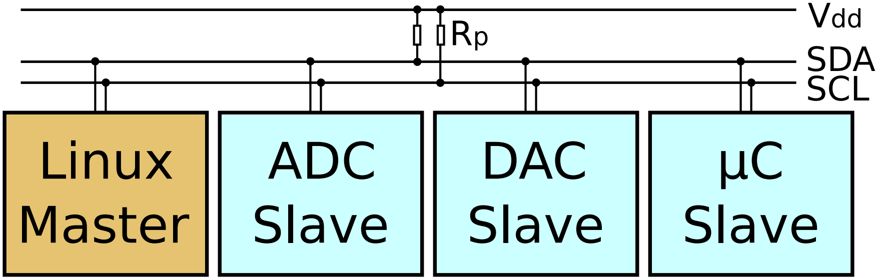
The Shiny New I2C Slave Framework

Wolfram Sang

Consultant/Renesas Kernel Team

6.10.2015, ELCE15

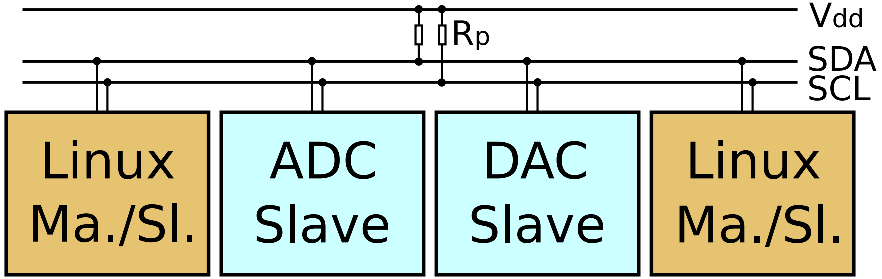
A typical I2C bus



[1]

[1]picture based on *this one* by Colin M.L. Burnett

The new era :)



[2]

[2]picture based on *this one* by Colin M.L. Burnett

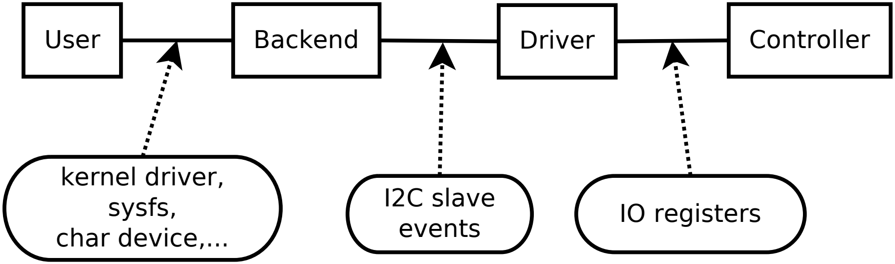
Finally...

~~At this time, Linux only operates I2C (or SMBus) in master mode; you can't use these APIs to make a Linux system behave as a slave/device, either to speak a custom protocol or to emulate some other device.~~

Use cases

- data delivery (sensor like)
- configuration (codec like)
- embedded controllers (e.g. nvec)
- avoid multi-master

Data flow


User
Backend
Driver
Controller
kernel driver, sysfs, char device,...
I2C slave events
IO registers

Driver side

We need support from the I2C bus driver

- activating slave support in the core is not enough
- usually extension to the I2C master driver

  `no slave only solutions please`
- watch your PM settings

  `a slave always needs to listen`

Registering a slave

```
static int rcar_reg_slave(struct i2c_client *slave)
{
    struct rcar_i2c_priv *priv = i2c_get_adapdata(slave->adapter);

    if (priv->slave)
        return -EBUSY;

    if (slave->flags & I2C_CLIENT_TEN)
        return -EAFNOSUPPORT;

    pm_runtime_forbid(rcar_i2c_priv_to_dev(priv));

    priv->slave = slave;
    rcar_i2c_write(priv, ICSAR, slave->addr);
    rcar_i2c_write(priv, ICSSR, 0);
    rcar_i2c_write(priv, ICSIER, SAR | SSR);
    rcar_i2c_write(priv, ICSCR, SIE | SDBS);

    return 0;
}
```

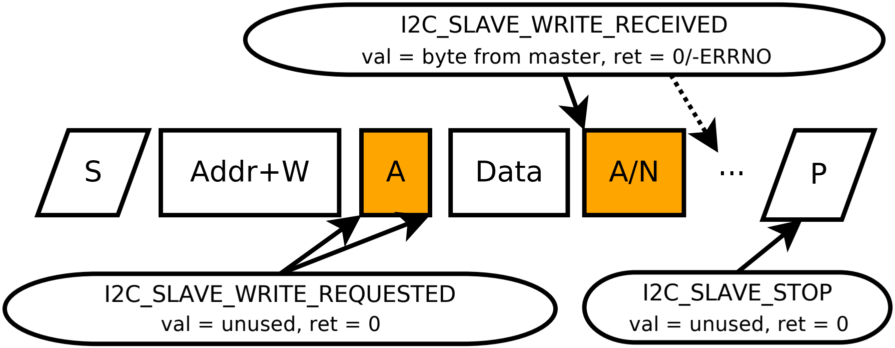
Slave events

Handler function

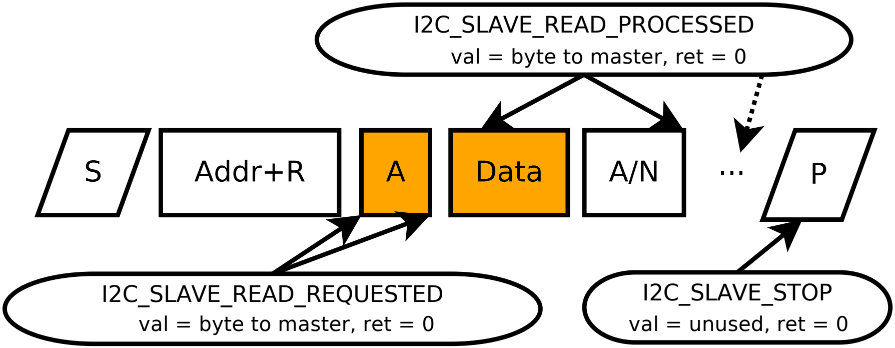
```
ret = i2c_slave_event(slave, event, &val)
```

- links driver and backend
- `val` carries the data. It is bidirectional.
- usually called from *interrupt context*!

Slave events (master writes)



Slave events (master reads)



Slave interrupt handler I

```
static bool rcar_i2c_slave_irq(struct rcar_i2c_priv *priv)
{
    u32 ssr_raw, ssr_filtered;
    u8 value;

    ssr_raw = rcar_i2c_read(priv, ICSSR) & 0xff;
    ssr_filtered = ssr_raw & rcar_i2c_read(priv, ICSIER);

    if (!ssr_filtered)
        return false;
```

Slave interrupt handler II

```c
/* address detected */
if (ssr_filtered & SAR) {
    /* read or write request */
    if (ssr_raw & STM) {
        i2c_slave_event(priv->slave, I2C_SLAVE_READ_REQUESTED, &value);
        rcar_i2c_write(priv, ICRXTX, value);
        rcar_i2c_write(priv, ICSIER, SDE | SSR | SAR);
    } else {
        i2c_slave_event(priv->slave, I2C_SLAVE_WRITE_REQUESTED, &value);
        rcar_i2c_read(priv, ICRXTX);    /* dummy read */
        rcar_i2c_write(priv, ICSIER, SDR | SSR | SAR);
    }

    rcar_i2c_write(priv, ICSSR, ~SAR & 0xff);
}
```

Slave interrupt handler III

```
/* master sent stop */
if (ssr_filtered & SSR) {
    i2c_slave_event(priv->slave, I2C_SLAVE_STOP, &value);
    rcar_i2c_write(priv, ICSIER, SAR | SSR);
    rcar_i2c_write(priv, ICSSR, ~SSR & 0xff);
}
```

Slave interrupt handler IV

```
/* master wants to write to us */
if (ssr_filtered & SDR) {
    int ret;

    value = rcar_i2c_read(priv, ICRXTX);
    ret = i2c_slave_event(priv->slave, I2C_SLAVE_WRITE_RECEIVED, &value);
    /* Send NACK in case of error */
    rcar_i2c_write(priv, ICSCR, SIE | SDBS | (ret < 0 ? FNA : 0));
    rcar_i2c_write(priv, ICSSR, ~SDR & 0xff);
}
```

Slave interrupt handler V

```
    /* master wants to read from us */
    if (ssr_filtered & SDE) {
        i2c_slave_event(priv->slave, I2C_SLAVE_READ_PROCESSED, &value);
        rcar_i2c_write(priv, ICRXTX, value);
        rcar_i2c_write(priv, ICSSR, ~SDE & 0xff);
    }

    return true;
}
```

Backends

- are standard i2c drivers
- are normally matched to i2c clients
- are HW independent
- provide a callback to handle slave events

Backend driver I

```c
static int i2c_slave_8bit_seconds_slave_cb(struct i2c_client *client,
                     enum i2c_slave_event event, u8 *val)
{
    switch (event) {
    case I2C_SLAVE_READ_REQUESTED:
    case I2C_SLAVE_READ_PROCESSED:
        /* Always get the most recent value */
        *val = get_seconds() & 0xff;
        break;

    case I2C_SLAVE_WRITE_REQUESTED:
    case I2C_SLAVE_WRITE_RECEIVED:
    case I2C_SLAVE_STOP:
    default:
        break;
    }

    return 0;
}
```

Backend driver II

```
static int i2c_slave_8bit_seconds_probe(struct i2c_client *client,
                    const struct i2c_device_id *id)
{
    return i2c_slave_register(client, i2c_slave_8bit_seconds_slave_cb);
};

static int i2c_slave_8bit_seconds_remove(struct i2c_client *client)
{
    i2c_slave_unregister(client);
    return 0;
}
```

The "read pointer" problem

```
...
    case I2C_SLAVE_READ_PROCESSED:
        /* The previous byte made it to the bus, get next one */
        eeprom->buffer_idx++;
        /* fallthrough */
    case I2C_SLAVE_READ_REQUESTED:
        spin_lock(&eeprom->buffer_lock);
        *val = eeprom->buffer[eeprom->buffer_idx];
        spin_unlock(&eeprom->buffer_lock);
        /*
         * Do not increment buffer_idx here, because we don't know if
         * this byte will be actually used. Read Linux I2C slave docs
         * for details.
         */
        break;
...
```

Current status

Drivers:	RCar, (Tegra), ((Davinci))
Backend:	EEPROM/memory simulator
Devicetree:	bindings clear \o/

Address spaces

Avoid address collisions. Enable loopbacks.

Devicetree

- I2C_TEN_BIT_ADDRESS

  ```
  address space: 0xa000 - 0xa3ff
  ```

- I2C_OWN_SLAVE_ADDRESS

  ```
  address space += 0x1000
  ```

- Example:

  ```
  reg = <(I2C_OWN_SLAVE_ADDRESS | 0x42)>;
  ```

Run time instantiation

```
echo slave-24c02 0x1064
    > /sys/bus/i2c/devices/i2c-1/new_device
```

Next steps

- more driver support
- more backends (if there is a user)
- no new features (unless there is a use case)

Thanks! <3

- Renesas

  `for funding this upstream solution`

- Renesas Kernel Team

  `especially Geert and Laurent for thorough review`

- Uwe Kleine-König

  `for in-depth discussions`

- Andrey Danin

  `for the Tegra slave implementation and nvec backend port`

Demo!

At the showcase this evening

The End

Thank you for your attention!

Questions? Comments?

- right now
- at the showcase this evening
- or anytime at this conference
- wsa@the-dreams.de