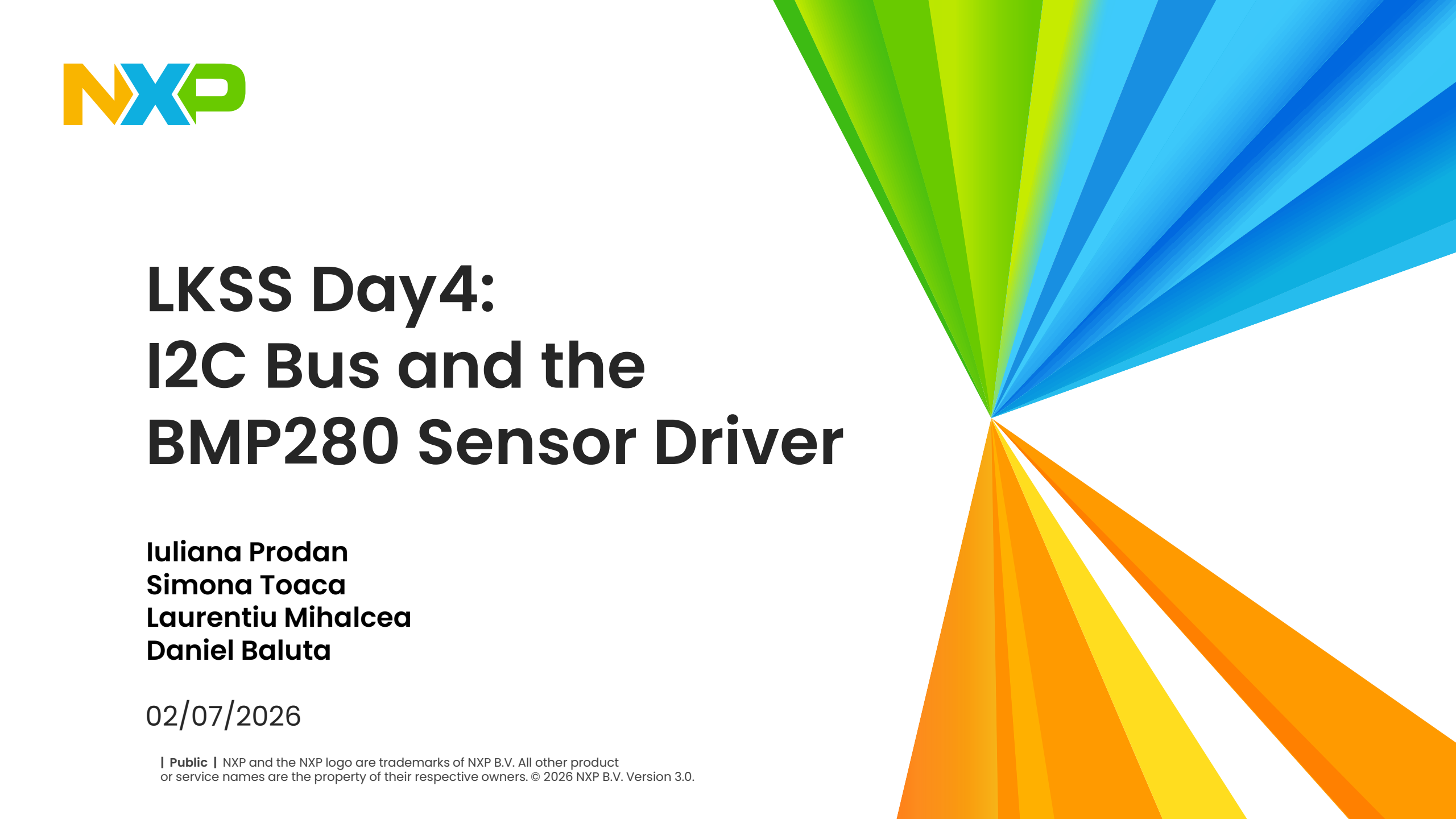


LKSS Day4: I2C Bus and the BMP280 Sensor Driver

Iuliana Prodan
Simona Toaca
Laurentiu Mihalcea
Daniel Baluta

02/07/2026

Content

I2C Protocol

Linux I2C Subsystem

BMP280 Sensor

Calibration & Compensation

Writing the Driver: probe to sysfs

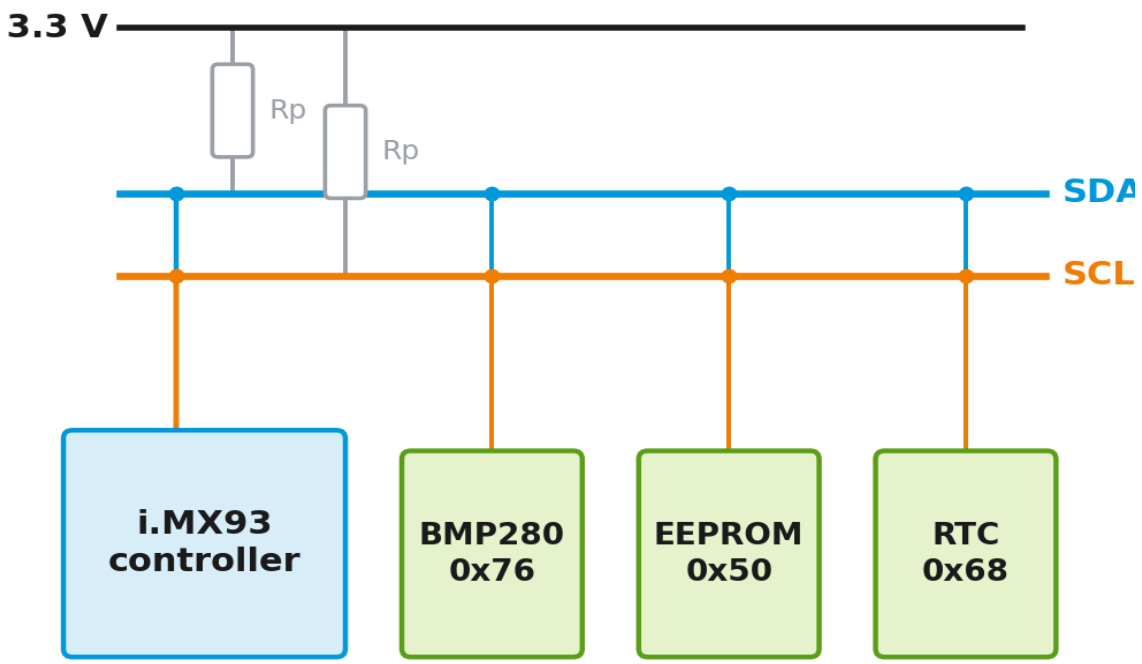
01

I2C Protocol

Two wires, many chips

I2C Protocol

- **One bus, two wires, many chips**
  - UART and GPIO: one wire, data one way
  - I2C: two wires
    - **Serial Clock (SCL)**
    - **Serial Data Line (SDA), bidirectional**
  - Devices are selected by address
  - Fewer pins, simpler boards

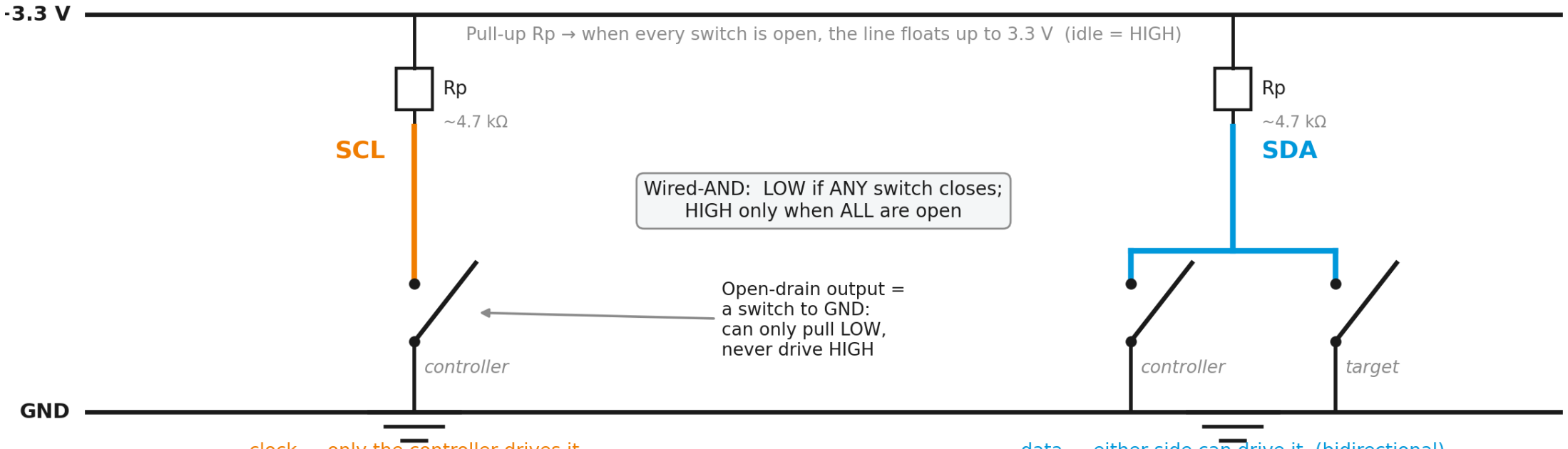


Every device shares the same two wires; each is selected by its address.

Notes and sources

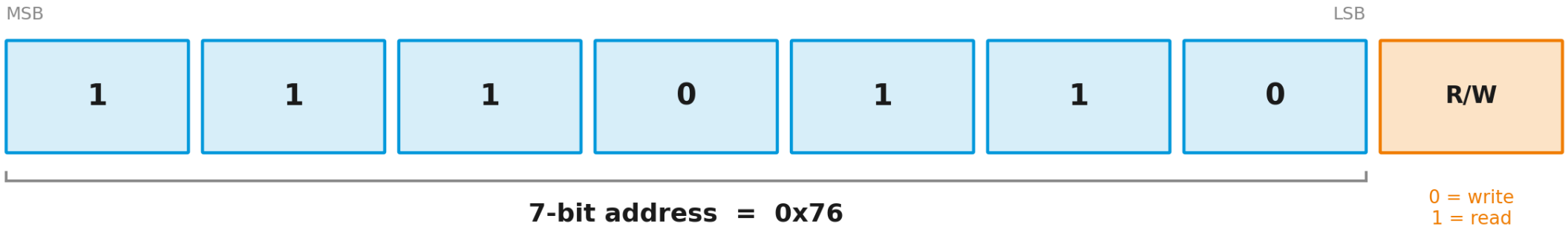
The Two Wires

- **SCL — Serial Clock**
  - driven by the controller (the i.MX93 FRDM)
  - every bit sampled on a clock edge
- **SDA — Serial Data**
  - bidirectional: carries bytes both ways
  - changes only while SCL is low
- **Open-drain outputs**
  - a device can only pull a line low
  - never drives it high
- **Pull-up resistors (~4.7 kΩ)**
  - return the idle line to 3.3 V
  - wired-AND: low wins if anyone pulls



Addressing: who is on the bus?

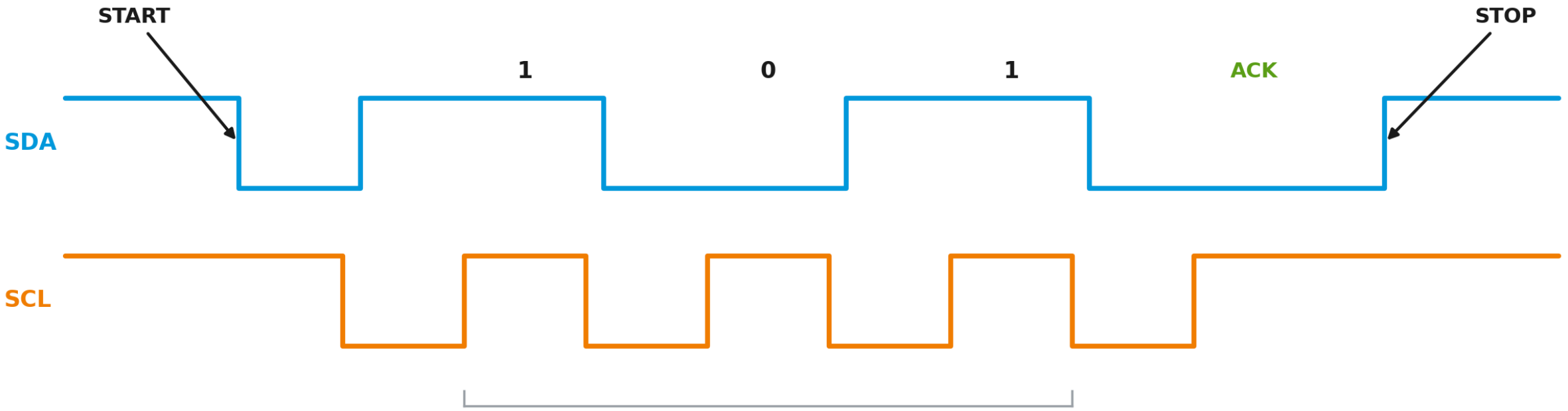
- Each target has a 7-bit address (0x00–0x7F)
- Up to ~120 devices per bus — some addresses are reserved
- The address is fixed in the chip
  - a strap pin sets the low bit(s) so identical chips can coexist
- BMP280: SDO → GND = 0x76, SDO → VDDIO = 0x77
  - this lab uses 0x76



The 7 address bits select the chip; the 8th bit says read or write. BMP280: SDO→GND ⇒ 0x76, SDO→VDDIO ⇒ 0x77

Notes and sources

Anatomy of a transaction



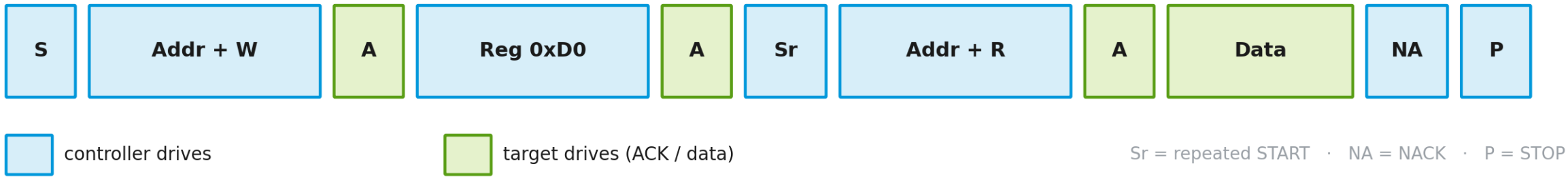
address / data bits — SDA stable while SCL is HIGH (sampled), changes while SCL is LOW

Notes and sources

Register access & timing

- **Write**
  - addr+W → register → data → STOP
- **Read**
  - addr+W → register (dummy write)
  - repeated START, addr+R → data
  - the dummy write sets the chip pointer

- **Bus speeds**
  - Standard 100 kHz · Fast 400 kHz · Fast+ 1 MHz
- **Clock stretching**
  - a target holds SCL low until ready
- **Terminology**
  - controller / target (formerly master / slave)



Notes and sources

02

Linux I2C Subsystem

Three layers, one tidy API

The Three Layer

Controller driver
the LPI2C hardware

- Knows the LPI2C registers
- Drives SCL and SDA
- Written by NXP — you never touch it

I2C core
the kernel glue

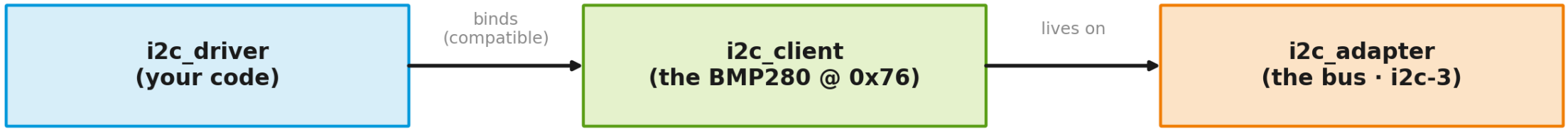
- Matches devices to drivers
- Emulates SMBus on raw transfers
- One call → a correct read

Your driver
your code

- An i2c_driver
- Gets an i2c_client (one chip)
- probe() runs at bind

Key structures & APIs

- `struct i2c_client` — one device: addr, adapter, dev
- `i2c_smbus_read_byte_data(client, reg)` — read a register
- `i2c_smbus_write_byte_data(client, reg, val)` — write a register
- `i2c_smbus_read_i2c_block_data(client, reg, len, buf)` — burst read
- SMBus helpers do the dummy-write / repeated-START
- Every read returns < 0 on error — always check!

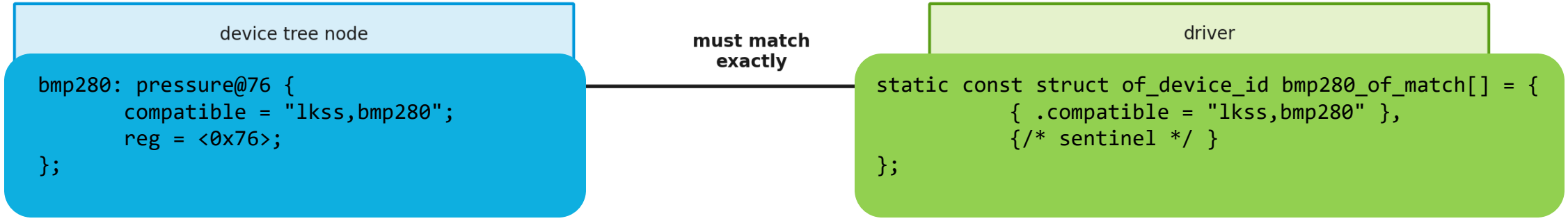


The I2C core matches driver ↔ client by the compatible string, then calls probe(), which hands you the i2c_client.

Describing the sensor: device tree

```
&lpi2c7 {                  /* bus on the EXT2 header */
    status = "okay";
    clock-frequency = <400000>;   /* 400 kHz */

    bmp280: pressure@76 {
        compatible = "lkss,bmp280"; /* compatible must match the driver of_match_table */
        reg = <0x76>;               /* SDO -> GND */
    };
};
```



Match → the core creates the i2c_client and runs probe(). No match → probe never runs.

Notes and sources

03

BMP280 sensor

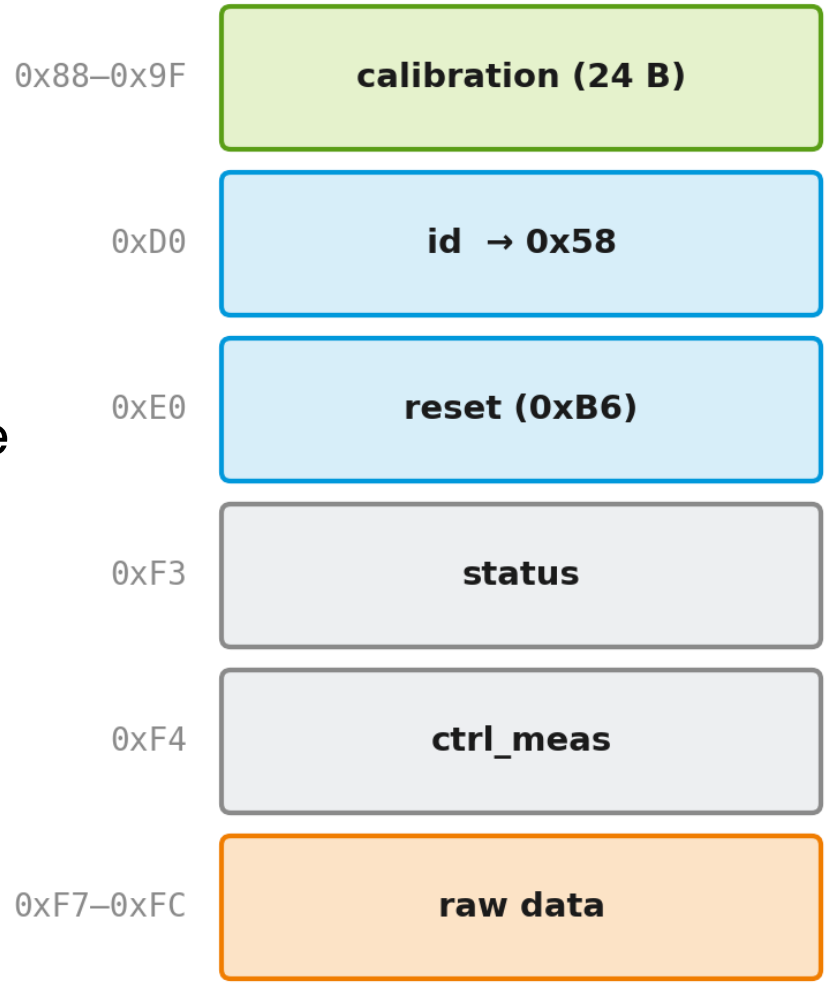
Pressure & temperature over I2C

Inside the BMP280 sensor

Identity & control who am I, how to measure	Calibration 0x88–0x9F	Raw output 20-bit ADC
• id (0xD0) reads 0x58 • reset, status • ctrl_meas, config	• 24 factory bytes • dig_T1..T3 (temperature) • dig_P1..P9 (pressure)	• pressure at 0xF7 • temperature at 0xFA • read both together

The register map

- `0xD0` `id` — chip ID, reads 0x58
- `0xE0` `reset` — write 0xB6 for a soft reset
- `0xF3` `status` — measuring / im_update
- `0xF4` `ctrl_meas` — oversampling + power mode
- `0xF7–0xF9 / 0xFA–0xFC` — raw pressure / temperature
- `0x88–0x9F` `calib` — 24 calibration bytes



Notes and sources

Power modes

- `Sleep` — idle, no measurements
- `Normal` — free-running at a fixed rate
- `Forced` — one measurement, then back to sleep
- We use forced mode: write ctrl_meas, wait, read
  - mode bits revert to 00 after each sample
  - re-trigger before every read

We use FORCED mode: the sensor returns to SLEEP after each sample → re-trigger before every read.

Notes and sources

04

Calibration & Compensation

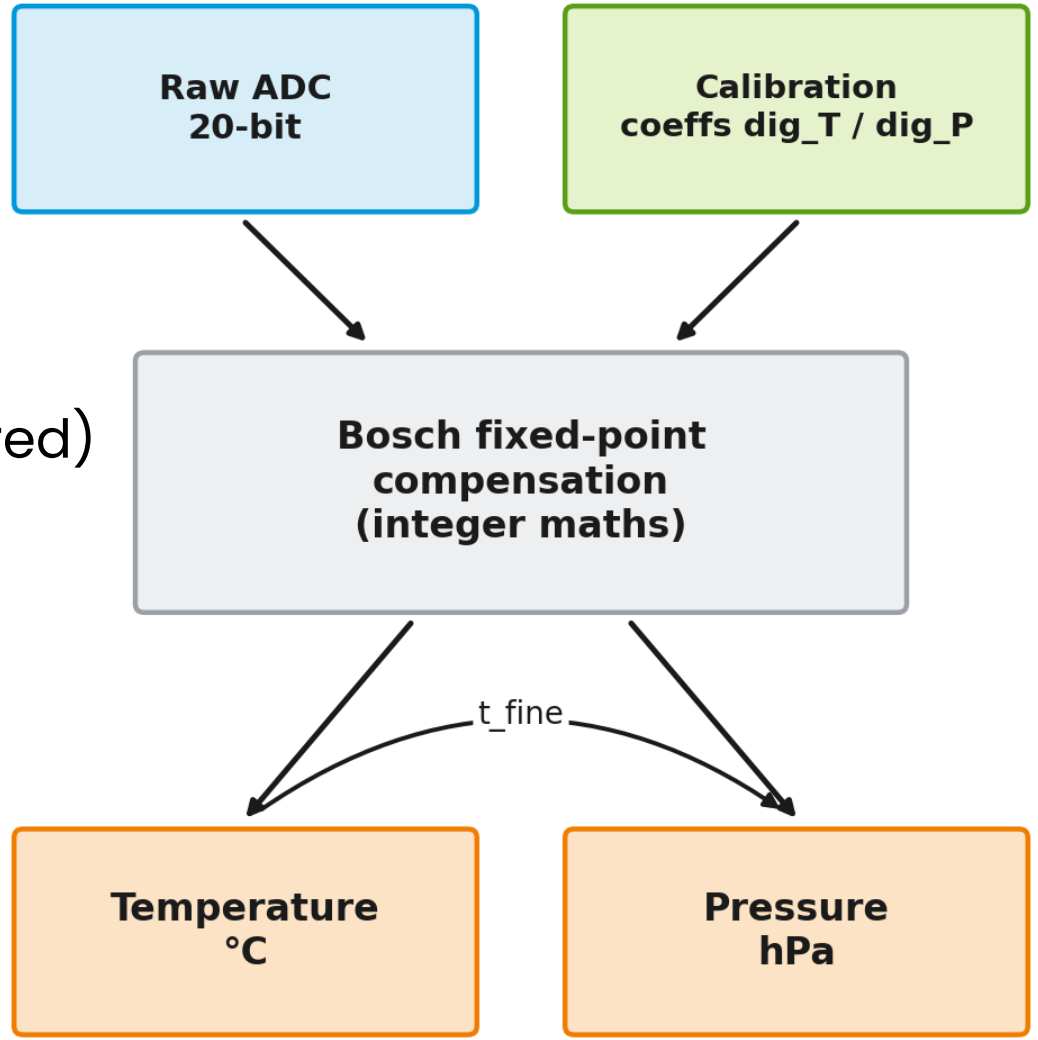
Raw counts aren’t physical units

From ADC counts to °C and hPa

- Factory-calibrated sensor
  - with stored compensation coefficients
- Apply Bosch fixed-point

compensation formulas to raw readings

- Integer-only calculations (no floating-point required)
  - Suitable for kernel-level execution
- Run temperature compensation first
- Temperature calculation generates `t_fine`
- Pressure compensation depends on `t_fine`

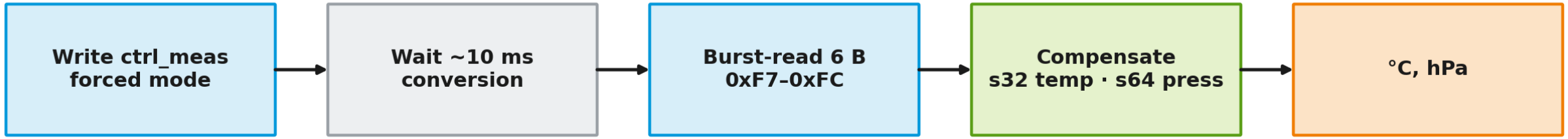


Notes and sources

The compensation math

```
/* temperature — all s32; produces t_fine */
v1 = (((adc_T>>3) - (dig_T1<<1)) * dig_T2) >> 11;
v2 = ((((adc_T>>4) - dig_T1) * ((adc_T>>4) - dig_T1) >> 12) * dig_T3) >> 14;
t_fine = v1 + v2;
temp   = (t_fine*5 + 128) >> 8;     /* 0.01 °C */

/* pressure — same idea, but s64: values overflow 32 bits */
```



mode reverts to sleep after each sample — re-trigger before every read

Notes and sources

i2c-tools

```
# find your bus   (LPI2C7 = 426d0000.i2c)
$ i2cdetect -l
$ i2cdetect -y 3                # scan bus 3 for devices

# read / write a single register
$ i2cget  -y 3 0x76 0xD0        # chip ID    -> 0x58
$ i2cset  -y 3 0x76 0xF4 0x27   # ctrl_meas  -> forced mode

# dump every register of the device
$ i2cdump -y 3 0x76

#  -y  skip the prompt
#  -f  force past a bound driver
#  76  free address
#  UU  driver owns it (get/set -> busy)
```


Notes and sources

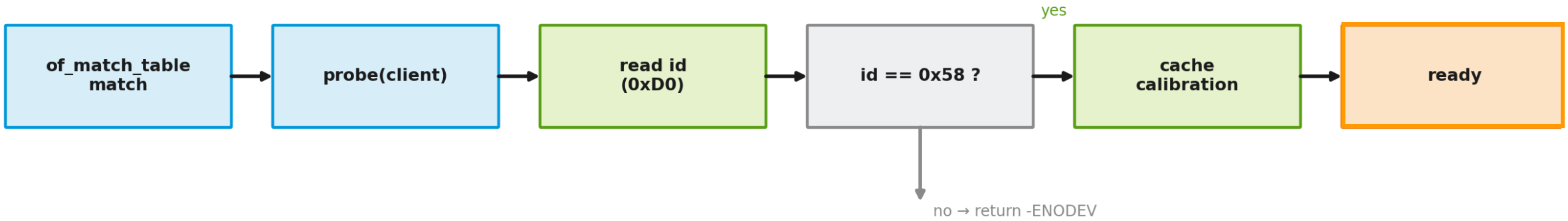
05

Writing the driver

Probe, read, expose

From probe to reading

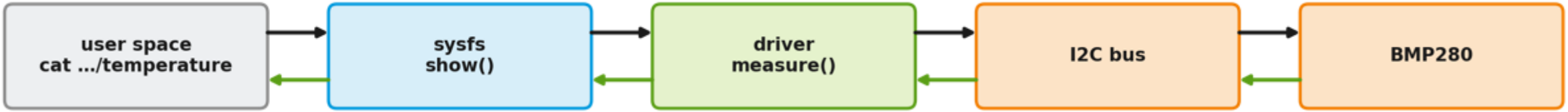
- **Bind & verify**
  - match via of_match_table ("`lkss,bmp280`")
  - probe() receives the i2c_client
  - read id (0xD0); expect 0x58
  - cache calibration once
- **Take a measurement**
  - write ctrl_meas → forced mode
  - wait ~10 ms for the conversion
  - burst-read 6 raw ADC bytes
  - compensate → °C and hPa



Notes and sources

Exposing data to userspace

- **sysfs — this lab**
  - DEVICE_ATTR_RO(temperature, pressure)
  - `show()` triggers a fresh read
  - `sysfs_emit()` returns the text
  - cat …/3-0076/temperature
- **IIO — the real driver**
  - standard framework for sensors
  - in_temp_input, in_pressure_input
  - buffers, triggers, iio_info
  - upstream `bmp280-core.c` uses it



← the value returns along the same path, shown as text (°C / hPa)

Notes and sources

Summary

- I2C: two shared wires, address-selected, controller / target
- A register read is a write-then-read — SMBus hides it
- The device tree binds your driver to the chip (compatible)
- BMP280: read calibration once, then compensate (integer maths)
- Expose readings via sysfs; IIO is the production path
- Check for < 0 on every bus access

Notes and sources

nxp.com