Deferred Problem

Issues With Complex Dependencies Between Devices in Linux Kernel

Andrzej Hajda

Samsung R&D Institute Poland

October 19, 2018

Problem Description - Hardware

Hardware: Device with One Interface

PCI cards, USB devices:

- only one control interface,
- connected to only one device (bus master) via control interface,
- all signals are supplied by the bus or does not need to be controlled by software (including power),
- usually are discoverable,
- devices forms nice tree of dependencies.

One-interface-per-device design comes from PC World (really?).

Hardware: Device with Multiple Interfaces

Integrated circuit (IC, chip) or Semiconductor intellectual property core (IP):

- connected to many devices - power supplies, clock lines, gpios, data buses,
- multiple control buses or no control bus at all (only power and gpio lines),
- requires close cooperation with connected devices,
- dependencies between devices forms complicated graphs (sometimes even with cycles).

Typical for Embedded World but not-limited to.

Hardware: Nice Example

MHL 3.0 Transmitter (Sil8620) pins:

- I2C slave control interface - to I2C master,

Hardware: Typical Example

MHL 3.0 Transmitter (Sil8620) pins:

- I2C slave control interface - to I2C master,
- 10 power lines - to different power supplies (at least 2),

Hardware: Typical Example

MHL 3.0 Transmitter (Sil8620) pins:

- I2C slave control interface - to I2C master,
- 10 power lines - to different power supplies (at least 2),
- interrupt line - to interrupt controller,

Hardware: Typical Example

MHL 3.0 Transmitter (Sil8620) pins:

- I2C slave control interface - to I2C master,
- 10 power lines - to different power supplies (at least 2),
- interrupt line - to interrupt controller,
- clock line - to clock controller,

Hardware: Typical Example

MHL 3.0 Transmitter (Sil8620) pins:

- I2C slave control interface - to I2C master,
- 10 power lines - to different power supplies (at least 2),
- interrupt line - to interrupt controller,
- clock line - to clock controller,
- reset line - to GPIO controller,

Hardware: Typical Example

MHL 3.0 Transmitter (Sil8620) pins:

- I2C slave control interface - to I2C master,
- 10 power lines - to different power supplies (at least 2),
- interrupt line - to interrupt controller,
- clock line - to clock controller,
- reset line - to GPIO controller,
- TMDS lines - to HDMI video source,

Hardware: Typical Example

MHL 3.0 Transmitter (Sil8620) pins:

- I2C slave control interface - to I2C master,
- 10 power lines - to different power supplies (at least 2),
- interrupt line - to interrupt controller,
- clock line - to clock controller,
- reset line - to GPIO controller,
- TMDS lines - to HDMI video source,
- MHL lines - to MHL connector,

Hardware: Typical Example

MHL 3.0 Transmitter (SiI8620) pins:

- I2C slave control interface - to I2C master,
- 10 power lines - to different power supplies (at least 2),
- interrupt line - to interrupt controller,
- clock line - to clock controller,
- reset line - to GPIO controller,
- TMDS lines - to HDMI video source,
- MHL lines - to MHL connector,
- Hot-Plug line - to GPIO controller,

Hardware: Nasty Example

MHL 3.0 Transmitter (Sil8620) pins:

- I2C slave control interface - to I2C master,
- 10 power lines - to different power supplies (at least 2),
- interrupt line - to interrupt controller,
- clock line - to clock controller,
- reset line - to GPIO controller,
- TMDS lines - to HDMI video source,
- MHL lines - to MHL connector,
- Hot-Plug line - to GPIO controller,
- I2C slave (DDC) - to I2C master,

Hardware: Nasty Example

MHL 3.0 Transmitter (Sil8620) pins:

- I2C slave control interface - to I2C master,
- 10 power lines - to different power supplies (at least 2),
- interrupt line - to interrupt controller,
- clock line - to clock controller,
- reset line - to GPIO controller,
- TMDS lines - to HDMI video source,
- MHL lines - to MHL connector,
- Hot-Plug line - to GPIO controller,
- I2C slave (DDC) - to I2C master,
- SPI slave control interface - to SPI master,

Hardware: Nasty Example

MHL 3.0 Transmitter (Sil8620) pins:

- I2C slave control interface - to I2C master,
- 10 power lines - to different power supplies (at least 2),
- interrupt line - to interrupt controller,
- clock line - to clock controller,
- reset line - to GPIO controller,
- TMDS lines - to HDMI video source,
- MHL lines - to MHL connector,
- Hot-Plug line - to GPIO controller,
- I2C slave (DDC) - to I2C master,
- SPI slave control interface - to SPI master,
- HSIC - for tunneling USB traffic in MHL stream,

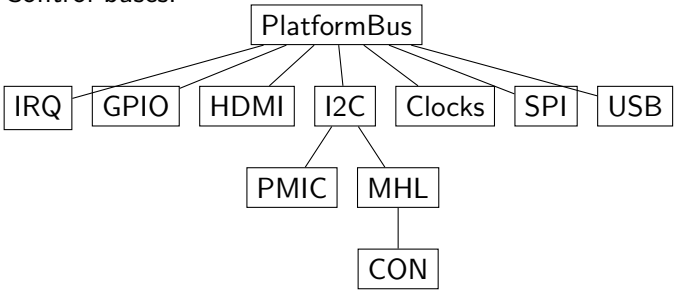
Hardware: Nasty Example

MHL 3.0 Transmitter (Sil8620) pins:

- I2C slave control interface - to I2C master,
- 10 power lines - to different power supplies (at least 2),
- interrupt line - to interrupt controller,
- clock line - to clock controller,
- reset line - to GPIO controller,
- TMDS lines - to HDMI video source,
- MHL lines - to MHL connector,
- Hot-Plug line - to GPIO controller,
- I2C slave (DDC) - to I2C master,
- SPI slave control interface - to SPI master,
- HSIC - for tunneling USB traffic in MHL stream,
- 9 GPIOs - for different purposes.

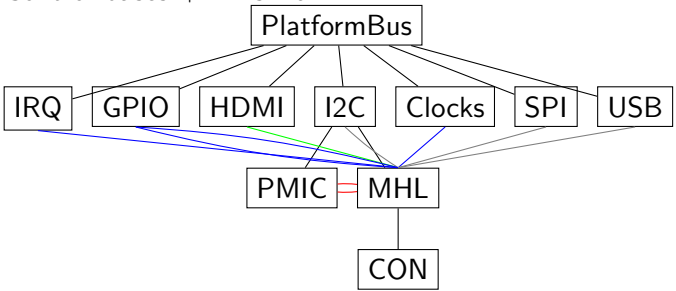
Hardware: Nice Picture

Control buses:



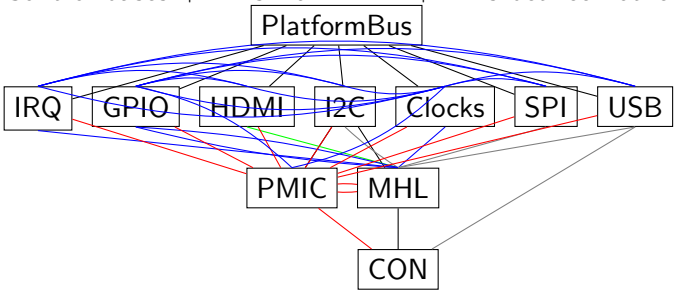
Hardware: Scary Picture

Control buses + links from MHL:



Hardware: Even Scarier Picture, but not the Scariest One

Control buses + links from MHL + links between other devices:



Problem Description - Software

Kernel: Driver Model

- The Kernel uses following entities:
  - device - a physical device that is attached to a bus,
  - driver - a software entity that can be associated with a device and performs operations with it,
  - bus - a device to which other devices can be attached.
- One device can be attached to only one bus (control interface).
- Devices without bus are attached to platform pseudo-bus.
- Devices forms tree based on control bus - bus master is a parent of devices on the bus.

It looks like Linux Kernel Driver Model is oriented towards handling one-interface devices.

Kernel: Driver Model - Matching Devices vs Drivers

- One driver can be bound to many devices, but device can be bound to only one driver at a time.
- Every time device is registered it is matched against registered drivers.
- Every time driver is registered it is matched against registered and unbound devices.
- If the match succeeds device is probed with the driver, if the probe succeeds the driver is bound to the device.
- In case driver is unregistered it is unbound from all bound devices.
- In case device is unregistered it is unbound from its driver.

Kernel: Driver Model - Life Time of Device and Driver

- Devices can be registered and unregistered at any time:
  - Kernel initialization (platform code, DeviceTree),
  - bus registration/unregistration,
  - physical device attach/detach,
  - ...
- Drivers can be registered and unregistered at any time:
  - Kernel initialization,
  - module insert/removal,
  - ...
- Devices can be also bound/unbound anytime on user demand.

Since unbound device is almost non-functional, for the rest of the presentation term device will used to describe bound device.

Kernel: Resources

- Some devices (consumers) use objects (resources) provided by other devices (providers).
- Sample resources present in Kernel:
  - natural candidates: clocks, regulators, gpios, interrupts, ...
  - framework convention: panels, bridges, camera sensors, ...
- Some resources are required - device cannot be bound without them.
- Some are optional - if available device can work better.

Kernel: Problem Description

So where is the problem?

- Probing order is undetermined: consumer can be probed before required resources appears.
- Resource can disappear (provider is unbound).
- ~~Consumer can disappear (consumer is unbound)~~.
- How to get optional resource? What if provider is unbound?
- What about semi-circular dependencies? (see next slide).
- Resource requirements are evaluated during probe.

Kernel: Problem Description - Semi-circular Dependency

What it is?

- Device A provides resource a, but to full work it requires resource b.
- Device B requires a then it can provide resource b.

If we assume resource gathering can be done only in probe it will never work.
But it can work this way:

- Device A registers resource a - probe time.
- Device B grabs a, then registers b - probe time.
- Device A grabs b - later (when?).

So lets treat b as optional resource for A, the problem becomes "optional resource" issue.
But does it happens in real world? Yes, in media, display stacks: core (ISP, DC) provides clock, required by peripherals (camera sensors, encoders), but it also requires resources provided by peripherals.

Current Solutions

Current Solutions: Changing Bind Order

We can use initcalls, modify Makefiles, DT files to change subsystem/driver/device registration order. Why is it BAD practice?

- Different platforms have different tree of dependencies, but the same devices.
- Different Kernel configurations can change bind order.
- It does not work with modules.
- It depends on too many factors to be maintainable in sane way.

Current Solutions: Disable Device/Driver Unplugging

Prevent modules from being unloaded, by playing with module ref-counting:

- Avoidance: Quite ugly and counter idea of modules.
- It protects only module from being unloaded, but does not protect the driver from unbinding.

Disallow users from binding/unbinding devices from the driver:

- Again not so pretty, again avoidance.
- Could make sense only in case of some core devices.
- It is just removing some sysfs entries, nothing more. There could be other ways to provoke device unbind.

Current Solutions: Deferred Probe - Description

How does it work?

- During device probe driver gathers required resources, if some are unavailable it exits with -EPROBE_ERROR.
- Driver core puts such device on special list for re-probing.
- Successful probe of any device triggers re-probing all deferred devices, with hope that successful probe caused appearance of some resources.
- To avoid too many re-probes trigger starts working in late_initcall - after registering all built-in device drivers.

Current Solutions: Deferred Probe - Pros and Cons

Advantages:

- Very simple, non-invasive mechanism.
- No big changes in drivers and the core.

Disadvantages:

- Suboptimal - some devices can be re-probed multiple times.
- Can significantly delay start of important subsystem.
- Does not handle resource disappearance.
- Does not handle optional resources.
- It assumes resource can be registered only in probe call - easy to fix if necessary.
- We really do not know if the resource will be available - client can defer forever.

Current Solutions: Device Links - Description

How does it work?

- One(who?) can create link from consumer to provider devices.
- Driver core guarantees then that consumer will not be probed before provider or provider will not be unbound before consumer.
- Suspend/resume order is also guaranteed, and runtime suspend/resume order can be, depending on the flags.
- To avoid circular dependencies link is verified before adding.

Current Solutions: Device Links - Pros and Cons

Advantages:

- Idea quite straightforward.
- Solves provider unbind problem.
- Integrated with PM and RPM.

Disadvantages:

- Since dependency is usually discovered in consumer probe it does not solve probe order in most cases.
- Does not solve optional resource issue.
- Requires mapping of the resource and provider to devices, this is not always the case (for example drm_dev has no associated device).
- Limitations of the link creation time causes subtle issues (solvable?).

Current Solutions: Device Links - Subtle Issue

Resource's lifetime is little different than lifetime of the provider device - resource usually is registered before device is probed and is unregistered before device is unbound.

Corner case:

Probe of Consumer	Probe of Provider
	register_resource(a);
r= get_resource(a);	
device_link_add(dev, r→dev);	// provider's probe not finished
return 0;	return 0;

In this example device link is created during probe of both: the provider and the consumer, but it means the provider is not yet probed - inconsistent state, does not work with the framework. On the other side when it should be created? Shall we defer till provider is fully probed?

Current Solutions: Components - Description

Features:

- It is specialized framework to synchronize multiple devices.
- Master device in probe calls component_master_add* with list of components to wait for and with bind/unbind callbacks.
- Component devices in probe calls component_add with provided bind/unbind callbacks - no specific order required.
- All bind callbacks are called when all components and master are added - synchronization point.
- Two stage device initialization: probe and bind.
- All-or-nothing strategy: bind does not occur until all components are ready, and if at least one component or master is to be removed unbind occurs.
- Here the master is a consumer, and the components are providers.

Current Solutions: Components - Pros and Cons

Advantages:

- It is optimal (no re-probing due to missing components).
- It handles correctly component/master unbind/re-bind.

Disadvantages:

- Specialized framework - solves only subset of the problems.
- No support for optional components.

Current Solutions: Missing Parts - Summary

- Optimal bind/probe order.
- Optional resources gathering.
- Unbind of providers of optional resources.

Proposition

Proposition: Resource Tracking

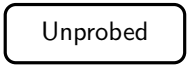
What if the consumer can track interesting resources appearance/disappearance?

- Instead of probe deferring consumer will register callbacks which will be called when requested resources appeared or will disappear.
- Lack of resource will not result in probe error, device will finish probe successfully, but will wait with further initialization till required resources are present.
- The same mechanism can be used for required and optional resources.

Proposition: Resource Tracking

What if the consumer can track interesting resources appearance/disappearance?

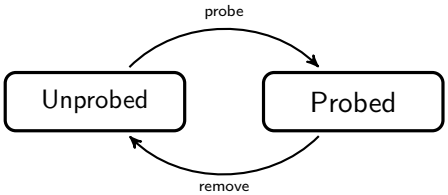
- Instead of probe deferring consumer will register callbacks which will be called when requested resources appeared or will disappear.
- Lack of resource will not result in probe error, device will finish probe successfully, but will wait with further initialization till required resources are present.
- The same mechanism can be used for required and optional resources.



Proposition: Resource Tracking

What if the consumer can track interesting resources appearance/disappearance?

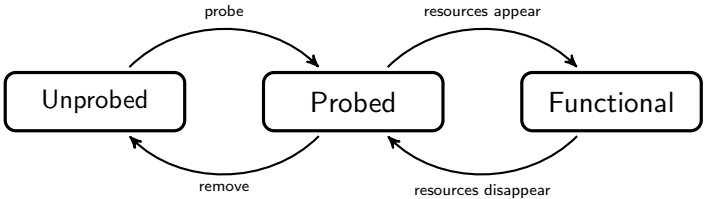
- Instead of probe deferring consumer will register callbacks which will be called when requested resources appeared or will disappear.
- Lack of resource will not result in probe error, device will finish probe successfully, but will wait with further initialization till required resources are present.
- The same mechanism can be used for required and optional resources.



Proposition: Resource Tracking

What if the consumer can track interesting resources appearance/disappearance?

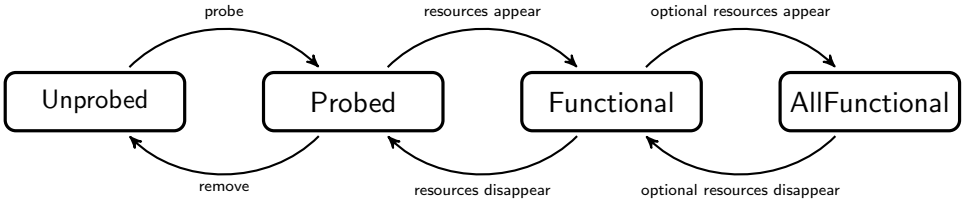
- Instead of probe deferring consumer will register callbacks which will be called when requested resources appeared or will disappear.
- Lack of resource will not result in probe error, device will finish probe successfully, but will wait with further initialization till required resources are present.
- The same mechanism can be used for required and optional resources.



Proposition: Resource Tracking

What if the consumer can track interesting resources appearance/disappearance?

- Instead of probe deferring consumer will register callbacks which will be called when requested resources appeared or will disappear.
- Lack of resource will not result in probe error, device will finish probe successfully, but will wait with further initialization till required resources are present.
- The same mechanism can be used for required and optional resources.



Proposition: Resource Tracking - Pseudo-code

How the code could look like?

Proposition: Resource Tracking - Pseudo-code

How the code could look like?

```
int dev_probe(...)
{
    // probe initialization




}

int dev_remove(...)
{




    // remove cleanup
}
```

Proposition: Resource Tracking - Pseudo-code

How the code could look like?

```
int dev_probe(...)
{
    // probe initialization

    restrack_register(req_res,
  ...required resources...
    );

}

int dev_remove(...)
{

    restrack_unregister(req_res);


    // remove cleanup
}
```

```
int req_res_on()
{
    // final initialization

}

int req_res_off()
{


    // cleanup
}
```

Proposition: Resource Tracking - Pseudo-code

How the code could look like?

```
int dev_probe(...)
{
    // probe initialization

    restrack_register(req_res,
  ...required resources...
    );

}

int dev_remove(...)
{

    restrack_unregister(req_res);


    // remove cleanup
}
```

```
int req_res_on()
{
    // final initialization

    restrack_register(opt_res,
  ...optional resources...
    );

}

int req_res_off()
{

    restrack_unregister(opt_res);


    // cleanup
}
```

```
int opt_res_on()
{
    // start using optional resources
}
```

```
int opt_res_off()
{
    // stop using optional resources
}
```

Proposition: Resource Tracking - Consumer's PoV (1)

How does it work from consumer's point of view?

- *_res_on callback is called:
  - immediately from restrack_register, if all tracked resources are available during registration,
  - otherwise immediately after all tracked resources becomes available.
- *_res_off callback is called:
  - immediately before one of tracked resources is to be removed,
  - otherwise immediately from restrack_unregister - if all tracked resources are available during unregistration.

Proposition: Resource Tracking - Consumer's PoV (2)

What happens when:

- all resources are available (AllFunctional) and optional resource is about to remove:
  - callback opt_res_off is called,
  - driver stops using optional resources.
- all resources are available (AllFunctional) and required resource is about to remove:
  - callback req_res_off is called,
  - it calls restrack_unregister(opt_res),
  - callback opt_res_off is called,
  - driver stops using optional resources,
  - callback req_res_off is continued,
  - driver stops using required resources.

Proposition: Resource Tracking - Consumer's PoV (3)

What happens when:

- all resources are available (AllFunctional) and device is about to unbind:
  - callback dev_remove is called,
  - it calls restrack_unregister(req_res),
  - callback req_res_off is called,
  - it calls restrack_unregister(opt_res),
  - callback opt_res_off is called,
  - driver stops using optional resources,
  - callback req_res_off is continued,
  - driver stops using required resources,
  - callback dev_remove is continued,
  - driver is unbound.

Proposition: Resource Tracking - Provider's PoV

How does it works from provider's point of view?

- after resource becomes available restrack_on(resource_id) is called - all trackers are notified.
- before resource removal restrack_off(resource_id) is called - all trackers are notified.

What is resource_id? Something to identify resource even if it is not available, for example:

- DT node of the provider plus optional additional id: `<&ldo3_reg>, <&cmu_disp CLK_PCLK_DSIM0>`.
- Pair of strings: providers_device_name and resource_name: `("pmic", "ldo3")`.

Proposition: Resource Tracking - Framework Requirements

Framework requirements:

- The framework must be re-entrant: many drivers are consumers and providers at the same time - they will call restrack_(on|off) in their restrack callbacks.
- The framework must be thread safe: different drivers can register trackers or create resources asynchronously.
- The only assumption is that particular tracker can be first registered and then unregistered, also particular resource can be first created, then destroyed. The sequence can be repeated multiple times.

Proposition: Resource Tracking - Advantages

Advantages:

- One framework solves all three missing issues.
- Can smoothly replace and/or co-exist with current solutions.
- Looks like natural extension of probe/remove mechanism.
- Limits the number of device states and transitions to necessary minimum.

Proposition: Resource Tracking - Implementation Issues

How to do it?

- Re-entrancy and thread safety poses implementation challenges.
- Different subsystems have different ways of identifying resources. resource_id can be non-trivial to implement, especially in case of pre-DT fuzzy lookup mechanisms.

Proposition: Resource Tracking - Implementation Details (1)

How re-entrancy and thread safety issues can be solved:

- There are four operations: resource-on, resource-off, track-register, track-unregister.
- Lets implement every operation as a task (callback + context on which it should be called).
- Serialize tasks by putting them on the queue.
- There will be one queue per resource_id, the queue will have also owner field.

Proposition: Resource Tracking - Implementation Details (2)

How tasks are queued/executed? There are three cases:

- Queue owner is not set - in this case the process becomes queue owner and executes this and all tasks which will be queued meantime to this queue until the queue is empty, then queue owner is zeroed, then function ends.
- Queue owner is set and the owner is the same as current process (re-entrancy) - again all tasks are processed, but the owner is not zeroed.
- Queue owner is set and the owner is different from the current process (other process is working on the queue) - the task is just added to the queue, if the process wants to wait for the task to be executed it should wait for end of queue processing.

Waiting is necessary for off/unregister tasks - callers should be sure that after call it can remove resource/tracker.

Proposition: Resource Tracking - Implementation Details (3)

Real world analogy:

- There are multiple workers which should perform different tasks on the machine.
- If some worker receives task, he goes to the machine.
- If the machine is available he takes it and performs this task, otherwise he gives the task to the worker holding the machine.
- If during his work it appears the task requires other subtasks to be done he performs them as well.
- If during his work another worker brings another task, he performs it as well.
- After finishing all tasks he leaves the machine.
- If another worker wants to be sure his task was done he should wait till machine is available.

Proposition: Resource Tracking - Implementation Details (4)

Can it deadlock?

- This mechanism requires lock only for adding/removing tasks to/from the queue - non-blocking operations.
- Waiting is done only in the 3rd case, but it does not block queue processing.
- Above reasoning suggests the algorithm itself is non-blocking - the only blockers can be due to bugs in drivers using this framework.
- Unfortunately there are still corner cases it can block - should be solvable.

Proposition: Resource Tracking - Implementation Details (4)

Can it deadlock?

- This mechanism requires lock only for adding/removing tasks to/from the queue - non-blocking operations.
- Waiting is done only in the 3rd case, but it does not block queue processing.
- Above reasoning suggests the algorithm itself is non-blocking - the only blockers can be due to bugs in drivers using this framework.
- Unfortunately there are still corner cases it can block - should be solvable.
- No, it is waste of time, let's try different approach.

Proposition: Resource Tracking - New Approach

Let's avoid wheel reinventing, try to use existing Kernel framework - asynchronous function calls.

- Global list of resource state structures - one state per resource.
- Every state contains: current state, linked list of resource trackers, mutex, async cookie.
- State is created on first user request: res-register, res-on.
- Modification of state is performed under the mutex.
- Notification callbacks are scheduled using async_schedule.
- If we need synchronization barrier we use async_synchronize_cookie: before/after resource-off, restrack-unregister.
- This is my fresh idea (3 days old) - untested, even not brainstormed, so please do not be cruel if if is flawed.

Proposition: Resource Tracking - The Code

OK this is theory, but where is the real code?

Proposition: Resource Tracking - The Code

OK this is theory, but where is the real code?

- The code already exists - RFC patchset was posted.
- It uses different synchronization mechanism (harder to read, different queue granularity, different waiting implementation, can deadlock).
- Beside it needs some polishing/testing/review, maybe some rework.
- Lacks support for non-DT lookup mechanisms - I have some ideas how to deal with it but I would prefer to add them later - it would require cleanup of fuzzy resource lookup mechanism per every framework - quite annoying ungrateful task.
- Hopefully next version will be posted in near future.

Proposition: Resource Tracking - Code Example

Old code

```
int dev_probe(...)
{
  ...
  ctx->pclk = devm_clk_get(dev, "pclk");
  if (IS_ERR(ctx->pclk)) {
    ret = PTR_ERR(ctx->pclk);
    if (ret != -EPROBE_DEFER)
      dev_err(dev, "cannot␣get␣pclk\n");
    return ret;
  }
  ctx->vdd = devm_regulator_get(dev, "vdd");
  if (IS_ERR(ctx->vdd)) {
    ret = PTR_ERR(ctx->vdd);
    if (ret != -EPROBE_DEFER)
      dev_err(dev, "cannot␣get␣vdd\n");
    return ret;
  }
  ...
}
```

New code

```
int dev_probe(...)
{
  rtrack = devm_restrack_register(dev, lcd_callback,
    regulator_bulk_restrack_desc(&ctx->supplies[0]),
    regulator_bulk_restrack_desc(&ctx->supplies[1]),
    clk_restrack_desc(&ctx->pll_clk, "pll_clk"),
    clk_restrack_desc(&ctx->bus_clk, "bus_clk"),
    phy_restrack_desc(&ctx->phy, "dsim"),
  );

  return PTR_ERR_OR_NULL(rtrack);
}
```

Proposition: Resource Tracking - TODO

- Ressurect old patches.
- Rework/redesign/analyze synchronization code.
- Consider non-DT lookup mechanism.
- Polish the code.

Summary

Discussion

Q & A

References

- Probe deferral mechanism: https://lwn.net/Articles/450178/
- Device links framework: https://www.kernel.org/doc/html/latest/driver-api/device_link.html
- Component framework (kernel-doc patch, not merged): https://lore.kernel.org/patchwork/patch/780548/
- Resource tracking framework: https://lwn.net/Articles/625454/