Google

Android drivers in Rust

Wedson Almeida Filho, 2021/09/22

Declaring a driver

```
static struct amba_driver pl061_gpio_driver =
{
     .drv = {
          .name     = "pl061_gpio",
#ifdef CONFIG_PM
          .pm  = &pl061_dev_pm_ops,
#endif
     },
     .id_table = pl061_ids,
     .probe    = pl061_probe,
};
module_amba_driver(pl061_gpio_driver);

MODULE_LICENSE("GPL v2");
```

Macro used to register AMBA driver

```
module_amba_driver! {
    type: PL061Device,
    name: b"pl061_gpio",
    author: b"Wedson Almeida Filho",
    license: b"GPL v2",
}
```

`PL061Device` holds details about device driver

Driver callbacks and IDs

```
static const struct amba_id pl061_ids[] = {
     {
          .id       = 0x00041061,
          .mask     = 0x000fffff,
     },
     { 0, 0 },
};

static int pl061_probe(
     struct amba_device *adev,
     const struct amba_id *id)
{
  [...]
}
```

Trait that needs to implemented by AMBA drivers

```
impl amba::Driver for PL061Device {
    type InnerData = DeviceData;
    type PowerOps = Self;

    declare_id_table! {
        (0x00041061, 0x000fffff),
    }

    fn probe(
        dev: &mut amba::Device,
        id: &(u32, u32, Option<()>),
    ) -> Result<Ref<DeviceData>> {
        [...]
    }
}
```

`DeviceData` holds details about device instance

Specifies the type (`Self`) containing the power management operations (more on this later)

Typed device ID info

```
impl amba::Driver for PL061Device {
    type IdInfo = u64;

    declare_id_table! {
        (0x00041062, 0x000fffff),
        (0x00041061, 0x000fffff, 30),
    }

    fn probe(
        dev: &mut amba::Device,
        id: &(u32, u32, Option<u64>),
    ) -> Result<Ref<DeviceData>> {
        [...]
    }
}
```

Developer decides which type to use

No id info here

When present, id info must have type `Self::IdInfo` (`u64` in this case)

No need to zero-terminate the list

Id info: optional `Self::IdInfo`

Power management

```
#ifdef CONFIG_PM
static int pl061_suspend(struct device *dev)
{
      struct pl061 *pl061 = dev_get_drvdata(dev);
      [...]
}
static int pl061_resume(struct device *dev)
{
      struct pl061 *pl061 = dev_get_drvdata(dev);
      [...]
}
static const struct dev_pm_ops pl061_dev_pm_ops = {
      .suspend = pl061_suspend,
      .resume = pl061_resume,
      .freeze = pl061_suspend,
      .restore = pl061_resume,
};
#endif
```

No need for conditional compilation in driver code

Trait that needs to implemented by drivers that support power management

```
impl power::Operations for PL061Device {
    type Data = Ref<DeviceData>;

    fn suspend(
        data: &Ref<DeviceData>) -> Result {
        [...]
    }

    fn resume(
        data: &Ref<DeviceData>) -> Result {
        [...]
    }

    fn freeze(
        data: &Ref<DeviceData>) -> Result {
        [...]
    }

    fn restore(
        data: &Ref<DeviceData>) -> Result {
        [...]
    }
}
```

Device state data is typed

IRQ Registration

```
static int queue_request_irq(
     struct nvme_queue *nvmeq)
{
     struct pci_dev *pdev =
          to_pci_dev(nvmeq->dev->dev);
     int nr = nvmeq->dev->ctrl.instance;
     return pci_request_irq(pdev,
          nvmeq->cq_vector, nvme_irq,
          NULL, nvmeq, "nvme%dq%d", nr,
          nvmeq->qid);
}
```

```
fn register_irq(
    self: &Ref<Self>,
    pci_dev: &pci::Device) -> Result {
    let irq = pci_dev.request_irq(
        self.cq_vector.into(),
        self.clone(),
        format_args!(
            "nvme{}q{}",
            self.data.instance,
            self.qid,
        ),
    )?;
    self.inner.lock().irq.replace(irq);
    Ok(())
}
```

IRQ context is typed

Typed string formatting

IRQ handler registration and context automatically dropped when `Self` (`NvmeQueue`) is.

IRQ Handling

Trait that needs to implemented by drivers that handle interrupts

```
static
irqreturn_t nvme_irq(int irq, void *data)
{
        struct nvme_queue *nvmeq = data;

        if (nvme_process_cq(nvmeq))
                return IRQ_HANDLED;
        return IRQ_NONE;
}
```

```
impl irq::Handler for NvmeDevice {
    type Data = Ref<NvmeQueue>;

    fn handle_irq(
        queue: &Ref<NvmeQueue>
    ) -> irq::Return {
        if queue.process_cq() {
            irq::Return::Handled
        } else {
            irq::Return::None
        }
    }
}
```

Context is typed and guaranteed to exist while handler is registered

Enum values are in namespace and not implicitly coerced

Locking

Size of write is determined at compile-time by type (`NvmeCommand`)

```
static void nvme_submit_cmd(
     struct nvme_queue *nvmeq,
     struct nvme_command *cmd,
     bool write_sq)
{
     spin_lock(&nvmeq->sq_lock);
     memcpy(nvmeq->sq_cmds + (nvmeq->sq_tail
<< nvmeq->sqes), cmd, sizeof(*cmd));
     if (++nvmeq->sq_tail == nvmeq->q_depth)
          vmeq->sq_tail = 0;
     nvme_write_sq_db(nvmeq, write_sq);
     spin_unlock(&nvmeq->sq_lock);
}
```

```
fn submit(&self,
    cmd: &NvmeCommand,
    write_sq: bool,
) {
    let mut inner = self.inner.lock();
    self.sq.write(
        inner.sq_tail.into(), *cmd);
    inner.sq_tail += 1;
    if inner.sq_tail == self.q_depth {
        inner.sq_tail = 0;
    }
    self.write_sq_db(
        write_sq, &mut inner);
}
```

Spin lock acquired:

- Automatically released when guard goes out of scope
- Protected data syntactically inaccessible before locking

`write_sq_db` requires the queue inner state with the locked acquire

Memory-mapped IO

```c
static
int pl061_get_direction(struct gpio_chip *gc,
      unsigned offset)
{
   struct pl061 *pl061 = gpiochip_get_data(gc);

   if (readb(pl061->base + GPIODIR) & BIT(offset))
         return GPIO_LINE_DIRECTION_OUT;

      return GPIO_LINE_DIRECTION_IN;
}
```

Argument is typed, as always

No mixing of error codes and return type

```rust
fn get_direction(
    data: &Ref<DeviceData>,
    offset: u32,
) -> Result<gpio::LineDirection> {
 let pl061 = data.resources().ok_or(Error::ENXIO)?;

 Ok(if pl061.base.readb(GPIODIR) & bit(offset) != 0 {
    gpio::LineDirection::Out
 } else {
    gpio::LineDirection::In
 })
}
```

Resources become unavailable when device is removed

- No arithmetic in driver code
- `pl061.base` minimum size known at compile time
- Offset checked at compile time
- `try_` variants when offsets are not known at compile time

Why should one consider Rust?

- The primary reason for Rust is security:
  - Memory safety in safe fragment
- However, Rust also increases development velocity, for example:
  - All context data is strongly typed
  - Resources are automatically released
  - Detects and rejects data races
  - Doesn't mix error code and return values
  - Data protected by synchronisation primitives (mutexes, spin locks) are only available when locked
  - Functions can specify that they need locked data structures
  - No dangling pointers
  - No use-after-free
  - etc.

Discussion

- Questions
- Concerns/objections
- Unforeseen difficulties
- General feedback
- Pain points when writing drivers in C
- Possible paths to upstreaming