# fw_devlink: Device dependency tracking

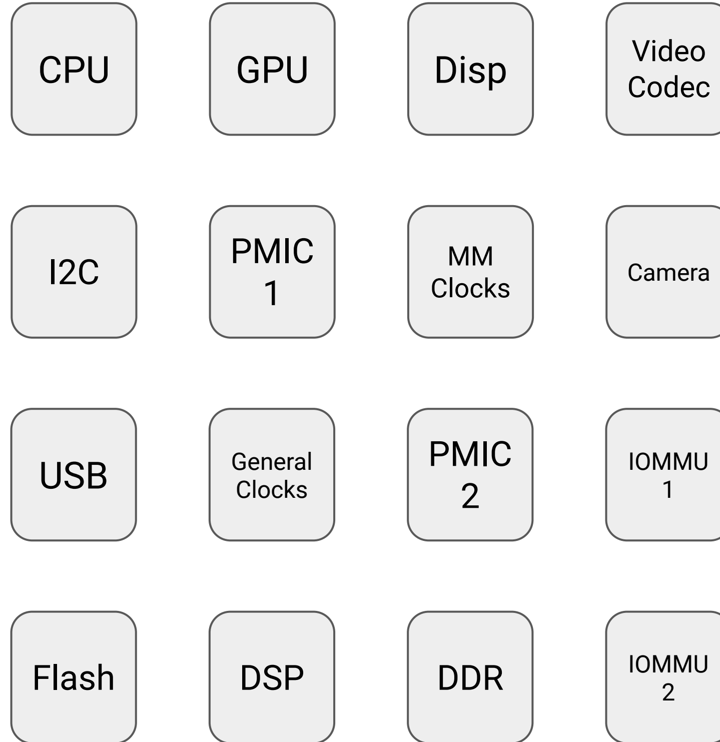
## What's new, leveraging it, and next steps

Saravana Kannan (Google)

LINUX
PLUMBERS
CONFERENCE Vienna, Austria / Sept. 18-20, 2024

# Introduction

# Devices in a typical SoC



A → B ***means*** A depends on resources provided by B

# Device dependencies (simplified)

CPU
GPU
Disp
Video Codec
I2C
PMIC 1
MM Clocks
Camera
USB
General Clocks
PMIC 2
IOMMU 1
Flash
DSP
DDR
IOMM 2

# Device tree example

```
soc: soc@0 {
        compatible = "simple-bus";
        interrupt-parent = <&gic>;

        cmu_misc: clock-controller@10010000 {
                compatible = "google,gs101-cmu-misc";
                clocks = <&cmu_top CMU_MISC_BUS>,
                         <&cmu_top CMU_MISC_SSS>;
        }

        gic: interrupt-controller@10400000 {
                compatible = "arm,gic-v3";
                #interrupt-cells = <4>;
                interrupt-controller;
                interrupts = <GIC_PPI 9 LEVEL_HIGH 0>;
        }

        ufs_0: ufs@14700000 {
                compatible = "google,gs101-ufs";
                interrupts = <GIC_SPI 532 LEVEL_HIGH 0>;
                clocks = <&cmu_hsi2 HSI2_UFS_EMBD_I_ACLK>;
                pinctrl-0 = <&ufs_rst_n &ufs_refclk_out>;    }
                phys = <&ufs_0_phy>;
        };
```

```
ufs_0_phy: phy@14704000 {
        compatible = "google,gs101-ufs-phy";
        clocks = <&ext_24_5m>;
};

usi1: usi@109000c0 {
        compatible = "google,gs101-usi";
        clocks = <&cmu_peric0 TOP0_PCLK_0>;
        hsi2c_1: i2c@10900000 {
                compatible = "google,gs101-hsi2c";
                clocks = <&cmu_misc TOP0_IPCLK_0>;
                interrupts = <GIC_SPI 6 LEVEL_HIGH 0>;
                pinctrl-0 = <&hsi2c1_bus>;
        };
}

watchdog_cl0: watchdog@10060000 {
        compatible = "google,gs101-wdt";
        clocks = <&cmu_misc MISC_WDT_CLUSTER0_PCLK>;
        interrupts = <GIC_SPI 765 LEVEL_HIGH 0>;
};
```

# fw_devlink: Overview

- Parses firmware to determine dependencies.
- Doesn't depend on drivers for correctness (needs to work for a fully modular kernel).
- Currently supports DT (33 different properties).
- Creates fwnode links to track consumer-supplier relationship between DT nodes.
- fwnode links are converted into device links when the consumer and supplier struct devices are created from the DT nodes.

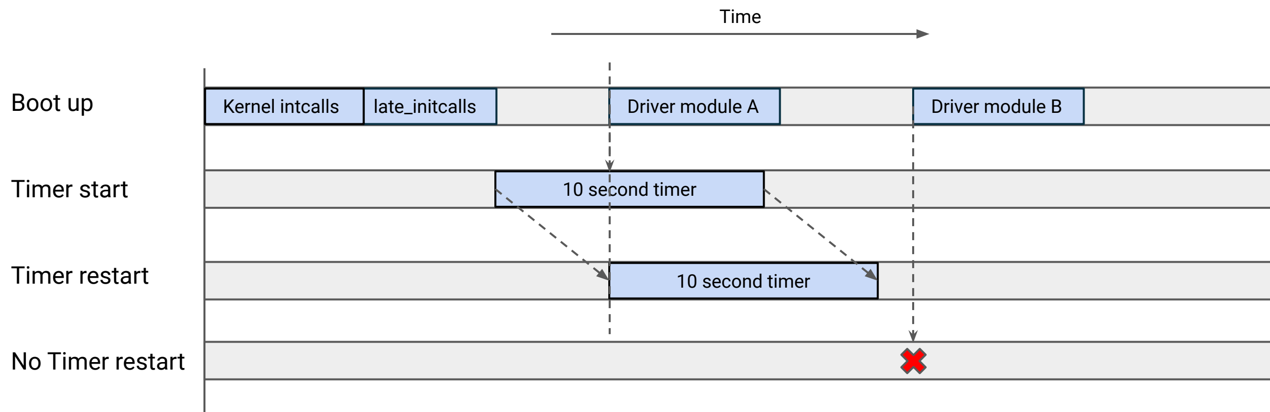
# What’s new(ish)?

# No more initcall chicken

- Your driver's initcall level is completely irrelevant when it comes to ensuring ordering with your suppliers.
- fw_devlink guarantees your suppliers will probe before your device.
- Don't need any special handling for optional supplier.
- If your supplier doesn't have a driver, fw_devlink will still allow your device to probe after deferred probe timeout expires.

# Smarter deferred probe timeout

- `deferred_probe_timeout` will now auto extend whenever a new driver is registered.
- This is true even when modules register drivers.
- So, as long as all the modules in the system are loaded early during boot, everything will work automatically if `deferred_probe_timeout` is set.
- Example with 10s `deferred_probe_timeout`:

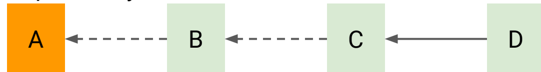


# Better functionality after timeout

- Smarter about relaxing (not blocking on) supplier dependencies after timeout expires.
- After timeout, if device probe retry order happens to be D, C, B and A:

Deferred probe list order:

Dependency info:




Without fw_devlink:

With fw_devlink:

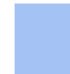
Consumer pointing to mandatory supplier.

Consumer pointing to optional supplier.

No driver available for this device.

Device with a matching (but not probed) driver.

Probes with full functionality


Probes with limited functionality

Fails to probe

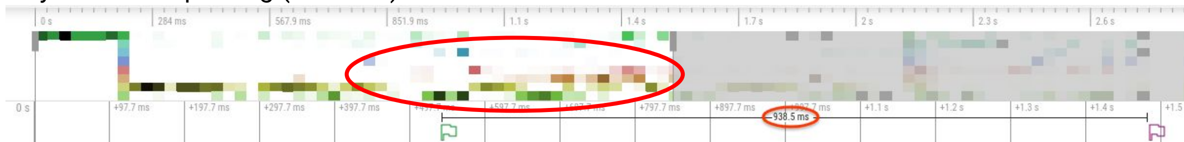
# Runtime PM enforcement

- `fw_devlink=rpm` is the default mode.
- Enforces runtime PM state between consumers and suppliers.
- Resuming a consumer automatically resumes the supplier.
- Runtime PM stability should be a lot easier to achieve.

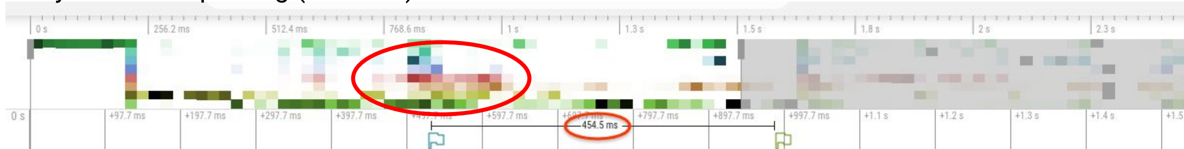
# Reliable async probing

- Parallelized async probing is dependable/stable now for most boards/systems. Give it a shot
- `driver_async_probe=*` will now enable async probing by default for all the drivers.
- `driver_async_probe=*,drvA,drvB,drvC` will enable async for all drivers except drivers A, B and C.
- To avoid character limit of `driver_async_probe`, you can also use `module.async_probe=1` and `drvA-module.async_probe=0` for modules.

Synchronous probing (~900ms)



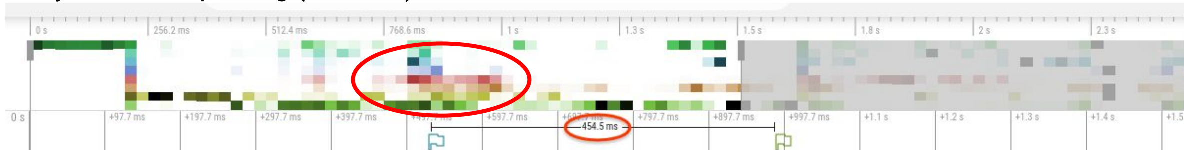

Asynchronous probing (~450ms)



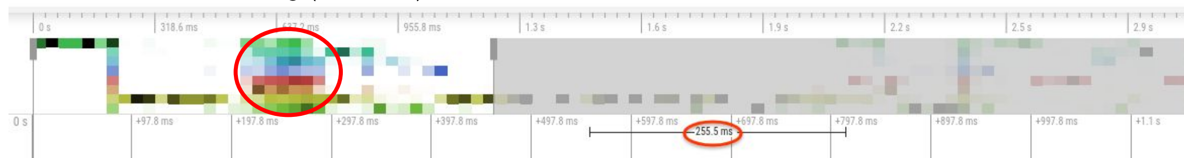
# Reliable parallel module loading

- Even better than async probing is actual parallel module loading.
- Needs userspace updates to load multiple modules in parallel.
- For example, Android 13 and later will load modules in parallel if you set `androidboot.load_modules_parallel=1` in the kernel command line.
- Do other userspace module loaders already support this? If not, might want to add support for this.

Asynchronous probing (~450ms)




Parallel module loading (~250ms)



# Reliable async suspend/resume

- Parallelized async suspend/resume is dependable/stable now. Give it a shot
- No command line option for this, but you can run this to enable it after boot to go full async:

  ```
  find /sys/devices/ -name async | while read -r filename; do echo
  enabled > "$filename"; done
  ```

- Very stable on downstream Pixel 6 with no additional driver fixes necessary.
- See also: My other talk on why doing it for every device might not be the most optimal configuration.

# sync_state(): Safe release of unused resources

- Bootloader might leave a resource on (say, a power domain) before jumping to kernel.
- It's safe to turn off the resource only after all the consumers (say, USB and display) have probed successfully.
- How do you know when that is? Don't reinvent the wheel.
- Drivers get a `.sync_state(dev)` callback when all the consumers of a device have probed.
- `/sys/devices/.../state_synced` is present if your device has a `sync_state()`.
- `1` means it has been called. `0` means it has not been called.
- If you want to globally timeout waiting for consumers, set `fw_devlink.sync_state` to `timeout`. Default is `strict` which waits forever. See also: `CONFIG_FW_DEVLINK_SYNC_STATE_TIMEOUT`
- If you want to use `strict` but force `sync_state()` for one supplier, write `1` to the supplier's `/sys/devices/.../state_synced`.

# Dependency info in sysfs

- `/sys/class/devlink` provides a lot of details about device dependencies.
- One folder per device link. Folder name format: `<supplier>--<consumer>`
- Check out Documentation for all the details.
- Each `/sys/devices/` has `supplier:*` and `consumer:*` symlinks to these device link folders.
- Very handy if you want to decide which drivers to modularize first, to upstream first, etc.

# Effectively leveraging fw_devlink and device links

# Follow the device-driver model

- All of this dependency tracking and all the benefits go away if you don't use the device-driver model.
- Don't directly parse a DT node and start providing services/APIs/resources.
- Don't use any variant of the `OF_DECLARE` macros like `CLK_OF_DECLARE`, `IRQCHIP_DECLARE` or `TIMER_OF_DECLARE`.
- The only valid users of these are the sched timer/clock and the root IRQ chip.
- Even if your driver is simple now, it'll inevitably cause issues in the future.
- Plenty of examples, but I don't want to shame anyone here.
- Please create a device out the DT node/fwnode and probe it with your driver.

# Setting fwnode for devices

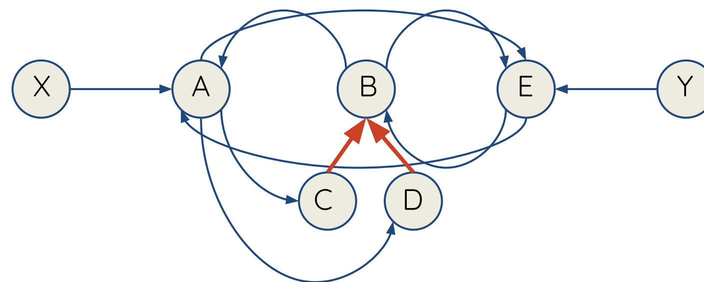
- If your framework/driver creates a new `struct device` from a `struct device_node`, the device's `.fwnode` field must be set using `device_set_node()`. Note: For now, do this only for "bus" devices.
- Create only one `struct device` from a `struct device_node`.
- If one fwnode / `struct device_node` has multiple devices, fw_devlink cannot determine which device a consumer depends on.
- If a DT node has multiple features, use one device that registers with multiple frameworks instead of creating multiple devices from one DT node. For example, a single device/driver can register with both the clocks and power domain frameworks.
- If features are clearly separate hardware blocks, represent them as subnodes in the device tree so that each `struct device` has its own `struct device_node`.
- This approach improves dependency tracking and resource management.

# To bus or not to bus? That is the question.

- Each `struct device` must belong to a `struct bus` or `struct class`.
- Think of a bus as representing a communication protocol. Eg: I2C, USB, PCIE, platform (memory mapped IO).
- Devices within a bus can have distinct functions and are probed by device-specific drivers.
- A class is a logical grouping of devices with similar functionality. Eg: GPUs, ethernet devices, LEDs, regulators, and RTCs.
- Devices in a class implement class-specific operations and are added (not probed) by drivers.
- Framework devs: **Don't use a bus in your framework if you don't intend to probe the devices.**
  - Too many examples of this.
  - This (rightfully) confuses fw_devlink and it'll wait indefinitely for your devices to probe.

# Break cycles using post-init-providers

- fw_devlink can detect cyclic dependencies in device tree.
- fw_devlink doesn't enforce ordering between devices in a cycle.
- Less ordering enforcement leads to less determinism and stability.
- A, B, C, D and E are part of a cycle and aren't ordered between them.
- X and Y are still ordered with respect to A and E respectively.
- Probe/suspend/resume/runtime PM cycles can't exist by definition.
- `post-init-providers` property in DT informs which dependency should be ignored to break the cycle.
- So, use `post-init-providers` to break cycles reported by fw_devlink to improve probe/suspend/resume determinism and stability.




Pointy end of arrow is at supplier/parent
Red arrows indicate parent-child relationship
These cycles are from real world scenarios

# Next steps

# Add ACPI support?

- I don't have much/any experience with ACPI.
- If inter-device dependency information can be derived from ACPI nodes (`struct acpi_device`) please work with me to add support for it.
- fw_devlink was designed with the intent of making it easy to add support for different firmware types.

# Device links & class devices

- **Managed devices links are device links that enforce probe ordering and auto consumer probe/unbind when the supplier probes/unbinds.**
- **Managed device links don't properly handle the case where the supplier belongs to a `struct class`.**
- **The consumer is indefinitely blocked from probing because the supplier never probes.**
- **The consumer is not unbound (if the device link was created after consumer probes) when the supplier is removed.**
- TODO:
  - Fix handling of `class` devices to treat addition/removal similar to probing/unbind of bus devices.
  - Fix auto probe/unbind to handle removal of `class` suppliers.

# More sync_state() support in frameworks

- Ulf is working on adding `sync_state()` support to power domains.
- I've signed up to finish up my patch for clocks, but haven't gotten around to it in a while.
- Need to fix the class stuff to try and add `sync_state()` support for regulators.
- Interconnect framework is the only framework with `sync_state()` support so far.

# Multiple devices from a fwnode/DT node

- As mentioned before, if one fwnode / `struct device_node` has multiple devices, fw_devlink cannot determine which device a consumer depends on.
- When this happens, fw_devlink just picks the first device that was created from a device node.
- No clear/good solution that doesn't involve the drivers.
- Might give a way to mark a device as "don't use as supplier" as a good enough solution for now.

# Thank you!

Questions?