PINCTRL (PIN CONTROL) subsystem

This document outlines the pin control subsystem in Linux

This subsystem deals with:

- Enumerating and naming controllable pins
- Multiplexing of pins, pads, fingers (etc) see below for details
- Configuration of pins, pads, fingers (etc), such as software-controlled biasing and driving mode specific pins, such as pull-up/down, open drain, load capacitance etc.

Top-level interface

Definition of PIN CONTROLLER:

- A pin controller is a piece of hardware, usually a set of registers, that can control PINs. It may be able to multiplex, bias, set load capacitance, set drive strength, etc. for individual pins or groups of pins.

Definition of PIN:

- PINS are equal to pads, fingers, balls or whatever packaging input or output line you want to control and these are denoted by unsigned integers in the range 0..maxpin. This numberspace is local to each PIN CONTROLLER, so there may be several such number spaces in a system. This pin space may be sparse - i.e. there may be gaps in the space with numbers where no pin exists.

When a PIN CONTROLLER is instantiated, it will register a descriptor to the pin control framework, and this descriptor contains an array of pin descriptors describing the pins handled by this specific pin controller.

Here is an example of a PGA (Pin Grid Array) chip seen from underneath:

```
      A   B   C   D   E   F   G   H

8     o   o   o   o   o   o   o   o

7     o   o   o   o   o   o   o   o

6     o   o   o   o   o   o   o   o

5     o   o   o   o   o   o   o   o

4     o   o   o   o   o   o   o   o

3     o   o   o   o   o   o   o   o

2     o   o   o   o   o   o   o   o

1     o   o   o   o   o   o   o   o
```

To register a pin controller and name all the pins on this package we can do this in our driver:

```
#include <linux/pinctrl/pinctrl.h>

const struct pinctrl_pin_desc foo_pins[] = {
        PINCTRL_PIN(0, "A8"),
        PINCTRL_PIN(1, "B8"),
        PINCTRL_PIN(2, "C8"),
        ...
        PINCTRL_PIN(61, "F1"),
        PINCTRL_PIN(62, "G1"),
        PINCTRL_PIN(63, "H1"),
};

static struct pinctrl_desc foo_desc = {
        .name = "foo",
        .pins = foo_pins,
        .npins = ARRAY_SIZE(foo_pins),
        .owner = THIS_MODULE,
};

int __init foo_probe(void)
{
        int error;

        struct pinctrl_dev *pctl;

        error = pinctrl_register_and_init(&foo_desc, <PARENT>,
                                          NULL, &pctl);
```

```
        if (error)
                return error;

        return pinctrl_enable(pctl);
}
```

To enable the pinctrl subsystem and the subgroups for PINMUX and PINCONF and selected drivers, you need to select them from your machine's Kconfig entry, since these are so tightly integrated with the machines they are used on. See for example arch/arm/mach-ux500/Kconfig for an example.

Pins usually have fancier names than this. You can find these in the datasheet for your chip. Notice that the core pinctrl.h file provides a fancy macro called PINCTRL_PIN() to create the struct entries. As you can see I enumerated the pins from 0 in the upper left corner to 63 in the lower right corner. This enumeration was arbitrarily chosen, in practice you need to think through your numbering system so that it matches the layout of registers and such things in your driver, or the code may become complicated. You must also consider matching of offsets to the GPIO ranges that may be handled by the pin controller.

For a padring with 467 pads, as opposed to actual pins, I used an enumeration like this, walking around the edge of the chip, which seems to be industry standard too (all these pads had names, too):

```
  0 ..... 104
466        105
  .         .
  .         .
358        224
 357 .... 225
```

Pin groups

Many controllers need to deal with groups of pins, so the pin controller subsystem has a mechanism for enumerating groups of pins and retrieving the actual enumerated pins that are part of a certain group.

For example, say that we have a group of pins dealing with an SPI interface on { 0, 8, 16, 24 }, and a group of pins dealing with an I2C interface on pins on { 24, 25 }.

These two groups are presented to the pin control subsystem by implementing some generic pinctrl_ops like this:

```
#include <linux/pinctrl/pinctrl.h>

struct foo_group {
        const char *name;
        const unsigned int *pins;
        const unsigned num_pins;
};

static const unsigned int spi0_pins[] = { 0, 8, 16, 24 };
static const unsigned int i2c0_pins[] = { 24, 25 };

static const struct foo_group foo_groups[] = {
        {
                .name = "spi0_grp",
                .pins = spi0_pins,
                .num_pins = ARRAY_SIZE(spi0_pins),
        },
        {
                .name = "i2c0_grp",
                .pins = i2c0_pins,
                .num_pins = ARRAY_SIZE(i2c0_pins),
        },
};


static int foo_get_groups_count(struct pinctrl_dev *pctldev)
{
        return ARRAY_SIZE(foo_groups);
}

static const char *foo_get_group_name(struct pinctrl_dev *pctldev,
                                      unsigned selector)
{
        return foo_groups[selector].name;
}

static int foo_get_group_pins(struct pinctrl_dev *pctldev, unsigned selector,
                              const unsigned **pins,
                              unsigned *num_pins)
{
        *pins = (unsigned *) foo_groups[selector].pins;
        *num_pins = foo_groups[selector].num_pins;
        return 0;
```

```
}

static struct pinctrl_ops foo_pctrl_ops = {
        .get_groups_count = foo_get_groups_count,
        .get_group_name = foo_get_group_name,
        .get_group_pins = foo_get_group_pins,
};


static struct pinctrl_desc foo_desc = {
...
.pctlops = &foo_pctrl_ops,
};
```

The pin control subsystem will call the .get_groups_count() function to determine the total number of legal selectors, then it will call the other functions to retrieve the name and pins of the group. Maintaining the data structure of the groups is up to the driver, this is just a simple example - in practice you may need more entries in your group structure, for example specific register ranges associated with each group and so on.

Pin configuration

Pins can sometimes be software-configured in various ways, mostly related to their electronic properties when used as inputs or outputs. For example you may be able to make an output pin high impedance, or "tristate" meaning it is effectively disconnected. You may be able to connect an input pin to VDD or GND using a certain resistor value - pull up and pull down - so that the pin has a stable value when nothing is driving the rail it is connected to, or when it's unconnected.

Pin configuration can be programmed by adding configuration entries into the mapping table; see section "Board/machine configuration" below.

The format and meaning of the configuration parameter, PLATFORM_X_PULL_UP above, is entirely defined by the pin controller driver.

The pin configuration driver implements callbacks for changing pin configuration in the pin controller ops like this:

```
#include <linux/pinctrl/pinctrl.h>
#include <linux/pinctrl/pinconf.h>
#include "platform_x_pindefs.h"

static int foo_pin_config_get(struct pinctrl_dev *pctldev,
                unsigned offset,
                unsigned long *config)
{
        struct my_conftype conf;

        ... Find setting for pin @ offset ...

        *config = (unsigned long) conf;
}

static int foo_pin_config_set(struct pinctrl_dev *pctldev,
                unsigned offset,
                unsigned long config)
{
        struct my_conftype *conf = (struct my_conftype *) config;

        switch (conf) {
                case PLATFORM_X_PULL_UP:
                ...
                }
        }
}

static int foo_pin_config_group_get (struct pinctrl_dev *pctldev,
                unsigned selector,
                unsigned long *config)
{
        ...
}

static int foo_pin_config_group_set (struct pinctrl_dev *pctldev,
                unsigned selector,
                unsigned long config)
{
        ...
}

static struct pinconf_ops foo_pconf_ops = {
        .pin_config_get = foo_pin_config_get,
        .pin_config_set = foo_pin_config_set,
```

```
        .pin_config_group_get = foo_pin_config_group_get,
        .pin_config_group_set = foo_pin_config_group_set,
};

/* Pin config operations are handled by some pin controller */
static struct pinctrl_desc foo_desc = {
        ...
        .confops = &foo_pconf_ops,
};
```

Interaction with the GPIO subsystem

The GPIO drivers may want to perform operations of various types on the same physical pins that are also registered as pin controller pins.

First and foremost, the two subsystems can be used as completely orthogonal, see the section named "pin control requests from drivers" and "drivers needing both pin control and GPIOs" below for details. But in some situations a cross-subsystem mapping between pins and GPIOs is needed.

Since the pin controller subsystem has its pinspace local to the pin controller we need a mapping so that the pin control subsystem can figure out which pin controller handles control of a certain GPIO pin. Since a single pin controller may be muxing several GPIO ranges (typically SoCs that have one set of pins, but internally several GPIO silicon blocks, each modelled as a struct gpio_chip) any number of GPIO ranges can be added to a pin controller instance like this:

```
struct gpio_chip chip_a;
struct gpio_chip chip_b;

static struct pinctrl_gpio_range gpio_range_a = {
        .name = "chip a",
        .id = 0,
        .base = 32,
        .pin_base = 32,
        .npins = 16,
        .gc = &chip_a;
};

static struct pinctrl_gpio_range gpio_range_b = {
        .name = "chip b",
        .id = 0,
        .base = 48,
        .pin_base = 64,
        .npins = 8,
        .gc = &chip_b;
};

{
        struct pinctrl_dev *pctl;
        ...
        pinctrl_add_gpio_range(pctl, &gpio_range_a);
        pinctrl_add_gpio_range(pctl, &gpio_range_b);
}
```

So this complex system has one pin controller handling two different GPIO chips. "chip a" has 16 pins and "chip b" has 8 pins. The "chip a" and "chip b" have different .pin_base, which means a start pin number of the GPIO range.

The GPIO range of "chip a" starts from the GPIO base of 32 and actual pin range also starts from 32. However "chip b" has different starting offset for the GPIO range and pin range. The GPIO range of "chip b" starts from GPIO number 48, while the pin range of "chip b" starts from 64.

We can convert a gpio number to actual pin number using this "pin_base". They are mapped in the global GPIO pin space at:

chip a:

- GPIO range : [32 .. 47]
- pin range : [32 .. 47]

chip b:

- GPIO range : [48 .. 55]
- pin range : [64 .. 71]

The above examples assume the mapping between the GPIOs and pins is linear. If the mapping is sparse or haphazard, an array of arbitrary pin numbers can be encoded in the range like this:

```
static const unsigned range_pins[] = { 14, 1, 22, 17, 10, 8, 6, 2 };

static struct pinctrl_gpio_range gpio_range = {
        .name = "chip",
        .id = 0,
        .base = 32,
        .pins = &range_pins,
```

```
        .npins = ARRAY_SIZE(range_pins),
        .gc = &chip;
};
```

In this case the pin_base property will be ignored. If the name of a pin group is known, the pins and npins elements of the above structure can be initialised using the function pinctrl_get_group_pins(), e.g. for pin group "foo":

```
pinctrl_get_group_pins(pctl, "foo", &gpio_range.pins,
                       &gpio_range.npins);
```

When GPIO-specific functions in the pin control subsystem are called, these ranges will be used to look up the appropriate pin controller by inspecting and matching the pin to the pin ranges across all controllers. When a pin controller handling the matching range is found, GPIO-specific functions will be called on that specific pin controller.

For all functionalities dealing with pin biasing, pin muxing etc, the pin controller subsystem will look up the corresponding pin number from the passed in gpio number, and use the range's internals to retrieve a pin number. After that, the subsystem passes it on to the pin control driver, so the driver will get a pin number into its handled number range. Further it is also passed the range ID value, so that the pin controller knows which range it should deal with.

Calling pinctrl_add_gpio_range from pinctrl driver is DEPRECATED. Please see section 2.1 of Documentation/devicetree/bindings/gpio/gpio.txt on how to bind pinctrl and gpio drivers.

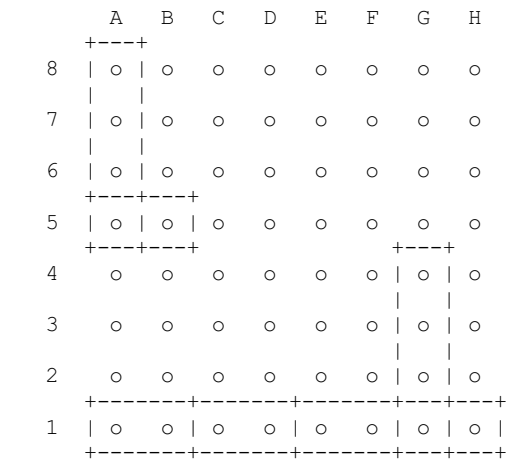
PINMUX interfaces

These calls use the pinmux_* naming prefix. No other calls should use that prefix.

What is pinmuxing?

PINMUX, also known as padmux, ballmux, alternate functions or mission modes is a way for chip vendors producing some kind of electrical packages to use a certain physical pin (ball, pad, finger, etc) for multiple mutually exclusive functions, depending on the application. By "application" in this context we usually mean a way of soldering or wiring the package into an electronic system, even though the framework makes it possible to also change the function at runtime.

Here is an example of a PGA (Pin Grid Array) chip seen from underneath:



This is not tetris. The game to think of is chess. Not all PGA/BGA packages are chessboard-like, big ones have "holes" in some arrangement according to different design patterns, but we're using this as a simple example. Of the pins you see some will be taken by things like a few VCC and GND to feed power to the chip, and quite a few will be taken by large ports like an external memory interface. The remaining pins will often be subject to pin multiplexing.

The example 8x8 PGA package above will have pin numbers 0 through 63 assigned to its physical pins. It will name the pins { A1, A2, A3 ... H6, H7, H8 } using pinctrl_register_pins() and a suitable data set as shown earlier.

In this 8x8 BGA package the pins { A8, A7, A6, A5 } can be used as an SPI port (these are four pins: CLK, RXD, TXD, FRM). In that case, pin B5 can be used as some general-purpose GPIO pin. However, in another setting, pins { A5, B5 } can be used as an I2C port (these are just two pins: SCL, SDA). Needless to say, we cannot use the SPI port and I2C port at the same time. However in the inside of the package the silicon performing the SPI logic can alternatively be routed out on pins { G4, G3, G2, G1 }.

On the bottom row at { A1, B1, C1, D1, E1, F1, G1, H1 } we have something special - it's an external MMC bus that can be 2, 4 or 8 bits wide, and it will consume 2, 4 or 8 pins respectively, so either { A1, B1 } are taken or { A1, B1, C1, D1 } or all of them. If we use all 8 bits, we cannot use the SPI port on pins { G4, G3, G2, G1 } of course.

This way the silicon blocks present inside the chip can be multiplexed "muxed" out on different pin ranges. Often contemporary SoC (systems on chip) will contain several I2C, SPI, SDIO/MMC, etc silicon blocks that can be routed to different pins by pinmux settings.

Since general-purpose I/O pins (GPIO) are typically always in shortage, it is common to be able to use almost any pin as a GPIO pin

if it is not currently in use by some other I/O port.

Pinmux conventions

The purpose of the pinmux functionality in the pin controller subsystem is to abstract and provide pinmux settings to the devices you choose to instantiate in your machine configuration. It is inspired by the clk, GPIO and regulator subsystems, so devices will request their mux setting, but it's also possible to request a single pin for e.g. GPIO.

Definitions:

- FUNCTIONS can be switched in and out by a driver residing with the pin control subsystem in the drivers/pinctrl/* directory of the kernel. The pin control driver knows the possible functions. In the example above you can identify three pinmux functions, one for spi, one for i2c and one for mmc.
- FUNCTIONS are assumed to be enumerable from zero in a one-dimensional array. In this case the array could be something like: { spi0, i2c0, mmc0 } for the three available functions.
- FUNCTIONS have PIN GROUPS as defined on the generic level - so a certain function is *always* associated with a certain set of pin groups, could be just a single one, but could also be many. In the example above the function i2c is associated with the pins { A5, B5 }, enumerated as { 24, 25 } in the controller pin space.

  The Function spi is associated with pin groups { A8, A7, A6, A5 } and { G4, G3, G2, G1 }, which are enumerated as { 0, 8, 16, 24 } and { 38, 46, 54, 62 } respectively.

  Group names must be unique per pin controller, no two groups on the same controller may have the same name.
- The combination of a FUNCTION and a PIN GROUP determine a certain function for a certain set of pins. The knowledge of the functions and pin groups and their machine-specific particulars are kept inside the pinmux driver, from the outside only the enumerators are known, and the driver core can request:
  - The name of a function with a certain selector (>= 0)
  - A list of groups associated with a certain function
  - That a certain group in that list to be activated for a certain function

  As already described above, pin groups are in turn self-descriptive, so the core will retrieve the actual pin range in a certain group from the driver.
- FUNCTIONS and GROUPS on a certain PIN CONTROLLER are MAPPED to a certain device by the board file, device tree or similar machine setup configuration mechanism, similar to how regulators are connected to devices, usually by name. Defining a pin controller, function and group thus uniquely identify the set of pins to be used by a certain device. (If only one possible group of pins is available for the function, no group name need to be supplied - the core will simply select the first and only group available.)

  In the example case we can define that this particular machine shall use device spi0 with pinmux function fspi0 group gspi0 and i2c0 on function fi2c0 group gi2c0, on the primary pin controller, we get mappings like these:

  ```
  {
          {"map-spi0", spi0, pinctrl0, fspi0, gspi0},
          {"map-i2c0", i2c0, pinctrl0, fi2c0, gi2c0}
  }
  ```

  Every map must be assigned a state name, pin controller, device and function. The group is not compulsory - if it is omitted the first group presented by the driver as applicable for the function will be selected, which is useful for simple cases.

  It is possible to map several groups to the same combination of device, pin controller and function. This is for cases where a certain function on a certain pin controller may use different sets of pins in different configurations.
- PINS for a certain FUNCTION using a certain PIN GROUP on a certain PIN CONTROLLER are provided on a first-come first-serve basis, so if some other device mux setting or GPIO pin request has already taken your physical pin, you will be denied the use of it. To get (activate) a new setting, the old one has to be put (deactivated) first.

Sometimes the documentation and hardware registers will be oriented around pads (or "fingers") rather than pins - these are the soldering surfaces on the silicon inside the package, and may or may not match the actual number of pins/balls underneath the capsule. Pick some enumeration that makes sense to you. Define enumerators only for the pins you can control if that makes sense.

Assumptions:

We assume that the number of possible function maps to pin groups is limited by the hardware. I.e. we assume that there is no system where any function can be mapped to any pin, like in a phone exchange. So the available pin groups for a certain function will be limited to a few choices (say up to eight or so), not hundreds or any amount of choices. This is the characteristic we have found by inspecting available pinmux hardware, and a necessary assumption since we expect pinmux drivers to present *all* possible function vs pin group mappings to the subsystem.

Pinmux drivers

The pinmux core takes care of preventing conflicts on pins and calling the pin controller driver to execute different settings.

It is the responsibility of the pinmux driver to impose further restrictions (say for example infer electronic limitations due to load, etc.) to determine whether or not the requested function can actually be allowed, and in case it is possible to perform the requested mux setting, poke the hardware so that this happens.

Pinmux drivers are required to supply a few callback functions, some are optional. Usually the set_mux() function is implemented, writing values into some certain registers to activate a certain mux setting for a certain pin.

A simple driver for the above example will work by setting bits 0, 1, 2, 3 or 4 into some register named MUX to select a certain function with a certain group of pins would work something like this:

```
#include <linux/pinctrl/pinctrl.h>
#include <linux/pinctrl/pinmux.h>

struct foo_group {
        const char *name;
        const unsigned int *pins;
        const unsigned num_pins;
};

static const unsigned spi0_0_pins[] = { 0, 8, 16, 24 };
static const unsigned spi0_1_pins[] = { 38, 46, 54, 62 };
static const unsigned i2c0_pins[] = { 24, 25 };
static const unsigned mmc0_1_pins[] = { 56, 57 };
static const unsigned mmc0_2_pins[] = { 58, 59 };
static const unsigned mmc0_3_pins[] = { 60, 61, 62, 63 };

static const struct foo_group foo_groups[] = {
        {
                .name = "spi0_0_grp",
                .pins = spi0_0_pins,
                .num_pins = ARRAY_SIZE(spi0_0_pins),
        },
        {
                .name = "spi0_1_grp",
                .pins = spi0_1_pins,
                .num_pins = ARRAY_SIZE(spi0_1_pins),
        },
        {
                .name = "i2c0_grp",
                .pins = i2c0_pins,
                .num_pins = ARRAY_SIZE(i2c0_pins),
        },
        {
                .name = "mmc0_1_grp",
                .pins = mmc0_1_pins,
                .num_pins = ARRAY_SIZE(mmc0_1_pins),
        },
        {
                .name = "mmc0_2_grp",
                .pins = mmc0_2_pins,
                .num_pins = ARRAY_SIZE(mmc0_2_pins),
        },
        {
                .name = "mmc0_3_grp",
                .pins = mmc0_3_pins,
                .num_pins = ARRAY_SIZE(mmc0_3_pins),
        },
};


static int foo_get_groups_count(struct pinctrl_dev *pctldev)
{
        return ARRAY_SIZE(foo_groups);
}

static const char *foo_get_group_name(struct pinctrl_dev *pctldev,
                                      unsigned selector)
{
        return foo_groups[selector].name;
}

static int foo_get_group_pins(struct pinctrl_dev *pctldev, unsigned selector,
                              const unsigned ** pins,
                              unsigned * num_pins)
{
        *pins = (unsigned *) foo_groups[selector].pins;
        *num_pins = foo_groups[selector].num_pins;
        return 0;
}

static struct pinctrl_ops foo_pctrl_ops = {
```

```
        .get_groups_count = foo_get_groups_count,
        .get_group_name = foo_get_group_name,
        .get_group_pins = foo_get_group_pins,
};

struct foo_pmx_func {
        const char *name;
        const char * const *groups;
        const unsigned num_groups;
};

static const char * const spi0_groups[] = { "spi0_0_grp", "spi0_1_grp" };
static const char * const i2c0_groups[] = { "i2c0_grp" };
static const char * const mmc0_groups[] = { "mmc0_1_grp", "mmc0_2_grp",
                                            "mmc0_3_grp" };

static const struct foo_pmx_func foo_functions[] = {
        {
                .name = "spi0",
                .groups = spi0_groups,
                .num_groups = ARRAY_SIZE(spi0_groups),
        },
        {
                .name = "i2c0",
                .groups = i2c0_groups,
                .num_groups = ARRAY_SIZE(i2c0_groups),
        },
        {
                .name = "mmc0",
                .groups = mmc0_groups,
                .num_groups = ARRAY_SIZE(mmc0_groups),
        },
};

static int foo_get_functions_count(struct pinctrl_dev *pctldev)
{
        return ARRAY_SIZE(foo_functions);
}

static const char *foo_get_fname(struct pinctrl_dev *pctldev, unsigned selector)
{
        return foo_functions[selector].name;
}

static int foo_get_groups(struct pinctrl_dev *pctldev, unsigned selector,
                        const char * const **groups,
                        unsigned * const num_groups)
{
        *groups = foo_functions[selector].groups;
        *num_groups = foo_functions[selector].num_groups;
        return 0;
}

static int foo_set_mux(struct pinctrl_dev *pctldev, unsigned selector,
                unsigned group)
{
        u8 regbit = (1 << selector + group);

        writeb((readb(MUX)|regbit), MUX);
        return 0;
}

static struct pinmux_ops foo_pmxops = {
        .get_functions_count = foo_get_functions_count,
        .get_function_name = foo_get_fname,
        .get_function_groups = foo_get_groups,
        .set_mux = foo_set_mux,
        .strict = true,
};

/* Pinmux operations are handled by some pin controller */
static struct pinctrl_desc foo_desc = {
        ...
        .pctlops = &foo_pctrl_ops,
        .pmxops = &foo_pmxops,
};
```

In the example activating muxing 0 and 1 at the same time setting bits 0 and 1, uses one pin in common so they would collide.

The beauty of the pinmux subsystem is that since it keeps track of all pins and who is using them, it will already have denied an impossible request like that, so the driver does not need to worry about such things - when it gets a selector passed in, the pinmux

subsystem makes sure no other device or GPIO assignment is already using the selected pins. Thus bits 0 and 1 in the control register will never be set at the same time.

All the above functions are mandatory to implement for a pinmux driver.

Pin control interaction with the GPIO subsystem

Note that the following implies that the use case is to use a certain pin from the Linux kernel using the API in <linux/gpio.h> with gpio_request() and similar functions. There are cases where you may be using something that your datasheet calls "GPIO mode", but actually is just an electrical configuration for a certain device. See the section below named "GPIO mode pitfalls" for more details on this scenario.

The public pinmux API contains two functions named pinctrl_gpio_request() and pinctrl_gpio_free(). These two functions shall *ONLY* be called from gpiolib-based drivers as part of their gpio_request() and gpio_free() semantics. Likewise the pinctrl_gpio_direction_[input|output] shall only be called from within respective gpio_direction_[input|output] gpiolib implementation.

NOTE that platforms and individual drivers shall *NOT* request GPIO pins to be controlled e.g. muxed in. Instead, implement a proper gpiolib driver and have that driver request proper muxing and other control for its pins.

The function list could become long, especially if you can convert every individual pin into a GPIO pin independent of any other pins, and then try the approach to define every pin as a function.

In this case, the function array would become 64 entries for each GPIO setting and then the device functions.

For this reason there are two functions a pin control driver can implement to enable only GPIO on an individual pin: .gpio_request_enable() and .gpio_disable_free().

This function will pass in the affected GPIO range identified by the pin controller core, so you know which GPIO pins are being affected by the request operation.

If your driver needs to have an indication from the framework of whether the GPIO pin shall be used for input or output you can implement the .gpio_set_direction() function. As described this shall be called from the gpiolib driver and the affected GPIO range, pin offset and desired direction will be passed along to this function.

Alternatively to using these special functions, it is fully allowed to use named functions for each GPIO pin, the pinctrl_gpio_request() will attempt to obtain the function "gpioN" where "N" is the global GPIO pin number if no special GPIO-handler is registered.

GPIO mode pitfalls

Due to the naming conventions used by hardware engineers, where "GPIO" is taken to mean different things than what the kernel does, the developer may be confused by a datasheet talking about a pin being possible to set into "GPIO mode". It appears that what hardware engineers mean with "GPIO mode" is not necessarily the use case that is implied in the kernel interface <linux/gpio.h>: a pin that you grab from kernel code and then either listen for input or drive high/low to assert/deassert some external line.

Rather hardware engineers think that "GPIO mode" means that you can software-control a few electrical properties of the pin that you would not be able to control if the pin was in some other mode, such as muxed in for a device.

The GPIO portions of a pin and its relation to a certain pin controller configuration and muxing logic can be constructed in several ways. Here are two examples:

```
(A)
                  pin config
                  logic regs
                  |               +- SPI
Physical pins --- pad --- pinmux -+- I2C
                  |       |       +- mmc
                  |       |       +- GPIO
                  |      pin
                  |      multiplex
                  |      logic regs
```

Here some electrical properties of the pin can be configured no matter whether the pin is used for GPIO or not. If you multiplex a GPIO onto a pin, you can also drive it high/low from "GPIO" registers. Alternatively, the pin can be controlled by a certain peripheral, while still applying desired pin config properties. GPIO functionality is thus orthogonal to any other device using the pin.

In this arrangement the registers for the GPIO portions of the pin controller, or the registers for the GPIO hardware module are likely to reside in a separate memory range only intended for GPIO driving, and the register range dealing with pin config and pin multiplexing get placed into a different memory range and a separate section of the data sheet.

A flag "strict" in struct pinmux_ops is available to check and deny simultaneous access to the same pin from GPIO and pin multiplexing consumers on hardware of this type. The pinctrl driver should set this flag accordingly.

```
(B)

                  pin config
                  logic regs
                  |               +- SPI
```

```
Physical pins --- pad --- pinmux -+- I2C
                  |       |       +- mmc
                  |       |
                  GPIO    pin
                          multiplex
                          logic regs
```

In this arrangement, the GPIO functionality can always be enabled, such that e.g. a GPIO input can be used to "spy" on the SPI/I2C/MMC signal while it is pulsed out. It is likely possible to disrupt the traffic on the pin by doing wrong things on the GPIO block, as it is never really disconnected. It is possible that the GPIO, pin config and pin multiplex registers are placed into the same memory range and the same section of the data sheet, although that need not be the case.

In some pin controllers, although the physical pins are designed in the same way as (B), the GPIO function still can't be enabled at the same time as the peripheral functions. So again the "strict" flag should be set, denying simultaneous activation by GPIO and other muxed in devices.

From a kernel point of view, however, these are different aspects of the hardware and shall be put into different subsystems:

- Registers (or fields within registers) that control electrical properties of the pin such as biasing and drive strength should be exposed through the pinctrl subsystem, as "pin configuration" settings.
- Registers (or fields within registers) that control muxing of signals from various other HW blocks (e.g. I2C, MMC, or GPIO) onto pins should be exposed through the pinctrl subsystem, as mux functions.
- Registers (or fields within registers) that control GPIO functionality such as setting a GPIO's output value, reading a GPIO's input value, or setting GPIO pin direction should be exposed through the GPIO subsystem, and if they also support interrupt capabilities, through the irqchip abstraction.

Depending on the exact HW register design, some functions exposed by the GPIO subsystem may call into the pinctrl subsystem in order to co-ordinate register settings across HW modules. In particular, this may be needed for HW with separate GPIO and pin controller HW modules, where e.g. GPIO direction is determined by a register in the pin controller HW module rather than the GPIO HW module.

Electrical properties of the pin such as biasing and drive strength may be placed at some pin-specific register in all cases or as part of the GPIO register in case (B) especially. This doesn't mean that such properties necessarily pertain to what the Linux kernel calls "GPIO".

Example: a pin is usually muxed in to be used as a UART TX line. But during system sleep, we need to put this pin into "GPIO mode" and ground it.

If you make a 1-to-1 map to the GPIO subsystem for this pin, you may start to think that you need to come up with something really complex, that the pin shall be used for UART TX and GPIO at the same time, that you will grab a pin control handle and set it to a certain state to enable UART TX to be muxed in, then twist it over to GPIO mode and use gpio_direction_output() to drive it low during sleep, then mux it over to UART TX again when you wake up and maybe even gpio_request/gpio_free as part of this cycle. This all gets very complicated.

The solution is to not think that what the datasheet calls "GPIO mode" has to be handled by the <linux/gpio.h> interface. Instead view this as a certain pin config setting. Look in e.g. <linux/pinctrl/pinconf-generic.h> and you find this in the documentation:

> PIN_CONFIG_OUTPUT:
> this will configure the pin in output, use argument 1 to indicate high level, argument 0 to indicate low level.

So it is perfectly possible to push a pin into "GPIO mode" and drive the line low as part of the usual pin control map. So for example your UART driver may look like this:

```
#include <linux/pinctrl/consumer.h>

struct pinctrl          *pinctrl;
struct pinctrl_state    *pins_default;
struct pinctrl_state    *pins_sleep;

pins_default = pinctrl_lookup_state(uap->pinctrl, PINCTRL_STATE_DEFAULT);
pins_sleep = pinctrl_lookup_state(uap->pinctrl, PINCTRL_STATE_SLEEP);

/* Normal mode */
retval = pinctrl_select_state(pinctrl, pins_default);
/* Sleep mode */
retval = pinctrl_select_state(pinctrl, pins_sleep);
```

And your machine configuration may look like this:

```
static unsigned long uart_default_mode[] = {
        PIN_CONF_PACKED(PIN_CONFIG_DRIVE_PUSH_PULL, 0),
};

static unsigned long uart_sleep_mode[] = {
        PIN_CONF_PACKED(PIN_CONFIG_OUTPUT, 0),
};
```

```
static struct pinctrl_map pinmap[] __initdata = {
        PIN_MAP_MUX_GROUP("uart", PINCTRL_STATE_DEFAULT, "pinctrl-foo",
                "u0_group", "u0"),
        PIN_MAP_CONFIGS_PIN("uart", PINCTRL_STATE_DEFAULT, "pinctrl-foo",
                        "UART_TX_PIN", uart_default_mode),
        PIN_MAP_MUX_GROUP("uart", PINCTRL_STATE_SLEEP, "pinctrl-foo",
                "u0_group", "gpio-mode"),
        PIN_MAP_CONFIGS_PIN("uart", PINCTRL_STATE_SLEEP, "pinctrl-foo",
                        "UART_TX_PIN", uart_sleep_mode),
};

foo_init(void) {
        pinctrl_register_mappings(pinmap, ARRAY_SIZE(pinmap));
}
```

Here the pins we want to control are in the "u0_group" and there is some function called "u0" that can be enabled on this group of pins, and then everything is UART business as usual. But there is also some function named "gpio-mode" that can be mapped onto the same pins to move them into GPIO mode.

This will give the desired effect without any bogus interaction with the GPIO subsystem. It is just an electrical configuration used by that device when going to sleep, it might imply that the pin is set into something the datasheet calls "GPIO mode", but that is not the point: it is still used by that UART device to control the pins that pertain to that very UART driver, putting them into modes needed by the UART. GPIO in the Linux kernel sense are just some 1-bit line, and is a different use case.

How the registers are poked to attain the push or pull, and output low configuration and the muxing of the "u0" or "gpio-mode" group onto these pins is a question for the driver.

Some datasheets will be more helpful and refer to the "GPIO mode" as "low power mode" rather than anything to do with GPIO. This often means the same thing electrically speaking, but in this latter case the software engineers will usually quickly identify that this is some specific muxing or configuration rather than anything related to the GPIO API.

Board/machine configuration

Boards and machines define how a certain complete running system is put together, including how GPIOs and devices are muxed, how regulators are constrained and how the clock tree looks. Of course pinmux settings are also part of this.

A pin controller configuration for a machine looks pretty much like a simple regulator configuration, so for the example array above we want to enable i2c and spi on the second function mapping:

```
#include <linux/pinctrl/machine.h>

static const struct pinctrl_map mapping[] __initconst = {
        {
                .dev_name = "foo-spi.0",
                .name = PINCTRL_STATE_DEFAULT,
                .type = PIN_MAP_TYPE_MUX_GROUP,
                .ctrl_dev_name = "pinctrl-foo",
                .data.mux.function = "spi0",
        },
        {
                .dev_name = "foo-i2c.0",
                .name = PINCTRL_STATE_DEFAULT,
                .type = PIN_MAP_TYPE_MUX_GROUP,
                .ctrl_dev_name = "pinctrl-foo",
                .data.mux.function = "i2c0",
        },
        {
                .dev_name = "foo-mmc.0",
                .name = PINCTRL_STATE_DEFAULT,
                .type = PIN_MAP_TYPE_MUX_GROUP,
                .ctrl_dev_name = "pinctrl-foo",
                .data.mux.function = "mmc0",
        },
};
```

The dev_name here matches to the unique device name that can be used to look up the device struct (just like with clockdev or regulators). The function name must match a function provided by the pinmux driver handling this pin range.

As you can see we may have several pin controllers on the system and thus we need to specify which one of them contains the functions we wish to map.

You register this pinmux mapping to the pinmux subsystem by simply:

```
ret = pinctrl_register_mappings(mapping, ARRAY_SIZE(mapping));
```

Since the above construct is pretty common there is a helper macro to make it even more compact which assumes you want to use pinctrl-foo and position 0 for mapping, for example:

```
static struct pinctrl_map mapping[] __initdata = {
        PIN_MAP_MUX_GROUP("foo-i2c.o", PINCTRL_STATE_DEFAULT,
                          "pinctrl-foo", NULL, "i2c0"),
};
```

The mapping table may also contain pin configuration entries. It's common for each pin/group to have a number of configuration entries that affect it, so the table entries for configuration reference an array of config parameters and values. An example using the convenience macros is shown below:

```
static unsigned long i2c_grp_configs[] = {
        FOO_PIN_DRIVEN,
        FOO_PIN_PULLUP,
};

static unsigned long i2c_pin_configs[] = {
        FOO_OPEN_COLLECTOR,
        FOO_SLEW_RATE_SLOW,
};

static struct pinctrl_map mapping[] __initdata = {
        PIN_MAP_MUX_GROUP("foo-i2c.0", PINCTRL_STATE_DEFAULT,
                          "pinctrl-foo", "i2c0", "i2c0"),
        PIN_MAP_CONFIGS_GROUP("foo-i2c.0", PINCTRL_STATE_DEFAULT,
                              "pinctrl-foo", "i2c0", i2c_grp_configs),
        PIN_MAP_CONFIGS_PIN("foo-i2c.0", PINCTRL_STATE_DEFAULT,
                            "pinctrl-foo", "i2c0scl", i2c_pin_configs),
        PIN_MAP_CONFIGS_PIN("foo-i2c.0", PINCTRL_STATE_DEFAULT,
                            "pinctrl-foo", "i2c0sda", i2c_pin_configs),
};
```

Finally, some devices expect the mapping table to contain certain specific named states. When running on hardware that doesn't need any pin controller configuration, the mapping table must still contain those named states, in order to explicitly indicate that the states were provided and intended to be empty. Table entry macro PIN_MAP_DUMMY_STATE serves the purpose of defining a named state without causing any pin controller to be programmed:

```
static struct pinctrl_map mapping[] __initdata = {
        PIN_MAP_DUMMY_STATE("foo-i2c.0", PINCTRL_STATE_DEFAULT),
};
```

Complex mappings

As it is possible to map a function to different groups of pins an optional .group can be specified like this:

```
...
{
        .dev_name = "foo-spi.0",
        .name = "spi0-pos-A",
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "spi0",
        .group = "spi0_0_grp",
},
{
        .dev_name = "foo-spi.0",
        .name = "spi0-pos-B",
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "spi0",
        .group = "spi0_1_grp",
},
...
```

This example mapping is used to switch between two positions for spi0 at runtime, as described further below under the heading "Runtime pinmuxing".

Further it is possible for one named state to affect the muxing of several groups of pins, say for example in the mmc0 example above, where you can additively expand the mmc0 bus from 2 to 4 to 8 pins. If we want to use all three groups for a total of 2+2+4 = 8 pins (for an 8-bit MMC bus as is the case), we define a mapping like this:

```
...
{
        .dev_name = "foo-mmc.0",
        .name = "2bit"
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "mmc0",
        .group = "mmc0_1_grp",
},
{
```

```
        .dev_name = "foo-mmc.0",
        .name = "4bit"
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "mmc0",
        .group = "mmc0_1_grp",
},
{
        .dev_name = "foo-mmc.0",
        .name = "4bit"
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "mmc0",
        .group = "mmc0_2_grp",
},
{
        .dev_name = "foo-mmc.0",
        .name = "8bit"
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "mmc0",
        .group = "mmc0_1_grp",
},
{
        .dev_name = "foo-mmc.0",
        .name = "8bit"
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "mmc0",
        .group = "mmc0_2_grp",
},
{
        .dev_name = "foo-mmc.0",
        .name = "8bit"
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "mmc0",
        .group = "mmc0_3_grp",
},
...
```

The result of grabbing this mapping from the device with something like this (see next paragraph):

```
p = devm_pinctrl_get(dev);
s = pinctrl_lookup_state(p, "8bit");
ret = pinctrl_select_state(p, s);
```

or more simply:

```
p = devm_pinctrl_get_select(dev, "8bit");
```

Will be that you activate all the three bottom records in the mapping at once. Since they share the same name, pin controller device, function and device, and since we allow multiple groups to match to a single device, they all get selected, and they all get enabled and disable simultaneously by the pinmux core.

Pin control requests from drivers

When a device driver is about to probe the device core will automatically attempt to issue pinctrl_get_select_default() on these devices. This way driver writers do not need to add any of the boilerplate code of the type found below. However when doing fine-grained state selection and not using the "default" state, you may have to do some device driver handling of the pinctrl handles and states.

So if you just want to put the pins for a certain device into the default state and be done with it, there is nothing you need to do besides providing the proper mapping table. The device core will take care of the rest.

Generally it is discouraged to let individual drivers get and enable pin control. So if possible, handle the pin control in platform code or some other place where you have access to all the affected struct device * pointers. In some cases where a driver needs to e.g. switch between different mux mappings at runtime this is not possible.

A typical case is if a driver needs to switch bias of pins from normal operation and going to sleep, moving from the PINCTRL_STATE_DEFAULT to PINCTRL_STATE_SLEEP at runtime, re-biasing or even re-muxing pins to save current in sleep mode.

A driver may request a certain control state to be activated, usually just the default state like this:

```
#include <linux/pinctrl/consumer.h>

struct foo_state {
struct pinctrl *p;
```

```
struct pinctrl_state *s;
...
};

foo_probe()
{
        /* Allocate a state holder named "foo" etc */
        struct foo_state *foo = ...;

        foo->p = devm_pinctrl_get(&device);
        if (IS_ERR(foo->p)) {
                /* FIXME: clean up "foo" here */
                return PTR_ERR(foo->p);
        }

        foo->s = pinctrl_lookup_state(foo->p, PINCTRL_STATE_DEFAULT);
        if (IS_ERR(foo->s)) {
                /* FIXME: clean up "foo" here */
                return PTR_ERR(foo->s);
        }

        ret = pinctrl_select_state(foo->s);
        if (ret < 0) {
                /* FIXME: clean up "foo" here */
                return ret;
        }
}
```

This get/lookup/select/put sequence can just as well be handled by bus drivers if you don't want each and every driver to handle it and you know the arrangement on your bus.

The semantics of the pinctrl APIs are:

- pinctrl_get() is called in process context to obtain a handle to all pinctrl information for a given client device. It will allocate a struct from the kernel memory to hold the pinmux state. All mapping table parsing or similar slow operations take place within this API.
- devm_pinctrl_get() is a variant of pinctrl_get() that causes pinctrl_put() to be called automatically on the retrieved pointer when the associated device is removed. It is recommended to use this function over plain pinctrl_get().
- pinctrl_lookup_state() is called in process context to obtain a handle to a specific state for a client device. This operation may be slow, too.
- pinctrl_select_state() programs pin controller hardware according to the definition of the state as given by the mapping table. In theory, this is a fast-path operation, since it only involved blasting some register settings into hardware. However, note that some pin controllers may have their registers on a slow/IRQ-based bus, so client devices should not assume they can call pinctrl_select_state() from non-blocking contexts.
- pinctrl_put() frees all information associated with a pinctrl handle.
- devm_pinctrl_put() is a variant of pinctrl_put() that may be used to explicitly destroy a pinctrl object returned by devm_pinctrl_get(). However, use of this function will be rare, due to the automatic cleanup that will occur even without calling it.

  pinctrl_get() must be paired with a plain pinctrl_put(). pinctrl_get() may not be paired with devm_pinctrl_put(). devm_pinctrl_get() can optionally be paired with devm_pinctrl_put(). devm_pinctrl_get() may not be paired with plain pinctrl_put().

Usually the pin control core handled the get/put pair and call out to the device drivers bookkeeping operations, like checking available functions and the associated pins, whereas select_state pass on to the pin controller driver which takes care of activating and/or deactivating the mux setting by quickly poking some registers.

The pins are allocated for your device when you issue the devm_pinctrl_get() call, after this you should be able to see this in the debugfs listing of all pins.

NOTE: the pinctrl system will return -EPROBE_DEFER if it cannot find the requested pinctrl handles, for example if the pinctrl driver has not yet registered. Thus make sure that the error path in your driver gracefully cleans up and is ready to retry the probing later in the startup process.

Drivers needing both pin control and GPIOs

Again, it is discouraged to let drivers lookup and select pin control states themselves, but again sometimes this is unavoidable.

So say that your driver is fetching its resources like this:

```
#include <linux/pinctrl/consumer.h>
#include <linux/gpio.h>

struct pinctrl *pinctrl;
```

```
int gpio;

pinctrl = devm_pinctrl_get_select_default(&dev);
gpio = devm_gpio_request(&dev, 14, "foo");
```

Here we first request a certain pin state and then request GPIO 14 to be used. If you're using the subsystems orthogonally like this, you should nominally always get your pinctrl handle and select the desired pinctrl state BEFORE requesting the GPIO. This is a semantic convention to avoid situations that can be electrically unpleasant, you will certainly want to mux in and bias pins in a certain way before the GPIO subsystems starts to deal with them.

The above can be hidden: using the device core, the pinctrl core may be setting up the config and muxing for the pins right before the device is probing, nevertheless orthogonal to the GPIO subsystem.

But there are also situations where it makes sense for the GPIO subsystem to communicate directly with the pinctrl subsystem, using the latter as a back-end. This is when the GPIO driver may call out to the functions described in the section "Pin control interaction with the GPIO subsystem" above. This only involves per-pin multiplexing, and will be completely hidden behind the gpio_*() function namespace. In this case, the driver need not interact with the pin control subsystem at all.

If a pin control driver and a GPIO driver is dealing with the same pins and the use cases involve multiplexing, you MUST implement the pin controller as a back-end for the GPIO driver like this, unless your hardware design is such that the GPIO controller can override the pin controller's multiplexing state through hardware without the need to interact with the pin control system.

System pin control hogging

Pin control map entries can be hogged by the core when the pin controller is registered. This means that the core will attempt to call pinctrl_get(), lookup_state() and select_state() on it immediately after the pin control device has been registered.

This occurs for mapping table entries where the client device name is equal to the pin controller device name, and the state name is PINCTRL_STATE_DEFAULT:

```
{
        .dev_name = "pinctrl-foo",
        .name = PINCTRL_STATE_DEFAULT,
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "power_func",
},
```

Since it may be common to request the core to hog a few always-applicable mux settings on the primary pin controller, there is a convenience macro for this:

```
PIN_MAP_MUX_GROUP_HOG_DEFAULT("pinctrl-foo", NULL /* group */,
                              "power_func")
```

This gives the exact same result as the above construction.

Runtime pinmuxing

It is possible to mux a certain function in and out at runtime, say to move an SPI port from one set of pins to another set of pins. Say for example for spi0 in the example above, we expose two different groups of pins for the same function, but with different named in the mapping as described under "Advanced mapping" above. So that for an SPI device, we have two states named "pos-A" and "pos-B".

This snippet first initializes a state object for both groups (in foo_probe()), then muxes the function in the pins defined by group A, and finally muxes it in on the pins defined by group B:

```
#include <linux/pinctrl/consumer.h>

struct pinctrl *p;
struct pinctrl_state *s1, *s2;

foo_probe()
{
        /* Setup */
        p = devm_pinctrl_get(&device);
        if (IS_ERR(p))
                ...

        s1 = pinctrl_lookup_state(foo->p, "pos-A");
        if (IS_ERR(s1))
                ...

        s2 = pinctrl_lookup_state(foo->p, "pos-B");
        if (IS_ERR(s2))
                ...
}
```

```
foo_switch()
{
        /* Enable on position A */
        ret = pinctrl_select_state(s1);
        if (ret < 0)
        ...

        ...

        /* Enable on position B */
        ret = pinctrl_select_state(s2);
        if (ret < 0)
        ...

        ...
}
```

The above has to be done from process context. The reservation of the pins will be done when the state is activated, so in effect one specific pin can be used by different functions at different times on a running system.

Debugfs files

These files are created in `/sys/kernel/debug/pinctrl`:

- `pinctrl-devices`: prints each pin controller device along with columns to indicate support for pinmux and pinconf
- `pinctrl-handles`: prints each configured pin controller handle and the corresponding pinmux maps
- `pinctrl-maps`: print all pinctrl maps

A sub-directory is created inside of `/sys/kernel/debug/pinctrl` for each pin controller device containing these files:

- `pins`: prints a line for each pin registered on the pin controller. The pinctrl driver may add additional information such as register contents.
- `gpio-ranges`: print ranges that map gpio lines to pins on the controller
- `pingroups`: print all pin groups registered on the pin controller
- `pinconf-pins`: print pin config settings for each pin
- `pinconf-groups`: print pin config settings per pin group
- `pinmux-functions`: print each pin function along with the pin groups that map to the pin function
- `pinmux-pins`: iterate through all pins and print mux owner, gpio owner and if the pin is a hog
- `pinmux-select`: write to this file to activate a pin function for a group:

```
    echo "<group-name function-name>" > pinmux-select
```