linaro™

Introducing the Power Sequencing Subsystem

Kernel Summit, Linux Plumbers Conference

Vienna, Austria, 2024

Bartosz Golaszewski
Linaro

About me

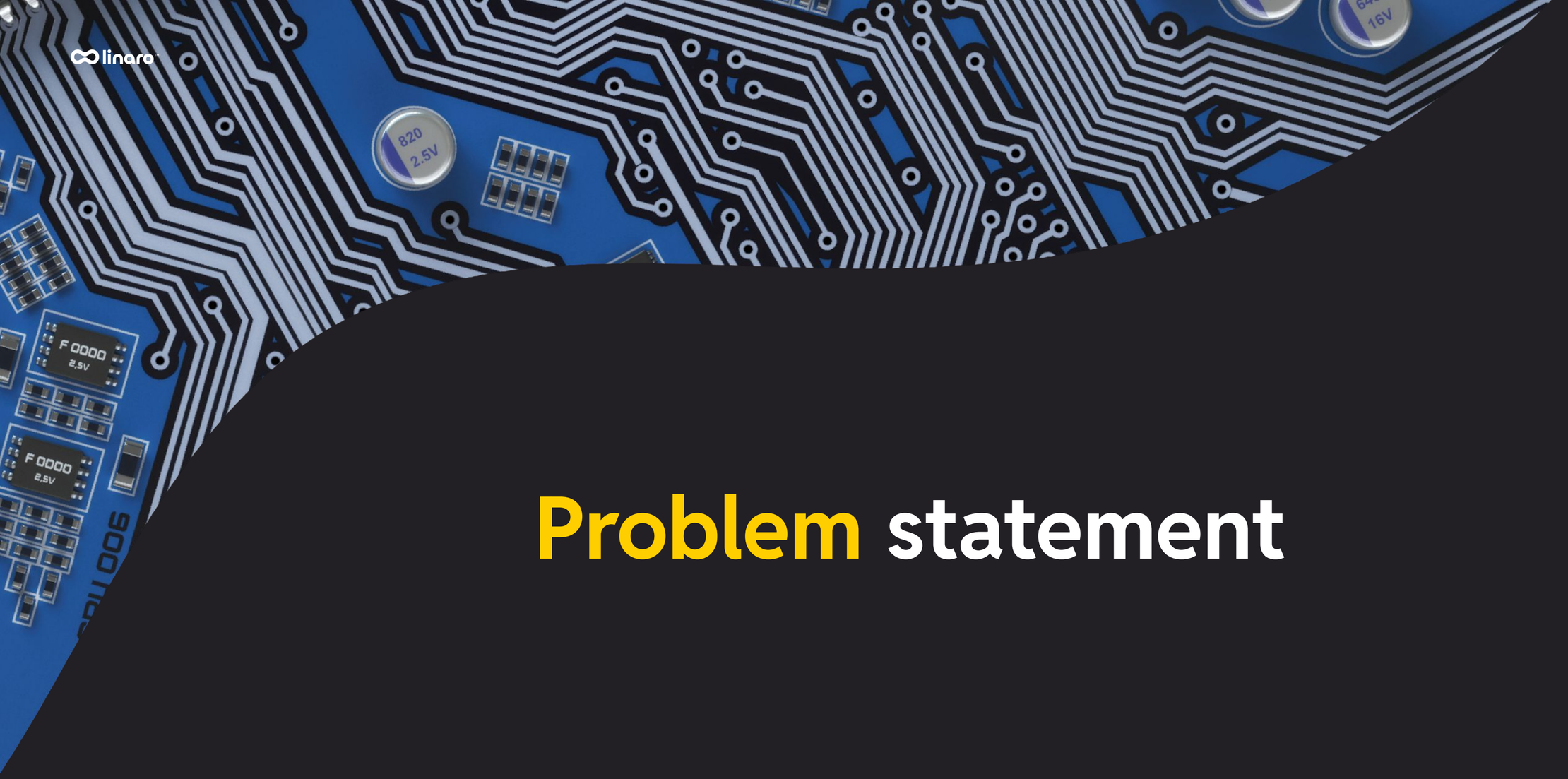
- Linux kernel developer for the Qualcomm Landing Team at Linaro
- 15 years of embedded linux experience
- Maintainer of the GPIO subsystem
- Author and maintainer of libgpiod
- Open-source contributor to many other projects
- Interested in complex software architecture

linaro™ is the software engine of the arm Ecosystem

Linaro empowers rapid product deployment within the dynamic arm Ecosystem.

- Our cutting-edge solutions, services and collaborative platforms facilitate the swift **development, testing, and delivery of arm-based innovations,** enabling businesses to stay ahead in today's competitive technology landscape.
- **Linaro** fosters an environment of collaboration, standardization and optimization among businesses and **open source ecosystems to accelerate the deployment of arm-based products and technologies** along with representing a pivotal role in open source discovery and adoption.
- **Automotive, Testing, Linux Kernel, Security, Cloud & Edge Computing, IoT & Embedded, AI, CI/CD, Toolchain, Virtualization**

Linaro has enabled trust, quality and collaboration since 2010



Problem statement

Dynamic bus chicken-and-egg problem

Dynamic bus chicken-and-egg problem

- Certain devices on dynamic busses need to be powered-up before they can be detected

Dynamic bus chicken-and-egg problem

- Certain devices on dynamic busses need to be powered-up before they can be detected
- The drivers will not power up these devices unless they are detected
  - Typically the drivers binding to these devices handle the resources

Dynamic bus chicken-and-egg problem

- Certain devices on dynamic busses need to be powered-up before they can be detected
- The drivers will not power up these devices unless they are detected
  - Typically the drivers binding to these devices handle the resources
- IOW We must power up the device to detect it but we must detect it to power it up

Sharing inter-dependent resources between devices

Sharing inter-dependent resources between devices

- Certain devices share resources (regulators, resets, clocks, GPIOs)

Sharing inter-dependent resources between devices

- Certain devices share resources (regulators, resets, clocks, GPIOs)
- Reference counting is not enough

Sharing inter-dependent resources between devices

- Certain devices share resources (regulators, resets, clocks, GPIOs)
- Reference counting is not enough
- Additional interactions between multiple devices must be considered

Sharing inter-dependent resources between devices

- Certain devices share resources (regulators, resets, clocks, GPIOs)
- Reference counting is not enough
- Additional interactions between multiple devices must be considered
  - Device may require a delay between enabling two resources

Sharing inter-dependent resources between devices

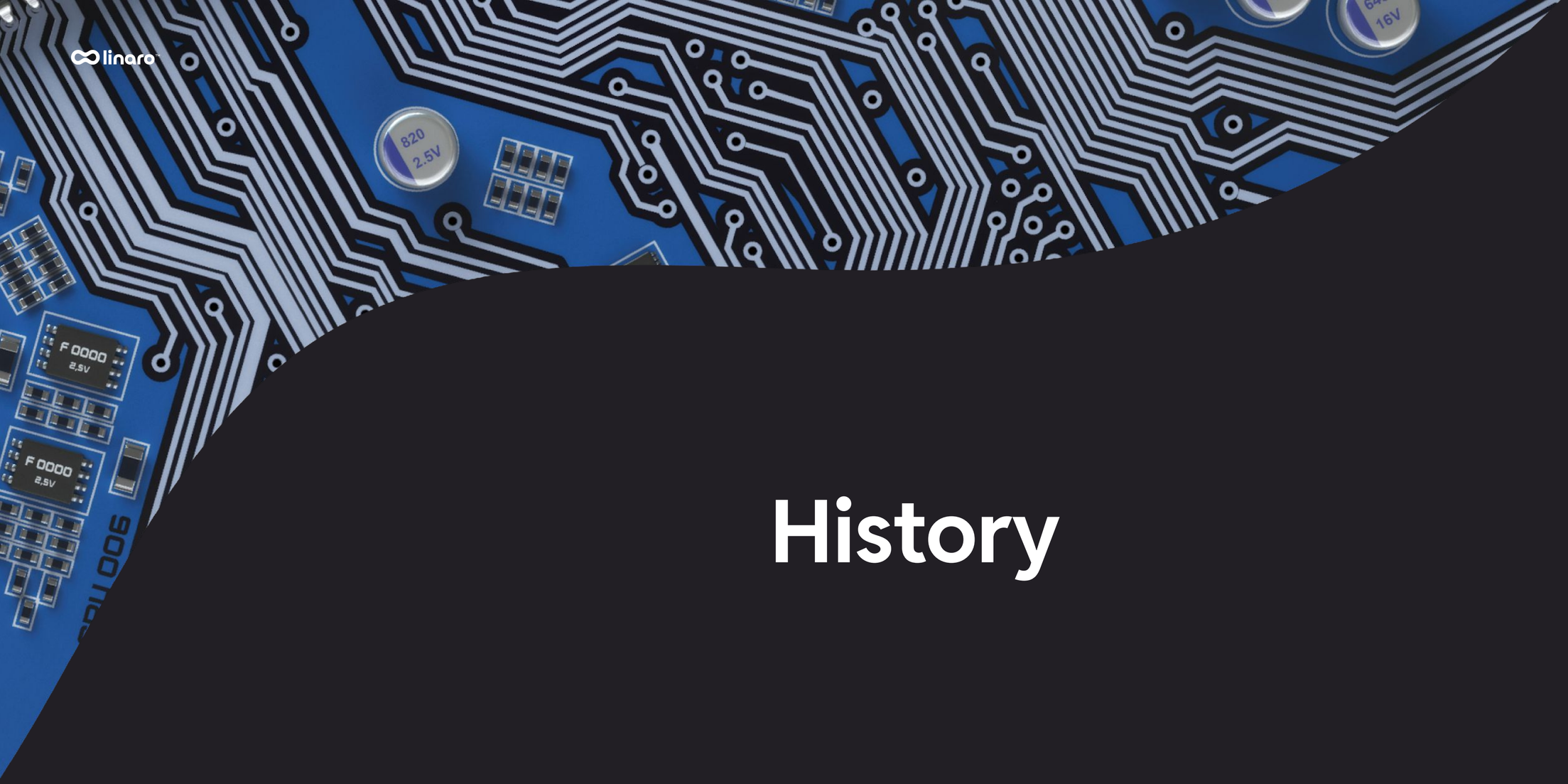
- Certain devices share resources (regulators, resets, clocks, GPIOs)
- Reference counting is not enough
- Additional interactions between multiple devices must be considered
  - Device may require a delay between enabling two resources
  - Resources may have a more complex dependency graph

Sharing inter-dependent resources between devices

- Certain devices share resources (regulators, resets, clocks, GPIOs)
- Reference counting is not enough
- Additional interactions between multiple devices must be considered
  - Device may require a delay between enabling two resources
  - Resources may have a more complex dependency graph
  - The actual power sequence may include specific timings

Sharing inter-dependent resources between devices

- Certain devices share resources (regulators, resets, clocks, GPIOs)
- Reference counting is not enough
- Additional interactions between multiple devices must be considered
  - Device may require a delay between enabling two resources
  - Resources may have a more complex dependency graph
  - The actual power sequence may include specific timings
- Code reuse for complex power-up sequences shared by multiple drivers



History

Previous attempts

Previous attempts

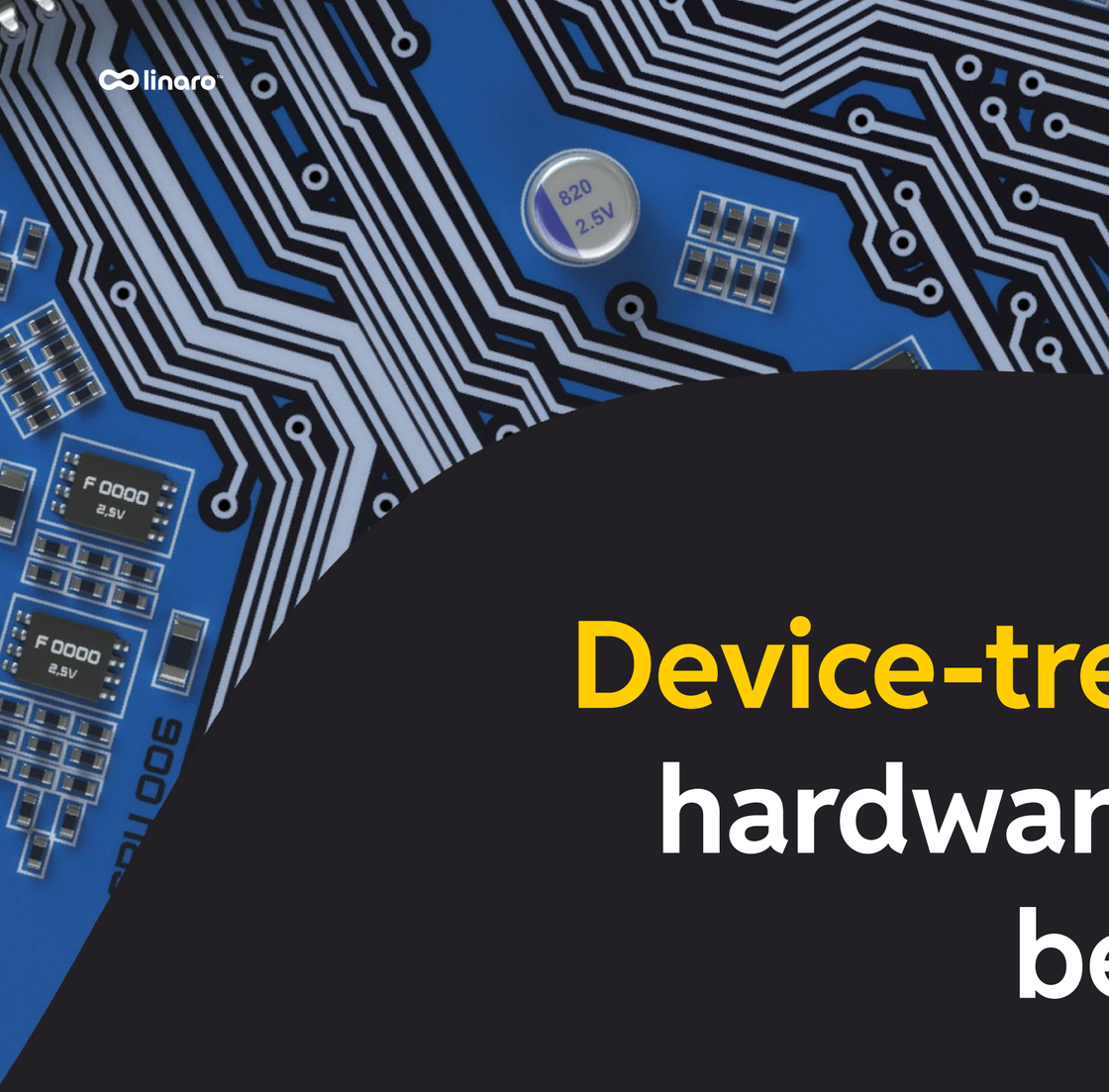
- MMC pwrseq
  - Source of regret for DT maintainers
  - Does the unspeakable in device-tree

Previous attempts

- MMC pwrseq
  - Source of regret for DT maintainers
  - Does the unspeakable in device-tree
- Power Sequencing subsystem by Dmitry Baryshkov
  - https://lore.kernel.org/netdev/20210829131305.534417-1-dmitry.baryshkov@linaro.org/
  - First attempt at enabling BT/WLAN chips in upstream
  - Shot down due to trying to do the unspeakable as well

Previous attempts

- MMC pwrseq
  - Source of regret for DT maintainers
  - Does the unspeakable in device-tree
- Power Sequencing subsystem by Dmitry Baryshkov
  - https://lore.kernel.org/netdev/20210829131305.534417-1-dmitry.baryshkov@linaro.org/
  - First attempt at enabling BT/WLAN chips in upstream
  - Shot down due to trying to do the unspeakable as well
- PCI slot/M.2 driver proposition at LPC 2023



Device-tree describes the hardware itself, not its behavior!

Example

```
// This is OK

foo {
   compatible = "foobar";
   enable-gpios = <&gpio0 0>;
   vdd-supply = <&host_pmic_out0>;
   resets = <&some_rst FOOBAR>;
};
```

```
// This is *NOT* OK

foo {
   compatible = "foobar";
   pwrseq = <&pwrseq_provider>;
};
```

Device-tree and driver code don’t have to correspond to each other 1:1

DT vs C

- Typically driver subsystems have common code handling device-tree nodes
  - For instance `gpiochip_add_data()` will parse the DT node looking for common GPIO chip properties

DT vs C

- Typically driver subsystems have common code handling device-tree nodes
  - For instance `gpiochip_add_data()` will parse the DT node looking for common GPIO chip properties
- This is just an implementation detail, there's no rule that states that a DT node called `pmic@0` must become a regulator provider and that its driver must call `regulator_register()`

DT vs C

- Typically driver subsystems have common code handling device-tree nodes
  - For instance `gpiochip_add_data()` will parse the DT node looking for common GPIO chip properties
- This is just an implementation detail, there's no rule that states that a DT node called `pmic@0` must become a regulator provider and that its driver must call `regulator_register()`
- We can actually do whatever makes sense with a DT node as long as it keeps on correctly describing the underlying hardware

High-level abstraction: Power Sequencing subsystem (pwrseq)

Power Sequencing subsystem concept

Power Sequencing subsystem concept

- Simple interface for consumers:
  - get()/put()
  - power_on/off()

Power Sequencing subsystem concept

- Simple interface for consumers:
  - get()/put()
  - power_on/off()
- Powerful for providers:
  - Concept of targets, units and dependencies
  - Run-time consumer <-> provider matching
  - Flexible interpretation of device nodes
    - May bind to nodes that would otherwise looks like they “belong” to a different subsystem

Power Sequencing consumers

```
struct pwrseq_desc *desc;
int ret;

desc = pwrseq_get(dev, "foo");
if (IS_ERR(desc))
   return PTR_ERR(desc);

ret = pwrseq_power_on(desc);
if (ret)
   return ret;

ret = pwrseq_power_off(desc);
if (ret)
   return ret;

pwrseq_put(desc);
```

Power Sequencing consumers

```
struct pwrseq_desc *desc;
int ret;

desc = pwrseq_get(dev, "foo");
if (IS_ERR(desc))
   return PTR_ERR(desc);

ret = pwrseq_power_on(desc);
if (ret)
   return ret;

ret = pwrseq_power_off(desc);
if (ret)
   return ret;

pwrseq_put(desc);
```

`dev` is the consumer device, “`foo`” is the name of the pwrseq target

Power Sequencing consumers

```
struct pwrseq_desc *desc;
int ret;

desc = pwrseq_get(dev, "foo");
if (IS_ERR(desc))
   return PTR_ERR(desc);

ret = pwrseq_power_on(desc);
if (ret)
   return ret;

ret = pwrseq_power_off(desc);
if (ret)
   return ret;

pwrseq_put(desc);
```

`dev` is the consumer device, `"foo"` is the name of the pwrseq target

`struct pwrseq_desc` is a proxy protecting the internal reference counting

Power Sequencing consumers

```
struct pwrseq_desc *desc;
int ret;

desc = pwrseq_get(dev, "foo");
if (IS_ERR(desc))
   return PTR_ERR(desc);

ret = pwrseq_power_on(desc);
if (ret)
   return ret;

ret = pwrseq_power_off(desc);
if (ret)
   return ret;

pwrseq_put(desc);
```

`dev` is the consumer device, `"foo"` is the name of the pwrseq target

`struct pwrseq_desc` is a proxy protecting the internal reference counting

`devm_pwrseq_get()` is also available

Power Sequencing providers

- Unit
  - Discrete part of the power on/off sequence
  - Binary state: enabled/disabled
    - Enable state is counted
  - May have a list of dependencies
    - They must be enabled before this unit
    - This unit must be disabled before any of its dependencies
  - Examples:
    - Enable clock
    - Enable a GPIO
    - Deassert a reset

Power Sequencing providers

- Target
  - Named unit that can be selected by consumers
  - Consists of the “target” unit and its dependencies
  - Multiple targets may share parts of the power sequence: for instance the `“bluetooth”` and `“wlan”` targets may share the `“regulator-enable”` unit
  - Examples:
    - `“bluetooth”` target
      - `“bluetooth-enable”` is the target unit
      - `“bluetooth-enable”` depends on `“clock-enable”`, `“regulators-enable”` and `“gpio-enable”`

Power Sequencing providers

- Descriptor
  - Opaque handle provided to consumers that want to use the power sequencer
  - References a single target
  - Assigned by `pwrseq_get()`
  - Can only ever increase and decrease the enable count of a target by 1

Power Sequencing providers

```
# cat /sys/kernel/debug/pwrseq
pwrseq.0:
  targets:
    target: [bluetooth] (target unit: [bluetooth-enable])
    target: [wlan] (target unit: [wlan-enable])
  units:
    unit: [regulators-enable] - enable count: 2
    unit: [clock-enable] - enable count: 2
    unit: [bluetooth-enable] - enable count: 1
      dependencies:
        [regulators-enable]
        [clock-enable]
    unit: [wlan-enable] - enable count: 1
      dependencies:
        [regulators-enable]
        [clock-enable]
```

Power Sequencing providers

- Run-time matching
  - Each provider driver provides a .match() callback
    - The functionality of this callback is entirely driver-specific
  - Consumer calls pwrseq_get()
  - The pwrseq core goes through the list of registered providers and calls the .match() callback passing it the (potential) consumer device
  - If the function returns non-zero, we assume it's a match
- Example matching mechanism:
  - We expect that the consumer takes the `vdd-supply` from the provider's node
  - We parse the `vdd-supply` phandle and see if it exists and leads us to the provider node

Low-level abstraction: PCI power control

PCI Power Control concept

```
&pcieport0 {
   wifi@0 {
      compatible = "pci17cb,1101";
      ...
   };
};
```

PCI Power Control concept

```
&pcieport0 {
   wifi@0 {
      compatible = "pci17cb,1101";
      ...
   };
};
```

PCI Core binds the DT node to the matching PCI device when it detects it on the host bridge

PCI Power Control concept

```
&pcieport0 {
   wifi@0 {
      compatible = "pci17cb,1101";
      ...
   };
};
```

PCI Core binds the DT node to the matching PCI device when it detects it on the host bridge

A DT node can be consumed by multiple devices

PCI Power Control concept

```
&pcieport0 {
   wifi@0 {
      compatible = "pci17cb,1101";
      ...
   };
};
```

PCI Core binds the DT node to the matching PCI device when it detects it on the host bridge

A DT node can be consumed by multiple devices

Idea: Let’s create platform devices for child nodes of the host bridge and use their platform drivers to power-up the device

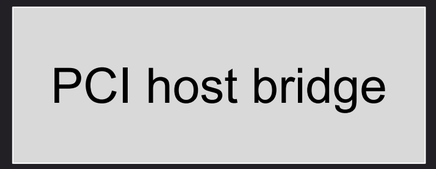
PCI Power Control concept

- Reuse DT nodes for known PCI devices
- Create platform devices that bind to these nodes
- Enable relevant resources
- Let PCI core know that it can now rescan the bus
- Let the platform device become the parent of the PCI device binding to the same node so that the suspend/resume callbacks of the former are always called after/before those of the latter respectively

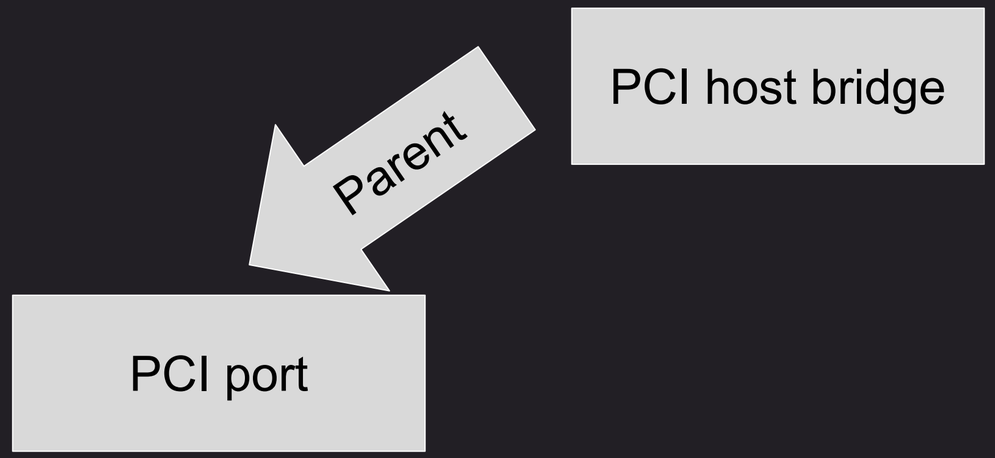
PCI Power Control implementation

- Minimal API
  - `pci_pwrctl_device_set/unset_ready()` + single configuration structure
- Populate platform devices for PCI of nodes when the host bridge is probed
- Mark the relevant OF node as reused to avoid pinctrl issues

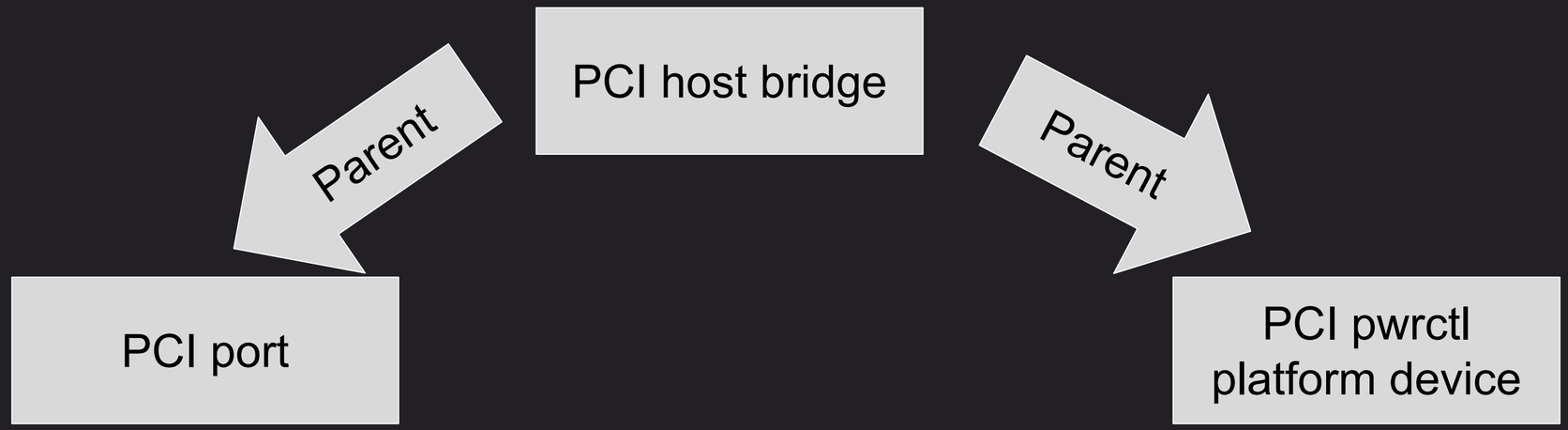
PCI Power Control implementation



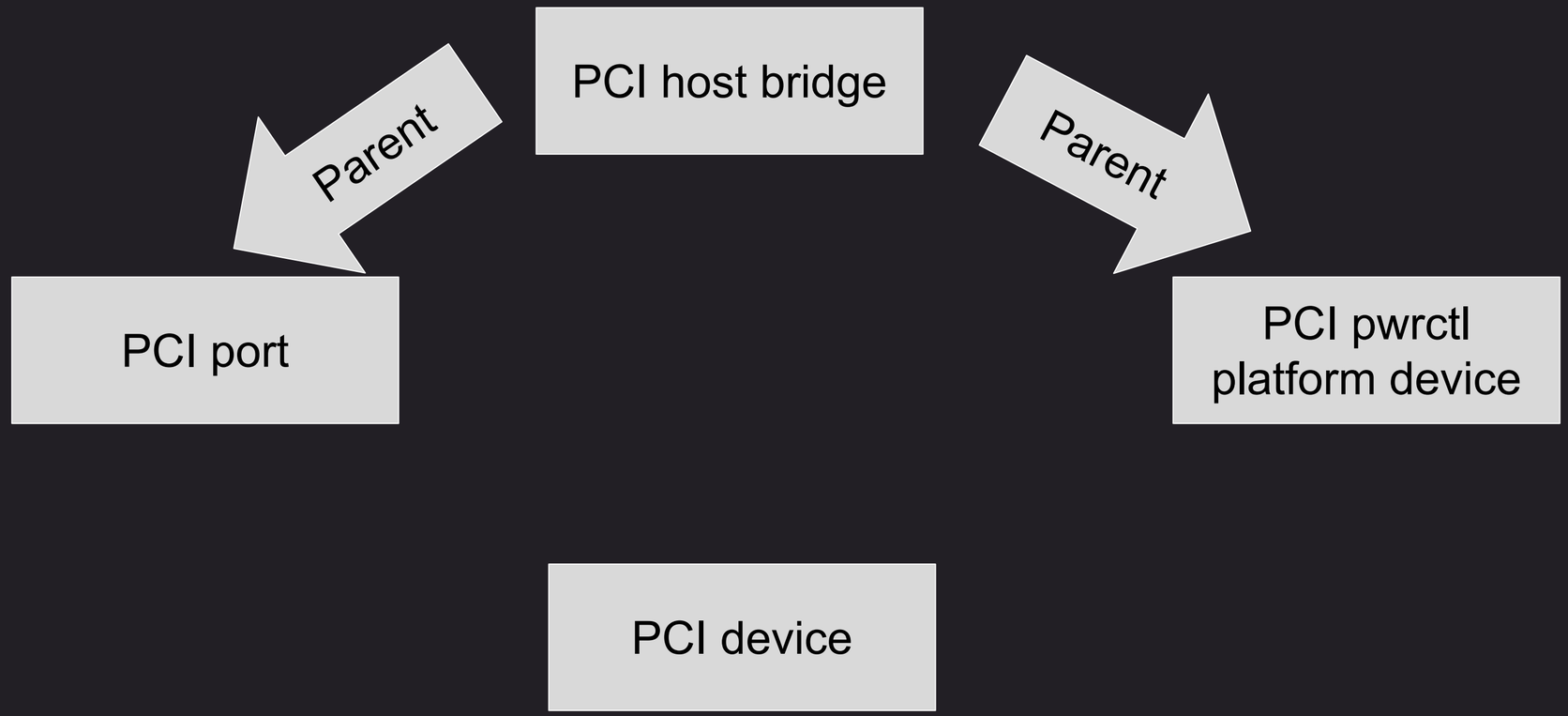
PCI Power Control implementation



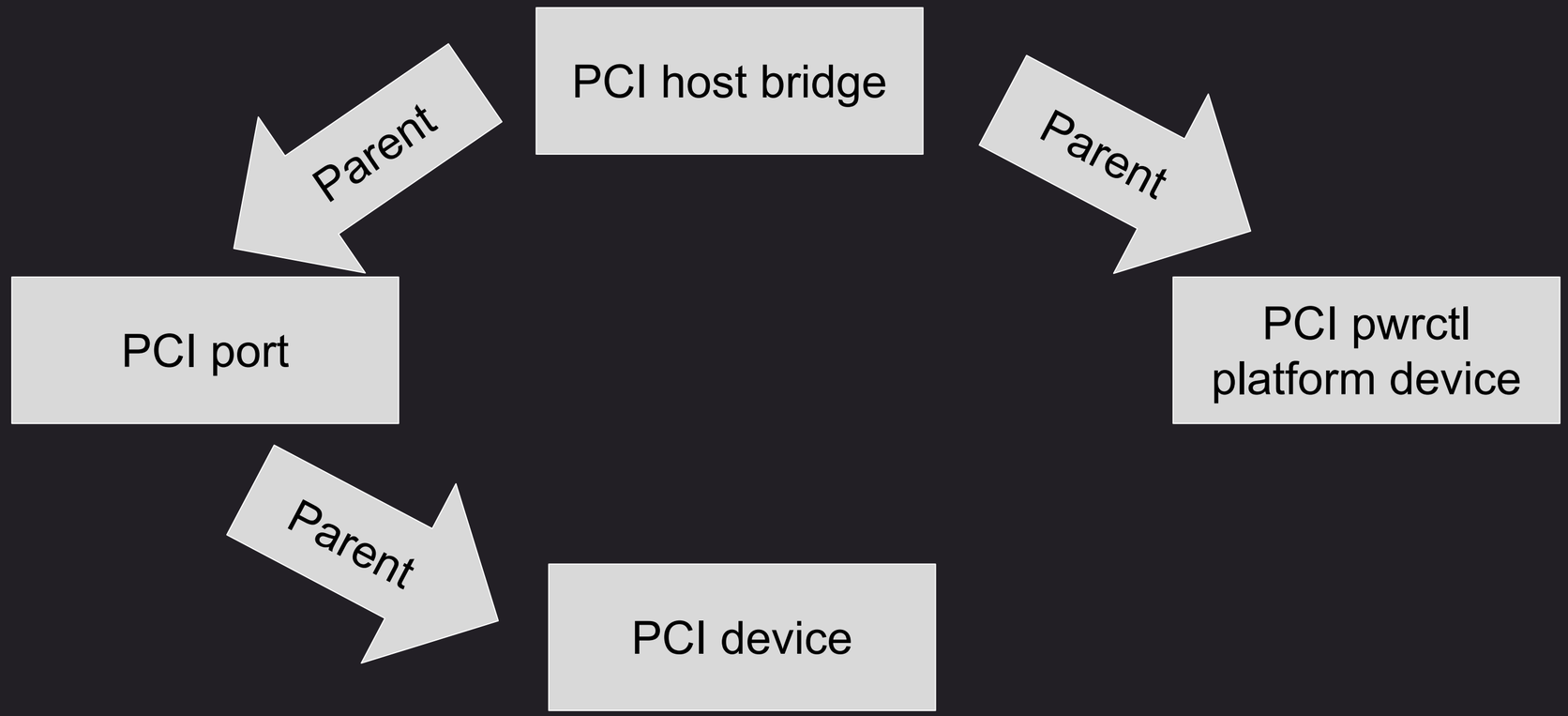
PCI Power Control implementation



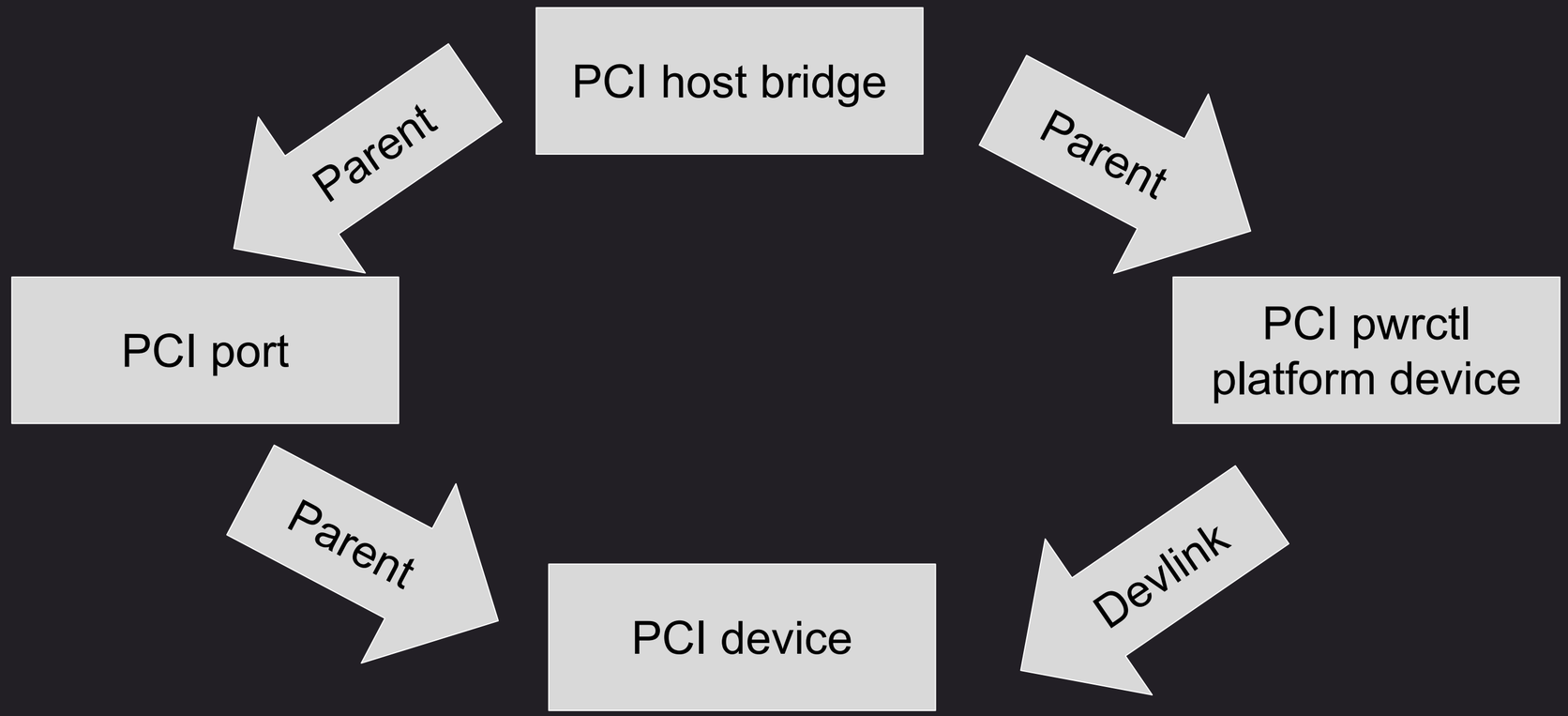
PCI Power Control implementation



PCI Power Control implementation



PCI Power Control implementation



By your powers combined:
pwrseq + PCI pwrctl

Where are we at?

- All core pwrseq and PCI pwrctl code is in mainline and has been released as part of v6.11
- Some creases are still being ironed-out
- RB5, sm8650-qrd, sm8650-hdk, sm8550-qrd, sc8280xp-crd and X13s use pwrseq for Bluetooth and WLAN
- Already got the first submission (although bad...) for a new pwrseq driver from Amlogic which further proves that this is something that was needed for a long time
- Process needs to be improved
  - which tree should the DT bindings go through

Q & A

Thank You!

Visit linaro.org

Contact me at:
bartosz.golaszewski@linaro.org