease() {
  pinMode(PIN_REQ, INPUT);   // high-impedance, the pull-up is on the Pi side
  reqActive = false;
}

static void enter(State s) {
  state = s;
  tState = millis();
  if (s != ST_ON) shutdownRequested = false;
}

void setup() {
  // LED pins: write HIGH (off) first, then make them outputs, never leave them high-impedance
  digitalWrite(PIN_LED_R, HIGH);
  digitalWrite(PIN_LED_G, HIGH);
  digitalWrite(PIN_LED_B, HIGH);
  pinMode(PIN_LED_R, OUTPUT);
  pinMode(PIN_LED_G, OUTPUT);
  pinMode(PIN_LED_B, OUTPUT);

  pinMode(PIN_PWR, OUTPUT);
  pwrSet(false);
  pinMode(PIN_BTN, INPUT_PULLUP);
  pinMode(PIN_PWROFF, INPUT);
  pinMode(PIN_REQ, INPUT);

  if (AUTO_START) {
    pwrSet(true);
    enter(ST_ON);
  }
}

void loop() {
  const uint32_t now = millis();

  // ---- Debounce the button -------------------------------------------------
  bool raw = (digitalRead(PIN_BTN) == LOW);
  if (raw != btnRaw) { btnRaw = raw; btnChange = now; }
  bool pressedEdge = false;
  if ((now - btnChange) >= DEBOUNCE_MS && btnStable != btnRaw) {
    btnStable = btnRaw;
    if (btnStable) { pressedEdge = true; btnPressedAt = now; }
  }
  if (waitRelease) {
    if (!btnStable) waitRelease = false;
    pressedEdge = false;
  }

  // ---- Check the PWROFF signal from the Pi ---------------------------------
  bool pwroffSeen = false;
  if (digitalRead(PIN_PWROFF) == HIGH) {
    if (pwroffHighSince == 0) pwroffHighSince = now ? now : 1;
    if ((now - pwroffHighSince) >= PWROFF_VALID_MS) pwroffSeen = true;
  } else {
    pwroffHighSince = 0;
  }

  // ---- End the REQ_N pulse -------------------------------------------------
```

```
  if (reqActive && (now - tReqStart) >= REQ_PULSE_MS) reqRelease();

  // ---- State machine ------------------------------------------------------
  switch (state) {
    case ST_OFF:
      pwrSet(false);
      if (pressedEdge) {
        pwrSet(true);
        pwroffHighSince = 0;
        enter(ST_ON);
      }
      break;

    case ST_ON:
      if (pressedEdge) { reqAssert(); shutdownRequested = true; }
      if (btnStable && (now - btnPressedAt) >= HARD_OFF_MS) {
        pwrSet(false);                // emergency off
        reqRelease();
        waitRelease = true;
        hardOffUntil = now + HARD_OFF_LED_MS;
        enter(ST_OFF);
        break;
      }
      if (pwroffSeen) enter(ST_SHUTDOWN);   // latched, even if the line drops again later
      break;

    case ST_SHUTDOWN:
      if ((now - tState) >= GRACE_MS) {
        pwrSet(false);
        reqRelease();
        enter(ST_OFF);
      }
      break;
  }

  // ---- RGB LED ------------------------------------------------------------
  switch (state) {
    case ST_OFF:
      if ((int32_t)(hardOffUntil - now) > 0) {
        ledSet(true, false, false);                 // emergency off: red
      } else {
        ledSet(false, (now % 3000) < 20, false);    // alive indicator: short green blink
      }
      break;
    case ST_ON:
      if (shutdownRequested) {
        bool on = ((now / 100) & 1);
        ledSet(on, on, false);                      // shutdown requested: yellow, fast
      } else if ((now - tState) < BOOT_SHOW_MS) {
        ledSet(false, false, true);                 // starting: blue
      } else {
        ledSet(false, true, false);                 // running: green
      }
      break;
    case ST_SHUTDOWN:
      ledSet(((now / 500) & 1), false, false);      // Pi shutting down: red, slow
      break;
  }
}
```

Orange Pi files

B.2 /etc/default/powerctl

```
# /etc/default/powerctl
# GPIO assignment of the J_PI lines on the Orange Pi 5.
# CHIP = gpiochip name or number, LINE = line number.
# The right values depend on the pin chosen on the 40-pin header.
# Look them up with `gpioinfo` and check the pinout in the board manual.

# J_PI.4  REQ_N   (input, low = shutdown requested)
REQ_CHIP=gpiochip3
REQ_LINE=4

# J_PI.3  PWROFF  (output, high = the Pi is about to power off)
OFF_CHIP=gpiochip3
OFF_LINE=5
```

B.3 /usr/local/bin/powerctl-req.sh

```
#!/bin/sh
# /usr/local/bin/powerctl-req.sh
# Waits for a falling edge on REQ_N (button press seen by the power manager) and shuts the Pi down.
# Syntax for libgpiod 1.x (gpiomon). With libgpiod 2.x the options must be adapted.
. /etc/default/powerctl

while true; do
  if gpiomon --num-events=1 --falling-edge "$REQ_CHIP" "$REQ_LINE" >/dev/null 2>&1; then
    logger -t powerctl "Shutdown requested by the button"
    systemctl poweroff
    sleep 60
  else
    sleep 2
  fi
done
```

B.4 /usr/local/bin/powerctl-off.sh

```
#!/bin/sh
# /usr/local/bin/powerctl-off.sh
# Runs on poweroff/halt only (not on reboot) and tells the power manager "going down now".
# The power manager evaluates >= 500 ms high, latches the request and switches the supply off after 25 s.
# Syntax for libgpiod 1.x (gpioset). With libgpiod 2.x the options must be adapted.
. /etc/default/powerctl

gpioset --mode=time --sec=2 "$OFF_CHIP" "$OFF_LINE=1"
```

B.5 powerctl-req.service

```
[Unit]
Description=Evaluate the shutdown request from the power manager
After=multi-user.target

[Service]
ExecStart=/usr/local/bin/powerctl-req.sh
Restart=always

[Install]
WantedBy=multi-user.target
```

B.6 powerctl-off.service

```
[Unit]
Description=Inform the power manager before poweroff
DefaultDependencies=no
Before=poweroff.target halt.target

[Service]
Type=oneshot
ExecStart=/usr/local/bin/powerctl-off.sh

[Install]
WantedBy=poweroff.target halt.target
```

Optional automatic shutdown (Orange Pi)

B.7 /usr/local/bin/powerctl-auto.py

```
#!/usr/bin/env python3
"""powerctl-auto: shut the machine down when sorting has finished.

Optional watchdog for the Orange Pi. It polls the local Sorter V2 backend. When pieces have been
sorted in this session and then nothing has happened for IDLE_MIN minutes while the hardware is
"ready", it waits GRACE_S seconds (cancelled if anything moves again) and then calls
POST /api/system/shutdown. The Linux poweroff triggers powerctl-off.service, which signals PWROFF
to the power manager, which switches the 24 V supply off 25 s later.

Disabled by default (AUTO_SHUTDOWN=0 in /etc/default/powerctl-auto).

The Sorter V2 backend has no "finished" state (lifecycle: initializing, paused, running, ready), so
"finished" is defined as "no new pieces for a while". A paused machine cannot be told apart from a
finished one over the REST API, so choose IDLE_MIN generously.

Endpoints used (Sorter V2 backend, loopback only, check the port for your version):
  GET  /api/system/status     -> {"hardware_state": ..., "hardware_error": ...}
  GET  /api/pieces/overview   -> {"total_pieces": ..., "distributed_pieces": ..., "last_seen": ...}
  POST /api/system/shutdown   -> powers the machine down
"""
import json
import os
import sys
import time
import urllib.error
import urllib.request

CONF_PATH = os.environ.get("POWERCTL_AUTO_CONF", "/etc/default/powerctl-auto")


def log(msg):
    print(f"powerctl-auto: {msg}", flush=True)


def load_conf(path):
    cfg = {
        "AUTO_SHUTDOWN": "0",
        "API_URL": "http://127.0.0.1:8000",
        "IDLE_MIN": "20",
        "GRACE_S": "60",
        "POLL_S": "30",
        "DRY_RUN": "0",
    }
    try:
        with open(path, encoding="utf-8") as f:
            for line in f:
                line = line.strip()
                if not line or line.startswith("#") or "=" not in line:
                    continue
                key, value = line.split("=", 1)
                cfg[key.strip()] = value.strip().strip('"').strip("'")
    except FileNotFoundError:
        pass
    return cfg


def api(cfg, method, path):
    req = urllib.request.Request(cfg["API_URL"].rstrip("/") + path, method=method)
    with urllib.request.urlopen(req, timeout=5) as resp:
        return json.loads(resp.read().decode("utf-8") or "{}")


def snapshot(cfg):
    status = api(cfg, "GET", "/api/system/status")
    overview = api(cfg, "GET", "/api/pieces/overview")
    state = status.get("hardware_state")
    error = status.get("hardware_error")
    sig = (overview.get("total_pieces"), overview.get("distributed_pieces"), overview.get("last_seen"))
    return state, error, sig


def main():
    cfg = load_conf(CONF_PATH)
    if cfg["AUTO_SHUTDOWN"] != "1":
        log("AUTO_SHUTDOWN is not 1, nothing to do")
        return 0

    idle_s = float(cfg["IDLE_MIN"]) * 60.0
    grace_s = float(cfg["GRACE_S"])
    poll_s = float(cfg["POLL_S"])
    dry_run = cfg["DRY_RUN"] == "1"
    log(f"active: idle {cfg['IDLE_MIN']} min, grace {cfg['GRACE_S']} s, poll {cfg['POLL_S']} s, dry run {dry_run}")

    baseline = None        # distributed_pieces when this service started
    armed = False          # true once pieces were sorted in this session
    last_sig = None
```

```
    last_change = time.monotonic()

    while True:
        try:
            state, error, sig = snapshot(cfg)
        except (urllib.error.URLError, OSError, ValueError) as exc:
            log(f"backend not reachable ({exc}); idle timer restarted")
            last_change = time.monotonic()
            time.sleep(poll_s)
            continue

        now = time.monotonic()
        if baseline is None:
            baseline = sig[1] or 0
            log(f"baseline: {baseline} pieces distributed so far")
        if sig != last_sig:
            last_sig = sig
            last_change = now
            if not armed and (sig[1] or 0) > baseline:
                armed = True
                log("sorting seen in this session, auto-shutdown armed")

        idle = now - last_change
        if armed and state == "ready" and not error and idle >= idle_s:
            log(f"no new pieces for {idle / 60:.1f} min, starting {grace_s:.0f} s grace period")
            cancelled = False
            end = time.monotonic() + grace_s
            while time.monotonic() < end:
                time.sleep(min(5.0, max(0.5, end - time.monotonic())))
                try:
                    state2, error2, sig2 = snapshot(cfg)
                except (urllib.error.URLError, OSError, ValueError):
                    cancelled = True
                    break
                if sig2 != sig or state2 != "ready" or error2:
                    cancelled = True
                    break
            if cancelled:
                log("activity during grace period, shutdown cancelled")
                last_change = time.monotonic()
                continue
            if dry_run:
                log("DRY RUN: would call POST /api/system/shutdown now")
                return 0
            log("calling POST /api/system/shutdown")
            try:
                api(cfg, "POST", "/api/system/shutdown")
            except (urllib.error.URLError, OSError, ValueError) as exc:
                log(f"shutdown call failed: {exc}")
                last_change = time.monotonic()
                continue
            time.sleep(300)   # the machine is going down; do not call again
            return 0

        time.sleep(poll_s)


if __name__ == "__main__":
    sys.exit(main())
```

B.8 /etc/default/powerctl-auto

```
# /etc/default/powerctl-auto
# Automatic shutdown when sorting has finished (optional, off by default).

# 1 = enabled, 0 = disabled
AUTO_SHUTDOWN=0

# Local Sorter V2 backend (loopback). Check the port for your version.
API_URL=http://127.0.0.1:8000

# Minutes without any new piece before the shutdown countdown starts.
# A paused machine looks the same as a finished one, so keep this generous.
IDLE_MIN=20

# Seconds of grace after the idle time. Any activity cancels the shutdown.
GRACE_S=60

# Polling interval in seconds.
POLL_S=30

# 1 = only log what would happen, do not shut down (use this for the first tests).
DRY_RUN=1
```

B.9 powerctl-auto.service

```
[Unit]
Description=Shut the machine down when sorting has finished (optional)
After=network.target

[Service]
ExecStart=/usr/bin/python3 /usr/local/bin/powerctl-auto.py
```

```
Restart=on-failure
RestartSec=30

[Install]
WantedBy=multi-user.target
```

Appendix C. Sources

- Sorter V2 project repository: github.com/basicallysource/sorter-v2 (electronics/KiCad, docs: lab/power, lab/servos, lab/pico, hardware pages, parts calculator bill of materials).
- Sorter V2 backend files used for the automatic shutdown: software/sorter/backend/server/routers/system.py (shutdown endpoint), server/routers/pieces.py (pieces overview), sorter_controller.py (lifecycle states), server/shared_state.py (hardware states).
- Mean Well HRPG-150 series datasheet, "HRPG-150-SPEC", 2017-07-14 (RS Online: docs.rs-online.com/7db2/0900766b81621e08.pdf).

This concept was prepared with the help of an AI assistant (Claude) from the sources above. It is a design proposal, not a verified or certified design.