PS1 Compute Blades: wiring, checking, EEPROM, labels (draft 9)

Printed from the pages published at https://docs.fpgas.online/en/latest/, built from commit f51b31947d9b932313eb9b186d1c4f88c372f7b9 of fpgas-online/fpgas.online-docs, fetched 2026-10-06. The published pages are the current ones; this is a copy.

1. Acorn wiring, sheet 3 (boards/acorn/wiring: parts-for-a-compute-blade, board-connectors, compute-blade, cables-for-a-compute-blade, verification, kernel-console-on-the-fpga-uart, troubleshooting)
2. PS1, sheet 42 (sites/ps1: compute-blades, checking-a-board-here)
3. fpgas-verify: checking an FPGA board from its Raspberry Pi, sheet 47 (verify/fpgas-verify: running-it, identity-and-labels, reading-the-result, acorn, checking-an-acorn-s-wiring, common-failures)
4. Bootloader EEPROM: upgrade and lock, sheet 66 (setup/bootloader-eeprom)
5. Labels for the PS1 Compute Blades and their FPGAs, sheet 93

DRAFT COPY 9, 6 OCT 2026, FOR REVIEW. NOT IN THIS COPY YET

- A procedure for converting a card to the fpgas.online design on a Compute Blade. Until a card is converted the check is expected to end in "fail: unconverted".
- Labels for the Acorn cards, and for pi14 and pi18 at ps1 (the last chapter says why).

THE ORDER OF WORK

1. Build the cables and fit them: the chapter "Acorn wiring". Its part "Cables for a Compute Blade" is the whole job in numbered steps, each with its picture, with its own parts and tools list; its last step fits the cables.
2. Boot the blade and log in: the chapter "PS1" says what was last read on each blade at ps1 and what each still needs.
3. Install the checking tool: the chapter "fpgas-verify", from "Checking an Acorn's wiring: After a fresh boot" on.
4. Run the check: the same chapter, the steps that follow the install.
5. EEPROM: in the chapter "Bootloader EEPROM: upgrade and lock", only the part "Compute Module 4, Compute Module 5 and the Compute Blade", sheet 80. The Raspberry Pi 5 steps before it (card, pads, soldering) are not for a blade: skip them.
6. Labels: the last chapter and the two label sheets after it.

READ THIS FIRST

- Every chapter is a page of https://docs.fpgas.online as published on the date in the foot; the site is the current copy.
- The cable-building steps have not been run by us on this hardware: they are written from the design. Each picture has a box listing what is not yet checked against a cable in the hand. Steps 2 and 7 check which wire is wire 1 with a meter before any wire is cut back: do not skip them, and tell us what you see.
- Where a chapter says "not yet run by us on this hardware", go carefully and tell us what you see.
- On a Compute Blade under kernel 6.18, JTAG could not run on pi16 at ps1 with the header's serial port on, and pi20 at ps1 now boots the same way. Booting with the serial port off is expected to free it and is not yet tried; in that boot the serial-port steps cannot run. The wiring chapter's "JTAG on a blade" has the detail.
- Log in as the visitor: ssh -p 11622 pi@ps1.fpgas.online for pi16 at ps1 (the port is 1NN22 for piNN). The password is the one each board page of https://ps1.fpgas.online/fpgas/ prints under "Use your own ssh client" (pi3.html, for example); sudo asks for the same one.
- The chapter "fpgas-verify" is printed as published today, so its tables also carry rows about Tiny Tapeout boards that were merged on 5 Oct 2026 (its "sdk" test). They do not concern the Compute Blades.
- Links are numbered in the text; their addresses are listed at the end of each chapter.

Acorn wiring

How a SQRL Acorn CLE-215+ (or a LiteFury/NiteFury) is wired to its Raspberry Pi host in the fpgas.online fleet. See SQRL Acorn and LiteFury[1] for the board itself and where each one is deployed.

The wiring depends on the **carrier the Pi sits in** — how many GPIOs it brings out — not on the site:

- **Raspberry Pi 5 with an M.2 HAT**[2]: the full 40-pin header is available. P2 goes to header pins 5-10 and P1 to header pins 19-26. JTAG has its own pins, `--pins 10:9:11:8` (TDI:TDO:TCK:TMS), the serial pair is on GPIO14/15, and both spare balls (J5, H5) are wired.
- **Compute Blade with a CM4 or CM5**: the blade brings out only GPIO2, 3, 4, 14 and 15. P1 goes to the Extension Port and JTAG is `--pins 2:3:4:14`. P2's serial pair goes to the 4-pin UART header, and J5 and H5 are not connected. The UART header's TX pin is the same GPIO14 as TMS, so the J2 wire has a 470 Ω resistor in it so that JTAG should win (designed so, not yet measured). No PS1 blade is wired this way yet; how each one is wired now is on Compute blades[3].

On both carriers the serial pair lands on the same GPIOs — K2 (FPGA TX) on GPIO15, J2 (FPGA RX) on GPIO14 — so one set of FPGA pin constraints and one set of host scripts serves every host.

The wiring sheets and pin tables on this page are generated in fpgas.online-test-designs[4] from a single table, `wiring.toml`, and copied here by `tools/sync_test_designs.py`. To change the wiring, change that table, not this page.

> **WARNING**
>
> **The VCC (3.3 V) wire of both P1 and P2 must never be connected to the host.** Cut it back and insulate it. 3.3 V from the Acorn into the header can damage the Pi.

Bill of Materials

Parts for a Compute Blade

The parts for **one** Compute Blade host; for several hosts, that many of each.

Have it	Qty	Part	Part number	What it is for
☐	1	Compute Blade (Uptime Lab)	—	its M.2 slot takes the card
☐	1	Raspberry Pi Compute Module 4 or 5	—	the host: it sits on the Compute Blade
☐	1	SQRL Acorn CLE-215+, CLE-215 or CLE-101 (the same card is sold as LiteFury and NiteFury)	—	M.2 M-key, 2280
☐	1	Molex Pico-EZmate cable assembly, 6 circuits, a plug at each end	Molex 0369200601	cut in half: one half is the P1 cable, the other the P2 cable. All six wires are black
☐	1	heat-shrink tube, about 2 mm, a few centimetres	—	over each cut-back wire end
☐	1	Dupont housing, 2×5, 2.54 mm pitch	—	over Extension Port pins 1 to 10; 5 of its 10 cavities stay empty
☐	1	Dupont housing, 1×4, 2.54 mm pitch	—	over UART pins 1 to 4; 1 of its 4 cavities stays empty
☐	8	Dupont female crimp terminal, 2.54 mm	—	one for each connected wire: GND, TCK, TDO, TMS, TDI, GND, J2, K2; buy a few more than this, as spares
☐	1	resistor, 470 Ω, 1/8 W axial	—	in series with J2
☐	1	heat-shrink tube, about 3 mm, a few centimetres	—	over the resistor and both joints

The tools to build its two cables, once for any number of hosts:

Have it	Tool	What it is for
☐	multimeter with a continuity buzzer	all six wires of the cable are black: each is checked from the plug to its cavity before the cable is fitted
☐	wire strippers for fine wire, and side cutters	to cut the cable in half, strip about 3 mm from each wire, and cut back the wires that are not connected
☐	crimping tool for 2.54 mm Dupont terminals	one crimp for each connected wire
☐	soldering iron and solder	to join the series resistor into its wire
☐	hot-air tool for the heat-shrink tube	to shrink the tube over each cut-back end (VCC always), and over the series resistor
☐	masking tape and a fine marker pen, or a paint pen	to flag the six wires with their numbers, and to mark the pin 1 corner of each housing

Board connectors

The Acorn has two 6-pin Molex Pico-EZmate connectors. Both are on the **underside** of the card (the face without the FPGA and heatsink), side by side along one long edge at the **fan end**, away from the M.2 edge connector. The silkscreen beside them reads `I/O` / `P2` and `JTAG` / `P1`; P2 is the one nearer the end of the card.

Pin 1 of each connector is the end **nearest the M.2 edge connector**, and it is GND on both:

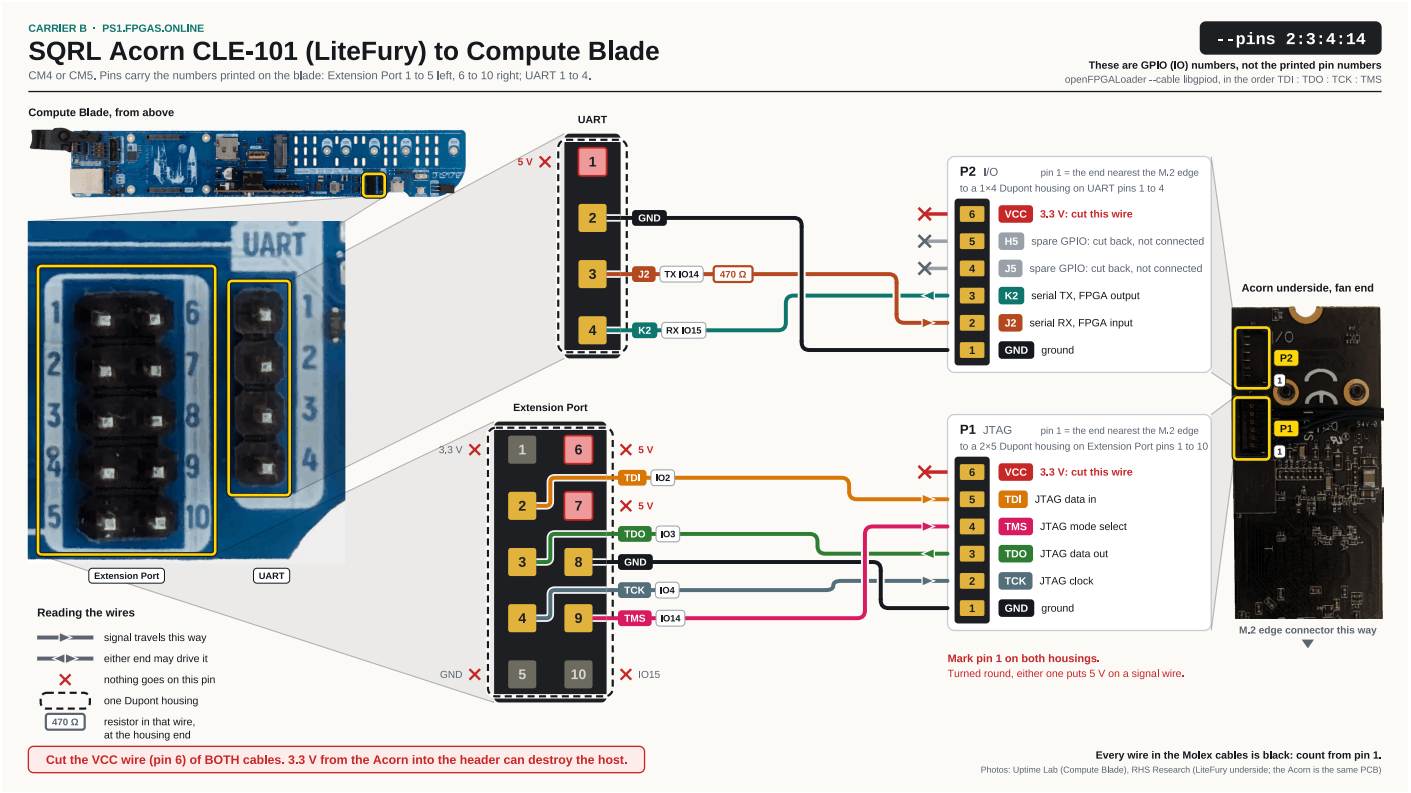
Pin	P2 (I/O)	P1 (JTAG)
1	GND	GND
2	J2 (LVCMOS33): serial RX, FPGA input	TCK: JTAG clock
3	K2 (LVCMOS33): serial TX, FPGA output	TDO: JTAG data out
4	J5 (LVCMOS33): spare GPIO	TMS: JTAG mode select
5	H5 (LVCMOS33): spare GPIO	TDI: JTAG data in
6	VCC: 3.3 V from the Acorn, **cut, never connected**	VCC: 3.3 V from the Acorn, **cut, never connected**

Sources: the “Pico-EZmate connectors pinout on Acorn, Nite/LiteFury” legend on the LiteX Acorn CLE-215 wiki[5], and the silkscreen of RHS Research’s 14-pin JTAG adapter (GND, TCK, TDO, TMS, TDI, VCC). The wiki legend names K2 and J2 from the host’s side (“UART RX (K2)”): K2 is the FPGA’s transmitter. The JTAG order is confirmed in service by pi20 at PS1, whose cable was built from that legend; it has not been checked with a meter on a bare card.

Cut one Pico-EZmate cable (plug at each end) in half to get the P1 and P2 cables, and strip about 3 mm from each wire. **All six wires are black**, so there is no colour to go by: count positions from pin 1, and buzz each wire through before crimping and again once it is in its housing (Building the cables).

Compute Blade

The Compute Blade[6] for a CM4 or CM5 does not have the 40-pin header.



[Acorn to Compute Blade wiring sheet: full size on a landscape sheet of its own, sheet 41.]

Connector	GPIOs	Pins
Extension Port (2×5)	GPIO2, GPIO3, GPIO4, GPIO14, GPIO15	printed 1-10
UART (1×4), beside the Extension Port	GPIO14, GPIO15	printed 1-4
UART Front (3-pin)	GPIO14, GPIO15	TX, RX, GND
Fan Unit (4-pin)	GPIO12, GPIO13	PWM0/UART5-TX, PWM1/UART5-RX

GPIO14 and GPIO15 are the same lines at every connector that carries them: the vendor's GPIO table[7] lists GPIO14 at "Expansion Module Port, UART Front(3pin), UART Back(4pin)" as "UART0 TX", and GPIO15 at the same three as "UART0 RX". The vendor publishes no schematic, so whether anything sits in series between the connectors is not known. GPIO8-11 (SPI0) are not brought out. GPIO2 and GPIO3, which carry TDI and TDO here, are also SDA1 and SCL1; I²C pull-ups on them are expected but have not been measured on a blade.

Pin numbering

The blade prints its own numbers: **1 to 5 down the left column of the Extension Port, 6 to 10 down the right**, and 1 to 4 from the top on the UART header (the vendor's "UART Back"). They are not Raspberry Pi header numbers, although the Extension Port's ten pins are electrically RPi header pins 1-

10 in the same arrangement. This page uses the printed numbers, because those are the ones in front of you at the bench. The legend above the Extension Port is spelled “Extention Port” on the board. TX and RX on the UART header are named from the blade’s side.

The Extension Port (2×5), by printed pin:

Pin	Silkscreen	GPIO	Same pin on a Raspberry Pi header	Same GPIO as	Wire
1	3.3V	—	1	—	empty
2	IO2	GPIO2 (I2C1_SDA)	3	—	P1:5 TDI
3	IO3	GPIO3 (I2C1_SCL)	5	—	P1:3 TDO
4	IO4	GPIO4 (GPCLK0)	7	—	P1:2 TCK
5	GND	—	9	—	empty
6	5V	—	2	—	**empty, 5 V**
7	5V	—	4	—	**empty, 5 V**
8	GND	—	6	—	P1:1 GND
9	IO14	GPIO14 (TXD0)	8	UART 3	P1:4 TMS
10	IO15	GPIO15 (RXD0)	10	UART 4	empty

The UART header (1×4), by printed pin:

Pin	Silkscreen	GPIO	Same GPIO as	Wire
1	5V	—	—	**empty, 5 V**
2	GND	—	—	P2:1 GND
3	TX	GPIO14 (TXD0)	Extension Port 9	P2:2 J2
4	RX	GPIO15 (RXD0)	Extension Port 10	P2:3 K2

The UART header’s 5 V pin can be an input or an output (vendor note), so it is live whenever the blade is powered.

P1: JTAG, on the Extension Port

P1 pin	Signal	→ Extension Port pin	Host name	GPIO	Direction	Series resistor	Same line as
1	GND	8	GND	—	—	—	—
2	TCK: JTAG clock	4	IO4	GPIO4 (GPCLK0)	Pi → FPGA	—	—
3	TDO: JTAG data out	3	IO3	GPIO3 (I2C1_SCL)	FPGA → Pi	—	—
4	TMS: JTAG mode select	9	IO14	GPIO14 (TXD0)	Pi → FPGA	—	J2 (UART 3)
5	TDI: JTAG data in	2	IO2	GPIO2 (I2C1_SDA)	Pi → FPGA	—	—
6	VCC: 3.3 V from the Acorn, **cut, never connected**	**cut, never connected**	—	—	—	—	—

P2: serial pair, on the UART header

P2 pin	Signal	→ UART pin	Host name	GPIO	Direction	Series resistor	Same line as
1	GND	2	GND	—	—	—	—
2	J2 (LVCMOS33): serial RX, FPGA input	3	TX	GPIO14 (TXD0)	Pi → FPGA	470 Ω	TMS (Extension Port 9)
3	K2 (LVCMOS33): serial TX, FPGA output	4	RX	GPIO15 (RXD0)	FPGA → Pi	—	—
4	J5 (LVCMOS33): spare GPIO	**not connected**: cut back and insulate	—	—	—	—	—
5	H5 (LVCMOS33): spare GPIO	**not connected**: cut back and insulate	—	—	—	—	—
6	VCC: 3.3 V from the Acorn, **cut, never connected**	**cut, never connected**	—	—	—	—	—

Cut the J5 and H5 wires back and insulate them like VCC: the blade’s five GPIOs are JTAG’s four plus the serial pair’s second line, so none is left for them. The fpgas.online Acorn design resets J5 and H5 to inputs, so the open ends do no harm.

Housings

P1 goes to a 2×5 Dupont housing over the whole Extension Port, and P2 to a 1×4 over the whole UART header. The unused cavities stay empty: Extension Port 1, 5, 6, 7 and 10, and UART 1. No cavity holds two wires.

Use full-length housings, not the shortest that holds the wires. A 2×3 on Extension Port rows 2-4 also fits one row toward pin 1, which puts the GND wire on pin 7 (5 V); a 1×3 on UART pins 2-4 also fits one pin toward pin 1, which puts GND on UART pin 1 (5 V). Either shorts the blade's 5 V rail, because the Acorn's ground is the blade's ground through the M.2 slot. A full-length housing has only one position, but it can still go on turned round — the 2×5 then puts TCK on pin 7, the 1×4 puts K2 on pin 1, both 5 V — so **mark pin 1 on each housing** and match it to printed pin 1.

> **WARNING**
>
> **Extension Port pins 6 and 7 and UART pin 1 are 5 V and sit inside a housing. Their cavities must stay empty.**

The shared line and the 470 Ω resistor

UART pin 3 *is* GPIO14, which is also Extension Port pin 9, where P1 puts TMS. The blade brings out five GPIOs and JTAG needs four, so one JTAG signal has to share a line with the serial pair: J2 and TMS share GPIO14.

The TMS wire goes straight to its pin. The J2 wire reaches the same line through a 470 Ω resistor fitted at the housing end of the wire. Whatever a loaded design does with J2, TMS should still get through: the worst case is an FPGA output fighting the JTAG driver through 470 Ω, 3.3 V / 470 Ω ≈ 7 mA, which the Artix-7 I/O and the Pi's GPIO should tolerate, with the direct driver winning the level (designed so, not measured: see the note below). Without the resistor the same fight is a short between two outputs, which crashes the host (see the warning under P2[8]). K2 needs no resistor: GPIO15 is not a JTAG pin on this carrier.

The serial port and JTAG cannot both have GPIO14 in one boot of a host on kernel 6.18: with the header's serial port on (`enable_uart=1`) the kernel's serial driver holds GPIO14 and JTAG cannot run, and with it off there is no `/dev/ttyAMA0` for the serial pair (see JTAG on a blade). Under kernel 6.12.75 pi20 ran JTAG and then used `/dev/ttyAMA0` in the same boot.

> **NOTE**
>
> 470 Ω is a chosen value, not a measured one: it limits the current to about 7 mA if the FPGA drives J2 against the Pi at 3.3 V. Whether `--detect` answers while a design drives J2, and whether `/dev/ttyAMA0` still transmits through the resistor, is **not yet run by us on this hardware**: no blade is wired this way yet. Check both on the first blade that is.

> **WARNING**
>
> **On a cable without the J2 resistor, a design that drives J2 costs you JTAG** (test-designs issue #4[9] item 1): GPIO14 is TMS, and once the FPGA drives it `openFPGALoader` cannot. The pin-ID design drives every P2 ball, so it does this every time. The way back is a PoE cycle of the blade's switch port, which restores everything in about 60 s: the flash bitstream reloads and `--detect`, the DNA read and the PCIe endpoint all come back. See PoE power control[10] and, for the PS1 blades, Power control[11]. Which blades have the resistor is on Compute blades[3].

JTAG on a blade

WARNING

JTAG needs the header's serial port off at boot. TMS is GPIO14, the serial port's TX pin. With `enable_uart=1` the kernel's serial driver holds GPIO14 whether or not a console or a getty uses the port, and kernel 6.18 does not hand a held pin to openFPGALoader, which then stops with `gpiod_line_request_set_values_subset: Assertion 'request' failed` (read on pi16, a CM5, on 2026-10-05: pi16 on 5 October 2026[12]). The serial port off means `enable_uart=0`, no `dtoverlay=uart0…` or `dtparam=uart0` line, and no `console=serial0` word on the kernel command line, which is expected to free GPIO14; in that boot there is no serial pair to test. **Booting a blade that way is not yet run by us on this hardware**, and nothing has been read on a CM4. pi20's JTAG answered under kernel 6.12.75 with the serial port on.

WARNING

Detach the PCIe endpoint before loading a bitstream, as on a Pi 5. The bus address differs per blade: `0000:01:00.0` on the CM4 blade pi14, `0001:01:00.0` on the CM5 blades pi16 and pi20. Take it from `lspci` or from Compute blades[3], and put it in `BDF` for the commands on this page:

```
$ BDF=0001:01:00.0           # a CM5 blade; 0000:01:00.0 on a CM4 blade
$ echo 1 | sudo tee /sys/bus/pci/devices/$BDF/remove
```

```
# Compute Blade JTAG pin order: TDI(GPIO2):TDO(GPIO3):TCK(GPIO4):TMS(GPIO14)
$ openFPGALoader --cable libgpiod --pins 2:3:4:14 --detect
$ openFPGALoader --cable libgpiod --pins 2:3:4:14 <bitstream.bit>
```

Which file `<bitstream.bit>` is, and where it comes from, is under The designs these steps load.

The `libgpiod` cable opens `/dev/gpiochip0`. On pi16 and pi20 (CM5s, kernel 6.18.50, 2026-10-05) `gpiodetect` listed the header's chip, `pinctrl-rp1`, as `gpiochip0` already, so no link was needed; do not copy the Pi 5's `gpiochip15` link. On any other blade run `gpiodetect` first: the header's chip is the one labelled `pinctrl-rp1` on a CM5 and `pinctrl-bcm2711` on a CM4 (not read by us on a CM4).

openFPGALoader 0.13.1 left GPIO2 and GPIO4 as **outputs** driving low when it exited on pi16 (2026-10-05); it drives TMS on GPIO14 too, so treat that as left an output as well (not observed: the run on pi16 stopped before it had GPIO14). Put them back before anything else uses the lines. On pi16 `pinctrl set 2,4 no pu` restored GPIO2 and GPIO4 to what they were before.

```
$ pinctrl set 2,4 no pu      # JTAG pins back to unconfigured with a pull-up
```

Give GPIO14 and GPIO15 back to the serial port only when the loaded design treats J2 as an input (the fpgas.online Acorn design does; pin-ID does not: see the warning under P2[8]). The UART function is a different alternate on each module, so run only the line for yours; `pinctrl funcs 14,15` lists them. This applies only to a boot in which the serial port is on (kernel 6.12.75, as on pi20). In a kernel 6.18 boot with the serial port off there is no `/dev/ttyAMA0` and nothing to give back.

On a **CM4** (BCM2711):

```
$ pinctrl set 14,15 a0       # GPIO14 = TXD0, GPIO15 = RXD0
```

On a **CM5** (RP1):

```
$ pinctrl set 14,15 a4       # GPIO14 = TXD0, GPIO15 = RXD0
```

Building the cables

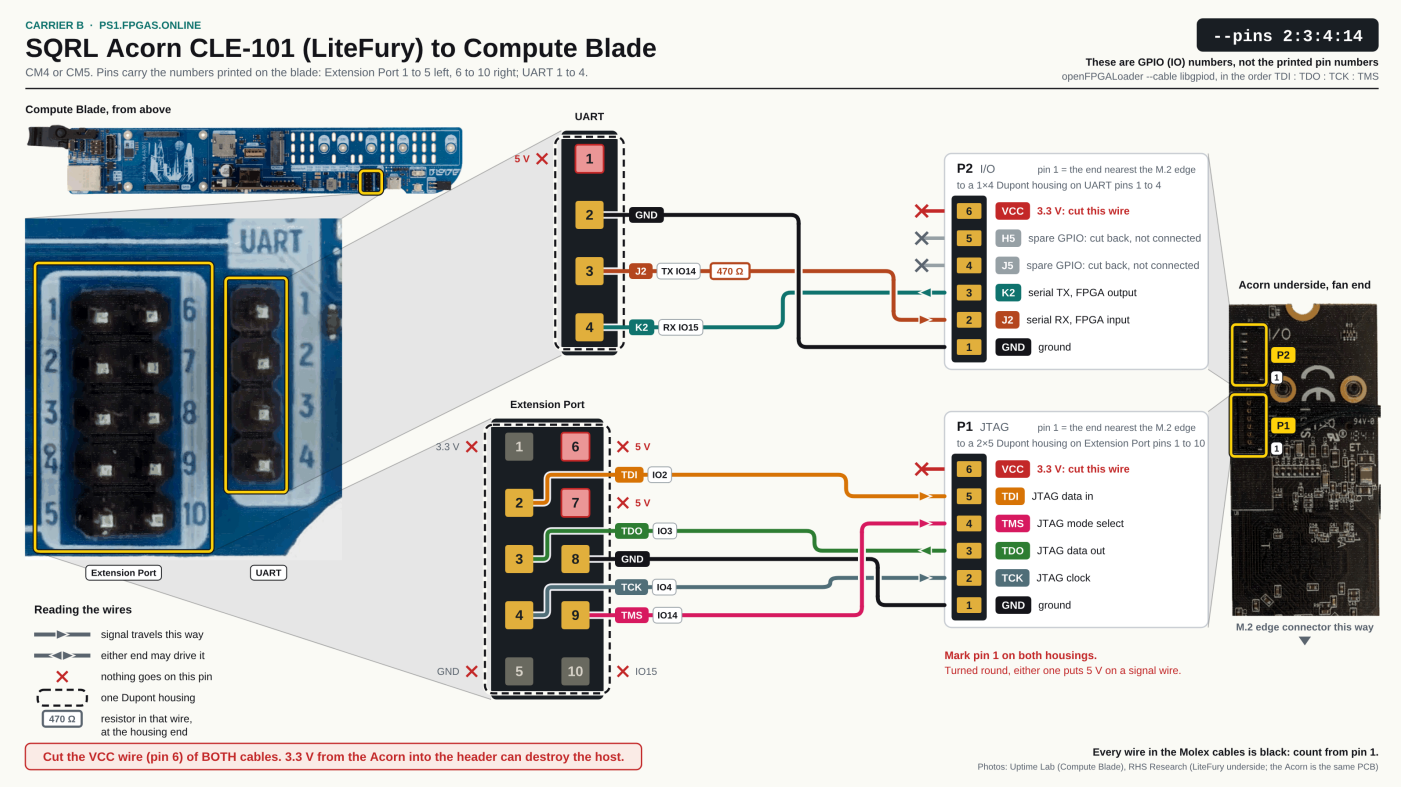
Cables for a Compute Blade

Not yet run by us on this hardware: written from the design.

What you will have

Two short cables from the Acorn's two connectors to the Compute Blade: the P1 cable carries JTAG, the P2 cable carries the serial port.

GND, TCK, TDO, TMS, TDI, J2 and K2 are the names on the pictures for each wire; J2 and K2 are the FPGA's pin names.



[The finished wiring: Acorn to Compute Blade: full size on a landscape sheet of its own, sheet 41.]

Parts and tools

The parts for **one** Compute Blade host; for several hosts, that many of each.

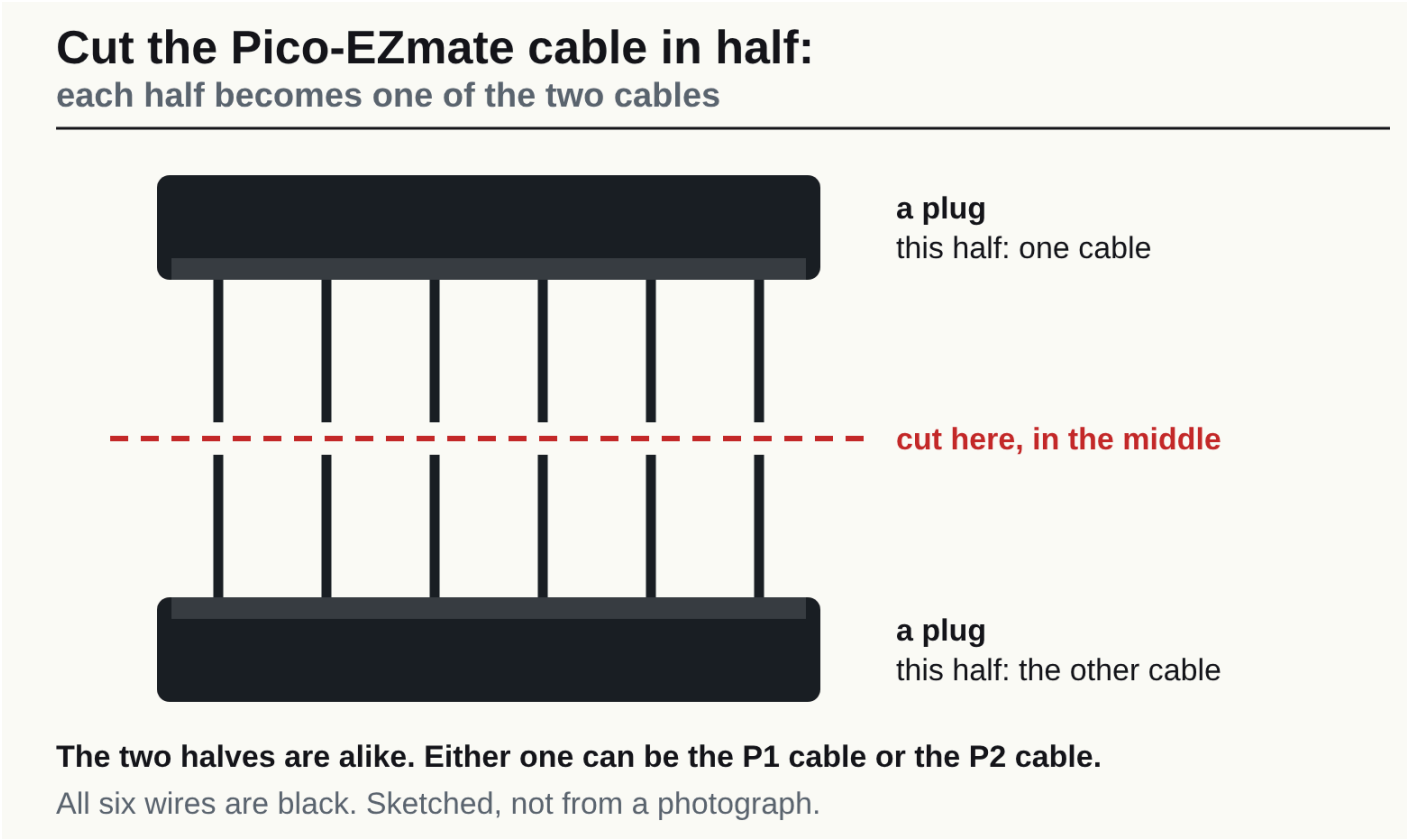
Have it	Qty	Part	Part number	What it is for
☐	1	Compute Blade (Uptime Lab)	—	its M.2 slot takes the card
☐	1	Raspberry Pi Compute Module 4 or 5	—	the host: it sits on the Compute Blade
☐	1	SQRL Acorn CLE-215+, CLE-215 or CLE-101 (the same card is sold as LiteFury and NiteFury)	—	M.2 M-key, 2280
☐	1	Molex Pico-EZmate cable assembly, 6 circuits, a plug at each end	Molex 0369200601	cut in half: one half is the P1 cable, the other the P2 cable. All six wires are black
☐	1	heat-shrink tube, about 2 mm, a few centimetres	—	over each cut-back wire end
☐	1	Dupont housing, 2×5, 2.54 mm pitch	—	over Extension Port pins 1 to 10; 5 of its 10 cavities stay empty
☐	1	Dupont housing, 1×4, 2.54 mm pitch	—	over UART pins 1 to 4; 1 of its 4 cavities stays empty
☐	8	Dupont female crimp terminal, 2.54 mm	—	one for each connected wire: GND, TCK, TDO, TMS, TDI, GND, J2, K2; buy a few more than this, as spares
☐	1	resistor, 470 Ω, 1/8 W axial	—	in series with J2
☐	1	heat-shrink tube, about 3 mm, a few centimetres	—	over the resistor and both joints

The tools to build its two cables, once for any number of hosts:

Have it	Tool	What it is for
☐	multimeter with a continuity buzzer	all six wires of the cable are black: each is checked from the plug to its cavity before the cable is fitted
☐	wire strippers for fine wire, and side cutters	to cut the cable in half, strip about 3 mm from each wire, and cut back the wires that are not connected
☐	crimping tool for 2.54 mm Dupont terminals	one crimp for each connected wire
☐	soldering iron and solder	to join the series resistor into its wire
☐	hot-air tool for the heat-shrink tube	to shrink the tube over each cut-back end (VCC always), and over the series resistor
☐	masking tape and a fine marker pen, or a paint pen	to flag the six wires with their numbers, and to mark the pin 1 corner of each housing

Steps

1. Cut the Molex cable in half with side cutters. Each half is one cable.



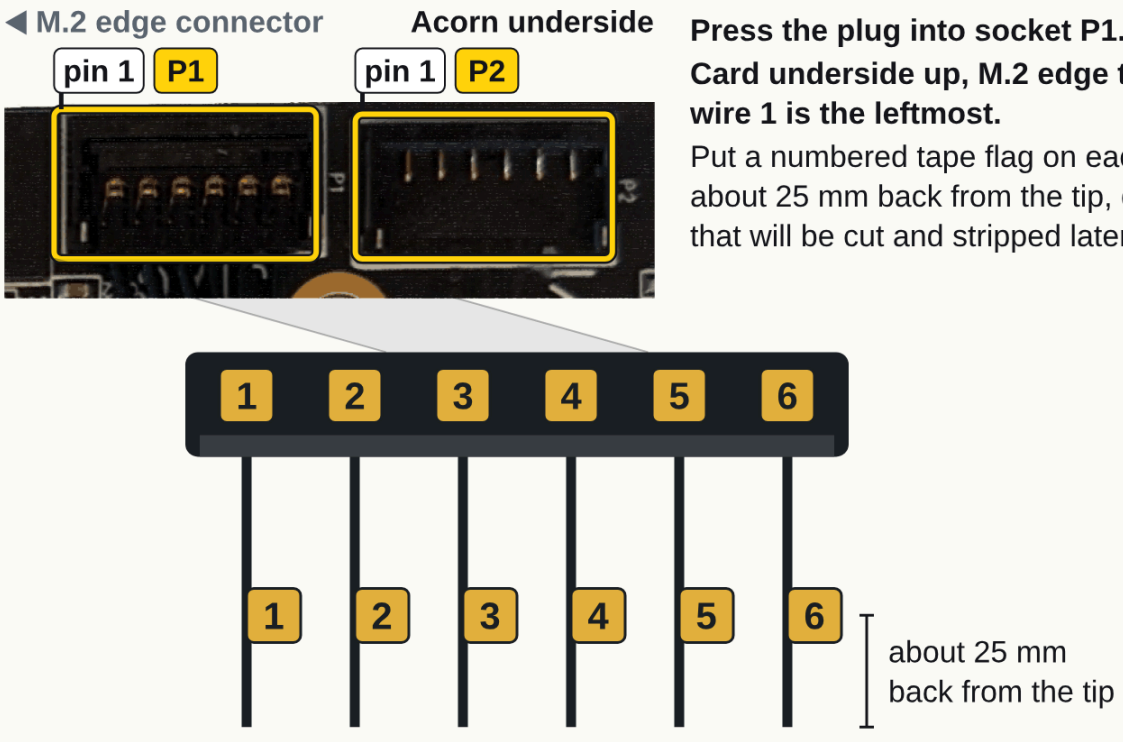
The P1 cable (JTAG)

2. Find wire 1 of the P1 cable and flag the wires, before cutting any wire back.

1. Press the plug of one half into socket P1 on the underside of the Acorn. Hold the card underside up with the M.2 edge to your left: wire 1 is the leftmost.
2. Put a numbered tape flag on each of the 6 wires, 1 at the left to 6, about 25 mm back from the tip, clear of the end that will be cut and stripped later.
3. With the plug still in the socket (the Acorn out of any slot, unpowered), set the meter to continuity. Press the probe tip, or a pin held to it, against the cut face of the wire flagged 1 (it is not stripped yet), and put the other probe on the plated half-round mounting pad at the end of the card: it must beep.
4. Do the same with the wire flagged 6: it must stay silent.
5. If wire 6 beeps instead, stop: the numbering is reversed; take the flags off and number from the other end. If neither wire beeps, strip about 3 mm from wires 1 and 6 and try again; if still neither beeps, stop: the ground point is not confirmed.
6. Take the plug out again.

P1 cable (JTAG): flag the 6 wires

nothing is cut in this step



Press the plug into socket P1.

Card underside up, M.2 edge to your left: wire 1 is the leftmost.

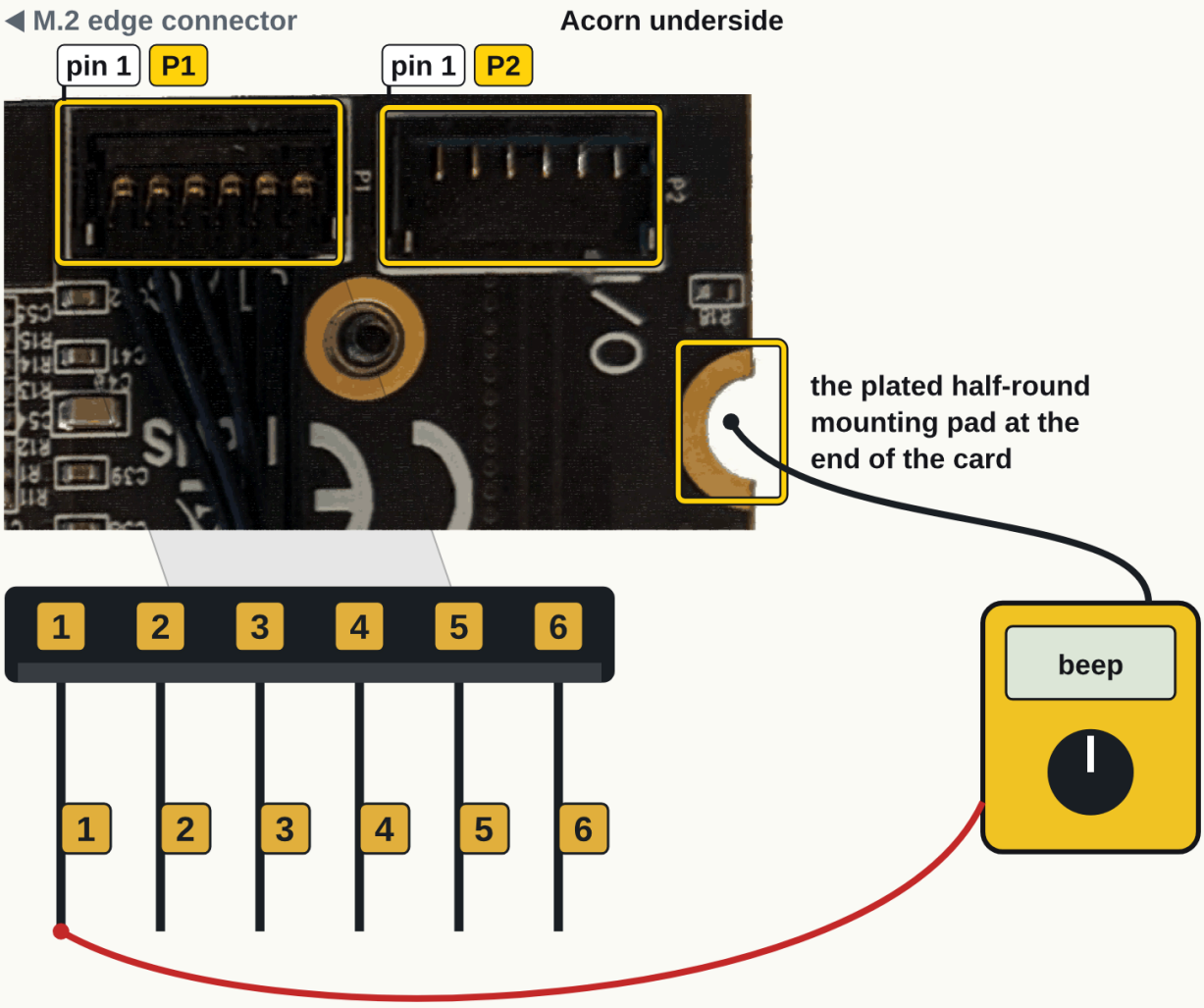
Put a numbered tape flag on each wire, 1 to 6, about 25 mm back from the tip, clear of the end that will be cut and stripped later.

All 6 wires stay whole in this step. Next: the meter check of wire 1.

All 6 wires are black. The plug and the flags are sketched.

Check which wire is wire 1, before cutting any wire back

the plug in socket P1, the Acorn out of any slot



Press the probe tip, or a pin held to it, against the cut face of the wire flagged 1; the other probe on the mounting pad: the meter must beep. Wire flagged 6: it must stay silent.

If wire 6 beeps instead, the numbering is reversed: take the flags off and number from the other end.

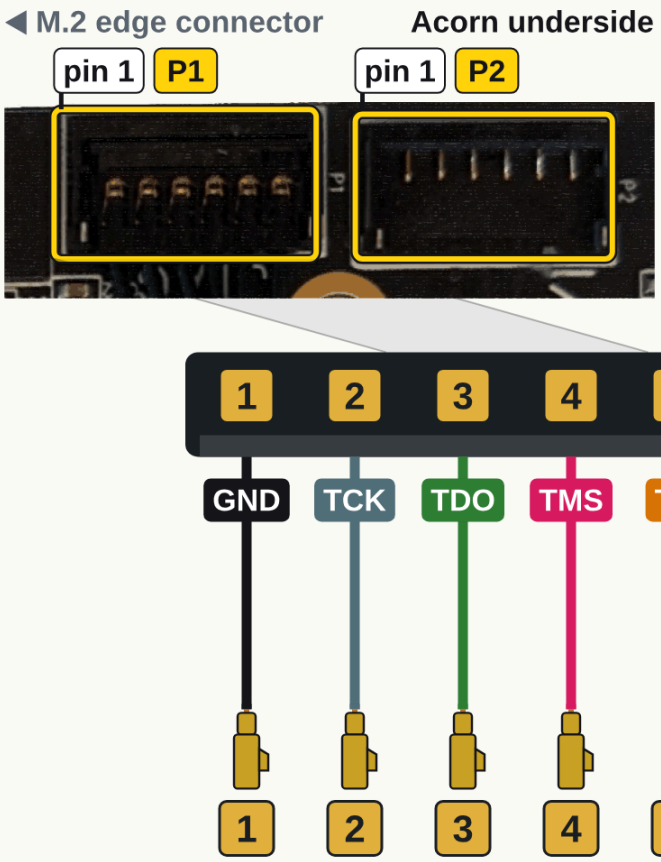
If neither wire beeps, strip about 3 mm from wires 1 and 6 and try again; if still neither beeps, stop: the ground point is not confirmed.

The pad is taken to be ground from the M.2 standard: not measured on this card. The plug, the flags and the meter are sketched.

3. Cut wire 6 off about 10 mm from the plug and shrink a piece of the 2 mm tube over the cut end. It goes in no cavity. Wire 6 is VCC, 3.3 V from the Acorn: it must never reach the host. Leave wires 1 to 5 at full length.

P1 cable (JTAG) for a Compute Blade:

which wires are cut back, which get a terminal



The plug is drawn under socket P1 to show which wire is which.

Card underside up, M.2 edge to your left: wire 1 is the leftmost.

Each wire keeps the flag you gave it in the step before.

The plug, seen from above as it sits in the socket.
Acorn turned underside up, M.2 edge to your left.

wire 6: cut off about 10 mm from the plug, heat shrink over the end
3.3 V from the Acorn: it must never reach the host.

Wires 1 to 5: leave at full length, strip about 3 mm, crimp a terminal on each.

Wire colours are for this picture only: the real wires are all black. Count from the plug's pin 1 end.

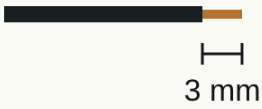
The plug and the terminals are sketched, not from a photograph.

4. Strip about 3 mm from wires 1 to 5. Crimp a Dupont terminal on each.

Crimp a terminal on each wire

Sketched, not from a photograph.

1. Strip about 3 mm.

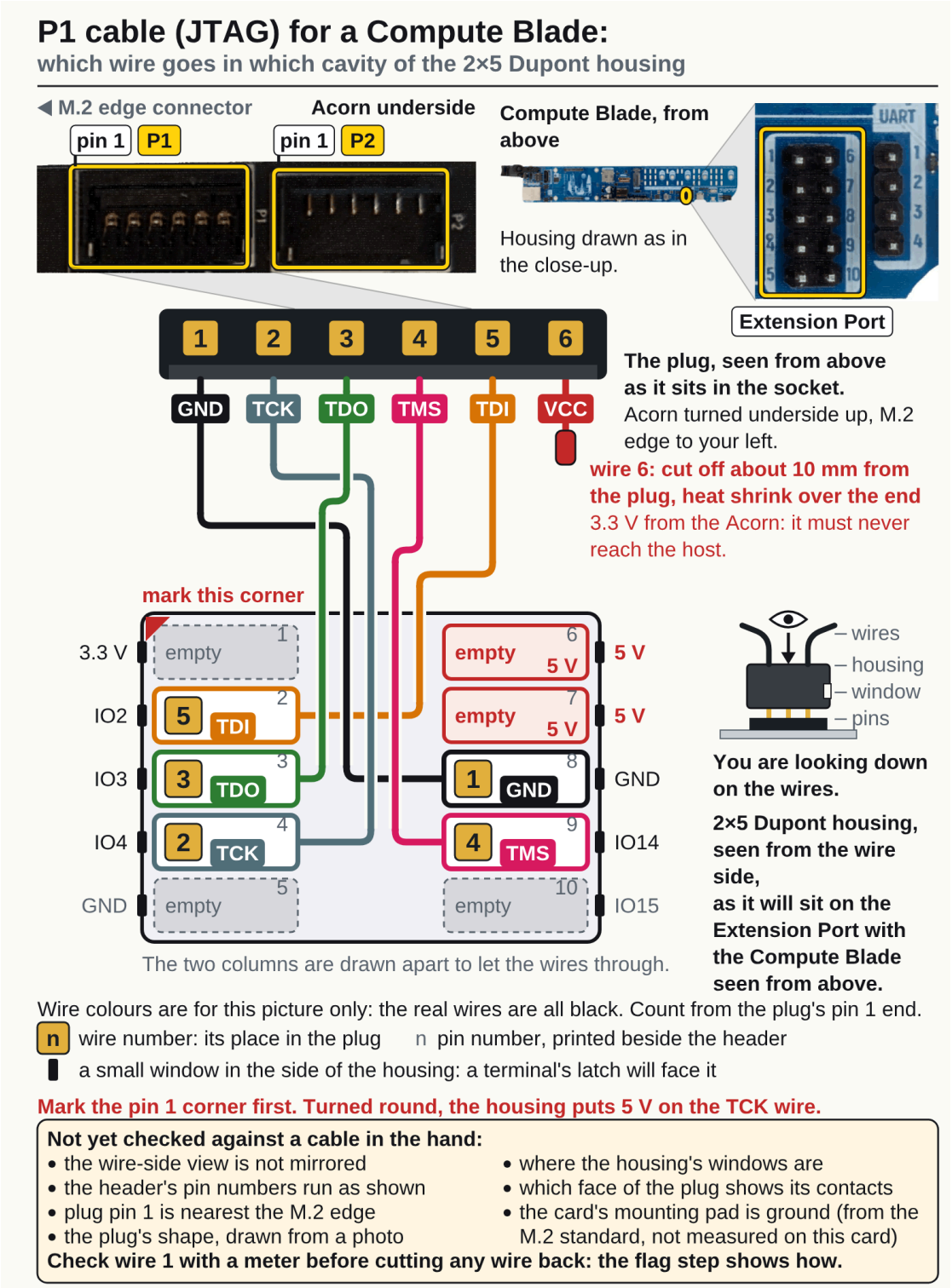


2. Crimp the terminal on.



One crimp grips the bare wire,
the other the insulation.

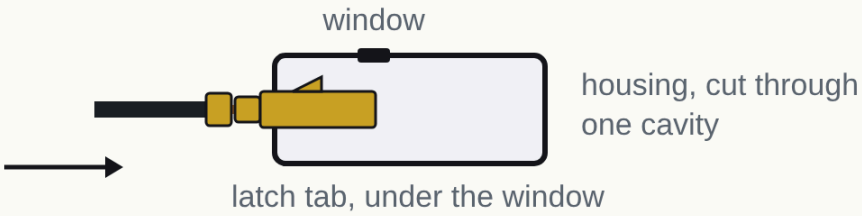
5. Hold the empty 2×5 housing with the wire openings facing you and its long side upright, as in the picture. Until it is marked, either way up is the same. Mark the top left corner with a paint pen or a dot of tape: that is the pin 1 corner. For each wire, read the number on its flag, find the same number in the picture, and push its terminal into that cavity, latch tab towards the window, until it clicks. Turned round, the housing puts 5 V on the TCK wire, which can destroy the host. Wire 6 goes in no cavity. Pull each wire gently: the terminal must stay in.



Push each terminal into its cavity

Sketched, not from a photograph.

1. Push it in, latch tab towards the window, until it clicks.



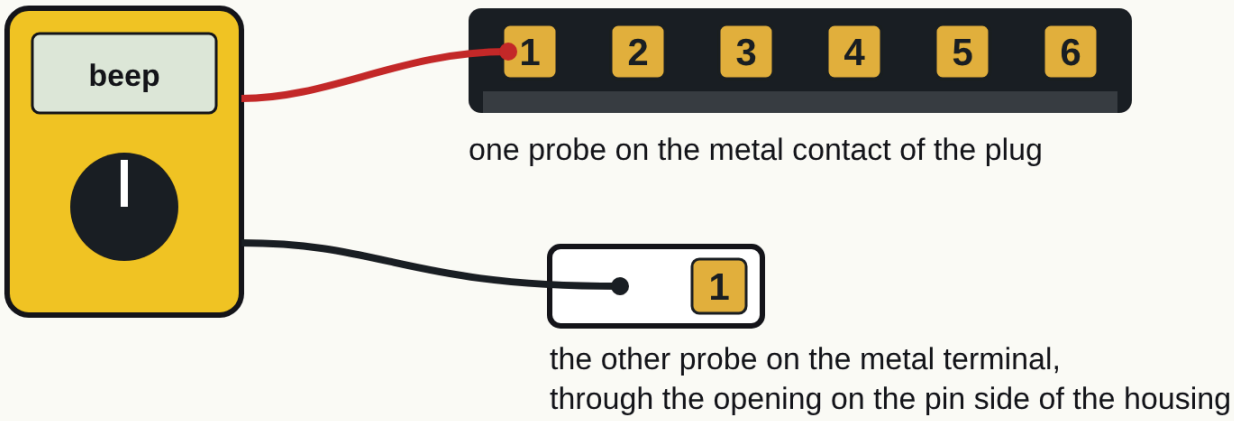
2. Pull the wire gently. The terminal must stay in.

- the window, as marked on the housing in the cavity picture

6. Check each wire with a meter on continuity. For each wire: one probe on its contact on the plug, the other on the terminal in the cavity the picture gives for that wire number, through the opening on the pin side of the housing: it must beep. Every other cavity must stay silent for that contact. The plug’s contacts are 1.2 mm apart: use a fine probe or a sewing pin held to the probe.

Check every wire with a meter

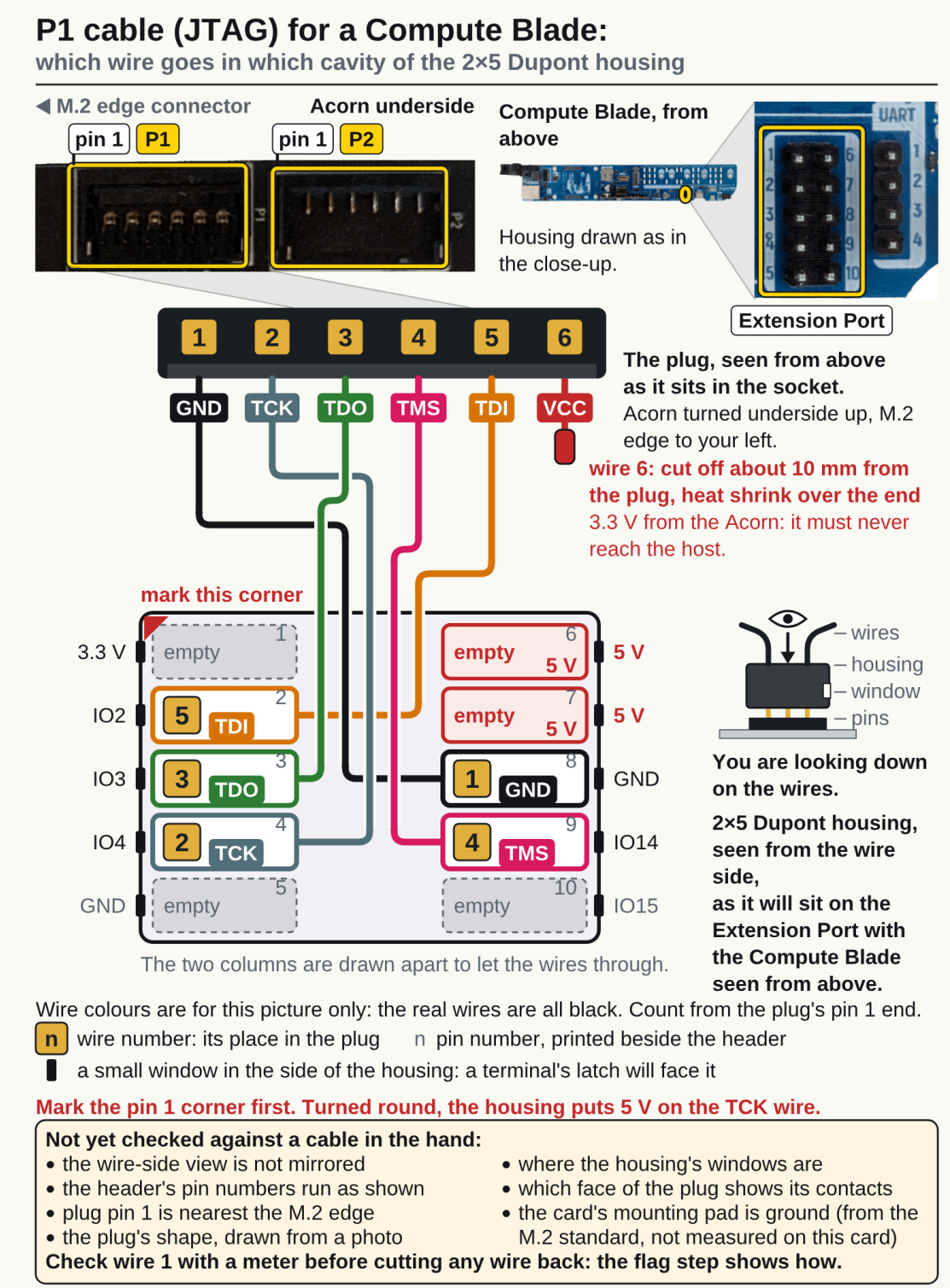
Sketched, not from a photograph.



Set the meter to continuity. For each wire, touch its contact on the plug and its terminal in the housing: the meter must beep.

Then try every other cavity: silent.

The P1 cavity picture again, to read each wire’s cavity from:



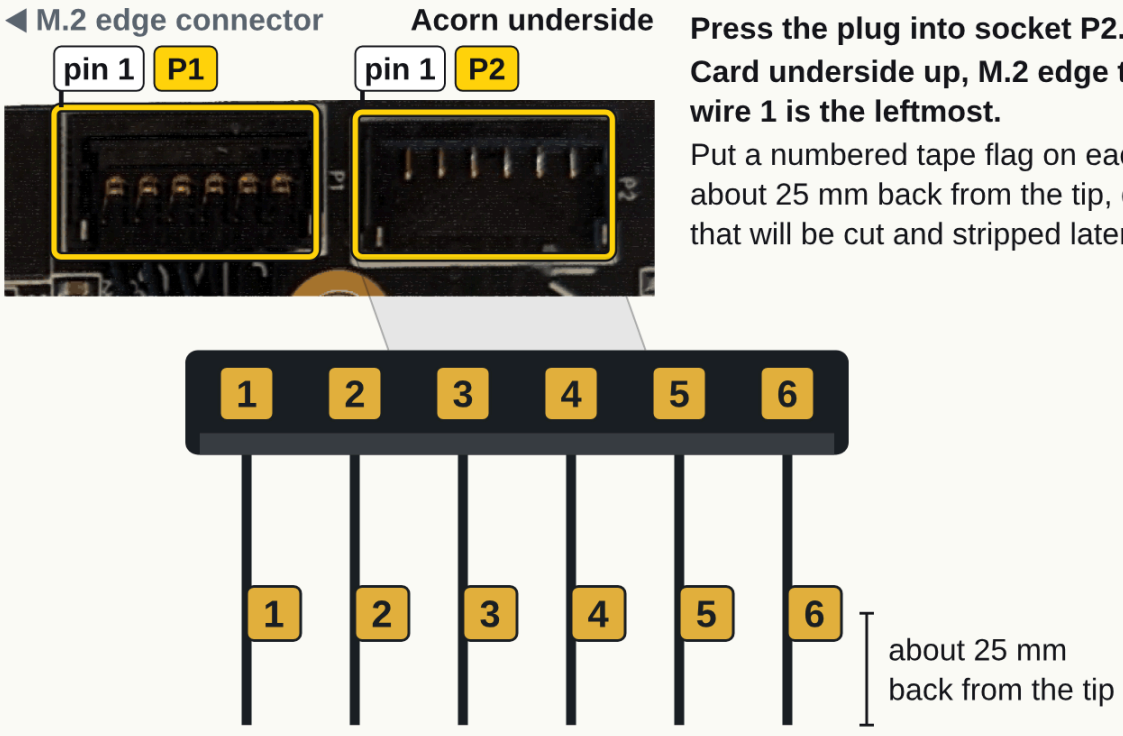
The P2 cable (I/O)

7. Find wire 1 of the P2 cable and flag the wires, before cutting any wire back.

1. Press the plug of the other half into socket P2 on the underside of the Acorn. Hold the card underside up with the M.2 edge to your left: wire 1 is the leftmost.
2. Put a numbered tape flag on each of the 6 wires, 1 at the left to 6, about 25 mm back from the tip, clear of the end that will be cut and stripped later.
3. With the plug still in the socket (the Acorn out of any slot, unpowered), set the meter to continuity. Press the probe tip, or a pin held to it, against the cut face of the wire flagged 1 (it is not stripped yet), and put the other probe on the plated half-round mounting pad at the end of the card: it must beep.
4. Do the same with the wire flagged 6: it must stay silent.
5. If wire 6 beeps instead, stop: the numbering is reversed; take the flags off and number from the other end. If neither wire beeps, strip about 3 mm from wires 1 and 6 and try again; if still neither beeps, stop: the ground point is not confirmed.
6. Take the plug out again.

P2 cable (I/O): flag the 6 wires

nothing is cut in this step



Press the plug into socket P2.

Card underside up, M.2 edge to your left: wire 1 is the leftmost.

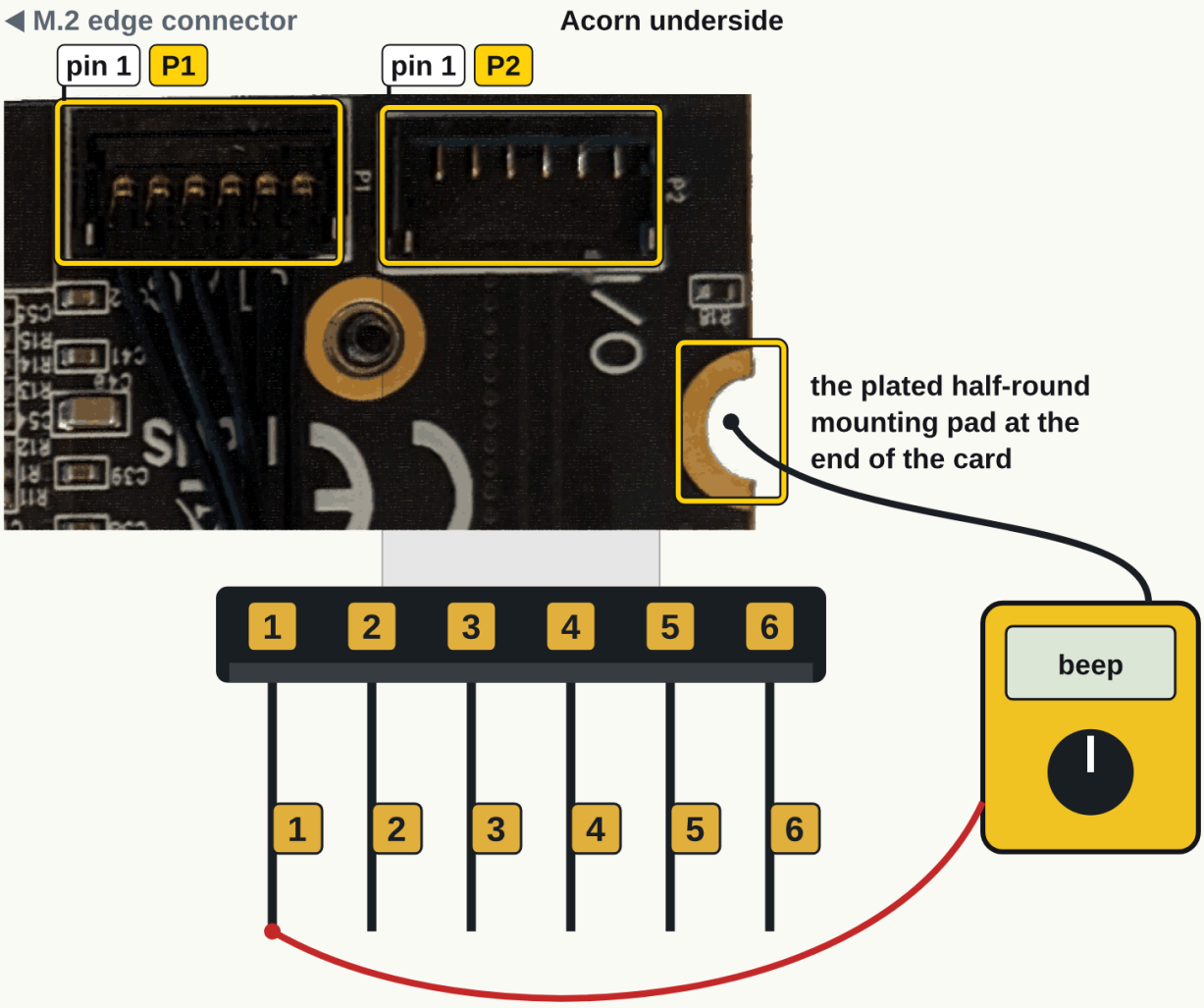
Put a numbered tape flag on each wire, 1 to 6, about 25 mm back from the tip, clear of the end that will be cut and stripped later.

All 6 wires stay whole in this step. Next: the meter check of wire 1.

All 6 wires are black. The plug and the flags are sketched.

Check which wire is wire 1, before cutting any wire back

the plug in socket P2, the Acorn out of any slot



Press the probe tip, or a pin held to it, against the cut face of the wire flagged 1; the other probe on the mounting pad: the meter must beep. Wire flagged 6: it must stay silent.

If wire 6 beeps instead, the numbering is reversed: take the flags off and number from the other end.

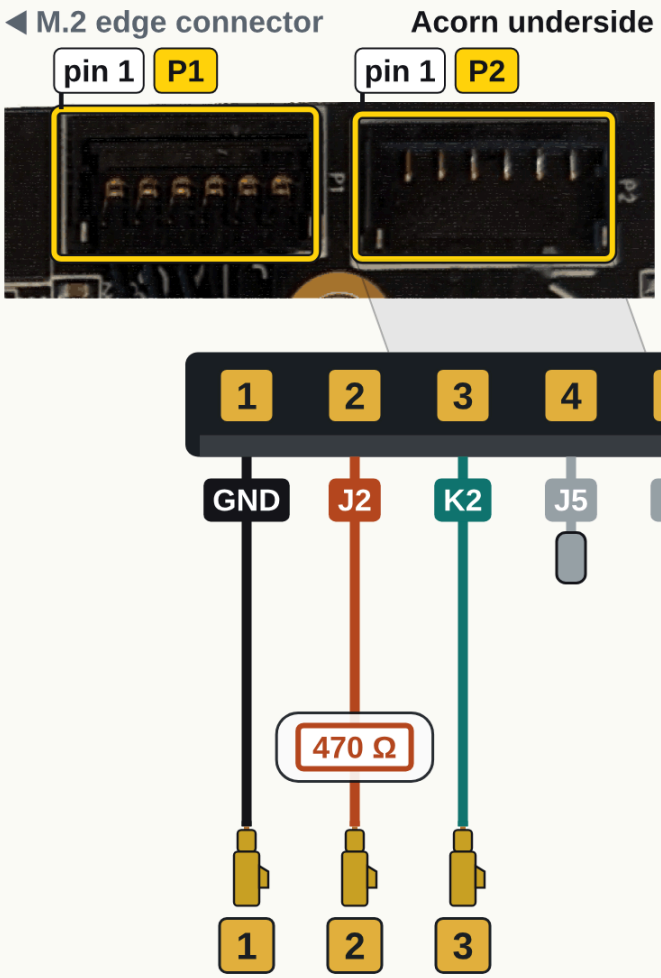
If neither wire beeps, strip about 3 mm from wires 1 and 6 and try again; if still neither beeps, stop: the ground point is not confirmed.

The pad is taken to be ground from the M.2 standard: not measured on this card. The plug, the flags and the meter are sketched.

8. Cut wires 4, 5 and 6 off about 10 mm from the plug and shrink a piece of the 2 mm tube over each cut end. They go in no cavity. Wire 6 is VCC, 3.3 V from the Acorn: it must never reach the host. Leave wires 1, 2 and 3 at full length.

P2 cable (I/O) for a Compute Blade:

which wires are cut back, which get a terminal



The plug is drawn under socket P2 to show which wire is which.

Card underside up, M.2 edge to your left: wire 1 is the leftmost.

Each wire keeps the flag you gave it in the step before.

The plug, seen from above as it sits in the socket.
Acorn turned underside up, M.2 edge to your left.

wire 6: cut off about 10 mm from the plug, heat shrink over the end
3.3 V from the Acorn: it must never reach the host.

wires 4 and 5: cut off about 10 mm from the plug, heat shrink over the end
spare GPIO: not used on a Compute Blade.

Wires 1, 2 and 3: leave at full length, strip about 3 mm, crimp a terminal on each.

Wire 2 (J2): the 470 Ω resistor, in heat shrink, is soldered in before its terminal is crimped.

Wire colours are for this picture only: the real wires are all black. Count from the plug's pin 1 end.

The plug and the terminals are sketched, not from a photograph.

9. Cut wire 2 (J2) about 15 mm from its free end. Slide a piece of the 3 mm tube onto the wire, clear of the cut. Strip about 3 mm from each cut end. Trim the resistor’s leads to about 5 mm each. Solder the 470 Ω resistor between the two cut ends. Slide the tube over the resistor and both joints and shrink it. Crimp the terminal only after this.

The 470 Ω resistor in wire 2 (J2) of the P2 cable

Sketched, not from a photograph.

1. Cut wire 2 about 15 mm from its free end.



2. Slide a piece of the 3 mm tube onto the wire, clear of the cut.



3. Strip 3 mm from each cut end. Trim the resistor's leads to 5 mm each. Solder it in.



4. Slide the tube over the resistor and both joints and shrink it.



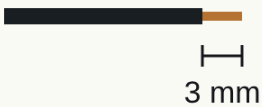
Crimp the terminal on this wire only after this.

10. Strip about 3 mm from wires 1, 2 and 3. Crimp a Dupont terminal on each.

Crimp a terminal on each wire

Sketched, not from a photograph.

1. Strip about 3 mm.

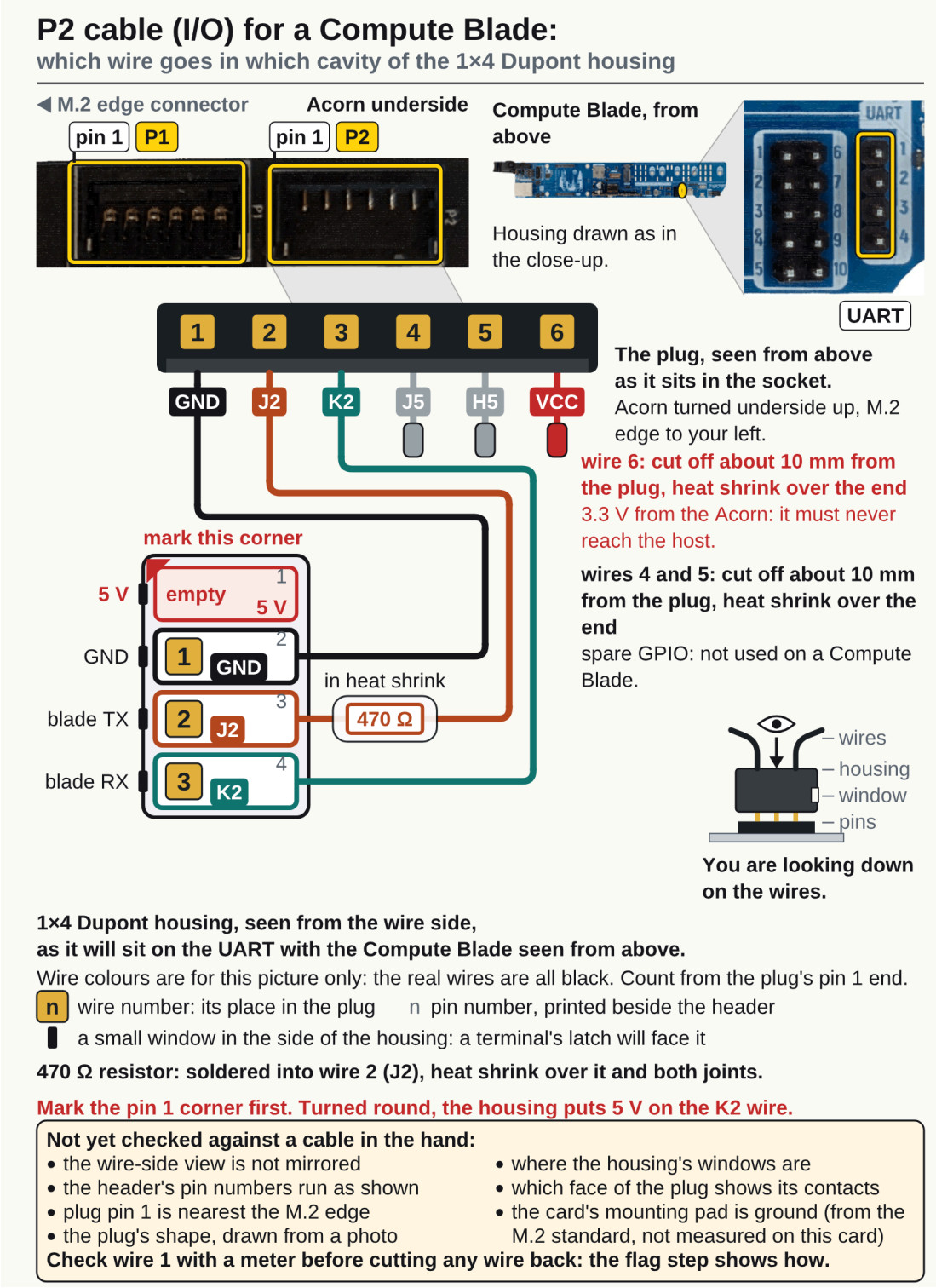


2. Crimp the terminal on.



One crimp grips the bare wire, the other the insulation.

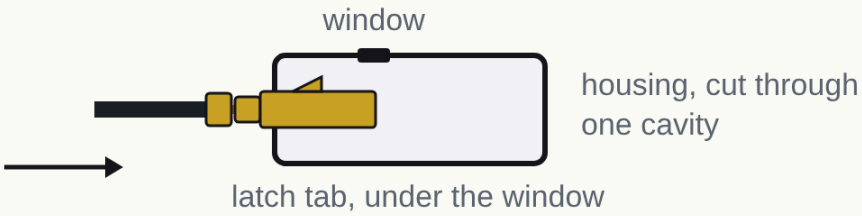
11. Hold the empty 1×4 housing with the wire openings facing you and its long side upright, as in the picture. Until it is marked, either way up is the same. Mark the top end with a paint pen or a dot of tape: that is the pin 1 corner. For each wire, read the number on its flag, find the same number in the picture, and push its terminal into that cavity, latch tab towards the window, until it clicks. Turned round, the housing puts 5 V on the K2 wire, which can destroy the host. Wires 4, 5 and 6 go in no cavity. Pull each wire gently: the terminal must stay in.



Push each terminal into its cavity

Sketched, not from a photograph.

1. Push it in, latch tab towards the window, until it clicks.



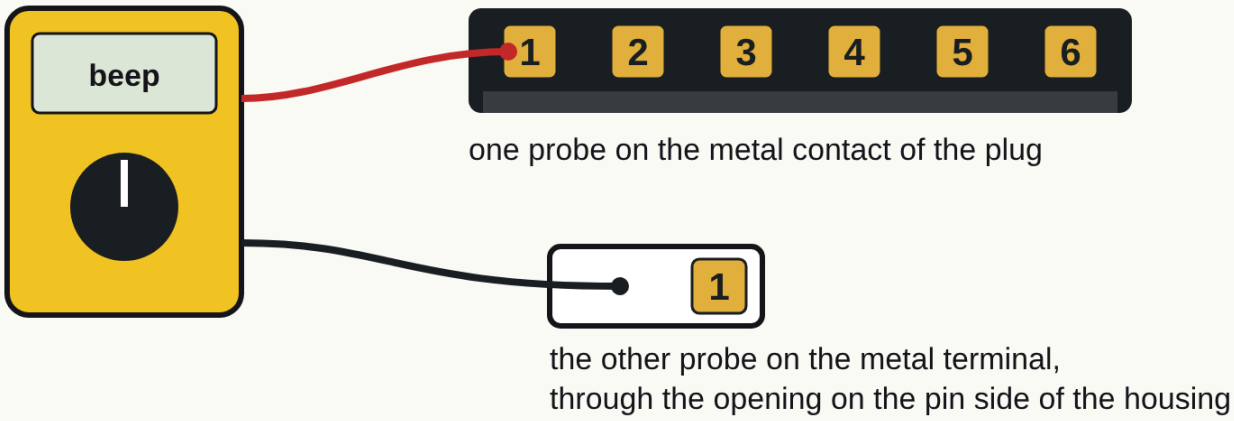
2. Pull the wire gently. The terminal must stay in.

- the window, as marked on the housing in the cavity picture

12. Check each wire with a meter on continuity. For each wire: one probe on its contact on the plug, the other on the terminal in the cavity the picture gives for that wire number, through the opening on the pin side of the housing: it must beep. Every other cavity must stay silent for that contact. The plug's contacts are 1.2 mm apart: use a fine probe or a sewing pin held to the probe.

Check every wire with a meter

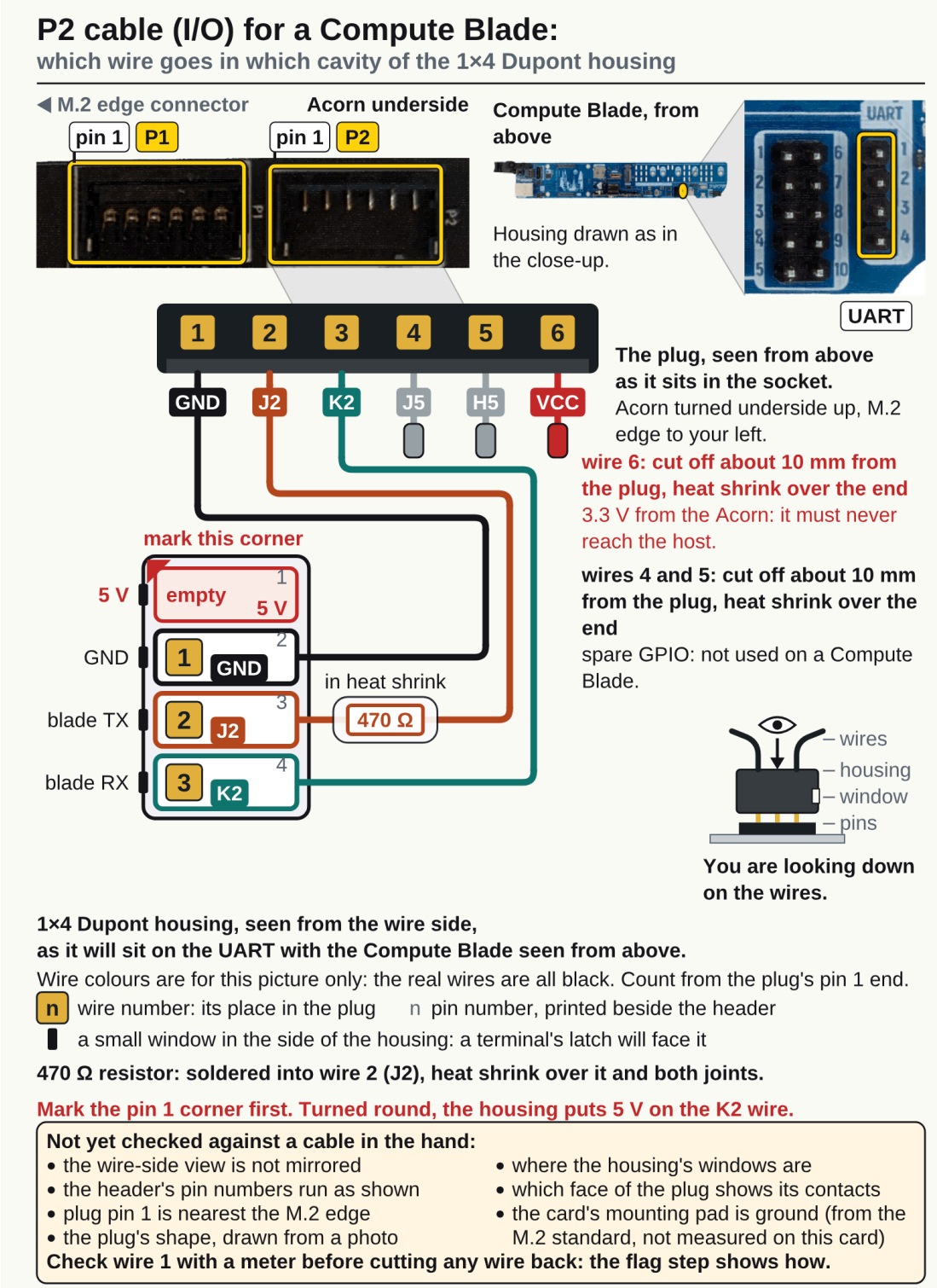
Sketched, not from a photograph.



Set the meter to continuity. For each wire, touch its contact on the plug and its terminal in the housing: the meter must beep.

Then try every other cavity: silent.

The P2 cavity picture again, to read each wire’s cavity from:

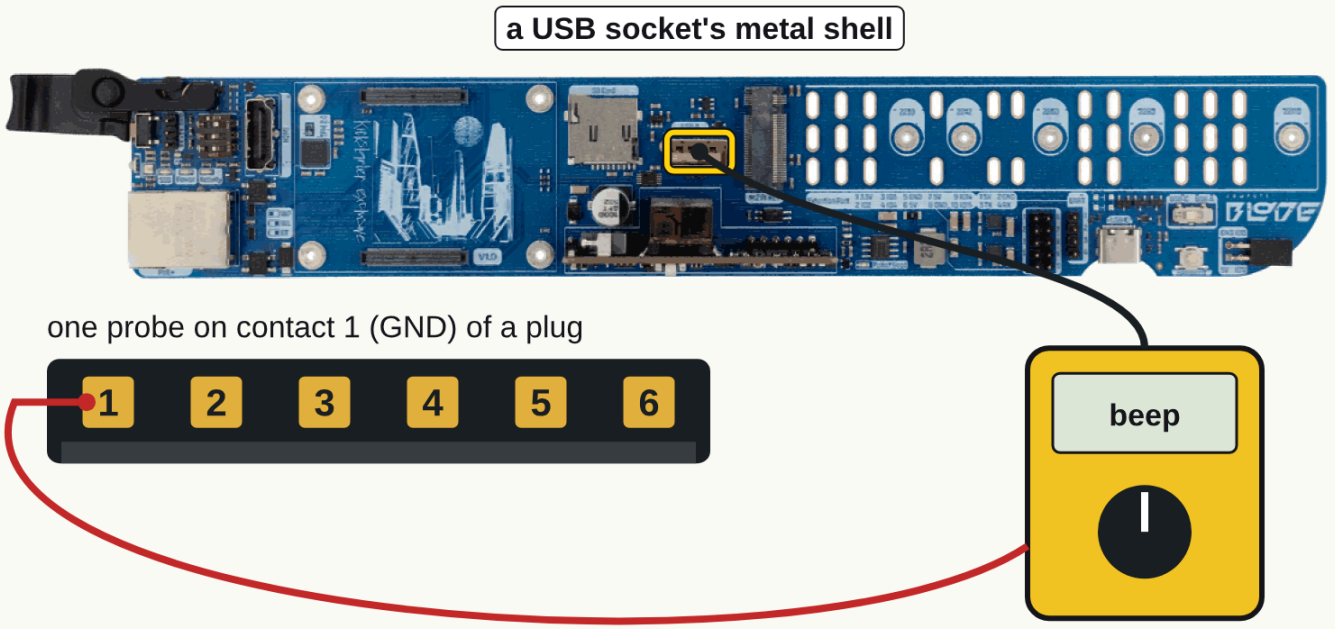


Fit the cables

13. This is a bench check; the housings come off again in the next step. Fit the P1 housing on the Extension Port with its marked corner on the pin printed 1 beside the header, and the P2 housing on the UART with its marked corner on the pin printed 1 beside the header. The Acorn is not in its slot and the plugs are free. With the host unplugged from power, put one meter probe on contact 1 (GND) of a plug and the other on the metal shell of a USB socket on the Compute Blade: it must beep. Do the same for the other plug. Then contact 6 of each plug (VCC, the wire you cut back): against the shell and against every other contact it must be silent.

Bench check on a Compute Blade, before power

housings on their headers, plugs free, host unplugged

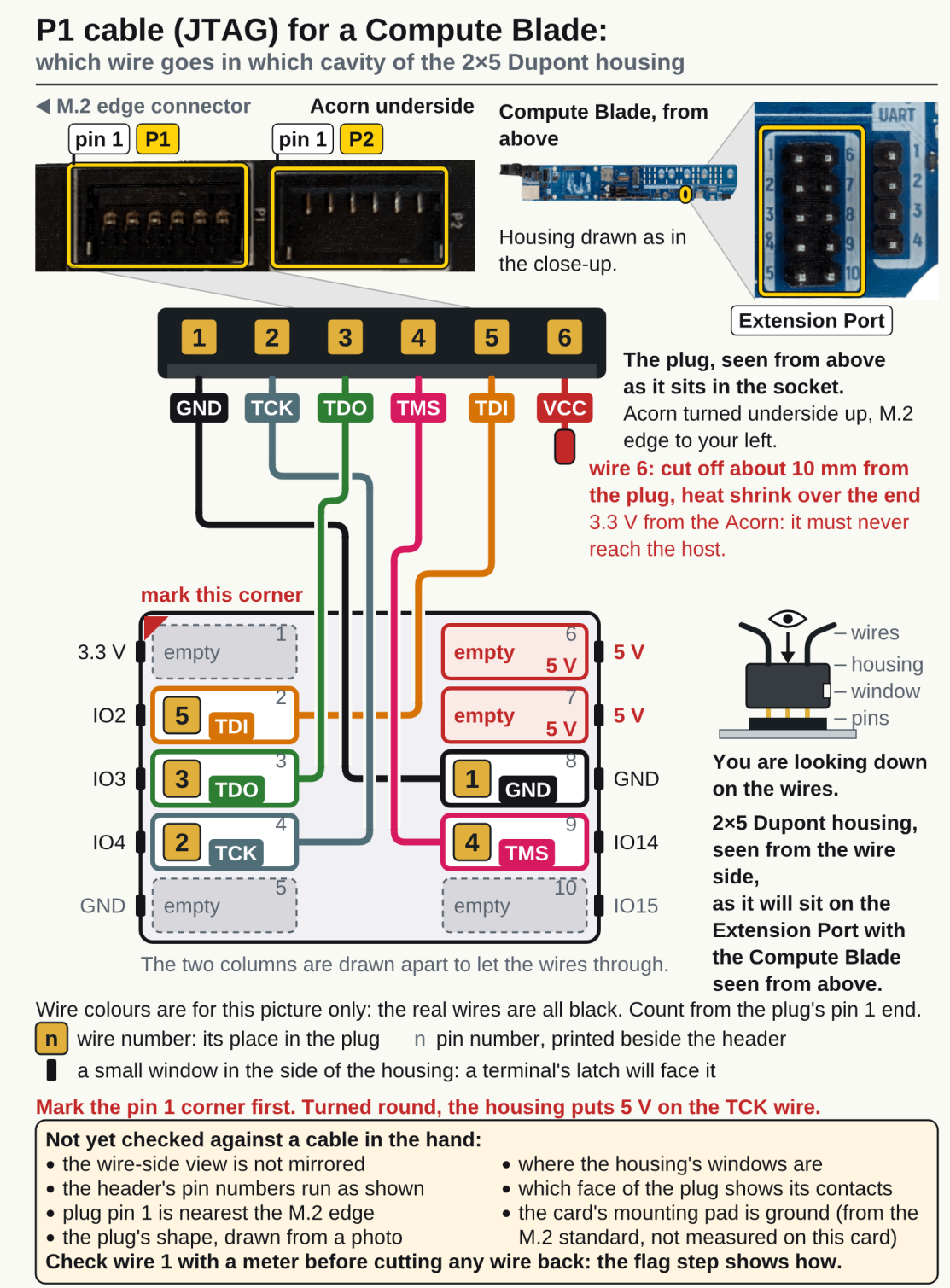


Contact 1 of each plug to the shell: the meter must beep.

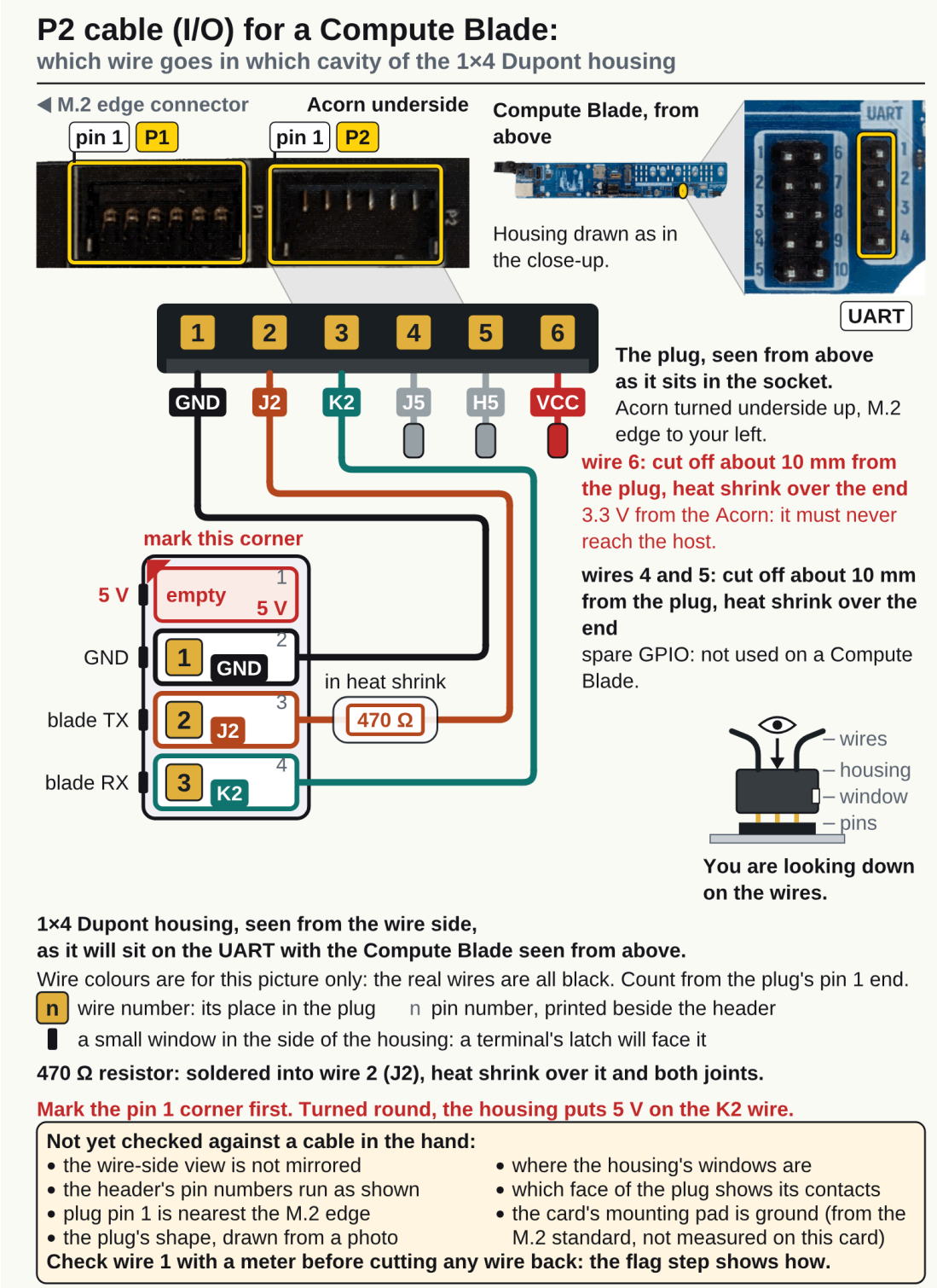
Contact 6 of each plug (VCC, cut back): silent to the shell and to every other contact.

The plug and the meter are sketched. The shell being ground is not measured on this host.

The P1 cavity picture again, for where its housing sits and which corner is marked:



The P2 cavity picture again, for where its housing sits and which corner is marked:



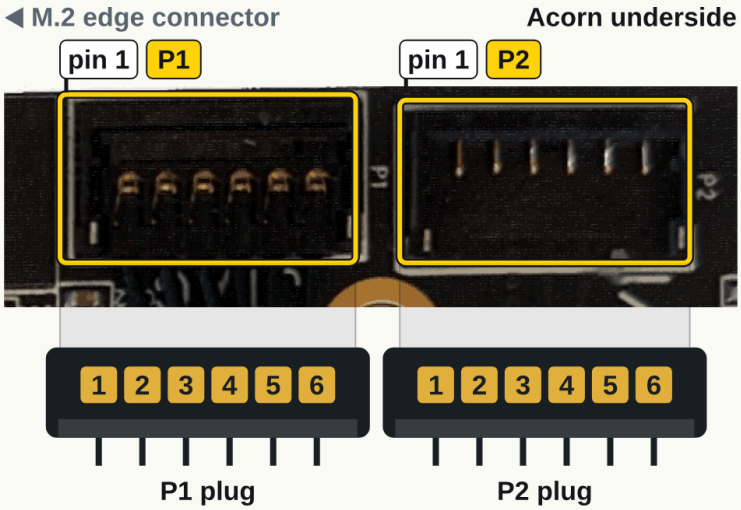
14. Fit the cables, in this order. The sockets are on the underside of the card and may not be reachable once it is in the slot.

1. Power off the Compute Blade: unplug its PoE cable.
2. If the housings are on the headers (after the bench check), take them off.
3. Press the P1 plug into socket P1 and the P2 plug into socket P2 on the underside of the Acorn, each the way round it was when you put the flags on, until fully seated.
4. Put the Acorn in the M.2 slot and fit its screw.
5. Fit the P1 housing on the Extension Port, marked corner on pin 1, and the P2 housing on the UART, marked corner on pin 1.

Fit the cables on a Compute Blade

in this order: the sockets may not be reachable once the card is in

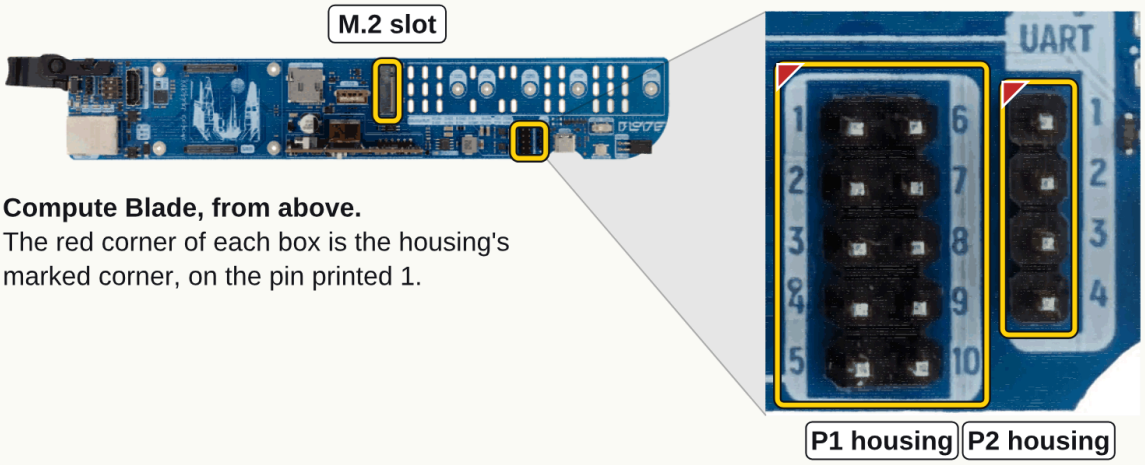
1. **Power off the Compute Blade: unplug its PoE cable.**
2. **If the housings are on the headers (after the bench check), take them off.**
3. **Press the P1 plug into socket P1 and the P2 plug into socket P2 on the underside of the Acorn, each the way round it was when you put the flags on, until fully seated.**



Wire 1 at the pin 1 end of each socket: the same way round as when you put the flags on.

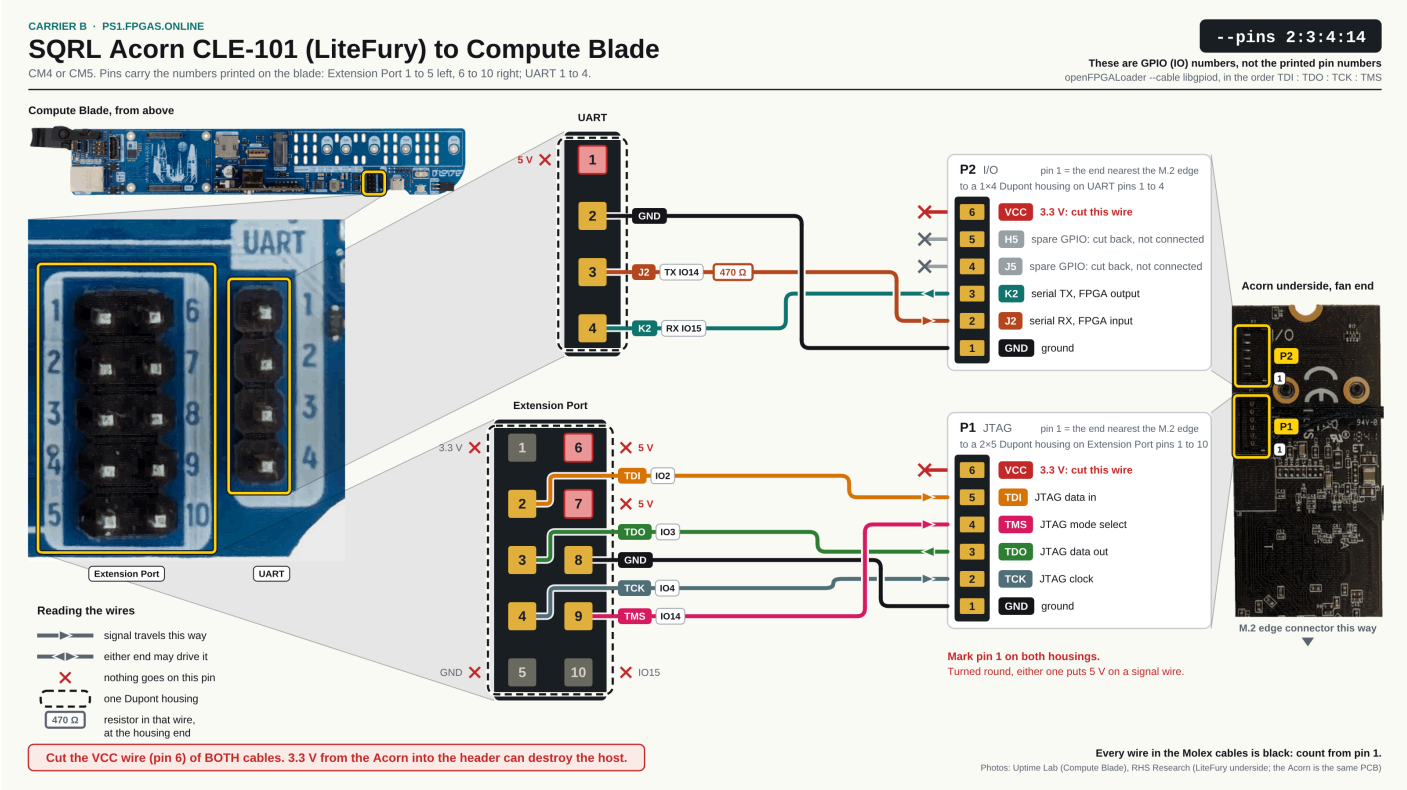
The plugs are sketched.

4. **Put the Acorn in the M.2 slot and fit its screw.**



Compute Blade, from above.
The red corner of each box is the housing's marked corner, on the pin printed 1.

5. **Fit the P1 housing on the Extension Port, marked corner on pin 1, and the P2 housing on the UART, marked corner on pin 1.**



[The finished wiring: Acorn to Compute Blade: full size on a landscape sheet of its own, sheet 41.]

Photos: Uptime Lab (Compute Blade), RHS Research (LiteFury underside; the Acorn is the same PCB).

Verification

The steps are the same on both carriers; the commands are not. What differs:

	Raspberry Pi 5	Compute Blade, CM4	Compute Blade, CM5
JTAG `--pins` (TDI:TDO:TCK:TMS)	`10:9:11:8`	`2:3:4:14`	`2:3:4:14`
GPIO chip for the `libgpiod` cable, which opens `/dev/gpiochip0`	`gpiochip15` under kernel 6.12 at Welland: link it as `gpiochip0` first	not read by us: run `gpiodetect` and link the chip labelled `pinctrl-bcm2711` as `gpiochip0` only if it is not that already	`pinctrl-rp1` was `gpiochip0` already on pi16 and pi20 (kernel 6.18.50, 2026-10-05): no link there
PCIe address of the card (`BDF` below)	`0001:01:00.0`	`0000:01:00.0` (pi14)	`0001:01:00.0` (pi16, pi20)
Root complex behind the slot	`1000110000.pcie`	not read by us: find it as in Step 5	`1000110000.pcie` (pi20, kernel 6.12.75)
FPGA serial port	`/dev/ttyAMA0`, GPIO14/15 at `a4`	`/dev/ttyAMA0`, GPIO14/15 at `a0`	`/dev/ttyAMA0`, GPIO14/15 at `a4`
J5 and H5	wired to GPIO3 and GPIO4	not wired	not wired
Boot configuration for the JTAG steps	as the fleet boots	not read by us on a CM4	`enable_uart=0` and no `console=serial0`: on pi16 (kernel 6.18.50, 2026-10-05) JTAG cannot run with `enable_uart=1`. Not yet run by us on this hardware

On a Compute Blade the JTAG steps and the serial step cannot share a boot under kernel 6.18 (JTAG on a blade): Steps 2, 4 and 5 load over JTAG and need the header's serial port off; a test of the serial pair (`fpgas-verify`'s `p2-uart` and `p2-serial`) needs it on, with a design already running that was loaded from flash. What has been run on a blade is the detach, load, rescan and re-probe of Steps 2 and 5, on pi20 (a CM5, kernel 6.12.75, its P2 pair on the Extension Port). Everything else in the blade blocks below is **not yet run by us on this hardware**. The J5 and H5 part of Step 4 does not apply on a blade.

Every blade block uses the card's address from `BDF`; set it once per shell:

On a **CM5** blade (pi16, pi20):

```
$ BDF=0001:01:00.0
```

On a **CM4** blade (pi14):

```
$ BDF=0000:01:00.0
```

The designs these steps load

Only one design is installed by a package: the fpgas.online Acorn design (the SoC that `fpgas-verify` checks). `fpgas-online-acorn-bitstreams` puts it in `/usr/share/fpgas-online/acorn-pcie/images/`, for each variant (`cle-215p`, `cle-215`, `cle-101`) as `acorn-<variant>-sqrl_acorn.bit` for a JTAG load, beside the two flash images. Steps 2 and 5 load that file; put its name in `SOC` once per shell. The PS1 cards are CLE-101; for another card put its variant in place of `cle-101`, here and in the two release file names below:

```
$ SOC=/usr/share/fpgas-online/acorn-pcie/images/acorn-cle-101-sqrl_acorn.bit
```

The loopback and pin-ID designs of Steps 3 and 4 are **not in any package**. They are built by CI and attached to the releases[13] of fpgas.online-test-designs, named `pmod-loopback_acorn-<variant>_vivado-vivado_sqrl_acorn.bit` and `pmod-pin-id_acorn-<variant>_vivado-vivado_sqrl_acorn.bit` in the release `vivado-bitstreams-v0.0-496-gf162f60`. That release's pin-ID build for the Acorn configures but never toggles a pin (Images[14]), so pin-ID has to be built from `main`. Download or build the file, copy it to the host, and put its name in `LOOPBACK` or `PINID`.

Steps 3 and 4 are the check by hand. To check the wiring of a card that runs the fpgas.online design, `fpgas-acorn-verify` needs none of these files: see Checking an Acorn's wiring[15].

Step 1: PCIe

On a **Raspberry Pi 5**:

```
$ lspci -nn -s 0001:01:00.0
```

On a **Compute Blade**:

```
$ lspci -nn -s $BDF
```

What it shows, after the address:

```
The factory Sqrl firmware:
  Processing accelerators [1200]: Squirrels Research Labs Acorn CLE-215+ [1e24:021f]
  Processing accelerators [1200]: Squirrels Research Labs Acorn CLE-101 [1e24:0101]
The vendor (RHS Research) XDMA sample image:
  Processing accelerators [1200]: Xilinx Corporation 7-Series FPGA Hard PCIe block (AXI/debug) [10ee:7011]
The fpgas.online Acorn design (or any LiteX x1 PCIe design):
  Memory controller [0580]: Xilinx Corporation Device [10ee:7021]
```

If the Acorn doesn't appear, check the M.2 seating (and on a Pi 5 the HAT's FPC cable), and `dmesg | grep -i pci`.

Step 2: JTAG

> **WARNING**
>
> **Detach the PCIe endpoint before every JTAG reconfiguration**, on either carrier. Reconfiguring the FPGA while its endpoint is enumerated is a surprise removal that the BCM2712 root complex (Pi 5, CM5) does not survive: the Pi drops SSH and reboots. With the endpoint removed first the load completes and the host is unaffected. Not measured on a CM4 (BCM2711); detach there too.

On a **Raspberry Pi 5**:

```
# 0. Detach the endpoint (bring it back as in Step 5, or reboot)
$ echo 1 | sudo tee /sys/bus/pci/devices/0001:01:00.0/remove
# The libgpiod cable opens gpiochip0, the header is gpiochip15
$ sudo ln -sfn /dev/gpiochip15 /dev/gpiochip0
# 1. Read-only check (safe without step 0)
$ openFPGALoader --cable libgpiod --pins 10:9:11:8 --detect
# Expected: idcode 0x3636093 (XC7A200T)
# 2. Load to SRAM. About 16 s for a 1.6 MB XC7A200T bitstream over libgpiod.
$ openFPGALoader --cable libgpiod --pins 10:9:11:8 $SOC
```

On a **Compute Blade** (booted with the header's serial port off):

```
# Which chip is the header? See the table above; do not copy the Pi 5 link.
$ gpiodetect
# 0. Detach the endpoint
$ echo 1 | sudo tee /sys/bus/pci/devices/$BDF/remove
# 1. Read-only check (safe without step 0)
$ openFPGALoader --cable libgpiod --pins 2:3:4:14 --detect
# Expected: idcode 0x3631093 (XC7A100T, an Acorn CLE-101 or LiteFury)
#           idcode 0x3636093 (XC7A200T, a CLE-215+ or NiteFury)
# openFPGALoader leaves its pins as outputs (seen after --detect on pi16): put them back
$ pinctrl set 2,4 no pu
# 2. Load to SRAM, and put the pins back again
$ openFPGALoader --cable libgpiod --pins 2:3:4:14 $SOC
$ pinctrl set 2,4 no pu
```

Never pass `--write-flash` here: an SRAM load is lost at power-off, so a reboot restores whatever is in flash, which makes every experiment safe. Writing the flash is covered in Acorn PCIe programming and multiboot[16].

> **WARNING**
>
> **Files staged under `/home/pi` do not survive a reboot** on a host whose root is an overlay in memory (`overlayroot=tmpfs` on a read-only NFS root, as on the fleet). The symptom is openFPGALoader printing `Open file … FAIL` in under 0.1 s: copy the bitstream again.

Step 3: UART and GPIO loopback

The loopback design (`pmod-loopback`) returns on K2 the inverse of what it sees on J2, and nothing else: GPIO14 → J2 → inverted → K2 → GPIO15. It does not touch J5 or H5; on a Raspberry Pi 5 those two wires are tested by `fpgas-verify` (`p2-gpio`) on a card that runs the fpgas.online design.

On a **Raspberry Pi 5**:

```
$ sudo systemctl stop serial-getty@ttyAMA0
$ sudo systemctl mask serial-getty@ttyAMA0
# Detach the endpoint first, as in Step 2, then load the loopback bitstream
$ echo 1 | sudo tee /sys/bus/pci/devices/0001:01:00.0/remove
$ openFPGALoader --cable libgpiod --pins 10:9:11:8 $LOOPBACK
# The loopback inverts each bit, so what comes back is not what was sent
$ stty -F /dev/ttyAMA0 115200 raw -echo
$ (sleep 1; echo test > /dev/ttyAMA0) &
$ timeout 3 cat /dev/ttyAMA0 | od -An -tx1
```

Some bytes arriving means J2 and K2 both carry; they are not the bytes sent, and because the design holds the line inverted while idle the UART may report framing errors or a break as well. No bytes at all means K2 or GPIO15 is not connected. These commands are written from the design's source: not yet run by us in this form.

On a **Compute Blade** under kernel 6.18 this step cannot be done by hand in one boot: the loopback design has to be loaded over JTAG, which needs the header's serial port off, and the test itself needs the serial port on (JTAG on a blade). The serial pair of a blade is checked by `fpgas-verify` (`p2-uart`, `p2-serial`) once the card runs the fpgas.online design from its flash; no PS1 blade card has that yet (see What each blade still needs[17]).

Step 4: pin ID

On a **Raspberry Pi 5**:

```
$ echo 1 | sudo tee /sys/bus/pci/devices/0001:01:00.0/remove   # detach first, as in Step 2
$ openFPGALoader --cable libgpiod --pins 10:9:11:8 $PINID
# Each ball transmits its own name at 1200 baud. Correctly wired:
# GPIO15 → "K2" (serial TX, on the Pi's RXD0)
# GPIO14 → "J2" (serial RX, on the Pi's TXD0)
# GPIO3  → "J5" (spare GPIO)
# GPIO4  → "H5" (spare GPIO)
```

On a **Compute Blade** (not yet run by us on a blade wired as on this page):

> **WARNING**
>
> **Not on a blade whose J2 wire has no 470 Ω resistor**, which is pi20 as it is wired today (Compute blades[3]). The moment the load finishes, the pin-ID design drives J2, and J2 is on GPIO14, which openFPGALoader has just left an output. With the resistor that is about 7 mA for a moment; without it, it is two outputs shorted together, which costs JTAG until a power cycle and can crash the host.

Like every JTAG load on a blade it needs a boot with the header's serial port off (JTAG on a blade). The load and the command that makes GPIO14 an input again are **one command line**, so that nothing is typed between them, and the pins are put back even if the load fails:

```
$ echo 1 | sudo tee /sys/bus/pci/devices/$BDF/remove   # detach first, as in Step 2
$ openFPGALoader --cable libgpiod --pins 2:3:4:14 $PINID; pinctrl set 14 ip pn; pinctrl set 2,4 no pu
# GPIO14 is now an input with no pull: do NOT set it to a0 or a4 while pin-ID runs.
# Only GPIO15 → "K2" and GPIO14 → "J2" answer: J5 and H5 are not connected.
# The design drives J2, which shares GPIO14 with TMS. The 470 Ω resistor in the
# J2 wire is there so that JTAG still works afterwards; without it JTAG is lost
# until a power cycle (see "The shared line and the 470 Ω resistor").
```

Only GPIO15 can be a hardware UART receiver on a Pi 5, so the other three are decoded from edge timestamps: the repository's pin-ID host scanner[18] requests both-edge events through gpiod (v1 or v2), rebuilds the 1200-baud frames from the kernel timestamps (833 µs per bit against nanosecond stamps), and finds the header chip by label, so it works on a Pi 5. `gpiomon` edge timestamps decoded by hand work the same way. Do not sample the pins by polling from Python: polling mis-frames the bytes even on a clean signal. Keep GPIO14 an input throughout (see the warning under P2[8]). Check the method on a positive control before trusting a negative: drive a spare Pi GPIO and confirm the monitor sees it. The design is described under Verifying wiring with the pin-id design[19]. Which of these readers works on a blade under kernel 6.18 is not known.

A passive check needs no bitstream: toggle the Pi's internal pull-up, then pull-down, on each line and see whether the line follows. A ~50 kΩ internal pull loses to any real driver, so a line that follows is floating (the far end is an FPGA input: J2) and one that stays put is driven (an FPGA output: K2). It cannot

read GPIO2 or GPIO3 where they carry I²C pull-ups. The same test shows whether P1 is mated: the Acorn pulls TCK up, so a TCK line that the Pi's pull-down cannot move has the card on the end of it, and one that follows has nothing.

Step 5: PCIe design

On a **Raspberry Pi 5**:

```
$ echo 1 | sudo tee /sys/bus/pci/devices/0001:01:00.0/remove   # detach first
$ openFPGALoader --cable libgpiod --pins 10:9:11:8 $SOC
$ echo 1 | sudo tee /sys/bus/pci/rescan
$ lspci -nn -d 10ee:
# Expected: Xilinx Corporation Device [10ee:7021] (the LitePCIe default for one lane)
```

On a **Compute Blade** (the load, rescan and re-probe as measured on pi20, a CM5; the `pinctrl` line as run on pi16; the root complex's name on a CM4 is not read by us, so take it from the `readlink` line):

```
$ echo 1 | sudo tee /sys/bus/pci/devices/$BDF/remove   # detach first
$ openFPGALoader --cable libgpiod --pins 2:3:4:14 $SOC
$ pinctrl set 2,4 no pu
$ echo 1 | sudo tee /sys/bus/pci/rescan
$ lspci -nn -d 10ee:
# Nothing? On pi20 a rescan was not enough: re-probe the root complex of the slot.
$ RC=$(readlink -f /sys/bus/pci/devices/${BDF%%:*}:00:00.0 | grep -o '[0-9a-f]*\.pcie')
$ echo $RC
1000110000.pcie
$ echo $RC | sudo tee /sys/bus/platform/drivers/brcm-pcie/unbind
$ echo $RC | sudo tee /sys/bus/platform/drivers/brcm-pcie/bind
$ lspci -nn -d 10ee:
# Expected: Xilinx Corporation Device [10ee:7021]
```

When the rescan is enough and when the re-probe is needed is under Bring the endpoint back after a JTAG load[20].

Kernel console on the FPGA UART

> **WARNING**
>
> **The kernel console must not be on the FPGA's UART.** If a host's kernel command line puts it there (`console=ttyAMA0`, or `console=serial0` on any host whose header UART is enabled: `uart0-pi5` on a Pi 5, `enable_uart=1` on a blade, as on pi16), loading any design that drives serial TX — the UART SoC, pin-ID, the GPIO loopback — reboots or crashes the host. A fleet host is meant to boot with `console=tty1` (Compute Blades) or `console=ttyAMA10` (Pi 5s); check this on any new host, and on a host that has been reinstalled: pi16 at PS1 had its console on the FPGA UART when read on 2026-10-05 (pi16 on 5 October 2026[12]).

The FPGA drives K2 at its own baud rate (1200 for pin-ID, 115200 for the UART SoC), and the kernel console reads the bytes as garbage, some of which are **SysRq commands** — `reboot(b)`, `crash(c)`, `poweroff(o)`, `kill-all-tasks(i)`. Apply both:

1. Keep the console off the FPGA UART: `console=tty1` on a Compute Blade; `console=ttyAMA10` (the debug connector) on a Pi 5, via `[pi5] cmdline=cmdline-pi5.txt`, so the getty systemd derives from the console lands there too.
2. Disable SysRq: `kernel.sysrq=0` on the kernel command line or in `/etc/sysctl.d/`.

Tracked in fpgas-online/todo#22[21] and test-designs issue #3[22].

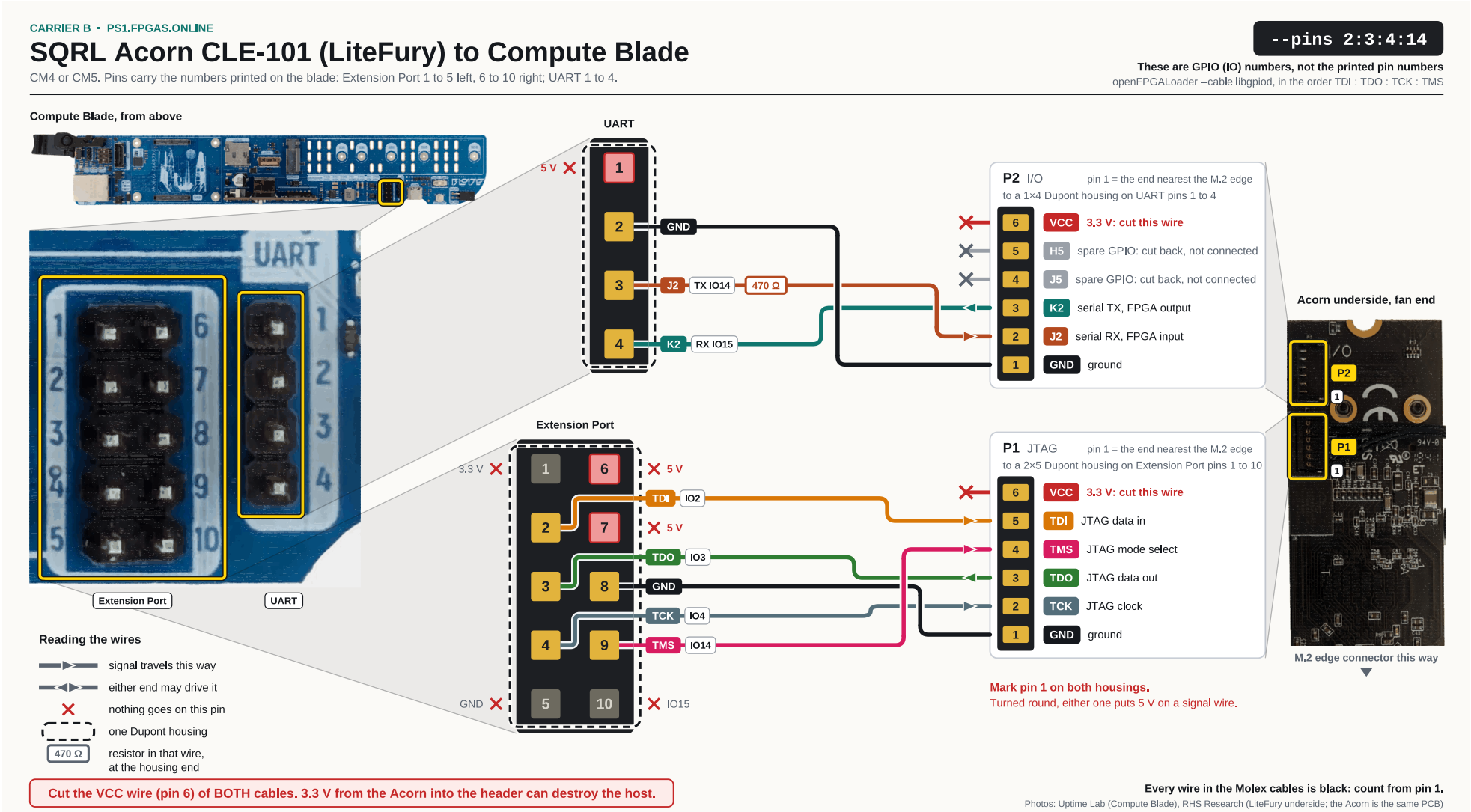
Troubleshooting

Problem	Likely cause	Fix
Acorn not on PCIe	M.2 not seated, FPC cable loose	Reseat the M.2 card, check the FPC
JTAG programming fails	SPI modules loaded (Pi 0-4 only), wrong pins	`rmmod spidev spi_bcm2835` on a Pi 0-4; check the pin order
`gpiod_line_request_set_values_subset: Assertion 'request' failed` on a Compute Blade	The serial driver holds GPIO14 (`enable_uart=1`; `dmesg`: `pin gpio14 already requested by 1f00030000.serial`)	JTAG needs the header's serial port off at boot (JTAG on a blade); not yet run by us on this hardware
`JTAG init failed with: Unable to open gpio chip` (Pi 5)	The `libgpiod` cable opens `/dev/gpiochip0`; the header is `gpiochip15`	`ln -sfn /dev/gpiochip15 /dev/gpiochip0`
`--detect` says `found 0 devices` but PCIe enumerates	P1 (JTAG) cable unmated or miswired	Check TCK for the Acorn's pull-up; reseat P1
Pi reboots or SSH drops during a JTAG load	FPGA reconfigured while its PCIe endpoint was enumerated	`echo 1 > /sys/bus/pci/devices/0001:01:00.0/remove` before loading (`0000:01:00.0` on a CM4 blade)
`Open file … FAIL` in < 0.1 s	The bitstream is gone: `/home/pi` is a tmpfs overlay, lost at reboot	Copy the file again
Pi crashes the instant a GPIO is set to output	Contention with an FPGA output on the same wire (pin-ID drives all four P2 balls)	Keep GPIO14 an input (`pinctrl set 14 ip pn`) while pin-ID runs
No `/dev/ttyAMA0` on a Pi 5	RP1 uart0 disabled; `disable-bt` does not enable it on bcm2712	`[pi5] dtoverlay=uart0-pi5`
No UART output	serial-getty holding the port, wrong baud, or K2/J2 not crossed over	Mask serial-getty, use 115200, run pin-ID and check GPIO15 reads `K2`
Pi reboots when a serial design loads	Kernel console on the FPGA UART; SysRq	Console to `ttyAMA10` (Pi 5) / `tty1` (blade), `kernel.sysrq=0`
GPIO pins don't respond	Cable wired incorrectly	Buzz each wire from its Pico-EZmate position (pin 1 = GND, nearest the M.2 edge) to the pin in the tables above

Problem	Likely cause	Fix
PCIe device missing after loading a design	Not rescanned, or (on a CM5 blade) a LiteX design that needs PERST#	Rescan; if still missing, re-probe the slot's root complex (procedure[20]); if it still does not link, check the build's I/O report has the lane on B10/B6
JTAG fails on a Compute Blade	Wrong pin order	Use `--pins 2:3:4:14`, not `--pins 10:9:11:8`
UART dead after JTAG on a Compute Blade	openFPGALoader left GPIO14 a plain output	Only in a boot with the serial port on (kernel 6.12), and only once the loaded design treats J2 as an input: on a CM4 `pinctrl set 14,15 a0`; on a CM5 `pinctrl set 14,15 a4` (one or the other), then open `/dev/ttyAMA0`. Never with pin-ID loaded
Board hung, ~0.4 W on PoE instead of ~8 W	Wedged Pi 5	PoE cycle the switch port; a Pi 5 needs over 90 s to come back

Links in this chapter

1. https://docs.fpgas.online/en/latest/boards/acorn/index.html
2. https://docs.fpgas.online/en/latest/boards/acorn/wiring.html#raspberry-pi-5
3. https://docs.fpgas.online/en/latest/sites/ps1.html#compute-blades
4. https://github.com/fpgas-online/fpgas.online-test-designs/tree/main/docs/wiring/acorn
5. https://github.com/enjoy-digital/litex/wiki/Use-LiteX-on-the-Acorn-CLE-215
6. https://computeblade.com/
7. https://docs.computeblade.com/blade/guides/gpio
8. https://docs.fpgas.online/en/latest/boards/acorn/wiring.html#p2-serial-pair-and-spare-gpios
9. https://github.com/fpgas-online/fpgas.online-test-designs/issues/4
10. https://docs.fpgas.online/en/latest/setup/network.html#poe-power-control
11. https://docs.fpgas.online/en/latest/sites/ps1.html#power-control
12. https://docs.fpgas.online/en/latest/sites/ps1.html#pi16-on-5-october-2026
13. https://github.com/fpgas-online/fpgas.online-test-designs/releases
14. https://docs.fpgas.online/en/latest/boards/acorn/pcie-programming.html#images
15. https://docs.fpgas.online/en/latest/verify/fpgas-verify.html#checking-an-acorn-s-wiring
16. https://docs.fpgas.online/en/latest/boards/acorn/pcie-programming.html
17. https://docs.fpgas.online/en/latest/sites/ps1.html#what-each-blade-still-needs
18. https://github.com/fpgas-online/fpgas.online-test-designs/blob/main/designs/pmod-pin-id/host/identify_pmod_pins.py
19. https://docs.fpgas.online/en/latest/boards/pin-id.html
20. https://docs.fpgas.online/en/latest/boards/acorn/pcie-programming.html#bring-the-endpoint-back-after-a-jtag-load
21. https://github.com/fpgas-online/todo/issues/22
22. https://github.com/fpgas-online/fpgas.online-test-designs/issues/3



Acorn to Compute Blade wiring sheet (https://docs.fpgas.online/en/latest/_downloads/bf3083dcc0862e154eb43b66073c3866/acorn-wiring-computeblade.svg)

PS1

The public deployment at Pumping Station: One[1] in Chicago, published as ps1.fpgas.online[2] and run by Carl Karsten. Eight Arty A7 boards on Raspberry Pi 3B/3B+/4B hosts, and four Raspberry Pi Compute Modules on Compute Blade[3] carriers with SQRL Acorn boards in the M.2 slot.

Of the four blades, **pi14**, **pi16** and **pi20** carry an Acorn; **pi18**'s M.2 slot is empty.

Hosts and boards

Compute blades

All four were up when last probed together (2026-09-20), and the table is that probe. On 2026-10-05 pi16 and pi20 were read again; what differed is under pi16 on 5 October 2026 and pi20 on 5 October 2026. The same day the visitor SSH ports of pi14 and pi18 did not answer, so nothing was read from them; why is not known.

Host	Switch Port	IP	RPi MAC	RPi Model	Board (PCIe ID)	FPGA DNA	PCIe Bus	JTAG (P1)	P2 cable	Status on 2026-09-20
pi14	e14	10.21.0.114	2c:cf:67:37:d4:bd	CM4 Rev 1.1 4 GB	Acorn CLE-101 `1e24:0101`	not read	0000:01	no response (P1 unmated)	untested	Online
pi16	e16	10.21.0.116	2c:cf:67:fb:91:e5	CM5 Lite Rev 1.0 8 GB	Acorn CLE-101 `1e24:0101`	not read	0001:01	2026-09-20: no response, TCK floating. 2026-10-05: cannot run (serial driver holds GPIO14)	untested	Online
pi18	e18	10.21.0.118	2c:cf:67:37:d5:08	CM4 Rev 1.1 4 GB	none: M.2 slot empty	—	—	n/a	n/a	Online
pi20	e20	10.21.0.120	2c:cf:67:fd:1e:be	CM5 Lite Rev 1.0 8 GB	vendor XDMA image `10ee:7011`	0x0028e5c45e304854	0001:01	2026-09-20 (kernel 6.12.75): OK, IDCODE `0x3631093`. 2026-10-05 (kernel 6.18.50, serial port on): not tried, expected not to run	on the Extension Port: K2 → GPIO15, J2 → GPIO14, no resistor; J5 and H5 not wired	Online

pi20 is the only blade whose JTAG has answered (on 2026-09-20), so it is the only one with a device DNA: `0x0028e5c45e304854`, an XC7A100T. The fpgas.online Acorn design ran on it from SRAM that day (Gen2 x1, the same ident and DNA over PCIe and over the UART bridge); by that probe its flash holds the vendor XDMA sample image, and on 2026-10-05 the card enumerated as `10ee:7011`, the ID of that sample design. “P1 unmated” on pi14 and pi16 is read off TCK: the Acorn pulls TCK up, and on pi20 the Pi’s pull-down cannot move it, while on pi14 and pi16 it floats exactly as on pi18, which has no card. Reseating P1 is the fix.

These boards are often called LiteFury. Their factory PCI ID identifies them as SQRL Acorn CLE-101: the same PCB family, XC7A100T with 512 MB of DDR3. See SQRL Acorn[4].

No blade is wired to the Compute Blade wiring[5] yet. That wiring puts P1 on the Extension Port and P2 on the 4-pin UART header, with a 470 Ω resistor in the J2 wire. pi20, the one blade whose wiring has been read, has its P2 serial pair on Extension Port pins 9 and 10 instead, sharing pin 9 (GPIO14) directly with TMS and with no resistor, so a design that drives J2 costs JTAG until a PoE cycle (why[6]), and its J5 and H5 are not wired. How pi14’s and pi16’s P2 cables are wired is not known: pi14’s P1 did not answer on 2026-09-20 and pi16’s JTAG cannot run today (pi16 on 5 October 2026), so nothing can be loaded to read them.

On all four blades the JTAG pins are `--pins 2:3:4:14` (it has only answered on pi20) and the FPGA UART is `/dev/ttyAMA0` at GPIO14/15. All four ran Debian’s openFPGALoader 0.13.1, which has `--read-dna`, when probed. PCIe is through the blade’s M.2 slot. On 2026-09-20 all four netbooted the trixie NFS root (arm64 then) with overlayroot, with `console=tty1` and `serial-getty@ttyAMA0` inactive, so the kernel console crash[7] could not happen. That no longer holds on pi16 or pi20 (their boot configuration, read on 2026-10-05, is below), and pi14 and pi18 have not been read since.

pi16 on 5 October 2026

Read over SSH as the visitor. Nothing was written to the host’s storage or to the Acorn; a report file in `/run` was made and removed, and GPIO2 and GPIO4, left as outputs by a hand-run `openFPGALoader --detect`, were put back.

	Read on pi16, 2026-10-05
Module	Compute Module 5 Lite Rev 1.0, 8 GB, MAC `2c:cf:67:fb:91:e5`
System	Raspbian 13 (trixie), 32-bit userspace on kernel `6.18.50+rpt-rpi-v8`; Debian’s openFPGALoader 0.13.1
Serial port	`enable_uart=1`, `console=serial0,115200`, and a serial getty active on `/dev/ttyAMA0`
Card	`1e24:0101` at `0001:01:00.0`: SQRL’s factory image, not converted
`fpgas-verify` 0.0.post1100	`pcie-link` (5.0 GT/s, x1) and `rp1-pio` pass; `jtag` cannot run; the tests that need the fpgas.online design are not run on a factory image, and `p2-gpio` is not run because J5 and H5 are not wired on a blade

Two things follow from the serial port being on:

- **JTAG cannot run.** TMS is GPIO14, which is also the serial port’s TX pin. The kernel’s serial driver holds it (`pin gpio14 already requested by 1f00030000.serial; cannot claim`), this kernel does not lend a held pin, and the serial driver has no `unbind` file and is the kernel console, so it cannot be

detached from the running system. openFPGALoader 0.13.1 then stops on a libgpiod assertion instead of saying so. Tracked in test-designs issue #127[8]. So whether pi16's P1 cable is mated cannot be told from a scan today; the "P1 unmated" in the table is the pull-up reading of 2026-09-20.
- **The kernel console is on the FPGA's UART**, which the wiring page[7] warns against: a design that drives serial TX can reboot or crash the host. It must be moved (`console=tty1`, no serial getty) before such a design is loaded.

Moving the console does not free JTAG. With `enable_uart=1` the serial driver holds GPIO14 whether or not a console or a getty uses the port. For JTAG the header's serial port itself has to be off at boot (`enable_uart=0`, and no `console=serial0` word), and in that boot the tests of the P2 serial pair cannot run. **Not yet run by us on this hardware.** The same applies to any blade on this kernel with the serial port on, pi20 included (next section).

pi20 on 5 October 2026

Read over SSH as the visitor, without sudo; nothing was installed, copied or changed.

	Read on pi20, 2026-10-05
Module	Compute Module 5 Lite Rev 1.0, 8 GB, MAC `2c:cf:67:fd:1e:be`
System	Raspbian 13 (trixie), 32-bit userspace on kernel `6.18.50+rpt-rpi-v8`; Debian's openFPGALoader 0.13.1
Serial port	`enable_uart=1`, `console=ttyAMA0,115200` on the kernel command line, and `serial-getty@ttyAMA0` active: the same boot settings as pi16
Card	`10ee:7011` at `0001:01:00.0`: the ID of Xilinx's XDMA sample design (`fpgas-verify` would report it as unconverted); not SQRL's factory image and not the fpgas.online design. Which image it is beyond that ID was not read
`fpgas-verify`	not installed, so it has not run here

pi20's JTAG answered on 2026-09-20 under kernel 6.12.75. On 2026-10-05 it ran the same kernel as pi16 with the same serial-port settings, so its JTAG is **expected not to run** for the same reason (the serial driver holds GPIO14). **Not tried:** no JTAG command was run on pi20 that day.

Where a blade's boot configuration is

Read on pi16 and pi20 themselves on 2026-10-05, not on the gateway: on both, the kernel command line has `nfsroot=10.21.0.1:/srv/nfs/rpi/trixie/root` and `console=ttyAMA0,115200`, and `/boot/firmware/config.txt` has `enable_uart=1` under `[all]` and no overlay for the UART. On pi16 `/boot/firmware/cmdline.txt` holds the same line with `console=serial0,115200`; that file was not read on pi20.

The settings to change for JTAG (`enable_uart` in `config.txt`, and the `console=serial0` word in `cmdline.txt`) are in those two files, but where their master copy is was not read. The likely place is the site's gateway, by this page's account under `/srv/nfs/rpi/trixie/boot`: the bootloader's boot order on both blades ends with the network (`BOOT_ORDER=0xf2461`) and the label read saw no storage device on either module, which makes a network boot likely. **Not read by us:** the directory the firmware fetches the two files from at boot, whether `/boot/firmware` on a blade is that directory or a copy, and whether one copy serves all four blades. Whoever runs the gateway makes the change. If one copy serves all four blades, it applies to all of them; it takes effect at a blade's next boot.

What each blade still needs

To reach the Compute Blade wiring[5], from what the table above records. None of us has wired a blade this way or converted a card on one yet: **not yet run by us on this hardware**.

Blade	Card	P1 (JTAG) cable	P2 (serial) cable	Host
pi14	fitted, factory image: to be converted	did not answer: reseat or refit on the Extension Port	not known: build to the UART header, 470 Ω in the J2 wire, J5 and H5 cut back	read its kernel command line and serial getty again (last read 2026-09-20)
pi16	fitted, factory image: to be converted	not known (see above): check, reseat or refit on the Extension Port	not known: build to the UART header, 470 Ω in the J2 wire, J5 and H5 cut back	take the kernel console and the getty off `/dev/ttyAMA0` (needed in any case); for JTAG, the serial port off at boot, as above
pi18	none: fit one	fit on the Extension Port	fit on the UART header, 470 Ω in the J2 wire, J5 and H5 cut back	read its kernel command line and serial getty again (last read 2026-09-20)
pi20	fitted, vendor sample image in flash: to be converted	answered on 2026-09-20: leave the cable. JTAG is expected not to run as the blade boots now (not tried)	move the serial pair from Extension Port pins 9 and 10 to the UART header, and add the 470 Ω resistor in the J2 wire	as pi16 (read 2026-10-05: the same kernel and serial-port settings): take the kernel console and the getty off `/dev/ttyAMA0`; for JTAG, the serial port off at boot

The parts are in the wiring page's Bill of Materials[9]. Once a blade is wired, check it with fpgas-verify.

pi20's serial pair is direct on GPIO14 and GPIO15 without the resistor. From the check's code, not from a run on a blade: `fpgas-verify`'s `p2-serial` test never has both ends driving a wire at once (the Pi's pins are inputs while the FPGA drives, and the FPGA's are inputs while the Pi drives), so it does not rely on the resistor, and the fpgas.online design treats J2 as an input. What the missing resistor does not survive is a design that drives J2, such as pin-ID.

The `RPi Model` column matters: a CM4 and a CM5 are not interchangeable, and what differs — the serial mux, and how many UARTs there are — is under Compute Module 4 versus Compute Module 5[10].

WARNING

Reconfiguring the FPGA over JTAG while its PCIe endpoint is enumerated is a surprise removal. Detach the endpoint first, using the host's own bus from the `PCIe Bus` column above:

```
$ BDF=0001:01:00.0           # pi16, pi20; 0000:01:00.0 on the CM4 blade pi14
$ echo 1 | sudo tee /sys/bus/pci/devices/$BDF/remove
```

Restore it by rebooting, or as described under Bring the endpoint back after a JTAG load[11] (on a blade a LiteX design needs a root-complex re-probe, not just a rescan). `--detect` and the other read-only queries are safe without this; **loading a bitstream is not**.

NOTE

The blades have no page under `https://ps1.fpgas.online/fpgas/`: pi14, pi16, pi18 and pi20 all return 404 (checked 2026-09-03).

Source: live probes (`lspci -nn`, `openFPGALoader --detect` and `--read-dna`, pull-up/pull-down on each P2 and JTAG line, the pin-ID check on pi20); MACs and switch ports cross-checked against infra `host_vars/ps1.fpgas.online.yml`.

Checking a board here

The hosts here are checked the same way as any machine outside the fleet: install the Acorn packages on the host and run `fpgas-verify`. See Installing the Acorn packages[12] and Checking a board: fpgas-verify[13], which covers running it, reading the result, updating the record and debugging a failure.

For an Acorn on a Compute Blade, Checking an Acorn's wiring[14] goes from a fresh boot to which wire a failing line points at, and On a Compute Blade[15] says what has and has not been run on a blade.

Links in this chapter

1. https://pumpingstationone.org/
2. https://ps1.fpgas.online/fpgas/
3. https://computeblade.com/
4. https://docs.fpgas.online/en/latest/boards/acorn/index.html
5. https://docs.fpgas.online/en/latest/boards/acorn/wiring.html#compute-blade
6. https://docs.fpgas.online/en/latest/boards/acorn/wiring.html#the-shared-line-and-the-470-resistor
7. https://docs.fpgas.online/en/latest/boards/acorn/wiring.html#kernel-console-on-the-fpga-uart
8. https://github.com/fpgas-online/fpgas.online-test-designs/issues/127
9. https://docs.fpgas.online/en/latest/boards/acorn/wiring.html#bill-of-materials
10. https://docs.fpgas.online/en/latest/setup/pi.html#compute-module-4-versus-compute-module-5
11. https://docs.fpgas.online/en/latest/boards/acorn/pcie-programming.html#bring-the-endpoint-back-after-a-jtag-load
12. https://docs.fpgas.online/en/latest/boards/acorn/index.html#installing-the-acorn-packages
13. https://docs.fpgas.online/en/latest/verify/fpgas-verify.html
14. https://docs.fpgas.online/en/latest/verify/fpgas-verify.html#checking-an-acorn-s-wiring
15. https://docs.fpgas.online/en/latest/verify/fpgas-verify.html#on-a-compute-blade

fpgas-verify: checking an FPGA board from its Raspberry Pi

`fpgas-verify` answers one question per Pi: **is this Pi and its FPGA board ready for users?** It finds the board, tests the board and its wiring to the Pi, and gives one result, pass or fail, with every fault it found.

- What the tool must do: verify-goals.md[1]. Where this page and that one disagree, verify-goals.md says what the tool should do.
- The code: `verify/`[2]. The design notes: plans/2026-09-26-fpgas-online-verify-design.md[3].
- `verify_hardware.py` (verify-hardware.md[4]) is a different tool: a developer's script that loads freshly built bitstreams from a workstation over SSH.

This page has two parts:

1. Using verify as a standalone tool: installing, running, reading the result, and what each board's check tests.
2. How verify is used in fpgas.online[5]: at every boot of every netbooted Pi, the events it sends the site, and the current results[6].

---

1. Using verify as a standalone tool

Running it

```
sudo fpgas-verify                          # check this host's board, as the boot does
sudo fpgas-arty-verify                     # check the Arty, whatever this host is set up for
sudo fpgas-arty-verify --test ddr          # run one test; the JSON report goes to stdout
sudo fpgas-acorn-verify --test pcie-link --test flash   # only some tests (these two only read)
sudo fpgas-verify --update                 # after flashing or swapping a board on purpose
fpgas-verify --list                        # the installed boards, and which this host checks

sudo fpgas-verify --identify               # who the board is, as JSON (see "Identity and labels")
sudo fpgas-verify --label --out labels.pdf # this Pi's and its board's labels, made by rpi-hwid

sudo fpgas-arty-debug test ddr             # load the DDR design and run its test, all output live
sudo fpgas-acorn-debug identify            # the same as fpgas-acorn-verify --identify
```

- Run them with `sudo`; `--help`, `--list` and `fpgas-<board>-debug list` do not need it.
- Nothing is sent anywhere unless a file says `[verify] publish = on` (next to `fpga-board`, below). A host that only has the packages installed publishes nothing, at boot or by hand; its summary says so in one line (`not published: no file says ...`), and there is no error. The fpgas.online Pi root sets it in `/etc/fpgas-verify/fleet.ini`, so a fleet Pi tells its site how the check went; there, `--no-publish` is how a run by hand stays private. The transcripts below were taken on fleet Pis, which is why they carry it; each shows the line that says whether it was published.
- `--test` runs part of the check. It is never published, never recorded, and its report goes to stdout unless `--report` says otherwise.

- Only one command uses a board at a time. A second one prints `waiting for another user of the <board> to finish...` and waits. Looking for a NeTV2 drives its JTAG pins, so the check and `fpgas-netv2-debug detect` take its lock for that too, waiting as for the check, and let it go before the check takes it again.
- The locks are files: `/run/fpgas-online/<board>.lock`, and the Acorn's `/run/lock/fpgas-acorn.lock` (shared with `fpgas-acorn-flash`). The packages' `/usr/lib/tmpfiles.d/fpgas-online-*.conf` create them at boot, root-owned and 0644, so no other user can create one first. A lock file that cannot be opened is an error naming the file and its owner: remove it, or reboot.
- Which board a host checks is `[verify] fpga-board = <board>` or `auto`, in `*.ini` files: the board's package puts one in `/usr/share/fpgas-online/verify/mode.d/`; one in `/etc/fpgas-verify/` overrides it.
- `[verify] publish = on` in the same files makes the check tell the fleet how it went (events[7]). It is off unless a file says so; a value that is neither `on` nor `off`, or two files that disagree, is an error and nothing is sent. Accepted values are `on`, `yes`, `true`, `1` and `off`, `no`, `false`, `0`, as for `power-cycle-check`. Every report and summary says whether the run was published and why: `"publish": {"on": true, "configured_by": "/etc/fpgas-verify/fleet.ini"}` and `published to the fleet: ...`, or `"publish": {"on": false, "why": "..."}` and `not published: ...`. On a fleet Pi, `not published: no file ...` in `journalctl -b -u fpgas-verify` means the root lost its `fleet.ini`.
- `[verify] power-cycle-check = on` in the same files switches on the Acorn's power-cycle check[8]. It is off unless a file says so: the fpgas.online Pi root sets it in `/etc/fpgas-verify/`; elsewhere it stays off.
- Options for the boot run go in `FPGAS_VERIFY_ARGS` in `/etc/default/fpgas-verify`. The package installs that file with every line a comment (it is what `EnvironmentFile=` in `systemctl cat fpgas-verify` names), so the boot run is plain `fpgas-verify` until you edit it. It is a configuration file: an upgrade keeps your edit.
- From a checkout, without installing: `PYTHONPATH=verify/src python3 -m fpgas_online_verify --help`.

Identity and labels

Two commands say who the board is, for rpi-hwid[9]'s labels. Neither runs the check, publishes anything or records any state.

Command	Does
`fpgas-verify --identify` (also `--board B`, `fpgas-<board>-verify --identify`, `fpgas-acorn-debug identify`)	prints one identity document (JSON) for every board found
`fpgas-verify --label [--out F] [--list]`	reads the identity, then runs `rpi-hwid labels --this-host [--out F] [--list]` to make this Pi's and its board's labels

`--identify`:

- The document and every field in it are described in identity.md[10]. It is printed with sorted keys and an indent of 1.
- What it reads live:

Board	Read live
Acorn	the running build, device DNA and flash identity over BAR0 (the check's `pcie-bar0`), and the IDCODE and DNA over P1 JTAG (`jtag`)
Arty	the JTAG IDCODE and the device DNA (the device DNA[11])
NeTV2	the JTAG IDCODE, from the scan that finds it, and the device DNA
Fomu, TT FPGA	how the board was found (its USB serial)

A TT board's Tiny Tapeout fields[12] are not read live: they come from the boot report (see below).

- Nothing is loaded into a board and no flash is written. Some reads do change state on the way, and each is put back:
  - Acorn, BAR0: memory decoding is switched on in the board's PCI COMMAND register for the read and switched off again if it was off. If a kernel driver (litepcie) is bound to the board, it is unbound for the read and bound again afterwards. To read the flash's IDs (RDID and OTPR) and its SFDP header (Read SFDP), the SoC's SPI master registers and the flash's chip select are written. No other CSR of the SoC is written; `--identify` refuses any other write, so the SoC is never reset through `ctrl_reset`.
  - Acorn, P1 JTAG, and NeTV2: openFPGALoader (or openocd) drives the Pi's JTAG GPIOs. Their state is read with `pinctrl` first and put back afterwards; a pin that was an output goes back as an input. Without `pinctrl` neither the scan nor the DNA read is run.
  - A SIGTERM during the read (rpi-hwid sends one to a slow `--identify`, then SIGKILL) exits 143 after putting all of this back and letting go of the locks; a second SIGTERM meanwhile is ignored.
- Each board is read under its lock, as a check is. `--identify` waits at most 30 s for it; a board still in use then has its fields missing, for the reason `board busy`. A NeTV2 is also looked for under its lock, since finding it drives its JTAG pins; one that could not be looked for, its lock busy, is in the document all the same, with its fields missing for that reason, even when other boards were found (`--identify` then exits 1). `--identify` holds one board's lock at a time: the NeTV2's is let go after its scan and taken again for its read, so no two locks are ever taken in an order that could deadlock with rpi-hwid's (acorn < arty < netv2).
- A field only the boot check reads comes from the boot report, `/run/fpgas-online/verify.json`, when the board there is this one by a key no other board has: a TT board's `usb_serial`, the Arty's USB serial, the Acorn's PCI slot, or the device DNA. These are the Arty's and NeTV2's flash fields, which need a design loaded for openFPGALoader's SPI-over-JTAG bridge, and a TT board's Tiny Tapeout fields with `tinytapeout_note` or `tinytapeout_error`, which need the port `fpgas-tt.service` holds. Those fields are listed in the board's `from_report`. An IDCODE names a part, not a board, so a board known only by its IDCODE (a NeTV2 whose DNA could not be read) gets nothing from the report: the fields stay missing, for the reason `no board-unique match in the boot report`.
- The exit status is 0 when every board's identity is whole (no field missing, no `<field>_error`), 1 otherwise. Each missing field is printed on stderr with why. The document is printed either way; readers use it and ignore the exit status.

- For an Arty or a NeTV2, `--identify` always exits 1 for now: their labels need the flash's unique ID (`flash_uid`), and neither the boot check nor `--identify` reads it yet (see Not done yet[13]). Their IDCODE and device DNA, and any flash fields the boot report has for them, are still in the document.
- For a TT board, `--identify` exits 0 only when the boot report has the fields rpi-hwid's Tiny Tapeout label needs (`mcu`, `chip`, `demoboard`, `demoboard_version`, `sdk`, besides `usb_serial`). So it exits 1 when rpi-hwid was not installed at boot.

`--label`:

1. reads the identity, as `--identify` does;
2. releases every board's lock;
3. writes the document to `/run/fpgas-online/identity-<pid>.json`, a new file readable by root alone (it never writes over a file or through a link already there);
4. runs `rpi-hwid labels --this-host [--out F] [--list]` with `FPGAS_VERIFY_IDENTITY` set to that file;
5. deletes the file, whatever happened: an error, Ctrl-C, or a SIGTERM (which exits 143).

Its exit status is rpi-hwid's, or 128 + N when rpi-hwid was killed by signal N, as a shell reports it (137 for SIGKILL). It exits 2, saying why, when rpi-hwid is not installed or when the file cannot be written: run it as root. `--list` (with `--label`) lists the labels rpi-hwid would make; `--out F` is where rpi-hwid writes them. `--label` writes no report or state and publishes nothing, so `--report`, `--state` and `--no-publish` are refused with it (exit 2).

Nesting: rpi-hwid gets the board's identity by running `fpgas-verify --identify`. So that the inner run never waits on a lock the outer run holds, reads a board twice or starts rpi-hwid again, a run with `FPGAS_VERIFY_IDENTITY` set (an empty value counts as not set):

Mode	Does
`--identify` (and `fpgas-acorn-debug identify`)	checks that the file is an identity document with `identity_version` 1 and prints it unchanged, byte for byte. It takes no lock and runs no board code
a missing or bad file	an error (exit 1). It never falls back to reading the hardware
any other mode, `--label` included	refuses (exit 2)

rpi-hwid is a soft dependency:

- fpgas-verify finds the `rpi-hwid` command on `PATH` and runs it. It never imports rpi-hwid's Python code.
- Without it, `--label` exits 2 and says how to install it: `sudo apt install python3-rpi-hwid`, or `uv tool install 'rpi-hwid[labels]'` (Python 3.11 or newer).
- The `fpgas-online-verify` deb lists `python3-rpi-hwid` in Suggests. The Python package has a `labels` extra (`fpgas-online-verify[labels]`).
- `--identify` does not need rpi-hwid.

Reading the result

There is one result: **pass** or **fail**. A fail is named for its worst cause:

Result	Pass or fail	Means
`pass`	pass	every test passed, and the board and its flash are the ones recorded
`changed`	fail	a different board, or a different flash, from the recorded state[14]
`fail`	fail	a test failed, or the board runs a design that is not ours (`unconverted: …` for an Acorn on SQRL's image)
`missing`	fail	no board found (with `auto`: none of the installed boards)
`error`	fail	the check itself could not run: a missing tool, a damaged package, no configuration

- Only `pass` exits 0. Anything else also leaves `fpgas-verify.service` failed.
- The check goes on after a fault wherever it can, so the report lists every fault it found.
- The summary goes to stderr; at boot, to the journal (`journalctl -b -u fpgas-verify`). It has two parts:
  - every test in the order it ran, with its result, and a failed test's last 8 output lines;
  - for a check that did not pass, a plain conclusion, last, so it is what is left on the terminal: `RESULT:` and what it means; each board with how many tests passed, failed and were not run; a `fault:` line for each thing wrong that is no one test's (the board runs SQRL's image, say), a `failed:` line for each failed test with its reason, and a `not run:` line for the tests that did not run and why (a test that does not apply to the setup is `not run`, and is not a failure); then `What to do:`, chosen from the faults found; then where the JSON report is.
- A reason is always one line, however long, so it can be searched for as it is (Common failures).
- The JSON report is in `/run/fpgas-online/verify.json`. It is replaced whole (written beside itself and renamed), so a reader sees the old report or the new one, never part of one. A `--report` path that is a symlink, a device or a pipe is written through instead, not replaced.

pass: an Acorn on the Pi 5 setup (pi-sw2-p47, 2026-10-02T03:56:17Z; the summary its report gives):

```
$ sudo fpgas-verify --no-publish
fpgas-verify: pass (mode auto, auto: USB/PCI IDs)
  not published: --no-publish
  acorn cle-215+: pass
    pcie-link  pass
    pcie-bar0  pass
    rp1-pio    pass
    jtag       pass
    flash      pass
    ddr        pass
    p2-uart    pass
    p2-serial  pass
    scratch    pass
    p2-gpio    pass
    flash 0x000000 match
    flash 0x400000 match
  state recorded (first run) in /var/lib/fpgas-online/verify-state.json
```

fail, a first install on someone's own hardware: a Compute Blade with a CM5 and an Acorn CLE-101 still on SQRL's factory image (ps1's `pi16`, 2026-10-05; the summary its boot report gives). Two things are wrong, and neither is the installation: the card has not been converted, and on this setup the JTAG test cannot have its TMS pin (#127[15]).

```
$ sudo fpgas-verify --no-publish

********************************************************************************
*** FPGA VERIFY: FAIL ********************************************************
fpgas-verify: fail (mode acorn, configured: acorn)
  not published: --no-publish
  acorn cle-101: fail
    unconverted: runs SQRL's factory image, not the fpgas.online design
    pcie-link  pass
    rp1-pio    pass
    jtag       fail: P1 JTAG: openFPGALoader --detect failed (exit -6) before scanning the JTAG chain:
openFPGALoader: line-request.c:199: gpiod_line_request_set_values_subset: Assertion `request' failed.
        openFPGALoader: line-request.c:199: gpiod_line_request_set_values_subset: Assertion `request'
failed.
    pcie-bar0  not run: unconverted: runs SQRL's factory image, not the fpgas.online design
    flash      not run: unconverted: runs SQRL's factory image, not the fpgas.online design
    ddr        not run: unconverted: runs SQRL's factory image, not the fpgas.online design
    p2-uart    not run: the board does not run a known build
    p2-serial  not run: unconverted: runs SQRL's factory image, not the fpgas.online design
    scratch    not run: unconverted: runs SQRL's factory image, not the fpgas.online design
    p2-gpio    not run: J5 and H5 are not wired on the Compute Blade setup
  state recorded (first run) in /var/lib/fpgas-online/verify-state.json

RESULT: FAIL: a board did not pass.
  acorn cle-101: fail (2 tests passed, 1 failed, 7 not run)
    fault: unconverted: runs SQRL's factory image, not the fpgas.online design
    failed: jtag: P1 JTAG: openFPGALoader --detect failed (exit -6) before scanning the JTAG chain:
openFPGALoader: line-request.c:199: gpiod_line_request_set_values_subset: Assertion `request' failed.
    not run: pcie-bar0, flash, ddr, p2-serial, scratch: unconverted: runs SQRL's factory image, not the
fpgas.online design
    not run: p2-uart: the board does not run a known build
    not run: p2-gpio: J5 and H5 are not wired on the Compute Blade setup
What to do:
  * The Acorn still runs the image it was sold with, not the fpgas.online one,
    so only its PCIe link and its JTAG could be tested. It has to be converted
    once (the fpgas.online image loaded over JTAG, then written to its flash
    with fpgas-acorn-flash):
    https://github.com/fpgas-online/fpgas.online-test-designs/blob/main/docs/hardware/acorn-pcie-
programming.md
  * openFPGALoader could not have one of the JTAG pins, because a driver holds
    it (on a Compute Blade the serial port holds GPIO14, which is also the
    JTAG TMS wire). The check cannot test JTAG on such a host yet:
    https://github.com/fpgas-online/fpgas.online-test-designs/issues/127
  * To look at the acorn board yourself: sudo fpgas-acorn-debug --help (sudo
    apt install fpgas-online-acorn-debug)
  * What each message means:
    https://docs.fpgas.online/en/latest/verify/fpgas-verify.html#common-failures
The whole report, for a program to read (JSON): /run/fpgas-online/verify.json
********************************************************************************
```

fail, with two faults: the check run against the tests' fake Acorn (`tests/acorn_fakes.py`[16]), with its PCIe link at x2 and a JTAG TDI wire that does not carry:

```
$ sudo fpgas-verify --no-publish

********************************************************************************
*** FPGA VERIFY: FAIL ********************************************************
fpgas-verify: fail (mode auto, auto: USB/PCI IDs)
```

```
  not published: --no-publish
  acorn cle-215+: fail
    pcie-link  fail: link is x2, expected x1
    pcie-bar0  pass
    rp1-pio    pass
    jtag       fail: device DNA over JTAG 0x1 is not the one over BAR0 0x54b48664b04854: TDI (or the DNA
readout) is wrong
        - 0 -> 0x13636093
        - 1 -> 0xffffffff
        {"dna": "0x0000000000000001"}
    flash      pass
    ddr        pass
    p2-uart    pass
    p2-serial  pass
    scratch    pass
    p2-gpio    pass
    flash 0x000000 match
    flash 0x400000 match
  state recorded (first run) in /var/lib/fpgas-online/verify-state.json

RESULT: FAIL: a board did not pass.
  acorn cle-215+: fail (8 tests passed, 2 failed)
    failed: pcie-link: link is x2, expected x1
    failed: jtag: device DNA over JTAG 0x1 is not the one over BAR0 0x54b48664b04854: TDI (or the DNA
readout) is wrong
What to do:
  * To look at the acorn board yourself: sudo fpgas-acorn-debug --help (sudo
    apt install fpgas-online-acorn-debug)
  * What each message means:
    https://docs.fpgas.online/en/latest/verify/fpgas-verify.html#common-failures
The whole report, for a program to read (JSON): /run/fpgas-online/verify.json
********************************************************************************
```

fail: a NeTV2 whose DDR test fails (pi-sw1-p12, at boot; the summary its report gives):

```
$ journalctl -b -u fpgas-verify -o cat
********************************************************************************
*** FPGA VERIFY: FAIL ********************************************************
fpgas-verify: fail (mode auto, auto: probed (netv2))
  published to the fleet: `publish = on` in /etc/fpgas-verify/fleet.ini
  netv2 a7-35: fail
    uart       pass
    ddr        fail: the test exited 1
               Read: 0x40000000-0x401c0000 1.7MiB
               Read: 0x40000000-0x401e0000 1.8MiB
               Read: 0x40000000-0x40200000 2.0MiB
              bus errors:  256/256
              addr errors: 0/8192
              data errors: 524288/524288
              Memtest KO
            RESULT: FAIL — DDR memory test had failures
    spiflash   pass
  state recorded (first run) in /var/lib/fpgas-online/verify-state.json

RESULT: FAIL: a board did not pass.
  netv2 a7-35: fail (2 tests passed, 1 failed)
    failed: ddr: the test exited 1
What to do:
  * To look at the netv2 board yourself: sudo fpgas-netv2-debug --help (sudo
    apt install fpgas-online-netv2-debug)
  * What each message means:
    https://docs.fpgas.online/en/latest/verify/fpgas-verify.html#common-failures
The whole report, for a program to read (JSON): /run/fpgas-online/verify.json
********************************************************************************
```

missing: a host set up for an Arty, with none attached:

```
$ sudo fpgas-verify --no-publish; echo "exit $?"

********************************************************************************
*** FPGA VERIFY: MISSING ***************************************************
fpgas-verify: missing (mode arty, -)
  no Digilent Arty A7 found: this host is set up for one, and nothing else is looked for
  not published: --no-publish

RESULT: MISSING: no board was found.
  no Digilent Arty A7 found: this host is set up for one, and nothing else is looked for
What to do:
  * Check the board's power and cables. `sudo fpgas-arty-debug detect` looks
    for it again without running the tests.
  * What each message means:
    https://docs.fpgas.online/en/latest/verify/fpgas-verify.html#common-failures
The whole report, for a program to read (JSON): /run/fpgas-online/verify.json
********************************************************************************

exit 1
```

What each board’s check tests

Acorn

The Acorn check (`suite.py`[17], `check.py`[18], `links.py`[19], `bist.py`[20]) tests the board as it booted from its flash:

- It loads nothing, never writes the flash and never reconfigures the FPGA.
- It runs every test it can, in the order below; a fault in one does not stop the others.
- Nothing is sent to the BARs of a board whose PCI IDs are not ours.
- On ours, nothing past the identifier read is sent over BAR0 or the P2 UART unless that identifier is a build of the installed release. The CSR addresses then come from that build’s `csr.json`; one that puts a CSR the check uses outside the 64 KiB of BAR0 it maps is an `error`.

Which board it is, from the PCI IDs alone:

PCI IDs	Is	Result
`10ee:7021`, subsystem `1e24:021f` (CLE-215+) or `1e24:0101` (CLE-101)	the fpgas.online Acorn SoC	tested
`1e24:021f` or `1e24:0101` as vendor:device	an Acorn on SQRL's factory image	`fail`, `unconverted: …`; only `pcie-link`, `rp1-pio` and `jtag` run
`10ee:7011`	the vendor XDMA sample (an Acorn or a NeTV2)	`fail`, `unconverted: …`
`10ee:0666`	a PCIe Screamer running PCILeech	`fail`: fpgas.online has no test design for this board yet
`10ee:7021`, subsystem `10ee:0007`, class `070001`, with a BAR2	a stock Xilinx XDMA design (most likely a PicoEVB)	`fail`: fpgas.online has no test design for this board yet
any other Xilinx or SQRL ID	not a design we built	`fail`

Which setup the host is. The Acorn has two setups, each wired its own way:

Host (`/proc/device-tree/model`)	Setup	JTAG `--pins` (TDI:TDO:TCK:TMS)	GPIO chip	J5 / H5
`Raspberry Pi 5 Model B`	Pi 5 with the Waveshare HAT	`10:9:11:8`	`raspberrypi,rp1-gpio`	GPIO3 / GPIO4
`Raspberry Pi Compute Module 4`	Compute Blade	`2:3:4:14`	`brcm,bcm2711-gpio`	not wired
`Raspberry Pi Compute Module 5`	Compute Blade	`2:3:4:14`	`raspberrypi,rp1-gpio`	not wired

- Both use openFPGALoader's `libgpiod` cable, and the P2 UART is `/dev/ttyAMA0`.
- Any other host is an `error`; the tests that do not need the wiring still run.

The expected data, one copy each, in the repository and installed by `fpgas-online-acorn-tools`:

File	Holds	In the repository	Installed
`wiring.toml`	each setup's wiring: which Pi GPIO each P1/P2 signal lands on, the JTAG cable and pins, the UART, and which hosts are that setup	`docs/wiring/acorn/wiring.toml`[21]	`/usr/lib/python3/dist-packages/fpgas_online_verify/boards/acorn/data/`
`expected.toml`	the figures the board must meet: PCIe link speed and width per setup, XADC ranges, and the least DRAM write and read bandwidth per variant	`docs/wiring/acorn/expected.toml`[22]	the same

From a checkout, the check reads them from the repository.

The tests, in the order they run:

Test	Over	Passes when
`pcie-link`	sysfs	`current_link_speed` and `current_link_width` are the setup's (5.0 GT/s, x1)
`pcie-bar0`	BAR0	the operational build runs (the golden build means the operational slot did not boot), the flash identifies itself and its SFDP header can be read (a flash with no SFDP passes), the device DNA is neither all zeros nor all ones, and the XADC temperature and VCCINT, VCCAUX and VCCBRAM are in range
`rp1-pio`	the Pi's kernel	Pi 5 / CM5 (BCM2712) only, not run elsewhere: `/dev/pio0` is a character device that opens read-write, which openfpgaloader-rp1pio needs. No bootloader or `config.txt` setting is read. When it does not, the test also says whether `rp1_fw` and `rp1_pio` are loaded, and the kernel's own `rp1-pio` / RP1 firmware lines say why, e.g. `failed to contact RP1 firmware` on a bootloader rp1_pio cannot talk to (pi-sw2-p47 and p48, bootloader 2024/11/05, 2026-10-03)
`jtag`	P1	`openFPGALoader --detect` finds one device, the variant's part in any silicon version (the JTAG IDCODE[23]), and `openFPGALoader --read-dna` reads a device DNA that is neither all zeros nor all ones. When BAR0's DNA is good, the two must match; otherwise the JTAG DNA is not compared. The IDCODE read does not use TDI; the DNA read does
`flash`	BAR0	both 4 MiB slots (golden at `0x000000`, operational at `0x400000`), read whole with read opcodes only, hold the release's images
`ddr`	BAR0	after the BIOS console is read out, the DRAM BIST makes two passes over the whole DRAM: no errors, and write and read bandwidth at least the variant's minimum
`p2-uart`	P2	the UARTBone identifier at 1200 baud is BAR0's; at 921600 baud the identifier, DNA and XADC readings are right and the DNA is BAR0's. The link is left at 1200 baud
`p2-serial`	BAR0 and the Pi's GPIO	J2/K2, borrowed from the UART by the `p2_serial` switch, carry 0 and 1 both ways; the switch goes back to serial by itself; the UARTBone then answers with BAR0's identifier
`scratch`	BAR0 and P2	the `ctrl` scratch register holds two patterns written over each bridge; its value is put back
`p2-gpio`	BAR0 and the Pi's GPIO	Pi 5 setup only: J5/H5 carry 0 and 1 both ways, FPGA to Pi and Pi to FPGA
`power-cycle` (opt-in, after `pcie-bar0`)	BAR0	the FPGA restarted since the last check: see below

Checking an Acorn’s wiring

The Acorn’s check doubles as a wiring test: each of its tests uses a known set of wires between the card and the Pi, so which tests pass and what a failing one says point at the wire. The wiring itself (which wire goes to which pin, and the parts) is on the Acorn wiring page[24], generated from `docs/wiring/acorn/wiring.toml`[21]; the parts to have on the bench, as a list to tick off, are generated from the same table for a Raspberry Pi 5[25] and for a Compute Blade[26].

After a fresh boot

On a host whose root file system is in memory (a netbooted Pi with `overlayroot=tmpfs`, as the Compute Blades at ps1 are), what you install is gone at the next boot, and so is the boot check’s unit. After each boot, install and run by hand:

```
# 1. The two apt repositories the packages come from.
sudo install -d -m0755 /etc/apt/keyrings
curl -fsSL https://apt.fpgas.online/apt.gpg | sudo tee /etc/apt/keyrings/apt.gpg > /dev/null
echo "deb [signed-by=/etc/apt/keyrings/apt.gpg] https://apt.fpgas.online/$(. /etc/os-release; echo
$VERSION_CODENAME)/ ./" \
  | sudo tee /etc/apt/sources.list.d/apt.list
curl -fsSL https://fpgas.online/fpgas.online-fpga-tools/fpgas.online-fpga-tools.gpg \
  | sudo tee /etc/apt/keyrings/fpgas.online-fpga-tools.gpg > /dev/null
echo "deb [signed-by=/etc/apt/keyrings/fpgas.online-fpga-tools.gpg] https://fpgas.online/fpgas.online-
fpga-tools/$(. /etc/os-release; echo $VERSION_CODENAME)/ ./" \
  | sudo tee /etc/apt/sources.list.d/fpgas.online-fpga-tools.list

# 2. The Acorn's packages.
sudo apt update
sudo apt install fpgas-online-acorn

# 3. What is installed, and which board this host is set up to check.
fpgas-verify --list

# 4. The check. Nothing is sent anywhere unless a file on the host says `publish = on`; --no-publish makes
sure.
sudo fpgas-acorn-verify --no-publish
```

- The check never writes the card’s flash and never loads a design into the FPGA. It does drive the P1 and P2 wires, which is how it tests them, and puts the Pi’s pins back as it found them; on a converted card it writes the design’s scratch register and puts the old value back (with the opt-in power-cycle check on, it leaves a marker there); and it records what it found on this host (`/var/lib/fpgas-online/verify-state.json`).
- It exits 0 only for a pass. The summary is on the terminal; the same as JSON is in `/run/fpgas-online/verify.json`.
- To keep the packages across boots, they have to go into the image the host boots from; that is the host owner’s root image, not something these packages do.
- The check needs no bitstream file of yours. The images it compares the card’s flash with, and the design that converts a card, are installed with it by `fpgas-online-acorn-bitstreams`, in `/usr/share/fpgas-online/acorn-pcie/images/` (for a CLE-101: `acorn-cle-101-sqrl_acorn.bit` and the two flash images, converting a card[27]). No package installs a pin-id or loopback design for the Acorn; the check does not use one.

Which test uses which wire

Test	Wires it uses	A pass shows	Runs on a card still on SQRL's image
`pcie-link`	the M.2 slot	the card is seated and the PCIe link is at the setup's speed and width	yes
`jtag`	P1: TCK, TMS, TDO for the IDCODE read; TDI as well for the device DNA read	all four JTAG wires, and that the FPGA is the variant's part	yes
`pcie-bar0`	the M.2 slot	the fpgas.online design is running and answers over PCIe	no: the board gets `fault: unconverted: …`; this test, `p2-uart`, `p2-serial`, `p2-gpio`, `flash`, `ddr` and `scratch` are listed as `not run` (`pcie-link`, `jtag` and `rp1-pio` still run)
`p2-uart`	P2: K2 (FPGA transmit) to the Pi's RXD (GPIO15), J2 (FPGA receive) from the Pi's TXD (GPIO14)	the serial pair, in the right direction	no
`p2-serial`	the same two wires, driven and read as plain pins in both directions	each of J2 and K2 on its own, so a crossed pair or one open wire is told apart	no
`p2-gpio`	P2: J5 to GPIO3, H5 to GPIO4, in both directions	the two spare wires	no; and never on a Compute Blade, whose cable does not carry them
`rp1-pio`	no wire: `/dev/pio0` on a Pi 5 or CM5	nothing about the wiring (not run on other hosts)	yes
`flash`, `ddr`, `scratch`	no wire of the cable (`scratch` also uses the serial pair)	nothing about the wiring	no

So on a card that has not been converted yet, `pcie-link` and `jtag` are the wiring tests; the P2 wires can only be tested once the card runs the fpgas.online design (converting a card[27]).

From what it says to the wire

The failing line	Look at
`pcie-link fail: link is x2, expected x1` (or a speed)	the M.2 seat; or the setup's expected figures are not this host's
`jtag fail: no device on the P1 JTAG chain`	the P1 cable is not plugged in, or TCK, TMS or TDO is open or on the wrong pin
`jtag fail: P1 JTAG chain has …, expected one …`	another device answers, or the card is not the variant the host expects
`jtag fail: device DNA over P1 JTAG reads 0x0: the DNA port is not being read` (or `reads 0x1fffffffffffff`), `openFPGALoader --read-dna` `read no device DNA over P1 JTAG`, or `device DNA over JTAG … is not the one over BAR0 …`	TDI: the IDCODE read worked without it
`jtag fail: … GPIO14 (TMS) is held by … (uart0): the kernel does not hand out a pin that is held …`	not a wire: on a Compute Blade the serial port has the TMS pin (below)
`p2-uart fail: no UARTBone reply on /dev/ttyAMA0 (P2 K2/J2)`	the serial pair: open, or crossed; `p2-serial` says which
`p2-serial fail: J2 -> GPIO14: the FPGA drove 1, the Pi read 0; K2 -> GPIO15: the FPGA drove 0, the Pi read 1; …` with the `01` and `10` lines swapped and `00` and `11` right	J2 and K2 are **crossed**: swap the two wires at the Pi end (on a Compute Blade the J2 wire carries the 470 Ω resistor: the resistor stays with J2)
`p2-serial` or `p2-gpio` naming one ball only	that one wire is **open**, or on the wrong pin. Before the FPGA drives, the test sets the Pi's pull against the level to come, so an open wire reads the opposite of what was driven; GPIO3 (the Pi 5's J5) has a pull-up of its own on the Pi, so an open J5 wire reads 1 whatever is driven
`p2-gpio fail: J5 -> GPIO3: …; H5 -> GPIO4: …` with the `01` and `10` lines swapped and `00` and `11` right	J5 and H5 are crossed

`p2-serial` and `p2-gpio` print what was driven and what was read, eight lines for two wires. The two digits are the two balls: the right-hand digit is J2 (or J5), the left-hand one K2 (or H5).

A crossed pair: read on acorn-olive at Welland, 4 October 2026, whose P2 pairs were both crossed. `FPGA drives 01` raises J2, which should arrive on GPIO14; it arrives on GPIO15:

```
    p2-serial  fail: J2 -> GPIO14: the FPGA drove 1, the Pi read 0; K2 -> GPIO15: the FPGA drove 0, the Pi
read 1; J2 -> GPIO14: the FPGA drove 0, the Pi read 1; K2 -> GPIO15: the FPGA drove 1, the Pi read 0;
GPIO14 -> J2: the Pi drove 1, the FPGA read 0; GPIO15 -> K2: the Pi drove 0, the FPGA read 1; GPIO14 ->
J2: the Pi drove 0, the FPGA read 1; GPIO15 -> K2: the Pi drove 1, the FPGA read 0; the UARTBone does not
answer on /dev/ttyAMA0 after the switch (no fpgas.online SoC answered at 1200 baud after a break)
        FPGA drives 00: Pi reads GPIO14=0 GPIO15=0
        FPGA drives 01: Pi reads GPIO14=0 GPIO15=1
        FPGA drives 10: Pi reads GPIO14=1 GPIO15=0
        FPGA drives 11: Pi reads GPIO14=1 GPIO15=1
        Pi drives 00: FPGA reads 00
        Pi drives 01: FPGA reads 10
        Pi drives 10: FPGA reads 01
        Pi drives 11: FPGA reads 11
```

One open wire: read on acorn-sycamore at Welland the same day, whose J5 wire did not reach GPIO3. GPIO3 reads 1 whatever the FPGA drives (the Pi's own pull-up on GPIO3 wins over the test's pull-down; on GPIO4 an open wire would read the opposite of what was driven), and on this card the FPGA read J5 as 1 whatever the Pi drove; H5 and GPIO4 follow each other:

```
    p2-gpio    fail: J5 -> GPIO3: the FPGA drove 0, the Pi read 1; J5 -> GPIO3: the FPGA drove 0, the Pi
read 1; GPIO3 -> J5: the Pi drove 0, the FPGA read 1; GPIO3 -> J5: the Pi drove 0, the FPGA read 1
        FPGA drives 00: Pi reads GPIO3=1 GPIO4=0
        FPGA drives 01: Pi reads GPIO3=1 GPIO4=0
        FPGA drives 10: Pi reads GPIO3=1 GPIO4=1
        FPGA drives 11: Pi reads GPIO3=1 GPIO4=1
        Pi drives 00: FPGA reads 01
        Pi drives 01: FPGA reads 01
        Pi drives 10: FPGA reads 11
        Pi drives 11: FPGA reads 11
```

A correctly wired pair reads back what was driven: `FPGA drives 01: Pi reads GPIO14=1 GPIO15=0`, `Pi drives 01: FPGA reads 01`, and so on for every pattern.

On a Compute Blade

What has been run on a Compute Blade, and what has not, as of 5 October 2026:

	State
Installing the packages and running the check (Raspberry Pi OS trixie, CM5)	run at ps1: the first failing example under Reading the result is that host's result, taken with 0.0.post1100, before the check named the pin's holder
`pcie-link`	run at ps1: passes (5.0 GT/s, x1)
`jtag` with the serial port on, kernel 6.18	run at ps1: **cannot work**. TMS is GPIO14, which is also the serial port's TX; that kernel does not lend a pin a driver has, and the serial driver cannot be detached while the system runs. From 0.0.post1111 the test fails saying so, without running the tool (#127[15])
`jtag` with the serial port off	**not yet run by us on this hardware**
`jtag` under kernel 6.12, serial port on	recorded as working on one ps1 blade (`--pins 2:3:4:14`), before these packages existed; not run with them
The `p2-uart` and `p2-serial` tests	**not yet run by us on this hardware**: they need a converted card
Converting a card on a Compute Blade	**not yet run by us on this hardware**; the written steps[27] are for the Pi 5 setup
A Compute Blade that passes the whole check	**not yet seen**
The `p2-serial` test on a blade whose J2 wire has no 470 Ω resistor (pi20 as wired on 5 October 2026: the pair on Extension Port pins 9 and 10)	**not yet run by us on this hardware**. From the code: while it runs to its end or raises an error, the test never has both ends of a wire driving at once (the Pi's pins are made inputs before the FPGA drives, and the FPGA's outputs are switched off before the Pi drives), so it does not rely on the resistor. What the resistor guards against is a design that drives J2 while JTAG or the serial port drives GPIO14; the fpgas.online design leaves J2 an input except while the host has switched J2/K2 to GPIO mode and enabled J2's output, which is what this test does, with the Pi's GPIO14 an input at that moment

JTAG and the serial pair share GPIO14 on a Compute Blade (J2 reaches it through 470 Ω, so JTAG wins electrically). Under kernel 6.18 they cannot both be had from one boot: with the header's serial port on, the kernel keeps GPIO14 for it. The configuration we expect to work for JTAG, and with it for converting a card, is the header's serial port off at boot. **Not yet run by us on this hardware**, and Raspberry Pi's documentation does not say that it frees GPIO14 on a Compute Module 5:

- in `config.txt`, the line `enable_uart=0`, written out (Raspberry Pi's documentation gives the default as 1 when the primary serial port is a PL011; we have not seen it left unset on a Compute Blade); and if the port is switched on by a `dtoverlay=uart0…` or `dtparam=uart0` line, that line has to go instead;
- in `cmdline.txt`, the word `console=serial0,115200` deleted from the one line, if it is there.

On a netbooted host those files may not be on the host: where the firmware reads `config.txt` and `cmdline.txt` from is not something we have read, and if it fetches them from a boot server, they are changed there, by whoever keeps that server, for every host booting from that tree. At ps1 the blades' root is the gateway's `/srv/nfs/rpi/trixie/root` (read from the kernel command line of pi16 and pi20, 5 October 2026); where the firmware's copies come from was not read by us.

Then check that the pin is free (the two commands below) before trying JTAG. With the serial port off, `/dev/ttyAMA0` is not there, so the `p2-uart`, `p2-serial` and `scratch` tests cannot pass in that boot; what a Compute Blade's check should count as its result in each of the two configurations is not settled.

To see who has the JTAG pins on a host, without running anything on the card:

```
pinctrl get 2,3,4,14,15     # the function each pin is switched to
gpiodetect                  # the header's chip: `pinctrl-rp1` on a CM5, `pinctrl-bcm2711` on a CM4
gpioinfo -c gpiochip0 | grep -E 'line +(2|3|4|14):'   # with that chip's name (gpiod 2; gpiod 1: `gpioinfo
gpiochip0`)
# a line shown with a consumer (`consumer="kernel"`) or `[used]` is one the kernel will not hand out
```

Common failures

It says	Meaning, and what to do
`missing: no <board> found: this host is set up for one…`	the board is not on USB/PCI (or JTAG). Check power and cables. A Fomu that has run a design needs a power cycle
`missing: …; a Fomu EVT was found on this host by an earlier check and is not there now: …`	the Fomu may still be plugged in and working: the last check that found it loaded its test design, which has no USB, and a reboot does not bring it back. If it is plugged in, power-cycle the Pi (its power or its PoE port), not a reboot. The check cannot tell this from a Fomu that was unplugged
`missing: none of the installed boards … was found`	nothing attached. Expected on a Pi with no FPGA, and still a fail
`error: the board did not say which Tiny Tapeout board it is, so no test was run and nothing was loaded`	the demo board could not be asked: the rest of the reason says why (rpi-hwid not installed, its `main.py` changed, its SDK did not start, rpi-hwid could not read it). Which Tiny Tapeout board it is[28]
`fail: wiring not run: the Pmod wiring test is not yet part of the boot check, …`	a demo board with a Tiny Tapeout chip. Nothing is known to be wrong with it: the check cannot test its cabling to the Pi yet, and does not pass a board untested. Its `sdk` test and identity are in the report. The `sdk` test[29]
`fail: sdk fail: a tt06 chip needs SDK 2.0.x on an RP2040, and the board runs SDK …` / `no SDK release is recorded as supporting …`	the Tiny Tapeout SDK on the demo board is not a release known to work with the chip it carries. The firmware is installed by whoever looks after the board; the check writes nothing to it. The `sdk` test[29]
`fail: a Raspberry Pi RP2 is on USB but is not running the Tiny Tapeout firmware`	the demo board's microcontroller is in its USB boot loader: power-cycle the board; if it comes back the same, its firmware is gone
`error: … is not installed`	a tool is missing: `mpremote` (bookworm: bookworm-backports), openocd, openFPGALoader
`error: no FPGA board is configured` / `conflicting fpga-board settings`	install one board's package, or fix `/etc/fpgas-verify/*.ini`

It says	Meaning, and what to do
`fail: power-cycle fail: the FPGA has not restarted since an earlier boot's check`	the Acorn kept its configuration across the Pi's restart (not seen after a soft reboot on the one Pi 5 measured): power-cycle the Pi (PoE). The power-cycle check[8]
`error: power-cycle-check is '…'; it is on or off` / `conflicting power-cycle-check settings`	fix `/etc/fpgas-verify/*.ini`
`error: … does not match its manifest` / `manifest.json is missing`	`sudo apt install --reinstall fpgas-online-<board>-bitstreams`
`fail: loading it failed (exit N)`	the programmer could not load the design: JTAG wiring, cable, or programmer support
`fail: python3 did not finish within 300 s`	the design never printed what the test waits for: wrong UART, or the design does not run
`fail: the test exited 1`	the test failed; its last lines are in the summary. `fpgas-<board>-debug test <test>` shows all of it
`fail: N/24 pins match expected wiring` (TT FPGA) or `N/18` (Arty)	the Pmod HAT cabling differs from the board's expected map: the table above that line shows each wire, what was expected on it and what was heard. The line after it says how many of the cabling's signal wires the test covers: all 24 on a TT FPGA board; 18 of 24 on an Arty, whose design cannot test the six wires on the three Pi pins two Pmods share
`fail: … is an XC7A100T, not the a7-35's XC7A35T …` / `P1 JTAG chain has … expected one …`	the JTAG IDCODE is not the variant's part: the wrong board, or the wrong `--variant`
`fail: the JTAG chain has N devices (…), not one`	more than the board's FPGA answers on its JTAG chain (an Arty or a NeTV2): another device wired into it, or a fault on the cable
`fail: unconverted: …`	an Acorn on SQRL's factory image (or the XDMA sample): convert it (acorn-pcie-programming.md[27])
`fail: … is not a design we built`	a Xilinx PCIe design the Acorn check does not know; its flash is not read
`fail: … has no test design for this board yet`	a Xilinx PCIe board that is not an Acorn (a PCIe Screamer, a PicoEVB)
`fail: running the golden image`	the Acorn's operational slot did not boot; it fell back to golden
`fail: link is x2, expected x1`	the Acorn's PCIe link is not the setup's (`expected.toml`)

It says	Meaning, and what to do
`fail: no device on the P1 JTAG chain` / `no UARTBone reply on /dev/ttyAMA0`	an Acorn's JTAG or P2 UART cable is off or miswired
`fail: P1 JTAG could not be probed: GPIO14 (TMS) is held by 1f00030000.serial (uart0): the kernel does not hand out a pin that is held…`	an Acorn's JTAG pin is in use by another driver, so the scan was not tried. Seen on a Compute Blade with a CM5 (kernel 6.18): TMS and the serial port's TX are both GPIO14, and with the serial port on the kernel does not lend the pin. JTAG cannot be tested there while the serial port is on; booting with it off should free the pin, which is not yet confirmed on hardware (#127[15])
`fail: … gpiod_line_request_set_values_subset: Assertion 'request' failed`	openFPGALoader could not have one of the JTAG pins, because a driver holds it. Seen on a Compute Blade with a CM5 (kernel 6.18), where the serial port holds GPIO14, which is also JTAG TMS: the JTAG test cannot run there yet (#127[15])
`fail: openFPGALoader printed no raw IDCODE scan (needs --verbose-level 2 output)`	the tool's version, not the board: openFPGALoader exited 0 but printed no `- 0 -> 0x...` lines at `--verbose-level 2`, so it is older than v0.9.0. Its last lines are in the report's `output`
`fail: … failed (exit N) before scanning the JTAG chain: …`	the scan tool exited with an error and printed no scan: the cable or gpiochip would not open, say. The reason ends with its last line of output; more is in the report's `output`
`fail: … exited N reading the IDCODE` / `openFPGALoader --detect exited N …`	the IDCODE scan reported an error, even if it printed an IDCODE; its last lines are in the report's `output`
`fail: device DNA over P1 JTAG reads 0x…: the DNA port is not being read`	the DNA read over JTAG is all zeros or all ones, which is no chip's DNA. The IDCODE read worked without TDI, so check the P1 TDI wire first; then run `openFPGALoader --read-dna` by hand
`fail: device DNA over JTAG reads 0x…: the DNA port is not being read`	an Arty's or NeTV2's DNA read is all zeros or all ones. Its IDCODE scan works without TDI, so check TDI first (the NeTV2's GPIO27); then run `openFPGALoader --read-dna` by hand
`fail: openFPGALoader --read-dna read no device DNA (exit N): …`	an Arty's or NeTV2's DNA read failed; the reason ends with its last line. An openFPGALoader older than 0.13.0 (Debian bookworm's) has no `--read-dna`: install fpgas.online's (Installing[30])
`fail: device DNA over JTAG … is not the one over BAR0`	the P1 TDI wire does not carry, or the DNA readout is wrong. Only a good BAR0 DNA is compared: one of all zeros or all ones is pcie-bar0's own fault (`device DNA over BAR0 reads 0x0: the DNA port is not being read`)
`fail: J5 -> GPIO3: the FPGA drove 0, the Pi read 1` (or the other way)	a P2 spare wire is cut or miswired
`fail: K2 -> GPIO15: …` / `GPIO14 -> J2: …`	a P2 serial wire is cut or miswired

It says	Meaning, and what to do
`fail: DRAM write … MB/s, below the … MB/s expected` / `… words wrong in the … half`	the DRAM is slow or broken; `selftest.py` shows more
`error: this host (…) is not an Acorn setup in wiring.toml`	an Acorn on a host neither setup has: add the host to `wiring.toml` if it is a real setup
`changed`	the board or its flash differs from the recorded state. Meant it? `sudo fpgas-verify --update`

Links in this chapter

1. https://docs.fpgas.online/en/latest/verify/goals.html
2. https://github.com/fpgas-online/fpgas.online-test-designs/tree/main/verify
3. https://github.com/fpgas-online/fpgas.online-test-designs/blob/main/docs/plans/2026-09-26-fpgas-online-verify-design.md
4. https://github.com/fpgas-online/fpgas.online-test-designs/blob/main/docs/verify-hardware.md
5. https://docs.fpgas.online/en/latest/verify/fpgas-verify.html#how-verify-is-used-in-fpgas-online
6. https://docs.fpgas.online/en/latest/verify/fpgas-verify.html#current-results
7. https://docs.fpgas.online/en/latest/verify/fpgas-verify.html#events
8. https://docs.fpgas.online/en/latest/verify/fpgas-verify.html#the-acorn-s-power-cycle-check-opt-in
9. https://github.com/mithro/rpi-hwid
10. https://docs.fpgas.online/en/latest/verify/identity.html
11. https://docs.fpgas.online/en/latest/verify/fpgas-verify.html#the-device-dna
12. https://docs.fpgas.online/en/latest/verify/identity.html#tiny-tapeout-fields
13. https://docs.fpgas.online/en/latest/verify/fpgas-verify.html#not-done-yet
14. https://docs.fpgas.online/en/latest/verify/fpgas-verify.html#the-report-and-the-recorded-state
15. https://github.com/fpgas-online/fpgas.online-test-designs/issues/127
16. https://github.com/fpgas-online/fpgas.online-test-designs/blob/main/tests/acorn_fakes.py
17. https://github.com/fpgas-online/fpgas.online-test-designs/blob/main/verify/src/fpgas_online_verify/boards/acorn/suite.py
18. https://github.com/fpgas-online/fpgas.online-test-designs/blob/main/verify/src/fpgas_online_verify/boards/acorn/check.py
19. https://github.com/fpgas-online/fpgas.online-test-designs/blob/main/verify/src/fpgas_online_verify/boards/acorn/links.py
20. https://github.com/fpgas-online/fpgas.online-test-designs/blob/main/verify/src/fpgas_online_verify/boards/acorn/bist.py
21. https://github.com/fpgas-online/fpgas.online-test-designs/blob/main/docs/wiring/acorn/wiring.toml
22. https://github.com/fpgas-online/fpgas.online-test-designs/blob/main/docs/wiring/acorn/expected.toml
23. https://docs.fpgas.online/en/latest/verify/fpgas-verify.html#the-jtag-idcode
24. https://docs.fpgas.online/en/latest/boards/acorn/wiring.html
25. https://github.com/fpgas-online/fpgas.online-test-designs/blob/main/docs/wiring/acorn/generated/acorn-pi5-bom.md
26. https://github.com/fpgas-online/fpgas.online-test-designs/blob/main/docs/wiring/acorn/generated/acorn-blade-bom.md
27. https://docs.fpgas.online/en/latest/boards/acorn/pcie-programming.html
28. https://docs.fpgas.online/en/latest/verify/fpgas-verify.html#which-tiny-tapeout-board-it-is
29. https://docs.fpgas.online/en/latest/verify/fpgas-verify.html#the-sdk-test
30. https://docs.fpgas.online/en/latest/verify/fpgas-verify.html#installing

Bootloader EEPROM: upgrade and lock

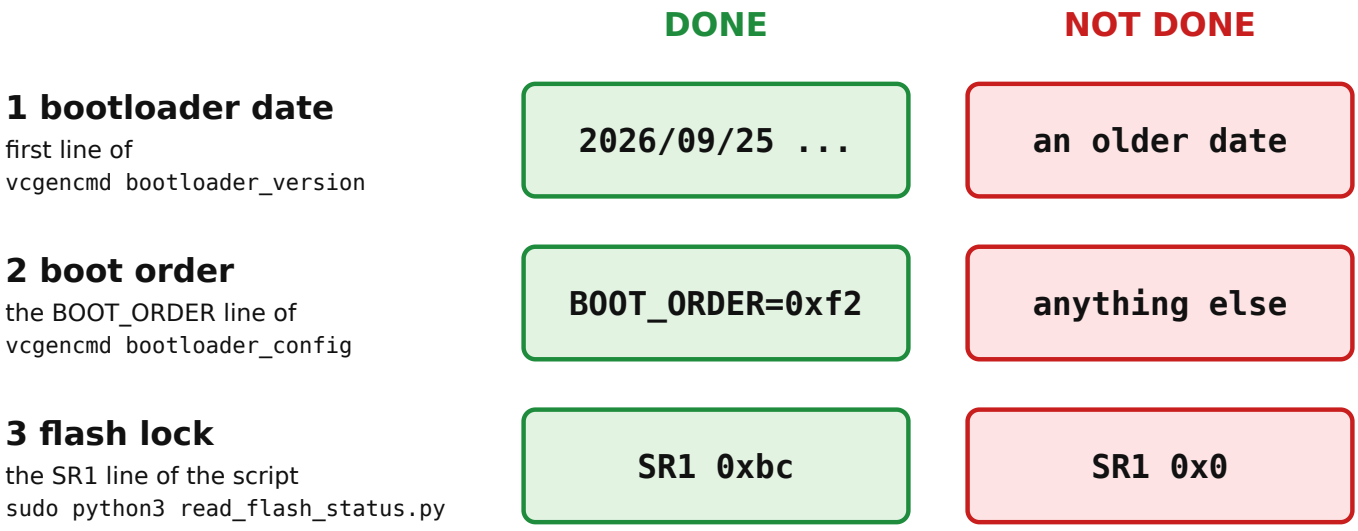
Every Raspberry Pi since the Pi 4 keeps its bootloader and the bootloader's settings in a small flash chip on the board, the bootloader EEPROM. On a fleet Pi that chip is locked, so that nobody with root on the Pi can change how it boots; a new bootloader therefore cannot simply be installed. Pick the part you need:

- **Check a Pi 5**: is its bootloader the fleet's, and is it locked?
- **Upgrade a locked Raspberry Pi 5**: six steps, for someone with the Pi in hand.
- **A Compute Module or a Compute Blade**: how to tell whether it needs anything (the two blades we read need nothing). There are no upgrade steps for it on this page.
- **What is behind this page**: what has been run and measured, what is not known, what does not work.

Check a Pi 5 (nothing is changed)

Three reads, for a **Raspberry Pi 5**. (This page has no check and no upgrade steps for a Pi 4.)

Is this Pi 5 done? Three reads, three answers



All three DONE: nothing to do.
Date or boot order NOT DONE: upgrade it (next part).
ONLY the lock NOT DONE (SR1 0x0): do NOT upgrade. Pull the network cable, plug it in again on the Pi's normal port, wait 2 minutes, read again.
Still 0x0: unplug the Pi and tell whoever runs the site.
A NEWER date, another SR1 value, or a jedec line that is not ef4015: STOP. This page does not cover that Pi; change nothing on it.

Open the web site of the place the Pi is at (for the Welland fleet, https://welland.fpgas.online), find this Pi in its list of boards and open the Pi's page. The web terminal on that page logs you in as user `pi`. (With ssh instead: `ssh pi@` followed by the address on that page; the same page shows the password.) Run:

```
$ vcgencmd bootloader_version
2026/09/25 15:59:09
version 1accd6659983613cd104973ce2432da14f1a7682 (release)
timestamp 1790351949
update-time 1791022751
capabilities 0x0000007f
$ vcgencmd bootloader_config
[all]
BOOT_UART=1
WAKE_ON_GPIO=1
POWER_OFF_ON_HALT=0
BOOT_ORDER=0xf2
NET_INSTALL_AT_POWER_ON=0
```

That is what a Pi that is done prints. Read 1 is the first line of the first command. Read 2 is the `BOOT_ORDER` line of the second.

Read 3 needs a small script on the Pi (a fleet Pi forgets it at every restart, so put it there each time). In the terminal on the Pi, type `cat > read_flash_status.py`, press Enter, paste the text below, press Enter, and then press Ctrl-D on that empty line. Nothing is printed when it worked.

```
# SPDX-License-Identifier: Apache-2.0
"""Print the part and the status registers of a Pi 5's bootloader flash.

    sudo python3 read_flash_status.py

Read commands only (9Fh JEDEC id, 05h SR1, 35h SR2, 15h SR3):
nothing on the flash changes.
"""
import array
import ctypes
import fcntl
import os
import struct


def xfer(fd, tx):
    txb = ctypes.create_string_buffer(bytes(tx), len(tx))
    rxb = ctypes.create_string_buffer(len(tx))
    msg = struct.pack("QQIIHBBBBBB",
                      ctypes.addressof(txb), ctypes.addressof(rxb),
                      len(tx), 1000000, 0, 8, 0, 0, 0, 0, 0)
    # SPI_IOC_MESSAGE(1)
    fcntl.ioctl(fd, 0x40206B00, array.array("B", msg), True)
    return rxb.raw


fd = os.open("/dev/spidev10.0", os.O_RDWR)  # the Pi 5's bootloader flash
print("jedec", xfer(fd, [0x9F, 0, 0, 0])[1:].hex())
for name, cmd in (("SR1", 0x05), ("SR2", 0x35), ("SR3", 0x15)):
    print(name, hex(xfer(fd, [cmd, 0])[1]))
```

Then run it. It only reads. (With ssh from a computer that has the script as a file, `downloaded here`[1], one command does both: `ssh pi@ADDRESS sudo python3 - < read_flash_status.py`.)

```
$ sudo python3 read_flash_status.py
jedec ef4015
SR1 0xbc
SR2 0x2
SR3 0x60
```

Read 3 is the `SR1` line: `0xbc` is locked, `0x0` is not locked. If the `jedec` line is anything other than `ef4015`, this is a flash chip this page does not cover: stop, and write nothing to that Pi.

Do not use `sudo rpi-eeprom-update` for this check: it prints “up to date” on Pis that are not on the fleet’s bootloader.

Reading BOOT_ORDER: start at the LAST digit and go left

0xf2: the fleet's Pi 5 setting. Network only.

f	2
start again: then	network: tried 1st

0xf12: network, then an SD card. Not the fleet's.

f	1	2
start again: then	SD card: tried 2nd	network: tried 1st

0xf2461: SD card, NVMe, USB, then network. Not the fleet's.

f	2	4	6	1
start again: then	network: tried 4th	USB: tried 3rd	NVMe: tried 2nd	SD card: tried 1st

Upgrade a locked Raspberry Pi 5

For a Raspberry Pi 5 whose check says NOT DONE. (How far each step has been tried by us is listed under What has been run.)

What you need on the bench

1 The Pi 5	out of its case, underside reachable
2 A microSD card	any size; everything on it is erased
3 A Linux computer	card reader, internet, git, python3, sha256sum
4 A soldering iron and solder	OR fine tweezers or a short wire to hold
5 Desoldering braid	if you solder: to take the bridge off again
6 The Pi's own network cable	on its switch port; it also powers the Pi
7 A multimeter	optional: to check the bridge

Step 1: make the card

The Pi stays plugged in and running during this step.

The finished card: these four files at the top level, nothing else

microSD card (one FAT32 partition)

recovery.bin	104314 bytes	the program the Pi runs from the card
pieeprom.bin	2097152 bytes	the 2026/09/25 bootloader with the fleet's settings
pieeprom.sig	80 bytes	the checksum of pieeprom.bin
config.txt	23 bytes	one line: eeprom_write_protect=0

On the computer, get Raspberry Pi’s bootloader files, at the version these steps were written with:

```
$ git clone https://github.com/raspberrypi/rpi-eeprom
$ cd rpi-eeprom
$ git checkout a72213d02af27f2dc3b86f584988aec407e46378
$ mkdir card
```

Make the settings file. Paste this whole block as it is:

```
$ cat > boot.conf <<'EOF'
[all]
BOOT_UART=1
WAKE_ON_GPIO=1
POWER_OFF_ON_HALT=0
BOOT_ORDER=0xf2
NET_INSTALL_AT_POWER_ON=0
EOF
```

Make the four files:

```
$ ./rpi-eeprom-config --config boot.conf --out card/pieeprom.bin \
      firmware-2712/default/pieeprom-2026-09-25.bin
$ ./rpi-eeprom-digest -i card/pieeprom.bin -o card/pieeprom.sig
$ cp firmware-2712/default/recovery.bin card/
$ echo eeprom_write_protect=0 > card/config.txt
```

Check them. The first command must print `[all]` and the five settings, and the second must list the four files with these sizes:

```
$ ./rpi-eeprom-config card/pieeprom.bin
[all]
BOOT_UART=1
WAKE_ON_GPIO=1
POWER_OFF_ON_HALT=0
BOOT_ORDER=0xf2
NET_INSTALL_AT_POWER_ON=0
$ wc -c card/*
     23 card/config.txt
2097152 card/pieeprom.bin
     80 card/pieeprom.sig
 104314 card/recovery.bin
2201569 total
```

If `BOOT_ORDER` there is not `0xf2`, stop and make `boot.conf` again: a card with another boot order leaves a Pi that does not start from the network.

Now the card. Find its name: run `lsblk` before you put the card in the reader and again after. The line that is new is the card. In this example (yours will differ) it is `sdb`:

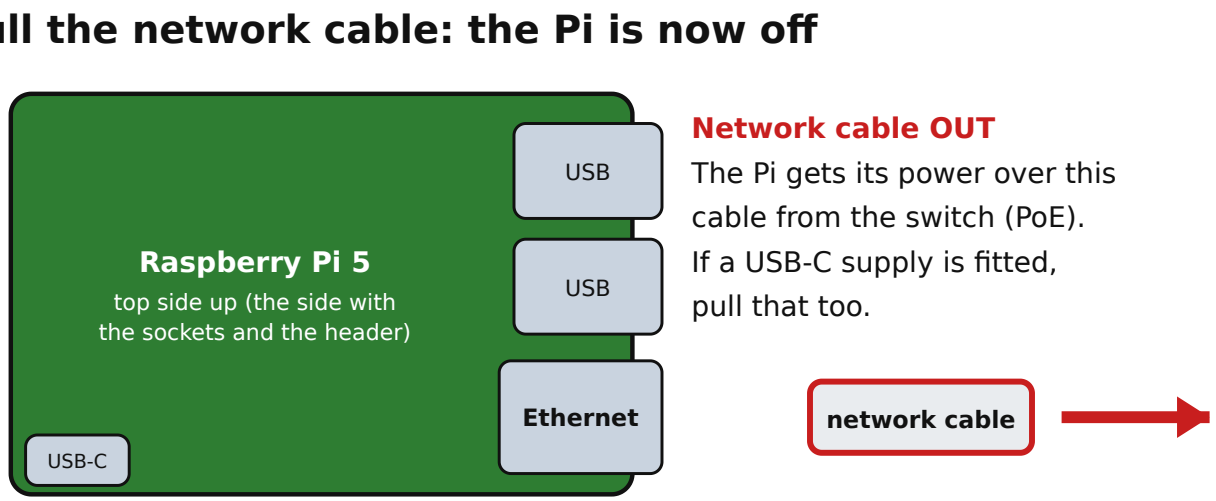
```
$ lsblk -d -o NAME,SIZE,MODEL
NAME      SIZE MODEL
nvme0n1 476.9G Samsung SSD 980
$ lsblk -d -o NAME,SIZE,MODEL
NAME      SIZE MODEL
nvme0n1 476.9G Samsung SSD 980
sdb      29.7G STORAGE DEVICE
```

The next commands erase everything on the card. Put your card’s name in the first line in place of `sdX`, and check it twice: the wrong name erases another disk. (For a name like `mmcblk0` the second line is `PART=${DISK}p1`.) The `umount` lets go of the card if your desktop opened it by itself; “not mounted” from it is fine.

```
$ DISK=/dev/sdX
$ PART=${DISK}1
$ sudo umount ${DISK}*
$ sudo wipefs -a $DISK
$ echo 'type=c' | sudo sfdisk $DISK
$ sudo mkfs.vfat -F 32 $PART
$ sudo mount $PART /mnt
$ sudo cp card/* /mnt/
$ ls /mnt
config.txt  pieeprom.bin  pieeprom.sig  recovery.bin
$ sudo umount /mnt
```

Take the card out of the reader.

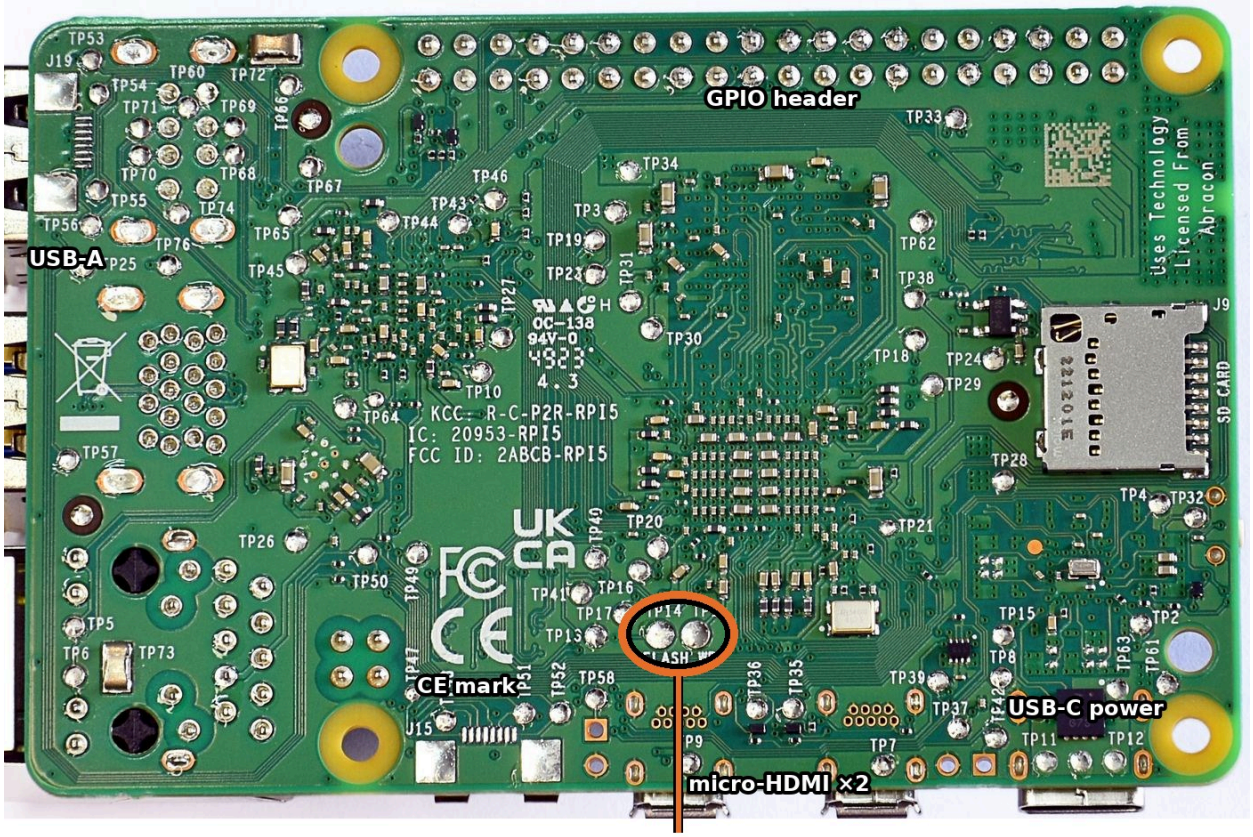
Step 2: network cable out



1. Pull the Pi's network cable. If the Pi also has a USB-C power supply, pull that too. The Pi is now off.
2. Take the Pi out of whatever holds it. Turn it over like the page of a book: the USB sockets, on your right in this drawing, end up on your left.

Step 3: join the two FLASH WP pads

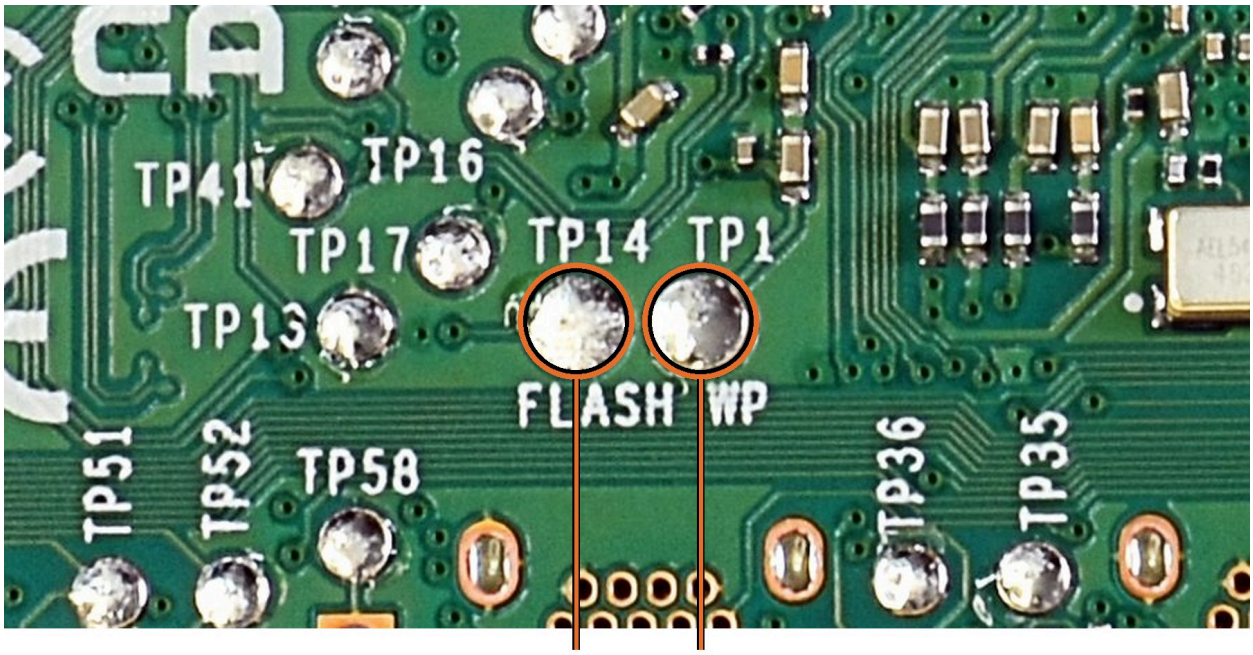
The Pi is off (step 2). This is its underside, USB-A sockets on the left, GPIO header along the top:



TP14 and TP1: the two pads marked FLASH WP

right of the CE mark, above the two micro-HDMI sockets

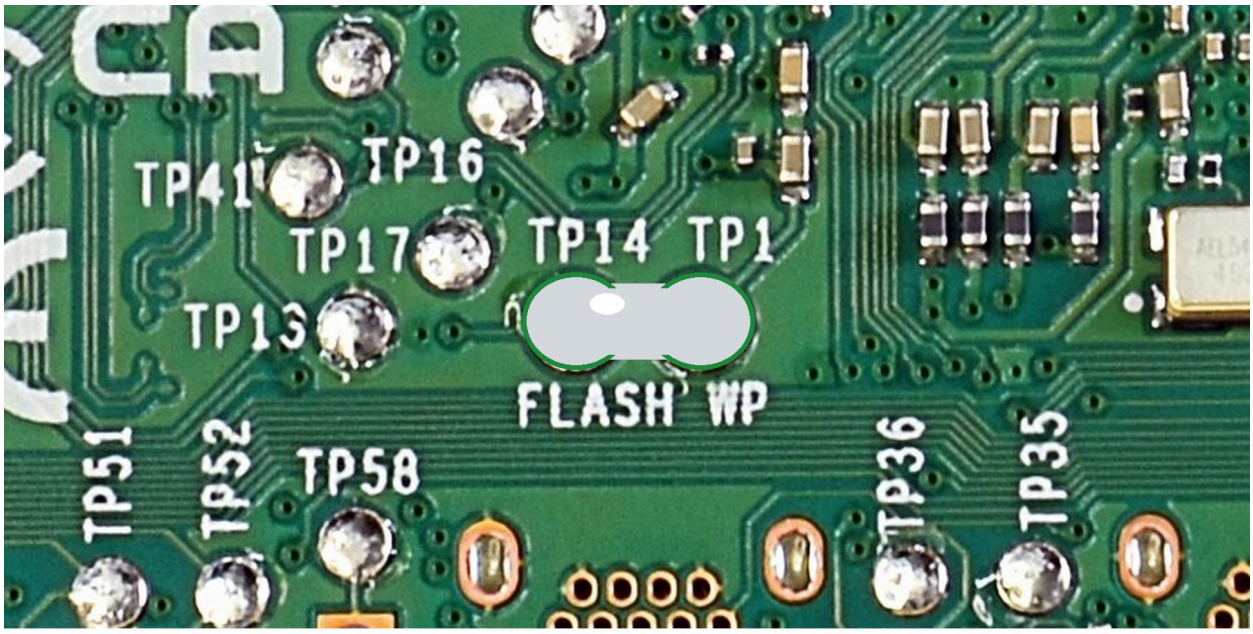
Photo: Suyash Dwivedi, Wikimedia Commons[2], CC BY-SA 4.0[3]; annotated, same licence.



TP14 (left): the flash chip's write-protect line **TP1 (right): power**

The board prints TP14 TP1 above the pads and FLASH WP below them.

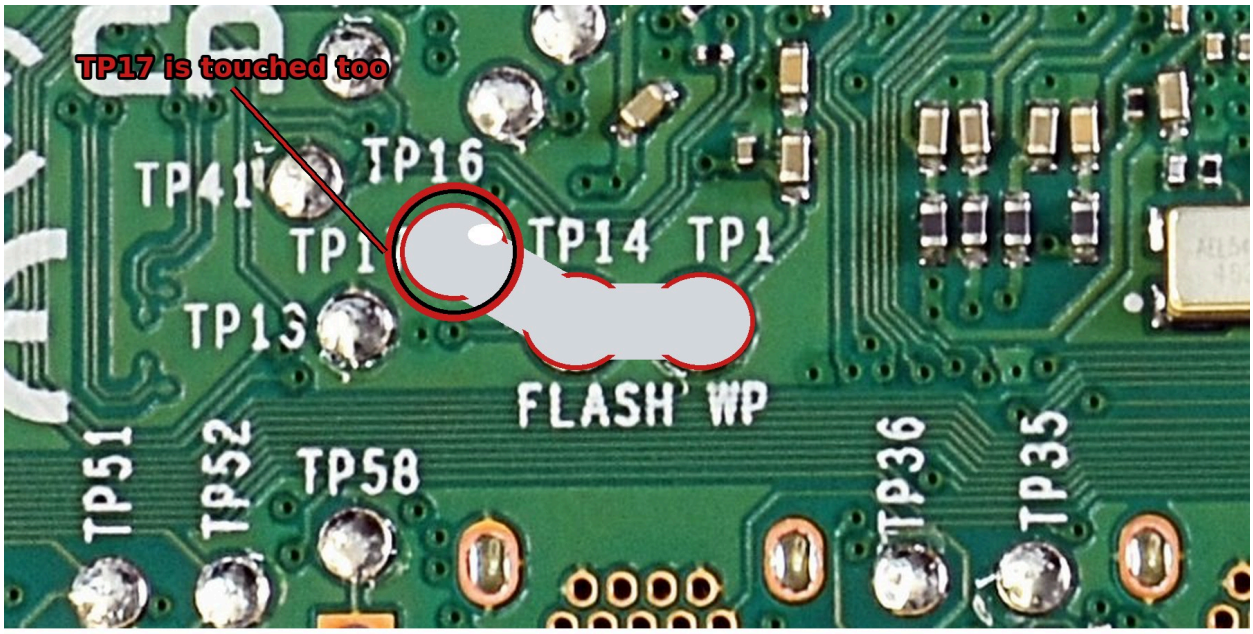
Photo: Suyash Dwivedi, Wikimedia Commons[2], CC BY-SA 4.0[3]; annotated, same licence.



RIGHT: one blob of solder over TP14 and TP1, touching nothing else

(Without an iron: a wire or tweezers held on both pads.)
A drawing on the photo.

Photo: Suyash Dwivedi, Wikimedia Commons[2], CC BY-SA 4.0[3]; annotated, same licence.



WRONG: the solder also reaches TP17

Take it off and make it again before any power goes on.
A drawing on the photo.

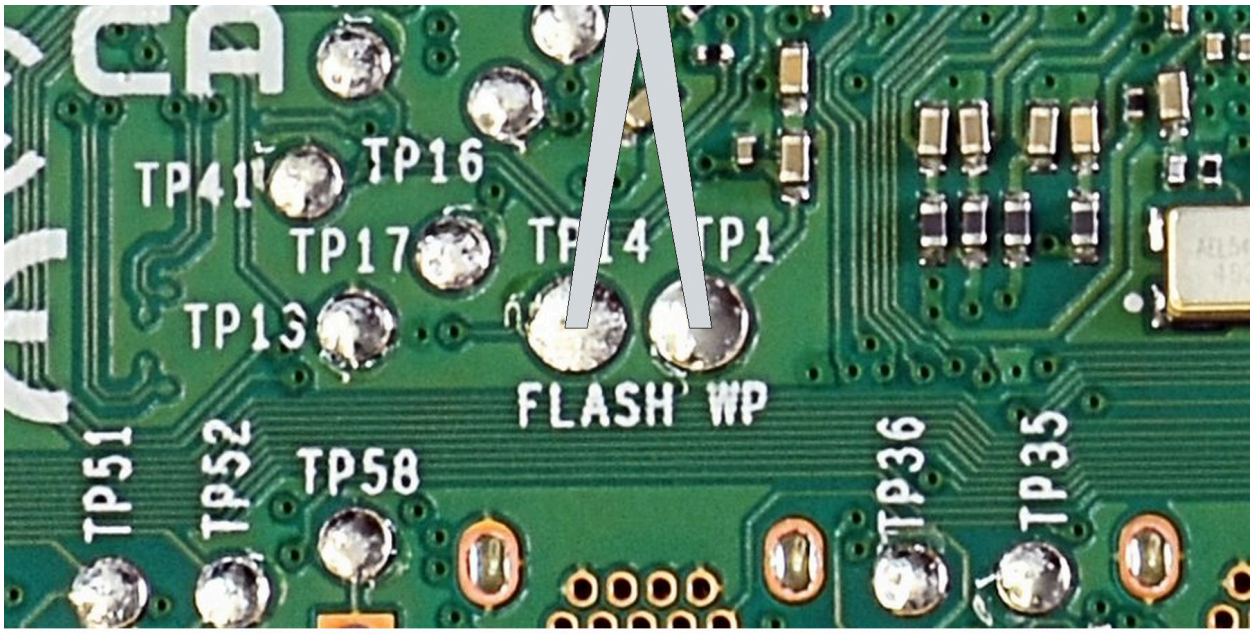
Photo: Suyash Dwivedi, Wikimedia Commons[2], CC BY-SA 4.0[3]; annotated, same licence.

WARNING

`TP1` carries power when the Pi is on. A bridge that touches anything besides `TP14` and `TP1` can damage the Pi. If yours does, take it off and make it again now, while the Pi is off.

1. Find the two pads.
2. Join `TP14` to `TP1` with a small blob of solder across both, as in the RIGHT picture.
3. Look at it against the RIGHT and WRONG pictures: one blob, on those two pads, touching nothing else. With a multimeter: `TP14` to `TP1` reads a short circuit.

No soldering iron? Skip 2 and 3. In step 4 you hold tweezers (or a short wire) on the two pads instead:

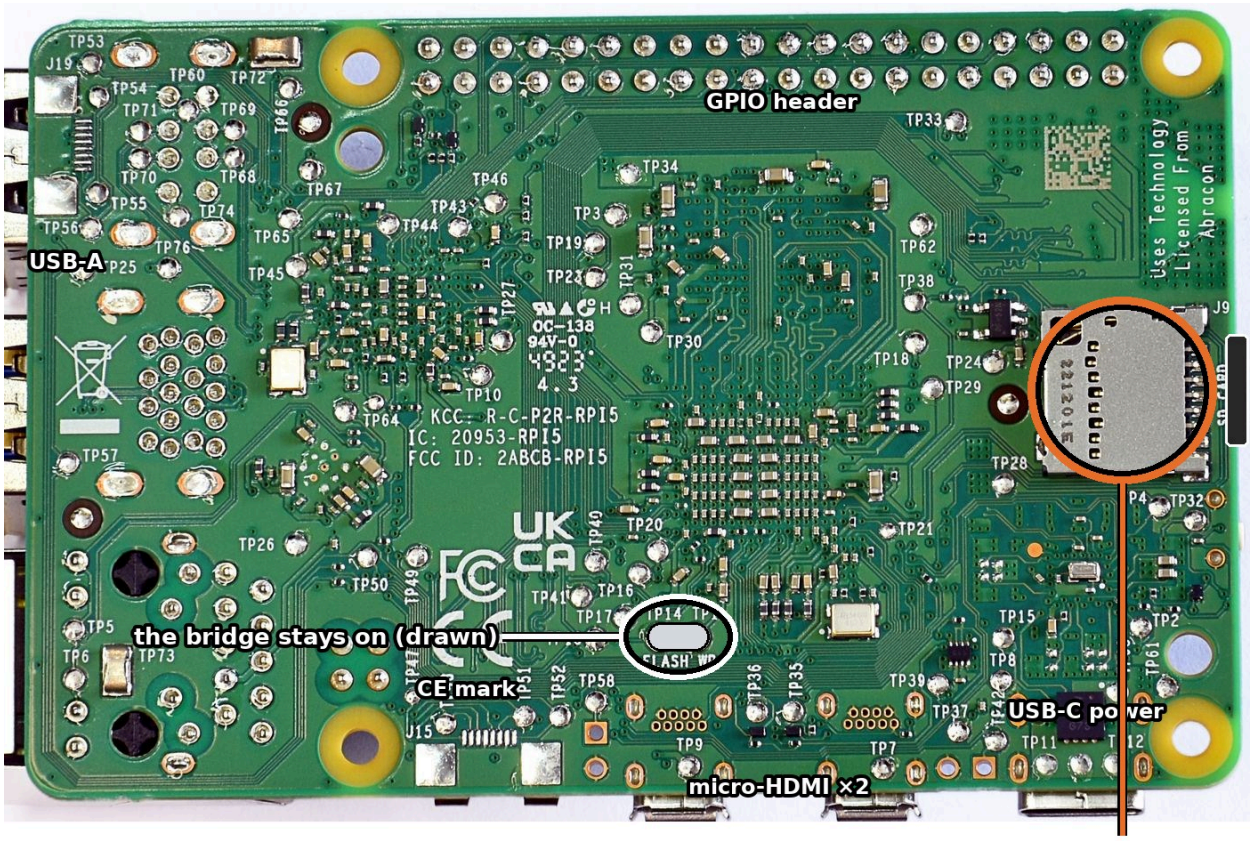


Without an iron: one tip of the tweezers on each pad

Hold them there for the whole 60 seconds of step 4.
A drawing on the photo.

Photo: Suyash Dwivedi, Wikimedia Commons[2], CC BY-SA 4.0[3]; annotated, same licence.

Step 4: card in, network cable in, wait 60 seconds

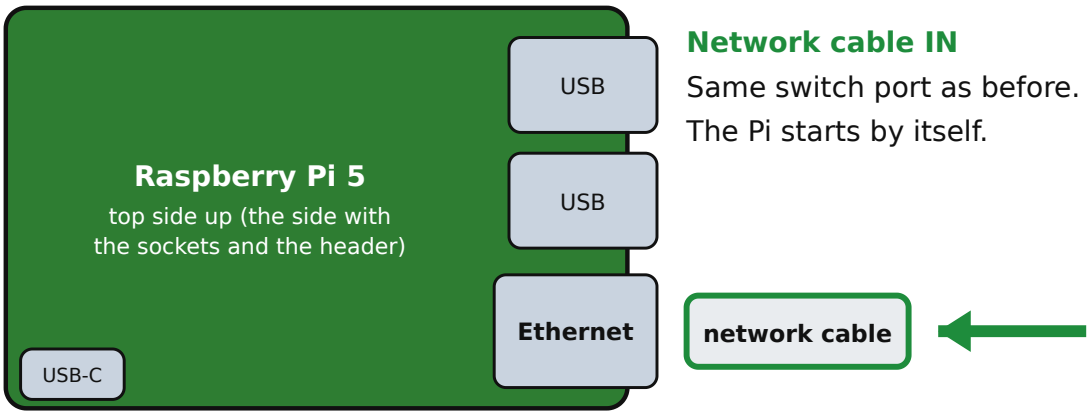


microSD slot: push the card in from the edge of the board, gold contacts facing the board, until it stops

Photo: Suyash Dwivedi, Wikimedia Commons[2], CC BY-SA 4.0[3]; annotated, same licence.

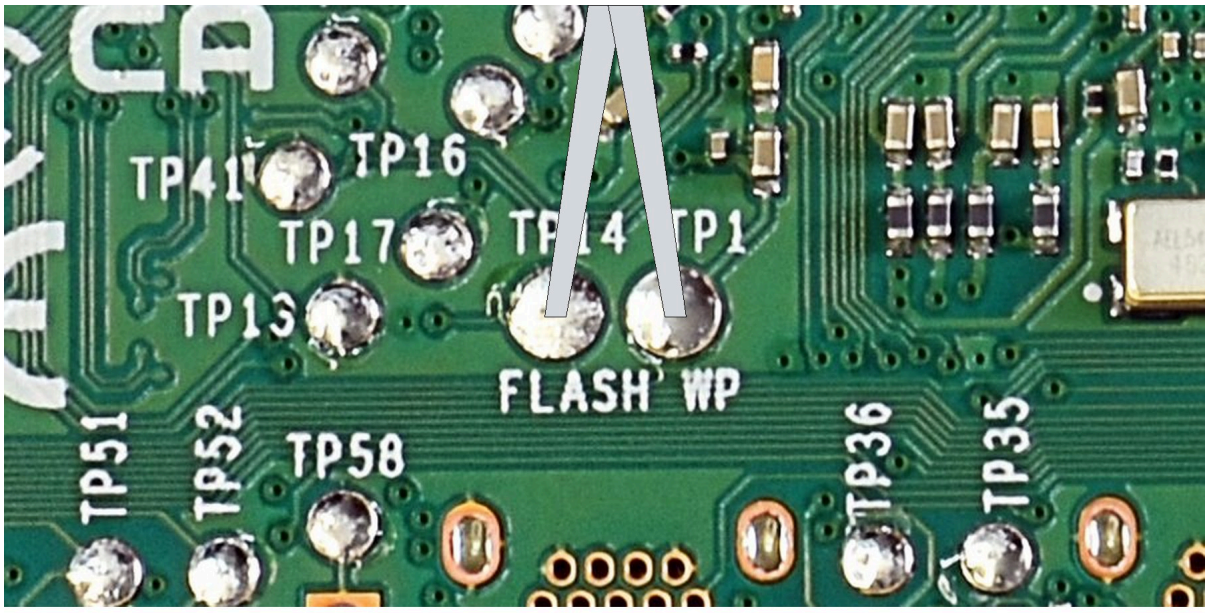
1. Push the card into the slot.

Plug the network cable in: the Pi is now on



2. Plug the network cable in, on the switch port the Pi was on. (The drawing shows the Pi top side up; yours is lying upside down. The Ethernet socket is the one beside the two USB blocks.)

   No solder bridge? Do this instead of 2:



Without an iron: one tip of the tweezers on each pad

Hold them there for the whole 60 seconds of step 4.
A drawing on the photo.

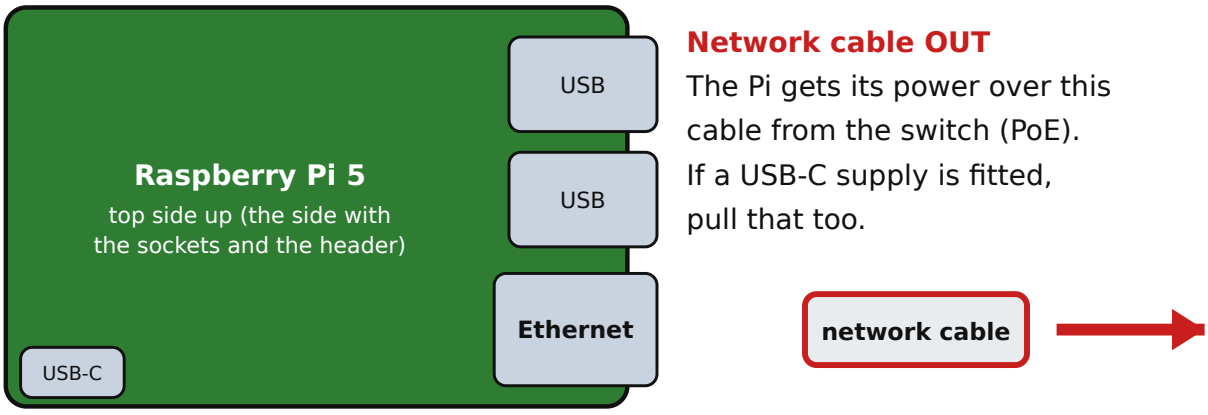
Photo: Suyash Dwivedi, Wikimedia Commons[2], CC BY-SA 4.0[3]; annotated, same licence.

- put one tip of the tweezers (or one end of the wire) on each pad;
- plug the network cable in with your other hand;
- keep the tips on both pads until the 60 seconds of item 3 are over.

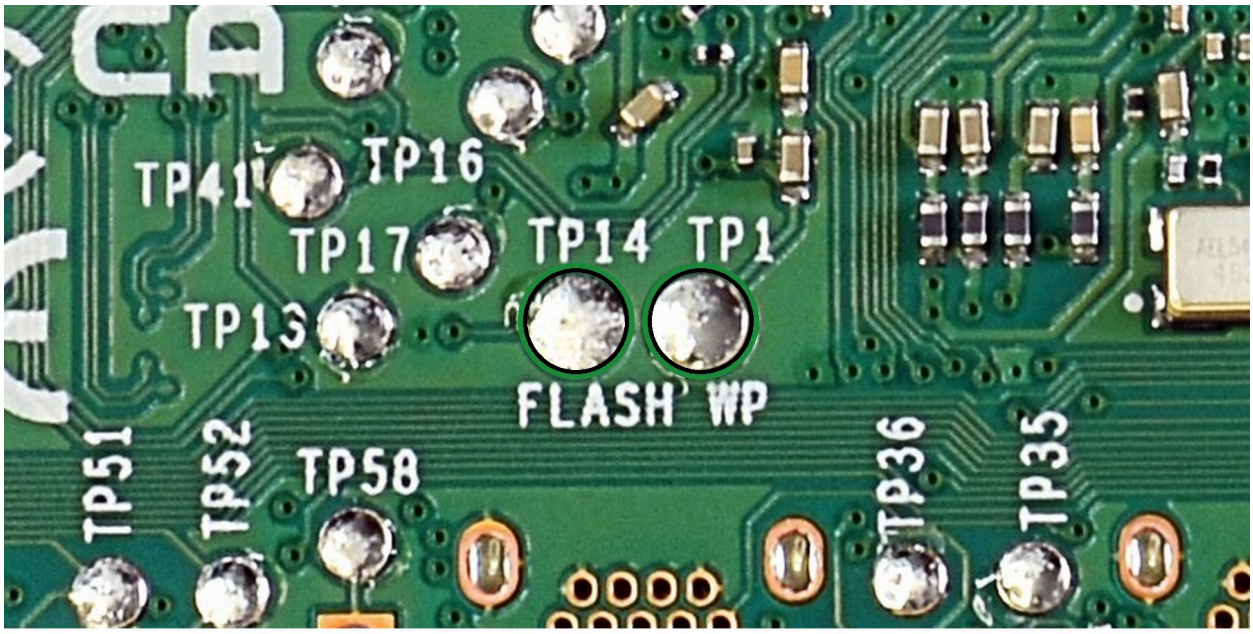
3. Wait 60 seconds by a clock. There is nothing to watch for: the Pi writes its new bootloader from the card and stops. Whatever the green light does, go on to step 5 when the 60 seconds are over.

Step 5: cable out, card out, bridge off, cable in

Pull the network cable: the Pi is now off



1. Pull the network cable.
2. Pull the card out of its slot.

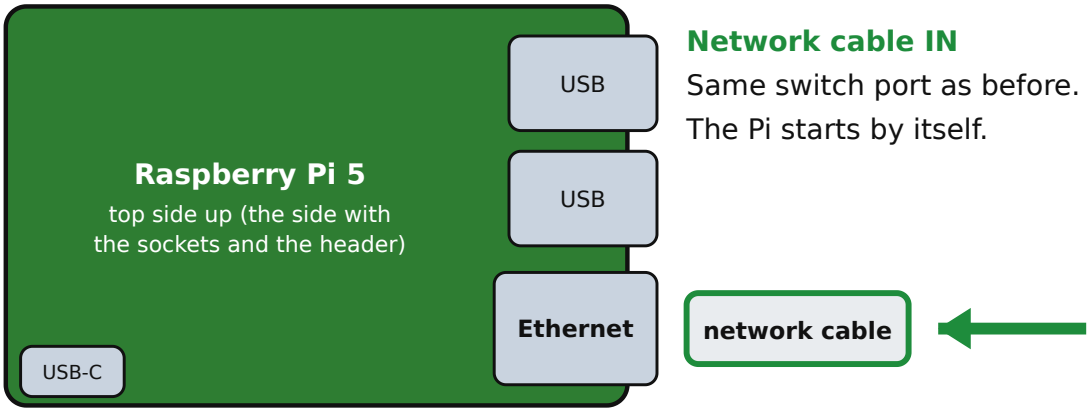


Bridge off: TP14 and TP1 are two separate pads again, and no solder went anywhere else.

Photo: Suyash Dwivedi, Wikimedia Commons[2], CC BY-SA 4.0[3]; annotated, same licence.

3. Take the bridge off: wick the solder away (or let go of the wire or tweezers). The two pads must be separate again, as in the picture.
4. Put the Pi back where it was.

Plug the network cable in: the Pi is now on



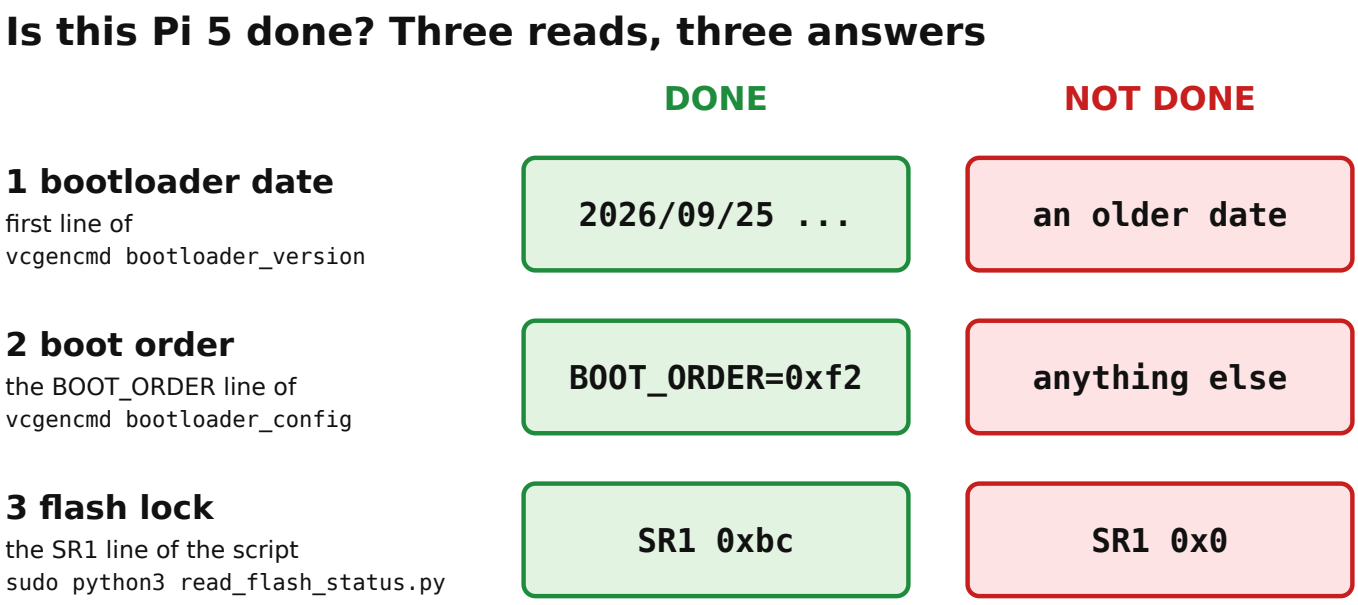
5. Plug the network cable in, on the same switch port: the port this Pi normally lives on. (Some sites keep one port aside for bringing up new Pis, an EEPROM service port. A Pi plugged in there is never locked: not that port.)

WARNING

Leave the Pi alone for two minutes now. It starts from the network and locks its flash again while it starts; do not pull the cable during that time.

Step 6: read it back

Do the three reads again, then find your result in the chart at the end of this step.



All three DONE: finished.
Anything else: the chart at the end of this step.
A NEWER date, another SR1 value, or a jedec line that is not ef4015:
STOP. This page does not cover that Pi; change nothing on it.

Open the web site of the place the Pi is at (for the Welland fleet, https://welland.fpgas.online), find this Pi in its list of boards and open the Pi's page. The web terminal on that page logs you in as user `pi`. (With ssh instead: `ssh pi@` followed by the address on that page; the same page shows the password.) Run:

```
$ vcgencmd bootloader_version
2026/09/25 15:59:09
version 1accd6659983613cd104973ce2432da14f1a7682 (release)
timestamp 1790351949
update-time 1791022751
capabilities 0x0000007f
$ vcgencmd bootloader_config
[all]
BOOT_UART=1
WAKE_ON_GPIO=1
POWER_OFF_ON_HALT=0
BOOT_ORDER=0xf2
NET_INSTALL_AT_POWER_ON=0
```

That is what a Pi that is done prints. Read 1 is the first line of the first command. Read 2 is the `BOOT_ORDER` line of the second.

Read 3 needs a small script on the Pi (a fleet Pi forgets it at every restart, so put it there each time). In the terminal on the Pi, type `cat > read_flash_status.py`, press Enter, paste the text below, press Enter, and then press Ctrl-D on that empty line. Nothing is printed when it worked.

```
# SPDX-License-Identifier: Apache-2.0
"""Print the part and the status registers of a Pi 5's bootloader flash.
```

```
    sudo python3 read_flash_status.py

Read commands only (9Fh JEDEC id, 05h SR1, 35h SR2, 15h SR3):
nothing on the flash changes.
"""
import array
import ctypes
import fcntl
import os
import struct


def xfer(fd, tx):
    txb = ctypes.create_string_buffer(bytes(tx), len(tx))
    rxb = ctypes.create_string_buffer(len(tx))
    msg = struct.pack("QQIIHBBBBBB",
                      ctypes.addressof(txb), ctypes.addressof(rxb),
                      len(tx), 1000000, 0, 8, 0, 0, 0, 0, 0)
    # SPI_IOC_MESSAGE(1)
    fcntl.ioctl(fd, 0x40206B00, array.array("B", msg), True)
    return rxb.raw


fd = os.open("/dev/spidev10.0", os.O_RDWR)  # the Pi 5's bootloader flash
print("jedec", xfer(fd, [0x9F, 0, 0, 0])[1:].hex())
for name, cmd in (("SR1", 0x05), ("SR2", 0x35), ("SR3", 0x15)):
    print(name, hex(xfer(fd, [cmd, 0])[1]))
```

Then run it. It only reads. (With ssh from a computer that has the script as a file, `downloaded here`[1], one command does both: `ssh pi@ADDRESS sudo python3 - < read_flash_status.py`.)

```
$ sudo python3 read_flash_status.py
jedec ef4015
SR1 0xbc
SR2 0x2
SR3 0x60
```

Read 3 is the `SR1` line: `0xbc` is locked, `0x0` is not locked. If the `jedec` line is anything other than `ef4015`, this is a flash chip this page does not cover: stop, and write nothing to that Pi.

Do not use `sudo rpi-eeprom-update` for this check: it prints “up to date” on Pis that are not on the fleet’s bootloader.

Your result:

Step 6: your result. Take the FIRST row that fits.

2026/09/25, BOOT_ORDER=0xf2 and SR1 0xbc	DONE. The Pi is back in service. Keep the card for the next Pi.
an older date (whatever the rest says)	Nothing was written. Go back to step 2. Check that the bridge is on both pads and the card holds the four files.
2026/09/25, but BOOT_ORDER is not 0xf2	The card's boot.conf was wrong. Make the card again (step 1, and check what it prints), then from step 2.
2026/09/25 and 0xf2, but SR1 0x0	Not locked. Pull the network cable, plug it in again on the Pi's normal port, wait 2 minutes, read again. Still 0x0: unplug the Pi and tell whoever runs the site.
no answer: the Pi is not back on the network after 5 minutes	Go back to step 2, with a card whose pieeprom.bin you have checked prints BOOT_ORDER=0xf2 (step 1).

Compute Module 4, Compute Module 5 and the Compute Blade

> **WARNING**
> **There are no upgrade steps for a Compute Module in a Compute Blade on this page, because we have not done one.** What this part gives you is how to tell whether your blade needs anything. Both blades we read are fine as they are and need nothing.

Does this blade need anything?

Log in to the blade the way you normally do and run these two. They change nothing and need no root.

```
$ vcgencmd bootloader_version
2025/12/08 19:29:54
version 2226a853bb9f5fd80392e3a4a89e457aeca88008 (release)
timestamp 1765222194
update-time 1774980226
capabilities 0x0000007f
$ vcgencmd bootloader_config
[all]
BOOT_UART=1
# Default BOOT_ORDER for provisioning
# SD -> NVMe -> USB -> Network
BOOT_ORDER=0xf2461
```

That is what one blade of ours (a Compute Module 5 Lite) printed while it was running from the network.

This blade's BOOT_ORDER, read from the LAST digit

0xf2461: SD card, NVMe, USB, then the network (the 2), then round again.

f	2	4	6	1
start again: then	network: tried 4th	USB: tried 3rd	NVMe: tried 2nd	SD card: tried 1st

IMPORTANT

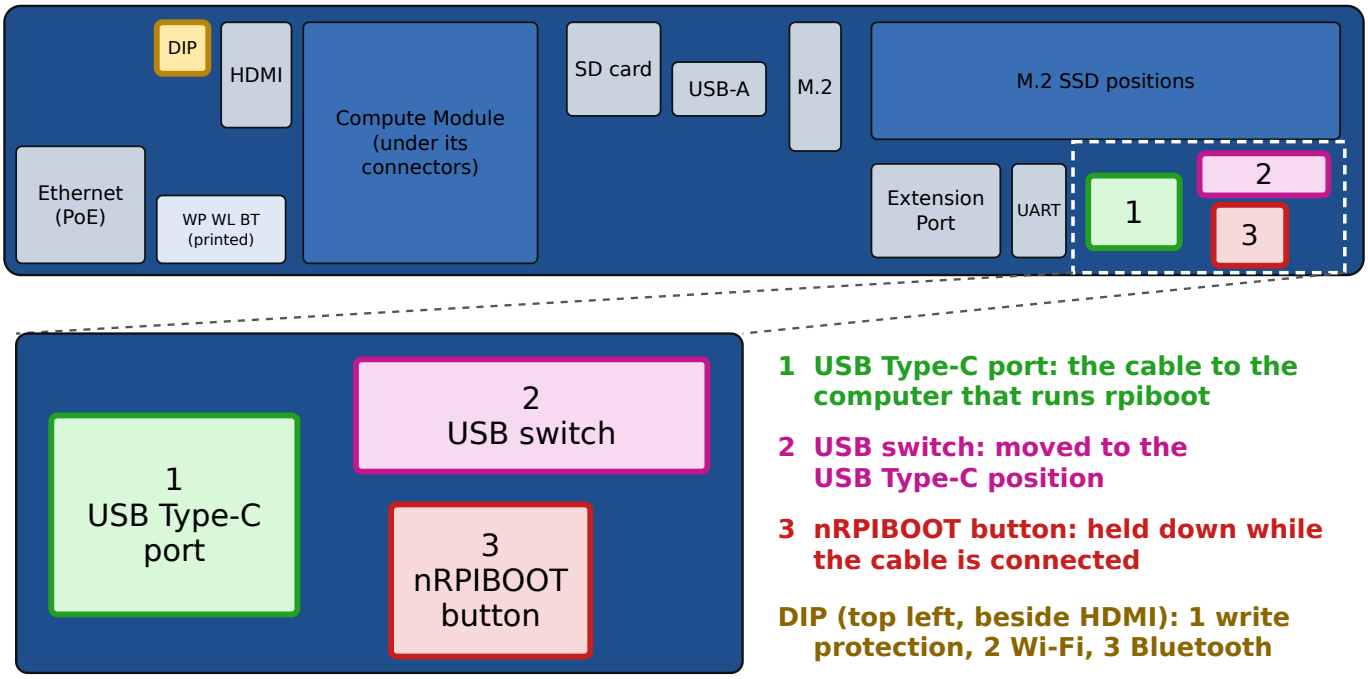
What to do

- Your blade starts from the network, and its `BOOT_ORDER` has a `2` in it (as `0xf2461` has): **do nothing.** Its bootloader is fine as it is.
- Your blade does not start from the network, or its `BOOT_ORDER` has no `2` in it: **do not try to change it from this page.** If the blade is part of fpgas.online, open an issue at fpgas-online/fpgas.online-docs[4] with the two outputs above. If it is your own, this page cannot help you further: what the blade's maker and Raspberry Pi document is quoted at the end of this page, untested by us.
- Your blade is meant to start from its own storage (an SSD or a card) and does: it is not broken, and nothing on this page applies to it.

A date in the first line that is older or newer than the one above is not a reason to act, and neither is "UPDATE AVAILABLE" from `rpi-eeprom-update`.

Why there are no steps: what a blade would need

Compute Blade, Dev model, component side (drawn from the maker's picture)



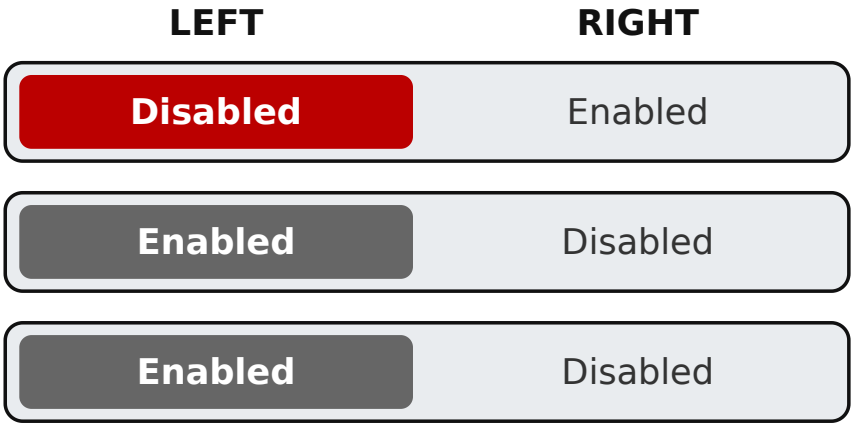
How to tell a Dev model: it has the USB Type-C socket and, beside it, the small nRPIBOOT button.

A drawing from documentation, not from a board in hand: positions are approximate.

[Outline of a Dev model Compute Blade with the USB Type-C port (1), the USB switch (2), the nRPIBOOT button (3) and the DIP switches marked: full size on a landscape sheet of its own, sheet 92.]

A Compute Module’s bootloader is written over USB from another computer, and that needs the three parts numbered 1, 2 and 3 in the drawing. The blade’s maker says only the Dev model has them. A failed write on a Compute Module can be repaired only the same way, which is why a blade that works is left alone.

Dev model Compute Blade: the DIP switches



Switch 1 at LEFT for a bootloader upgrade: that is our reading of the two makers' documents. Untested by us.

Drawn with every switch at LEFT. Which way is left on the board is the maker's wording; we have not held one. The maker: change switch positions only when the blade is unplugged from power.

The Dev model also has the switch that holds the bootloader’s write protection.

What is behind this page

You do not need this part to do the job.

What has been run

- **Upgrade of a locked Pi 5, steps 2 to 6:** once, on one fleet Pi 5, on 3 Oct 2026, and not exactly as written here. How the card used that day was made was not recorded. The Pi sat on the site’s EEPROM service port for the write and was moved to a normal port afterwards, where it read locked (`SR1 0xbc`) when next looked at; the lock coming back in the first start after a card write was not watched. That a normal fleet start locks an unlocked flash within that one start was measured on another occasion (`SR1` from `0x00` to `0xbc`).
- **The times in the steps** (60 seconds for the write, two minutes for the start, five minutes before calling a Pi gone) are generous round figures of ours, not measurements of the write.
- **Step 1:** the commands up to the four files were run on a PC on 5 Oct 2026 (raspberrypi/rpi-eeprom at commit `a72213d`, fetched with `git clone --depth 1` on the day that commit was its newest); the outputs shown are from that run. Formatting a card and booting a Pi from a card made this way: **not yet run by us.**
- **The check:** the outputs are from fleet Pi 5s on 4 Oct 2026 (four boards for the status registers). The read script printed exactly as shown has **not yet been run in that form**; the same code inside another tool produced the values.
- **The board page and its web terminal** have no picture on this page yet.

- **The pictures of the bridge** are drawn on a photo. We have no photo of a real bridge, of a card going in, or of the LED during a write.
- **The LED during the write:** Raspberry Pi's documentation says a steady rapid blink means success and an error pattern means failure. On our one run it blinked 3 long, 3 short, an error pattern, although the write had succeeded. That is why step 4 says to ignore it.
- **Three rows of the step 6 chart** ("but BOOT_ORDER is not 0xf2", "but SR1 0x0" and "no answer"): not seen by us. A Pi that is written but not locked can be changed by anyone with root on it, which on a public site is every visitor: that is why its row says to tell whoever runs the site if it stays unlocked.
- **That a Pi 5 runs `recovery.bin` from a card whatever its flash holds** is Raspberry Pi's documentation, not yet needed and so not yet run by us.
- **Compute Module, Compute Blade:** only the two reads. Nothing that changes a bootloader has been run by us on that hardware.

What two fleet Pi 5s printed

A Pi that still needed the upgrade, 4 Oct 2026, beside the done one shown in the check:

```
$ vcgencmd bootloader_version
2026/05/26 16:01:25
version 086b83e3332dfc8927c56762771d082f3077a1ae (release)
timestamp 1779807685
update-time 0
capabilities 0x0000007f
$ vcgencmd bootloader_config
[all]
BOOT_UART=1
BOOT_ORDER=0xf12
NET_INSTALL_AT_POWER_ON=1
```

`sudo rpi-eeprom-update` printed `BOOTLOADER: up to date` on both: it compares with the package installed in the root, not with what the fleet wants. The second PS1 blade read on 5 Oct 2026 (pi20) printed `2025/11/05 17:37:18` and the same settings as the blade shown above (pi16).

The lock, bit by bit

Status register 1 of the boot flash (Winbond W25Q16)

locked: reads 0xbc

bit 7: SRP	bit 6: SEC	bit 5: TB	bit 4: BP2	bit 3: BP1	bit 2: BP0	bit 1: WEL	bit 0: BUSY
1	**0**	**1**	**1**	**1**	**1**	**0**	**0**

not locked: reads 0x00

bit 7: SRP	bit 6: SEC	bit 5: TB	bit 4: BP2	bit 3: BP1	bit 2: BP0	bit 1: WEL	bit 0: BUSY
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**

SRP = 1: the register itself is protected while /WP is low.
TB = 1 with BP2 BP1 BP0 = 111: the whole chip is protected.

The script prints `0x00` as `0x0`. The W25Q16JV datasheet also says that with QE = 1, as these flashes have, the `/WP` pin is not used; what that means for the lock is under Not yet known.

Why the flash is locked

See EEPROM write protect[5].

What was measured on the Pi 5

All of it on Pi 5 boards with the W25Q16 flash, netbooting the fleet root, on 3 and 4 Oct 2026. Observations 2 to 5 are from one board.

	Observation
1	`eeprom_write_protect=1` in the `config.txt` served over netboot **is applied**: SR1 went from `0x00` to `0xbc` in one boot (one board). All four Pi 5s we read were locked this way, two of them never handled for it.
2	`eeprom_write_protect=0` served over netboot **was not applied**: SR1 stayed `0xbc` over four boots. The `TP14` to `TP1` bridge was believed fitted during those boots, but that was not checked and cannot be read from software (observation 6).
3	With the flash protected, the netboot self-update fetched `pieeprom.sig` and `pieeprom.upd`, wrote nothing, tried once more and then booted the operating system. **Nothing anywhere reported the failed update.**
4	An SD card holding `recovery.bin`, `pieeprom.bin` and a `config.txt` with `eeprom_write_protect=0` (whether `pieeprom.sig` was on it was not recorded), with the `TP14` and `TP1` pads bridged, cleared the protection and wrote the new bootloader. The activity LED blinked 3 long, 3 short although the write had succeeded.
5	With the flash unprotected and already holding the offered image, the netboot self-update found nothing to do and booted on.
6	The flash's `/WP` pin goes to no SoC GPIO: software cannot read whether the pads are bridged.
7	A bootloader of `2026/05/26` already provides `/dev/pio0` (RP1 PIO). The upgrade to `2026/09/25` is for a uniform fleet.

Observation 1 does not fit a plain reading of Raspberry Pi's description (quoted further down), where the setting is "for `recovery.bin`": the bootloader applied it from a netbooted `config.txt`. For observation 2 the simplest explanation is a bridge that was not making contact; the other is that the bootloader sets but does not clear the protection outside `recovery.bin`. Which it is has not been found out (the bootloader's UART log during such a boot has not been captured), so an unlock over netboot is **not shown to work**, and not shown to be impossible.

Not yet known

- **Whether the lock holds against root on the Pi.** Raspberry Pi's documentation says the Pi 5's `/WP` pin is low by default, so the status register cannot be changed without bridging the pads. The W25Q16JV datasheet says that with `QE=1` (which these flashes have) the `/WP` function is disabled. If the datasheet is right, root could clear the protection with two SPI commands. **Nobody has tried it.** Until it is tried, treat the lock as protection against accident and against the standard tools, not as proven against a determined visitor.
- Whether the SD card route works without the bridge.
- Whether `eeprom_write_protect=0` over netboot unlocks a Pi 5 whose bridge is known to be fitted (observation 2).
- What `rpi-eeprom-update -a` or `rpi-eeprom-config --apply` does on a locked, netbooted fleet Pi. Neither has been run by us there.
- Whether a Pi 5 that does not boot any more is repaired by the same card. Raspberry Pi documents that it is (below); we have not had such a board.

What does not work

- A netboot self-update against the protected flash: nothing is written and nothing reports it (observation 3). `rpi-eeprom-update -a` and `rpi-eeprom-config --apply` stage the same kind of update for the bootloader to apply, so the same outcome is expected; not yet run by us.
- Serving `eeprom_write_protect=0` to the Pi over netboot did not unlock it when tried, with the caveat of observation 2.
- `flashrom`: Raspberry Pi's documentation says it "does not support clearing of the write-protect regions and will fail to update the EEPROM if write-protect regions are defined".

What Raspberry Pi says (Pi 4 and Pi 5)

From the `config.txt` documentation, `eeprom_write_protect`[6]:

> Configures the EEPROM `write status register`. This can be set either to mark the entire EEPROM as write-protected, or to clear write-protection.
>
> This option must be used in conjunction with the EEPROM `/WP` pin which controls updates to the EEPROM `Write Status Register`. Pulling `/WP` low (CM4 `EEPROM_nWP` or on a Raspberry Pi 4 `TP5`) does NOT write-protect the EEPROM unless the `Write Status Register` has also been configured.
>
> […]
>
> `eeprom_write_protect` settings in `config.txt` for `recovery.bin`.
>
> […]
>
> On Raspberry Pi 5 `/WP` is pulled low by default and consequently write-protect is enabled as soon as the `Write Status Register` is configured. To clear write-protect pull `/WP` high by connecting `TP14` and `TP1`.

Values: `1` protects the whole EEPROM, `0` clears the protection, `-1` (the default) does nothing.

From Raspberry Pi boot EEPROM, `recovery.bin`[7]:

> At power on, the ROM found on BCM2711 and BCM2712 looks for a file called `recovery.bin` in the root directory of the boot partition on the SD card. If a valid `recovery.bin` is found then the ROM executes this instead of the contents of the EEPROM. This mechanism ensures that the bootloader flash image can always be reset to a valid image with factory default settings.

and, on the same page:

> If the bootloader update image is called `pieeprom.bin` then `recovery.bin` will stop after the update has completed. On success the HDMI output will be green and the green activity LED is flashed rapidly. If the update fails, the HDMI output will be red and an error code will be displayed by the activity LED.

The service port at Welland

At Welland one switch port is kept as an EEPROM service port. The gateway serves a Pi on that port its own boot tree, holding the target `pieeprom.upd` and `pieeprom.sig` and a `config.txt` with `eeprom_write_protect=0`, and no other port sees those files. With today's findings it upgrades only a Pi

whose flash is **not** protected (a new Pi, or one cleared with the card), and it is where an upgrade can be watched. Making it a managed feature that upgrades a locked Pi without a visit is open work.

What the Compute Blade's maker documents

> **WARNING**
> Untested by us. Do not follow these as steps.

Only the **Dev** model of the Compute Blade has the parts for the USB route. The maker says so where it describes writing an operating system to a module's eMMC, in a guide written for the Compute Module 4 (image guide[8]); the same port and button are what a bootloader flash over USB needs:

> eMMC can only be imaged on Dev blade as it has USB Type-C port, USB switch, and nRPIBOOT button

and, for putting the module into USB boot:

> From the usbboot directory run `sudo ./rpiboot`
>
> Move the USB switch to the USB Type-C position. Then, while holding down the nRPIBOOT button on the blade connect the USB Type-C cable.
>
> The Device will reconnect several times. On the last time it will appear as a USB media device.

The USB switch[9] is there because "The compute module can only operate one USB port at a time." The maker's usbboot guide[10] covers building `rpiboot` and says the configuration is edited in `/usbboot/<firmware directory>/config.txt`; for the flash itself it points to Raspberry Pi's "Flash Compute Module bootloader EEPROM", which is the procedure printed again below.

The write-protect pin is on a DIP switch, again on the Dev model only. From the maker's DIP switch guide[11]:

> DIP Switch is ONLY populated on Dev model Compute Blades.
>
> Only change switch positions when the Blade is unplugged from power.
>
> To enable write protection, changes are required to `config.txt`. It can be found in the directory `/boot/firmware/`, or flash the EEPROM using usbboot adding the following line to the `config.txt` file `eeprom_write_protect=1` This will pull the `EEPROM_nWP` pin low. This will enable the DIP switch.

Read with Raspberry Pi's description (quoted below), that is the same two-part lock: `eeprom_write_protect=1` sets the flash's status register, and switch 1 is what holds `EEPROM_nWP` low. Raspberry Pi requires `EEPROM_nWP` not to be low while the bootloader is flashed, and the maker's table calls the left position "Disabled", so left is the position for an upgrade. That last step is our inference from the two documents, not something either states or we have tried.

What is not known about the blade

- **Which model the PS1 blades are.** We have not recorded it. The maker's pages name a Dev model and a TPM model. On a blade that is not a Dev model the maker describes no USB Type-C port, no nRPIBOOT button and no DIP switch, so, as far as those pages go, neither the USB route nor the

hardware lock exists on the blade itself; the module would then have to go into a carrier that has them (Raspberry Pi describes the route on its own IO boards), which we have not done.
- **What a blade without the DIP switch does with `EEPROM_nWP`** (tied low, tied high or left open). The maker's pages we read do not say.
- **Whether a blade's bootloader can be updated from the running system.** Raspberry Pi's self-update needs update files in the boot file system it booted from ("For network boot make sure that the TFTP `boot` directory can be mounted through NFS and that `rpi-eeprom-update` can write to it"), and it "does not update the bootloader atomically". On a netboot root like PS1's, a read-only export under a tmpfs overlay, a file written on the blade never reaches the boot directory the bootloader reads, so we expect it not to work there. Not tried by us on any Compute Module.

What Raspberry Pi documents for a Compute Module

> **WARNING**
>
> Untested by us on this hardware. Do not follow these as steps on a blade: they are written for Raspberry Pi's own IO boards.

> **WARNING**
>
> **Not yet run by us on this hardware.** Everything in this section is Raspberry Pi's documentation, quoted from raspberrypi/documentation[12] at commit `287523e6`. No fpgas.online Compute Module has had its bootloader upgraded or locked by this procedure. Read the bootloader state first (the `vcgencmd` commands above work on a Compute Module; the device node of the boot flash and its part have not been read by us on a CM4 or CM5).

The SD card route above **does not exist** on a Compute Module. From Compute Module EEPROM bootloader[13]:

> On Compute Modules with an EEPROM bootloader, ROM never runs `recovery.bin` from SD/eMMC. These Compute Modules disable the `rpi-eeprom-update` service by default, because eMMC is not removable and an invalid `recovery.bin` file could prevent the system from booting.
>
> You can override this behaviour with `self-update` mode. In `self-update` mode, you can update the bootloader from USB MSD or network boot.
>
> WARNING: `self-update` mode does not update the bootloader atomically. If a power failure occurs during an EEPROM update, you could corrupt the EEPROM.

and from the Pi boot EEPROM page: "Reflashing the bootloader over USB is also the only option available for CM4 and CM4S."

Flashing the bootloader over USB (`rpiboot`).

This needs a second computer with `rpiboot`[14] (`sudo apt install rpiboot`, or built from source), a USB cable to the carrier's USB device port, and the carrier's `nRPI_BOOT` jumper or button, which makes the module wait for a USB host instead of booting. Raspberry Pi describes it for its own IO boards ("Fit `nRPI_BOOT` to J2 (`disable eMMC Boot`) on the IO board jumper"); another carrier brings `nRPI_BOOT` and the USB device port out in its own way, to be looked up in that carrier's documentation.

> To flash the bootloader EEPROM:

1. Set up the hardware as you would when flashing the eMMC, but ensure `EEPROM_nWP` is *not* pulled low.
2. Run the following command to write `recovery/pieeprom.bin` to the bootloader EEPROM: `./rpiboot -d recovery`
3. When complete, `EEPROM_nWP` can be pulled low again.

Setting the configuration and the lock.

To modify the Compute Module EEPROM bootloader configuration:

1. Navigate to the `usbboot/recovery` directory.
2. If you require a specific bootloader release, replace `pieeprom.original.bin` with the equivalent from your bootloader release.
3. Edit the default `boot.conf` bootloader configuration file to define a `BOOT_ORDER`: For network boot, use `BOOT_ORDER=0xf2`. For SD/eMMC boot, use `BOOT_ORDER=0xf1`. For USB boot failing over to eMMC, use `BOOT_ORDER=0xf15`. For NVMe boot, use `BOOT_ORDER=0xf6`.
4. Run `./update-pieeprom.sh` to generate a new EEPROM image `pieeprom.bin` image file.
5. If you require EEPROM write-protection, add `eeprom_write_protect=1` to `/boot/firmware/config.txt`. When enabled in software, you can lock hardware write-protection by pulling the `EEPROM_nWP` pin low.
6. Run the following command to write the updated `pieeprom.bin` image to EEPROM: `../rpiboot -d .`

So on a Compute Module the lock has two parts, as on the Pi 4: the status register, set by `eeprom_write_protect=1`, and the `EEPROM_nWP` pin of the module's connector, which the carrier must pull low for the status register to stay as set. Raspberry Pi's step 5 names `/boot/firmware/config.txt` while step 6 flashes from the `usbboot/recovery` directory, which has a `config.txt` of its own; which of the two the setting has to be in is not clear from the text and not tried by us. Whether a given carrier pulls `EEPROM_nWP` low, leaves it open or brings it to a jumper is a property of that carrier. For the upgrade it must **not** be low; for the lock it must be.

Differences from the Pi 5.

- No card route and no `recovery.bin` from storage. Raspberry Pi names USB `rpiboot` as the only way to reflash a CM4; for a CM5 we infer the same from the passage above. A Compute Module whose bootloader is broken therefore needs its carrier's USB device port and `nRPI_BOOT`.
- Self-update over netboot exists but "does not update the bootloader atomically": a power cut during the write can leave a module that only `rpiboot` repairs.
- Our Pi 5 finding that a protected flash ignores a netboot self-update silently (observation 3) may hold here too: read the version after any upgrade.

Sources

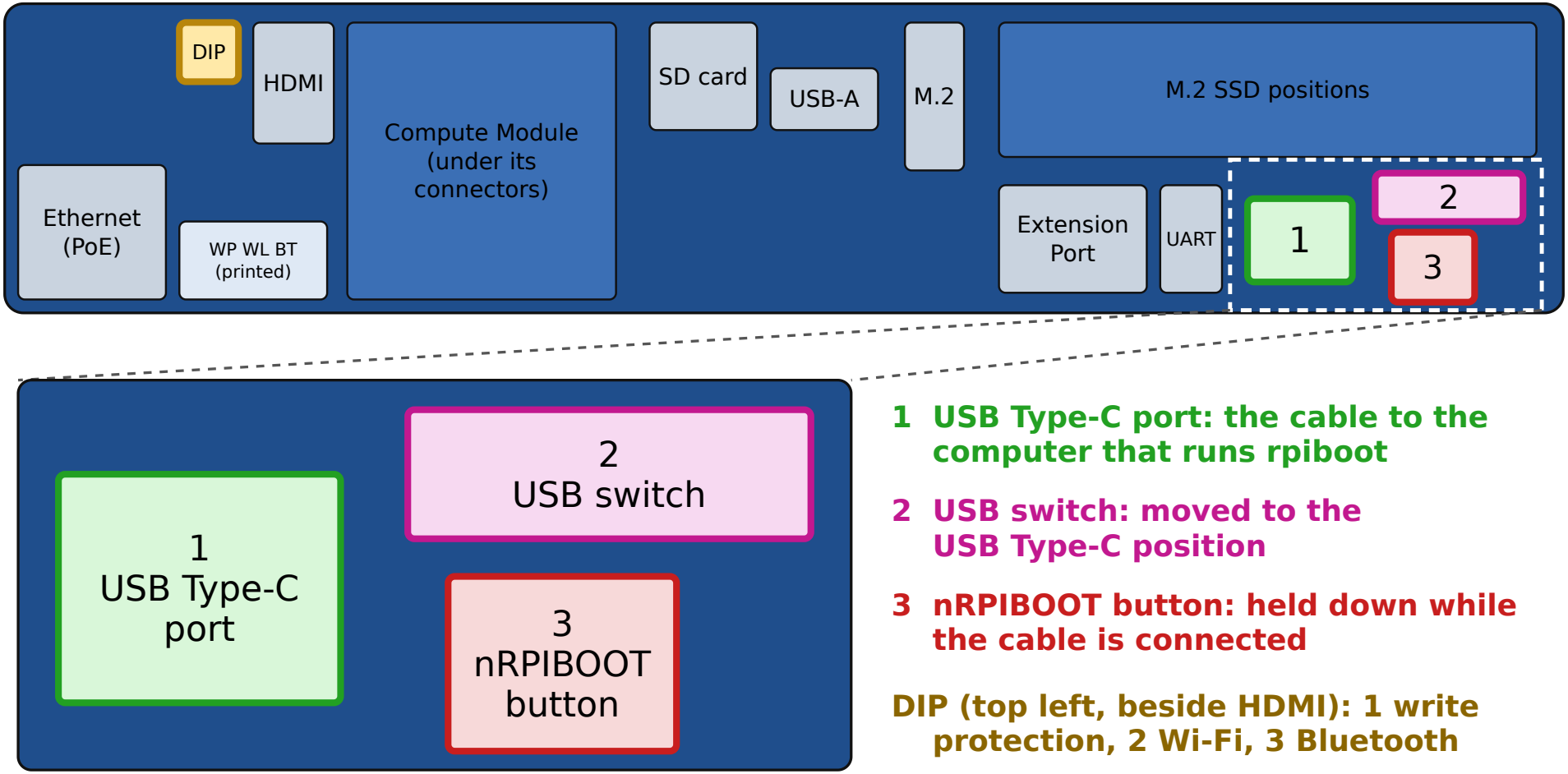
- Measurements: fpgas.online Welland fleet, 3 and 4 Oct 2026, Pi 5.

- Raspberry Pi documentation[12], commit `287523e6`: `computers/config_txt/boot.adoc`, `computers/raspberry-pi/boot-eeprom.adoc`, `computers/raspberry-pi/eeprom-bootloader.adoc`, `computers/compute-module/cm-bootloader.adoc`, `computers/compute-module/cm-emmc-flashing.adoc`.
- Winbond W25Q16JV datasheet (status register bits, `/WP` and `QE`).
- Compute Blade documentation[15] (Uptime Lab), read 5 Oct 2026: getting-started/image, guides/dip, guides/usb, advanced-guides/usbboot.
- Bootloader reads of two PS1 Compute Blades (Compute Module 5 Lite), 5 Oct 2026.
- Figures: `docs/setup/bootloader-eeprom/make_figures.py` in this repository draws them. The photographs are crops of “Raspberry Pi5 8GB Bottom View (1).jpg” by Suyash Dwivedi, Wikimedia Commons[2], CC BY-SA 4.0[3]; the annotated versions are under the same licence.
- Step 1 as run: raspberrypi/rpi-eeprom[16] at commit `a72213d`, 5 Oct 2026, on a PC.

Links in this chapter

1. https://docs.fpgas.online/en/latest/_downloads/b72401e912e4af9ca6c53c913af9663f/read_flash_status.py
2. https://commons.wikimedia.org/wiki/File:Raspberry_Pi5_8GB_Bottom_View_(1).jpg
3. https://creativecommons.org/licenses/by-sa/4.0/
4. https://github.com/fpgas-online/fpgas.online-docs/issues
5. https://docs.fpgas.online/en/latest/setup/netboot.html#eeprom-write-protect
6. https://www.raspberrypi.com/documentation/computers/config_txt.html#eeprom_write_protect
7. https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#recovery-bin
8. https://docs.computeblade.com/blade/getting-started/image
9. https://docs.computeblade.com/blade/guides/usb
10. https://docs.computeblade.com/blade/advanced-guides/usbboot
11. https://docs.computeblade.com/blade/guides/dip
12. https://github.com/raspberrypi/documentation
13. https://www.raspberrypi.com/documentation/computers/compute-module.html#compute-module-eeprom-bootloader
14. https://github.com/raspberrypi/usbboot
15. https://docs.computeblade.com/
16. https://github.com/raspberrypi/rpi-eeprom

Compute Blade, Dev model, component side (drawn from the maker's picture)



1 USB Type-C port: the cable to the computer that runs rpiboot

2 USB switch: moved to the USB Type-C position

3 nRPIBOOT button: held down while the cable is connected

DIP (top left, beside HDMI): 1 write protection, 2 Wi-Fi, 3 Bluetooth

How to tell a Dev model: it has the USB Type-C socket and, beside it, the small nRPIBOOT button.

A drawing from documentation, not from a board in hand: positions are approximate.

Outline of a Dev model Compute Blade with the USB Type-C port (1), the USB switch (2), the nRPIBOOT button (3) and the DIP switches marked (https://docs.fpgas.online/en/latest/_images/blade-dev.svg)

LABELS FOR THE PS1 COMPUTE BLADES AND THEIR FPGAS

- The two sheets after this one carry the Compute Module labels of pi16 and pi20, made from reads of 5 Oct 2026. First on plain US Letter paper, labels at true size (63.5 x 38.1 mm) with a line to cut along; then laid out for Avery 5163 / 8163 stickers (4 x 2 inch, ten to the sheet), each label in the middle of its sticker.
- Print both at 100 %, no scaling. The sticker layout is not yet tried on a real sheet: print it on plain paper first and hold it against the sticker sheet.
- "HAT none" on these labels means the firmware reported no HAT. The header's pins were deliberately not scanned, because on a Compute Blade they are JTAG wires.
- Not here: the Acorn (FPGA) labels. An Acorn label carries the card's flash ID as well as its device DNA, and the flash ID is read over PCIe through the fpgas.online design, so every card's label waits for that card's conversion.
- Not here: labels for pi14 and pi18. Their visitor ssh ports did not answer on 5 Oct 2026, so nothing was read from them that day; nothing else is known about them from that day.
- How labels are made: https://docs.fpgas.online/en/latest/verify/fpgas-verify.html#identity-and-labels

Plain US Letter: labels at true size (63.5 x 38.1 mm). Print at 100 %, no scaling; cut along the lines.

Raspberry Pi Compute Module 5 Lite

8 GB · Rev 1.0 · rev code d041a0

HAT none

serial
9fb8cfc7
cb291e63

eth MAC
2c:cf:67:fb:91:e5

wlan MAC
radio disabled, not readable

Raspberry Pi Compute Module 5 Lite

8 GB · Rev 1.0 · rev code d041a0

HAT none

serial
d85a19af
de59093d

eth MAC
2c:cf:67:fd:1e:be

wlan MAC
radio disabled, not readable

Avery 5163 / 8163 (4 x 2 in), grid from Avery's template. Not yet tried on real stock: print on plain paper first and hold it against the sheet. Print at 100 %, no scaling.

Raspberry Pi Compute Module 5 Lite

8 GB · Rev 1.0 · rev code d041a0

HAT none

serial
9fb8cfc7
cb291e63

eth MAC
2c:cf:67:fb:91:e5

wlan MAC
radio disabled, not readable

Raspberry Pi Compute Module 5 Lite

8 GB · Rev 1.0 · rev code d041a0

HAT none

serial
d85a19af
de59093d

eth MAC
2c:cf:67:fd:1e:be

wlan MAC
radio disabled, not readable