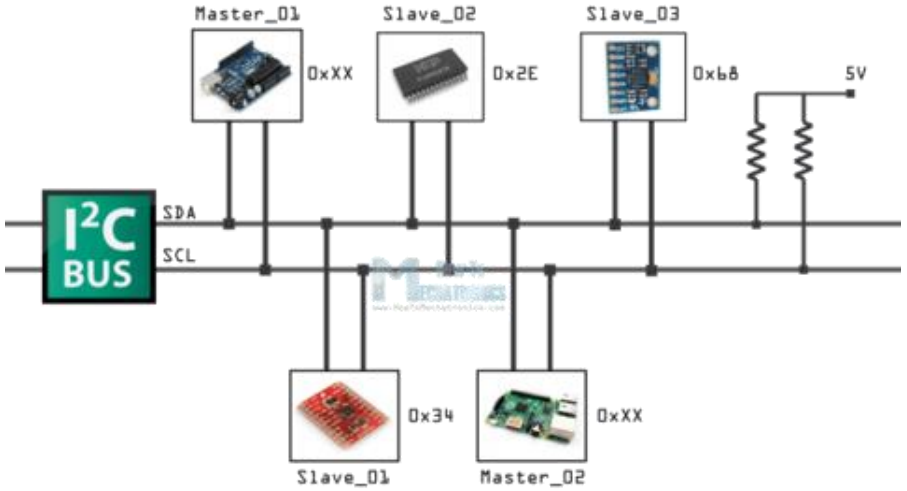
I2C Driver

Rights to copy

What is I2C?

- A very commonly used low-speed bus to connect on-board and external devices to the processor.
- Uses only two wires: SDA for the data, SCL for the clock.
- It is a master/slave bus: only the master can initiate transactions, and slaves can only reply to transactions initiated by masters.
- In a Linux system, the I2C controller embedded in the processor is typically the master, controlling the bus.
- Each slave device is identified by a unique I2C address. Each transaction initiated by the master contains this address, which allows the relevant slave to recognize that it should reply to this particular transaction.



The I2C subsystem

- Like all bus subsystems, the I2C subsystem is responsible for:
  - Providing an API to implement I2C controller drivers
  - Providing an API to implement I2C device drivers, in kernel space
  - Providing an API to implement I2C device drivers, in user space
- The core of the I2C subsystem is located in drivers/i2c/.
- The I2C controller drivers are located in drivers/i2c/busses/.
- The I2C device drivers are located throughout drivers/, depending on the type of device (ex: drivers/input/ for input devices).

Registering an I2C device driver

- Like all bus subsystems, the I2C subsystem defines a struct i2c_driver that inherits from struct device_driver, and which must be instantiated and registered by each I2C device driver.
  - As usual, this structure points to the ->probe() and ->remove() functions.
  - It also contains an id_table field that must point to a list of *device IDs* (which is a list of tuples containing a string and some private driver data). It is used for non-DT based probing of I2C devices.
- The i2c_add_driver() and i2c_del_driver() functions are used to register/unregister the driver.
- If the driver doesn't do anything else in its init()/exit() functions, it is advised to use the module_i2c_driver() macro instead.

Registering an I2C device driver: example

```
static const struct i2c_device_id <driver>_id[] = {
        { "<device-name>", 0 },
        { }
};
MODULE_DEVICE_TABLE(i2c, <driver>_id);

#ifdef CONFIG_OF
static const struct of_device_id <driver>_dt_ids[] = {
        { .compatible = "<vendor>,<device-name>", },
        { }
};
MODULE_DEVICE_TABLE(of, <driver>_dt_ids);
#endif

static struct i2c_driver <driver>_driver = {
        .probe          = <driver>_probe,
        .remove         = <driver>_remove,
        .id_table       = <driver>_id,
        .driver = {
                .name   = "<driver-name>",
                .owner  = THIS_MODULE,
                .of_match_table = of_match_ptr(<driver>_dt_ids),
        },
};

module_i2c_driver(<driver>_driver);
```

MODULE_DEVICE_TABLE() macro allows depmod to extract at compile time the relation between device identifiers and drivers, so that drivers can be loaded automatically by udev

CONFIG_OF
Open Firmware => DT

Registering an I2C device: non-DT

- On non-DT platforms, the struct i2c_board_info structure allows to describe how an I2C device is connected to a board.
- Such structures are normally defined with the I2C_BOARD_INFO() helper macro.
  - Takes as argument the device name and the slave address of the device on the bus.
- An array of such structures is registered on a per-bus basis using i2c_register_board_info(), when the platform is initialized.

Registering an I2C device, non-DT example

```
static struct i2c_board_info <board>_i2c_devices[] __initdata = {
        {
                I2C_BOARD_INFO("cs42l51", 0x4a),
        },
};

void board_init(void)
{
        /*
         * Here should be the registration of all devices, including
         * the I2C controller device.
         */

        i2c_register_board_info(0, <board>_i2c_devices,
                                ARRAY_SIZE(<board>_i2c_devices));

        /* More devices registered here */
}
```

Registering an I2C device, in the DT

- In the Device Tree, the I2C controller device is typically defined in the .dtsi file that describes the processor.
  - Normally defined with status = "disabled".
- At the board/platform level:
  - the I2C controller device is enabled (status = "okay")
  - the I2C bus frequency is defined, using the clock-frequency property.
  - the I2C devices on the bus are described as children of the I2C controller node, where the reg property gives the I2C slave address on the bus.

Registering an I2C device, DT example

- Definition of the I2C controller, sun7i-a20.dtsi file

```
i2c0: i2c@01c2ac00 {
        compatible = "allwinner,sun7i-a20-i2c",
                     "allwinner,sun4i-a10-i2c";
        reg = <0x01c2ac00 0x400>;
        interrupts = <GIC_SPI 7 IRQ_TYPE_LEVEL_HIGH>;
        clocks = <&apb1_gates 0>;
        status = "disabled";
        #address-cells = <1>;
        #size-cells = <0>;
};
```

- Definition of the I2C device, sun7i-a20-olinuxino-micro.dts file

```
&i2c0 {
        pinctrl-names = "default";
        pinctrl-0 = <&i2c0_pins_a>;
        status = "okay";

        axp209: pmic@34 {
                compatible = "x-powers,axp209";
                reg = <0x34>;
                interrupt-parent = <&nmi_intc>;
                interrupts = <0 IRQ_TYPE_LEVEL_LOW>;

                interrupt-controller;
                #interrupt-cells = <1>;
        };
};
```

probe() and remove()

- The ->probe() function is responsible for initializing the device and registering it in the appropriate kernel framework. It receives as argument:
  - A struct i2c_client pointer, which represents the I2C device itself. This structure inherits from struct device.
  - A struct i2c_device_id pointer, which points to the I2C device ID entry that matched the device that is being probed.
- The ->remove() function is responsible for unregistering the device from the kernel framework and shut it down. It receives as argument:
  - The same struct i2c_client pointer that was passed as argument to ->probe()

Probe/remove example

```
static int <driver>_probe(struct i2c_client *client,
                          const struct i2c_device_id *id)
{
        /* initialize device */
        /* register to a kernel framework */

        i2c_set_clientdata(client, <private data>);
        return 0;
}

static int <driver>_remove(struct i2c_client *client)
{
        <private data> = i2c_get_clientdata(client);
        /* unregister device from kernel framework */
        /* shut down the device */
        return 0;
}
```

Communicating with the I2C device: raw API

- The most **basic API** to communicate with the I2C device provides functions to either send or receive data:
  - int i2c_master_send(struct i2c_client *client, const char *buf, int count);
    - Sends the contents of buf to the client.
  - int i2c_master_recv(struct i2c_client *client, char *buf, int count);
    - Receives count bytes from the client, and store them into buf.

Communicating with I2C device: message transfer

- The message transfer API allows to describe **transfers** that consists of several **messages**, with each message being a transaction in one direction:
  - int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msg, int num);
  - The struct i2c_adapter pointer can be found by using client->adapter
  - The struct i2c_msg structure defines the length, location, and direction of the message.

I2C: message transfer example

```
struct i2c_msg msg[2];
int error;
u8 start_reg;
u8 buf[10];

msg[0].addr = client->addr;
msg[0].flags = 0;
msg[0].len = 1;
msg[0].buf = &start_reg;
start_reg = 0x10;

msg[1].addr = client->addr;
msg[1].flags = I2C_M_RD;
msg[1].len = sizeof(buf);
msg[1].buf = buf;

error = i2c_transfer(client->adapter, msg, 2);
```

write

read

SMBus (System Management Bus)

- SMBus is a subset of the I2C protocol.
- It defines a standard set of transactions, for example to read or write a register into a device.
- Linux provides SMBus functions that *should be used* instead of the raw API, if the I2C device supports this standard type of transactions. The driver can then be used on both SMBus and I2C adapters (can't use I2C commands on SMBus adapters).

SMBus Protocol

- Example:
  - i2c_smbus_read_byte_data(): read one byte of data from a device register.
    - S Addr Wr [A] Comm [A] S Addr Rd [A] [Data] NA P
  - i2c_smbus_write_byte_data(): write one byte of data to a device register.
    - S Addr Wr [A] Comm [A] Data [A] P
- See Documentation/i2c/smbus-protocol for details.

```
S (1 bit)       : Start bit
P (1 bit)       : Stop bit
Rd/Wr (1 bit) : Read/Write bit. Rd equals 1, Wr equals 0.
A, NA (1 bit)  : Accept and reverse accept bit.
Addr (7 bits)  : I2C 7 bit address.
Comm (8 bits): Command byte, a data byte which often selects a register on the device.
Data (8 bits)  : A plain data byte.
Count (8 bits): A data byte containing the length of a block operation.

[..]: Data sent by I2C device, as opposed to data sent by the host adapter.
```

List of SMBus functions

- Read/write one byte
  - s32 i2c_smbus_read_byte(const struct i2c_client *client);
  - s32 i2c_smbus_write_byte(const struct i2c_client *client, u8 value);
- Write a command byte, and read or write one byte
  - s32 i2c_smbus_read_byte_data(const struct i2c_client *client, u8 command);
  - s32 i2c_smbus_write_byte_data(const struct i2c_client *client, u8 command, u8 value);
- Write a command byte, and read or write one word
  - s32 i2c_smbus_read_word_data(const struct i2c_client *client, u8 command);
  - s32 i2c_smbus_write_word_data(const struct i2c_client *client, u8 command, u16 value);
- Write a command byte, and read or write a block of data (max 32 bytes)
  - s32 i2c_smbus_read_block_data(const struct i2c_client *client, u8 command, u8 *values);
  - s32 i2c_smbus_write_block_data(const struct i2c_client *client, u8 command, u8 length, const u8 *values);
- Write a command byte, and read or write a block of data (no limit)
  - s32 i2c_smbus_read_i2c_block_data(const struct i2c_client *client, u8 command, u8 length, u8 *values);
  - s32 i2c_smbus_write_i2c_block_data(const struct i2c_client *client, u8 command, u8 length, const u8 *values);

I2C functionality

- Not all I2C controllers support all functionalities.
- The I2C controller drivers therefore tell the I2C core which functionalities they support.
- An I2C device driver must check that the functionalities they need are provided by the I2C controller in use on the system.
- The i2c_check_functionality() function allows to make such a check.
- Examples of functionalities:
  - I2C_FUNC_I2C to be able to use the raw I2C functions,
  - I2C_FUNC_SMBUS_BYTE_DATA to be able to use SMBus commands to write a command and read/write one byte of data.

```
if (i2c_check_functionality(client->adapter, I2C_FUNC_SMBUS_READ_I2C_BLOCK))
                    return i2c_smbus_read_i2c_block_data(client, command, length, values);
```

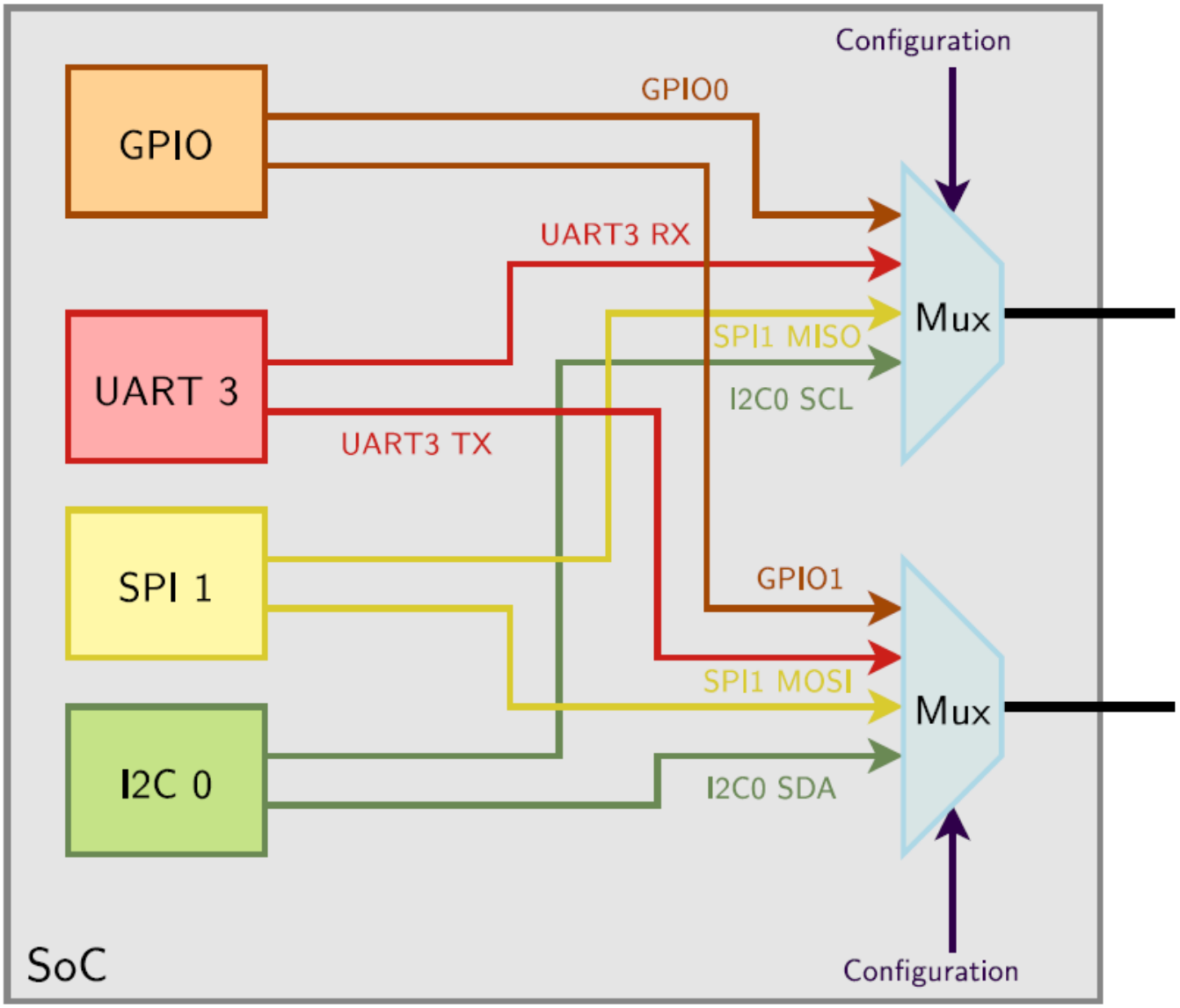
- See include/uapi/linux/i2c.h for the full list of existing functionalities.

Pin Muxing

What is pin muxing?

- Modern SoCs (System on Chip) include more and more hardware blocks, many of which need to interface with the outside world using *pins*.
- However, the physical size of the chips remains small, and therefore the number of available pins is limited.
- For this reason, not all of the internal hardware block features can be exposed on the pins simultaneously.
- The pins are **multiplexed**: they expose either the functionality of hardware block A **or** the functionality of hardware block B.
- This *multiplexing* is usually software configurable.

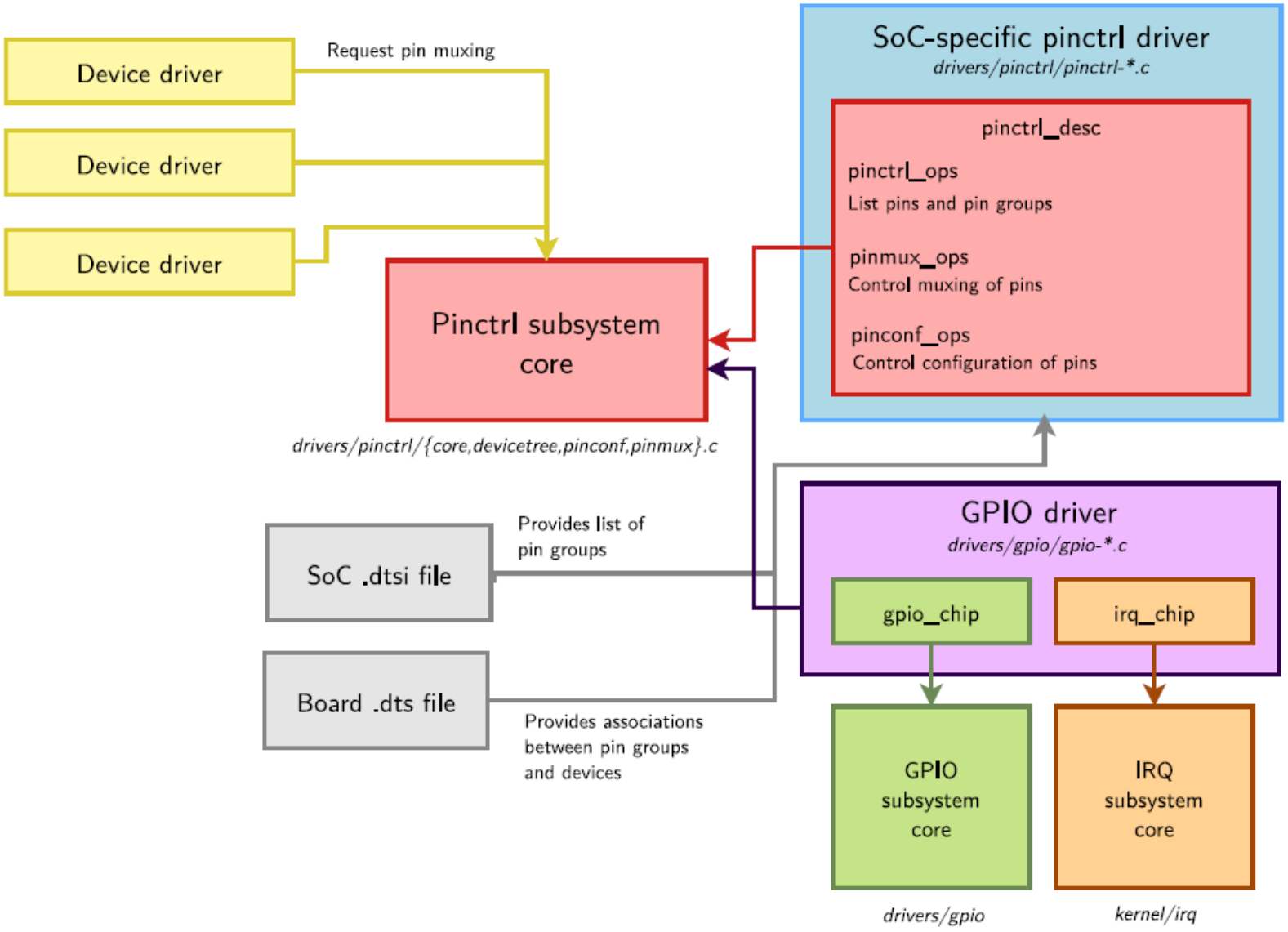
Pin muxing diagram



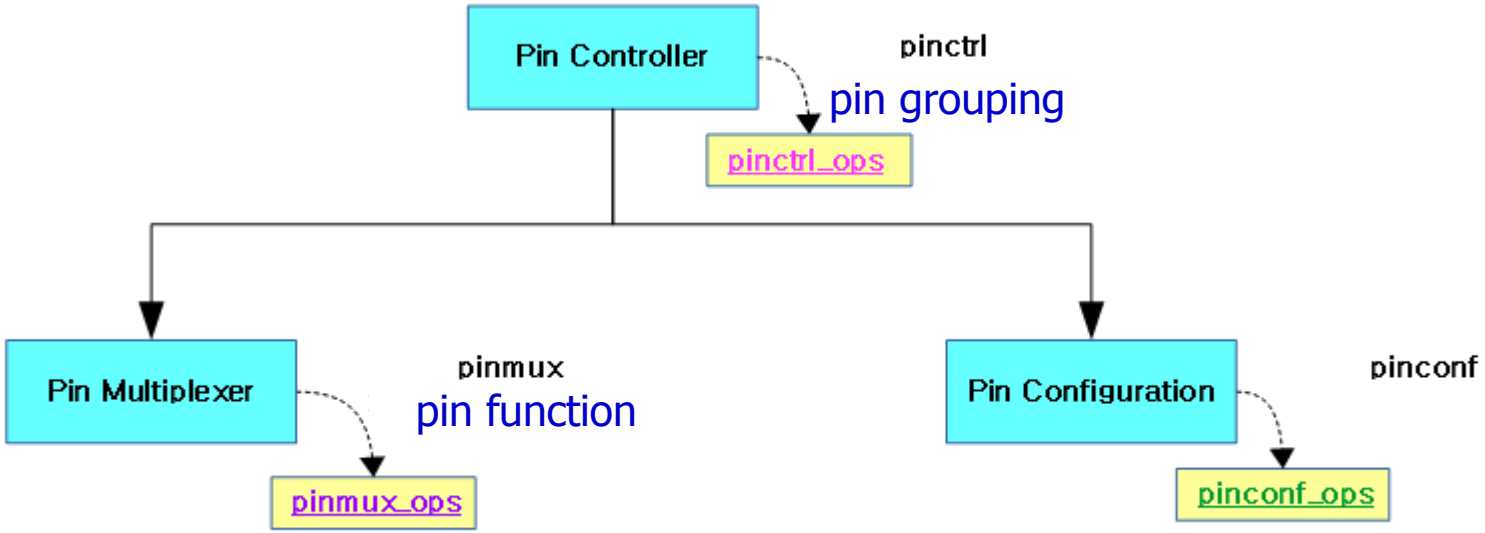
Pin muxing in the Linux kernel

- Since Linux 3.2, a pinctrl subsystem has been added.
- This subsystem, located in drivers/pinctrl/ provides a generic subsystem to handle pin muxing. It offers:
  - A pin muxing driver interface, to implement the system-on-chip specific drivers that configure the muxing.
  - A pin muxing consumer interface, for device drivers.
- Most pinctrl drivers provide a Device Tree binding, and the pin muxing must be described in the Device Tree.
  - The exact Device Tree binding depends on each driver.
  - Each binding is documented in Documentation/devicetree/bindings/pinctrl.

pinctrl subsystem diagram

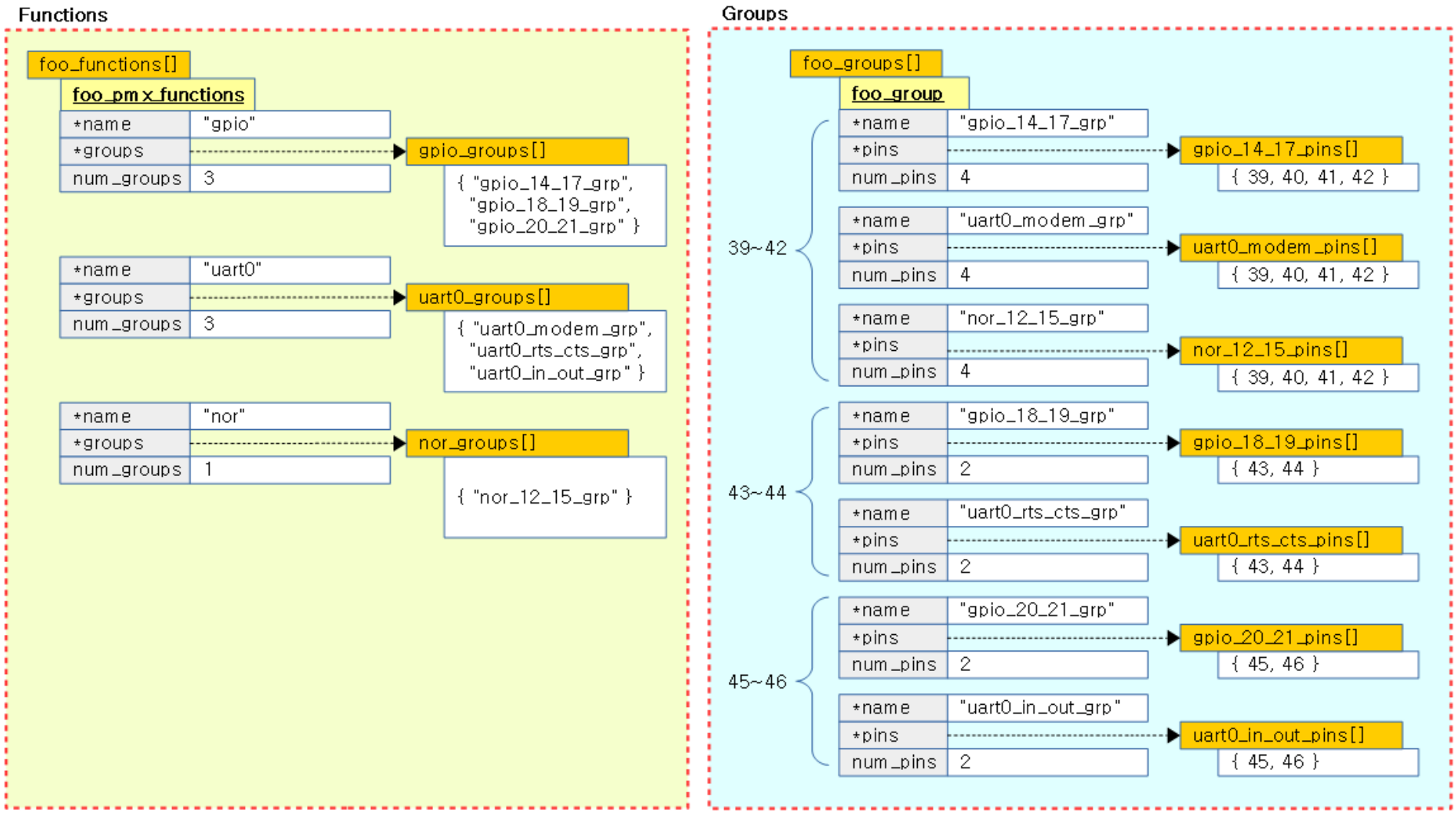


Pin Controller

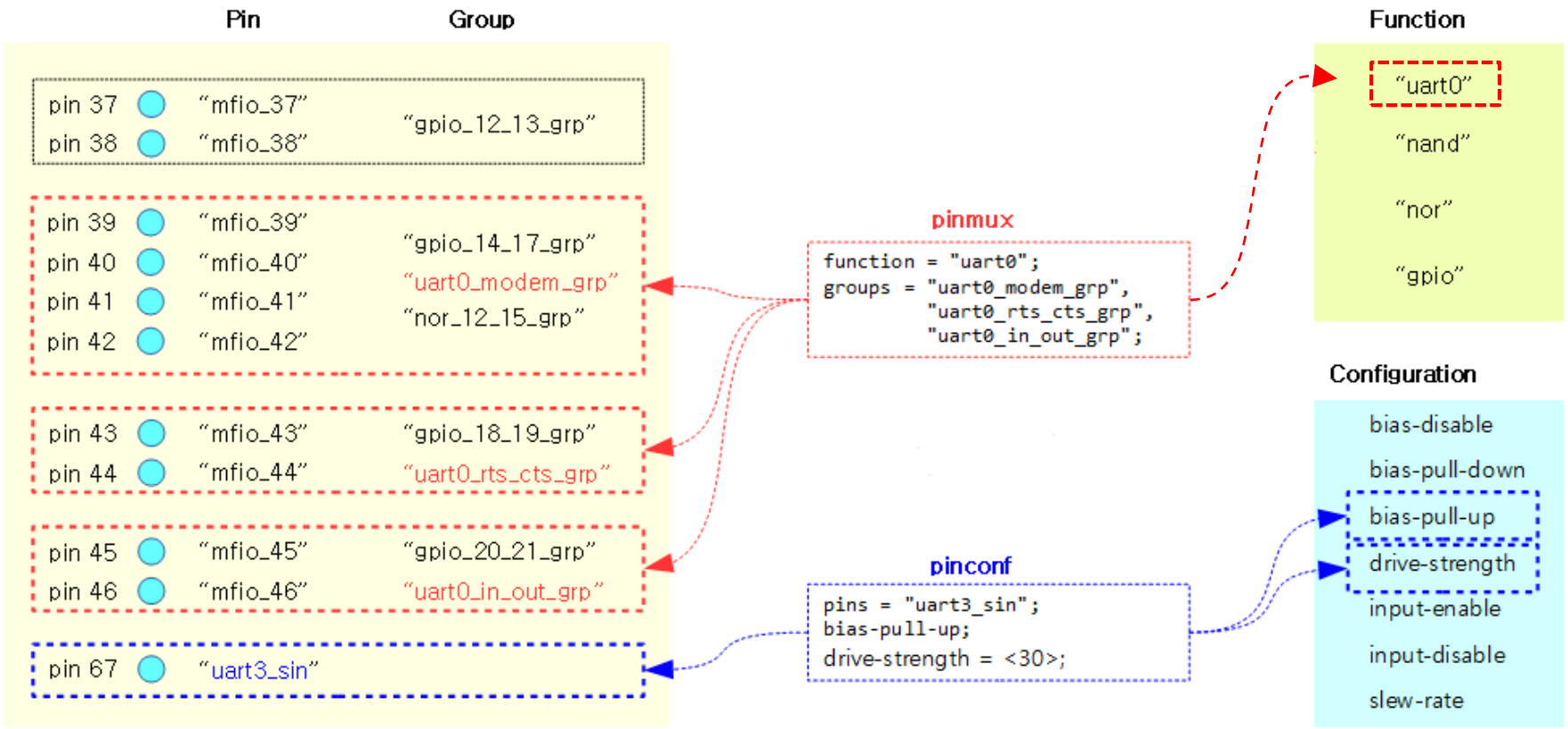



```
1static struct pinctrl_desc bcm2835_pinctrl_desc = {
2        .name = MODULE_NAME,
3        .pins = bcm2835_gpio_pins,
4        .npins = ARRAY_SIZE(bcm2835_gpio_pins),
5        .pctlops = &bcm2835_pctl_ops,
6        .pmxops = &bcm2835_pmx_ops,
7        .confops = &bcm2835_pinconf_ops,
8        .owner = THIS_MODULE,
9};
```

Pin Function & Group



pinmux & pinconf



Device Tree binding for consumer devices

- The devices that require certain pins to be muxed will use the pinctrl-<x> and pinctrl-names Device Tree properties.
- The pinctrl-0, pinctrl-1, pinctrl-<x> properties link to a pin configuration for a given state of the device.
- The pinctrl-names property associates a name to each state. The name default is special, and is automatically selected by a device driver, without having to make an explicit pinctrl function call.
- In most cases, the following is sufficient:

```
i2c@11000 {
        pinctrl-0 = <&pmx_twsi0>;
        pinctrl-names = "default";
        ...
};
```

- See Documentation/devicetree/bindings/pinctrl/pinctrl-bindings.txt for details.

Defining pinctrl configurations

- The different pinctrl configurations must be defined as child nodes of the main pinctrl *device* (which controls the muxing of pins).
- The configurations may be defined at:
  - SoC level (.dtsi file), for pin configurations that are often shared between multiple boards
  - board level (.dts file) for configurations that are board specific.
- The pinctrl-<x> property of the consumer device points to the pin configuration it needs through a DT phandle.
- The description of the configurations is specific to each pinctrl driver.
- See Documentation/devicetree/bindings/pinctrl for the DT bindings

Example on OMAP/AM33xx

- On OMAP/AM33xx, the pinctrl-single driver is used. It is common between multiple SoCs and simply allows to configure pins by writing a value to a register.
  - In each pin configuration, a pinctrl-single,pins value gives a list of *(register, value)* pairs needed to configure the pins.
- To know the correct values, one must use the SoC and board datasheets.

Pin configuration

```
/* Excerpt from am335x-boneblue.dts */

&am33xx_pinmux {
   ...
   i2c2_pins: pinmux_i2c2_pins {
      pinctrl-single,pins = <
         AM33XX_IOPAD(0x978, PIN_INPUT_PULLUP | MUX_MODE3)
         /* (D18) uart1_ctsn.I2C2_SDA */
         AM33XX_IOPAD(0x97c, PIN_INPUT_PULLUP | MUX_MODE3)
         /* (D17) uart1_rtsn.I2C2_SCL */
      >;
   };
};
```

Client

```
&i2c2 {
   pinctrl-names = "default";
   pinctrl-0 = <&i2c2_pins>;

   status = "okay";
   clock-frequency = <400000>;
   ...

   pressure@76 {
      compatible = "bosch,bmp280";
      reg = <0x76>;
   };
};
```

Example on Allwinner SoC