PIN CONTROL OVERVIEW

2012-05-16

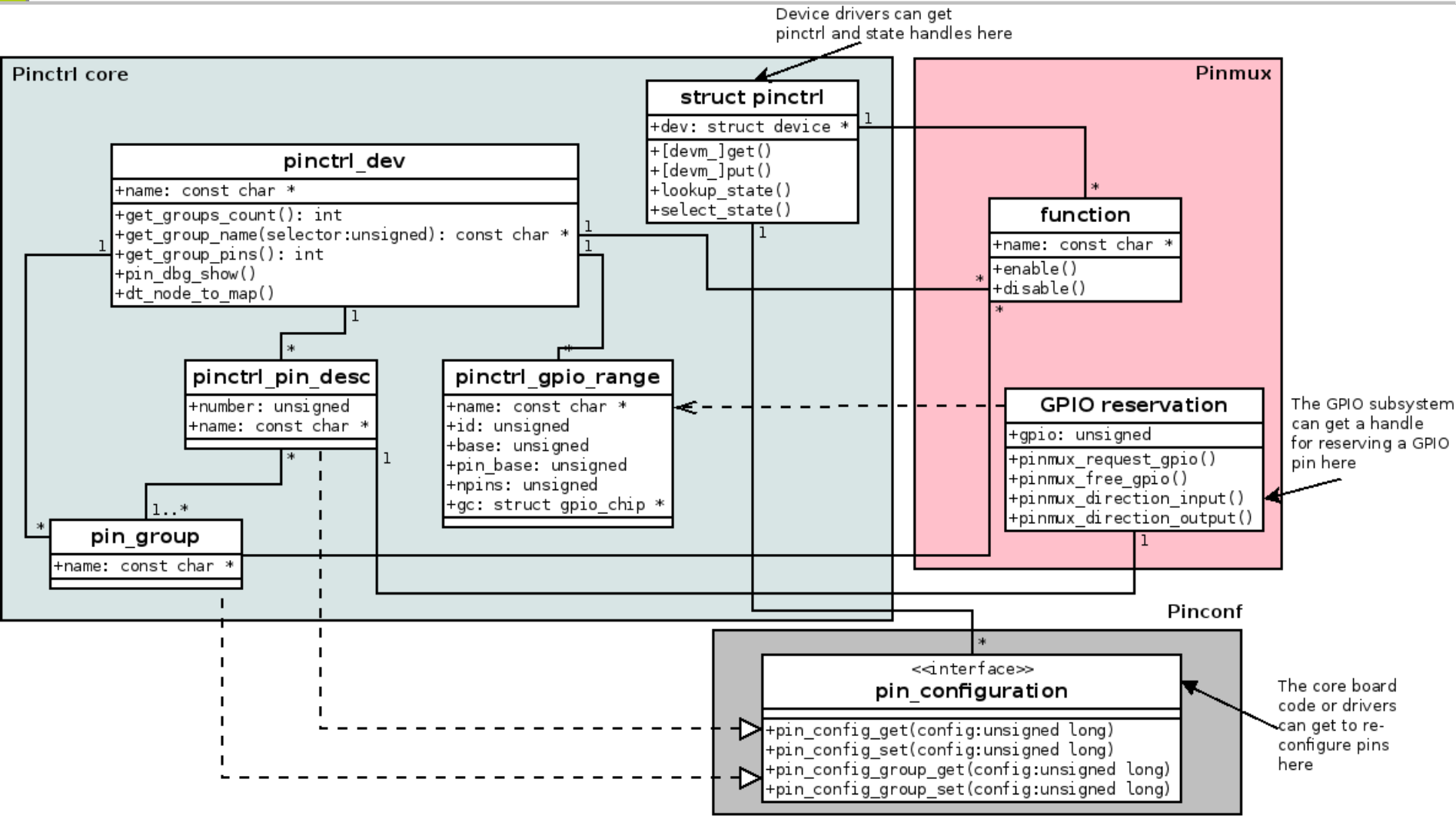
Linus Walleij
linus.walleij@linaro.org

Linaro CONNECT

Pins are driver resources

- Hardware base address for MMIO
- IRQ number
- DMA channels
- Clocks
- Regulators
- Pins
- Platform data and callbacks

The Subsystem is Simple...



Linaro CONNECT

Read the Docs

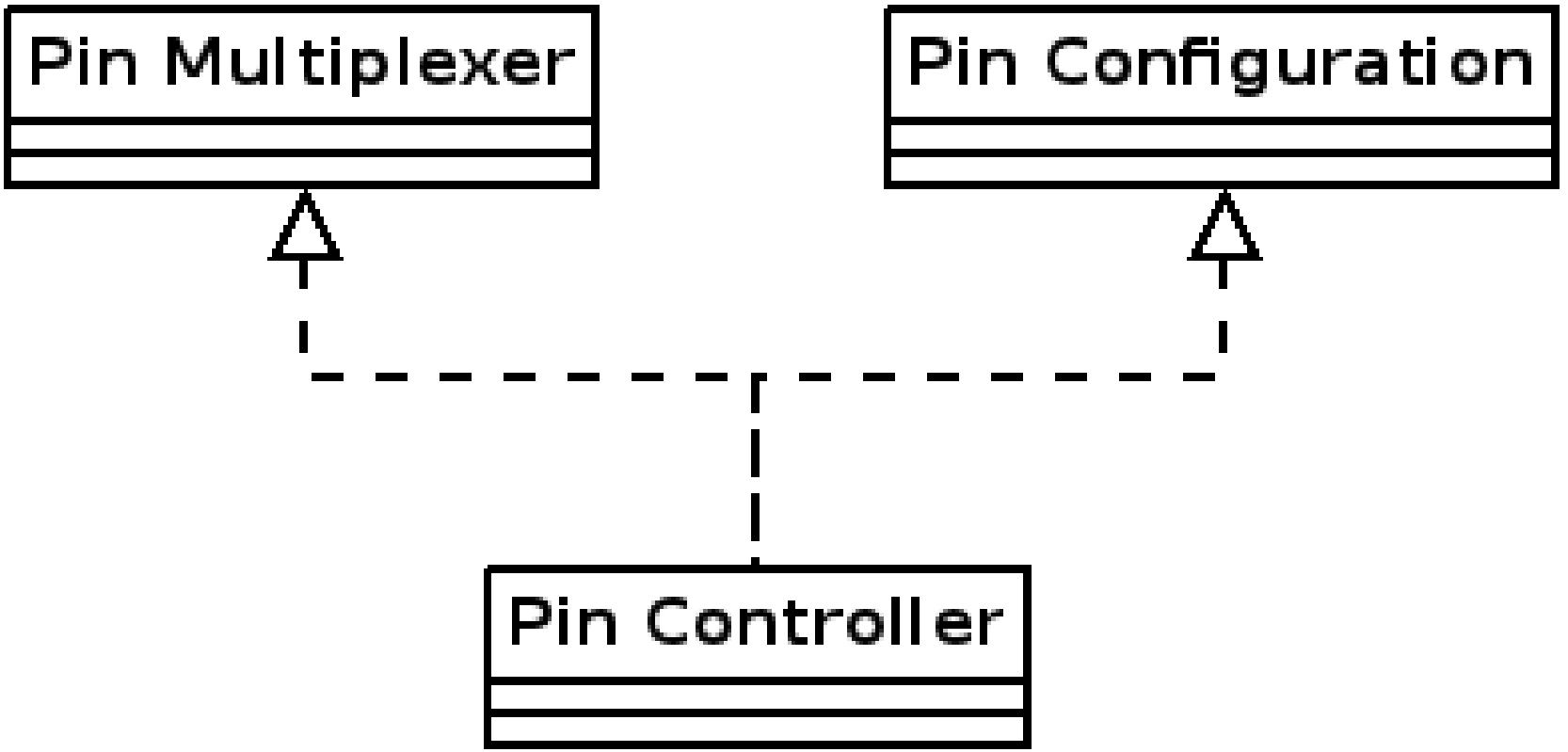
- Check Documentation/pinctrl.txt
- This is actually intended to be textbook quality, and was used as basis for the LWN article: http://lwn.net/Articles/468759/
- Then I had a big lecture on pin control at the Embedded Linux Conference: http://video.linux.com/videos/pin-control-subsystem

  http://www.df.lth.se/~triad/papers/pincontrol.pdf

Creating a Pin Control Driver

- Add PINCTRL_FOO in drivers/pinctrl/Kconfig
- Add pinctrl-foo.o into drivers/pinctrl/Makefile
- Add select PINCTRL and select PINCTRL_FOO to your SoC/system's Kconfig in arch/arm/mach-foo/Kconfig
- Edit drivers/pinctrl/pinctrl-foo.c
- Add some hogs in you SoC code
- Modify some drivers to request pins

Pin Control Can Do Two Things



How Drivers Use Pin Control

```
#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(s);
     }

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

drivers/tty/serial/amba-pl011.c

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

struct uart_amba_port {
      struct pinctrl          *pinctrl;
      struct pinctrl_state    *pins_default;
      struct pinctrl_state    *pins_sleep;
};

static int pl011_startup(struct uart_port *port) {
      if (!IS_ERR(uap->pins_default)) {
            retval = pinctrl_select_state(uap->pinctrl, uap->pins_default);
            if (retval)
                  dev_err(port->dev,
                        "could not set default pins\n");
      }
}

static void pl011_shutdown(struct uart_port *port) {
      if (!IS_ERR(uap->pins_sleep)) {
            retval = pinctrl_select_state(uap->pinctrl, uap->pins_sleep);
            if (retval)
                  dev_err(port->dev,
                        "could not set pins to sleep state\n");
      }
}

static int pl011_probe(struct amba_device *dev, const struct amba_id *id) {
      uap->pinctrl = devm_pinctrl_get(&dev->dev);
      if (IS_ERR(uap->pinctrl)) {
            ret = PTR_ERR(uap->pinctrl);
            goto unmap;
      }
      uap->pins_default = pinctrl_lookup_state(uap->pinctrl,
                                    PINCTRL_STATE_DEFAULT);
      if (IS_ERR(uap->pins_default))
            dev_err(&dev->dev, "could not get default pinstate\n");

      uap->pins_sleep = pinctrl_lookup_state(uap->pinctrl,
                                    PINCTRL_STATE_SLEEP);
      if (IS_ERR(uap->pins_sleep))
            dev_dbg(&dev->dev, "could not get sleep pinstate\n");
}
```

Pin Control per se

- Enumerate and present all pins/pads/balls/fingers (etc) on the system
- Cross-reference pins with GPIO lines
- Sort pins into groups (discrete sets)
- Provide handles for consumers (e.g. Drivers) to get an abstract struct pinctrl * representing the pins and groups used by that driver
- Functions for putting the handles into different named states “default”, “idle”, “sleep” etc.

```
#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),
     .maxpin = 63,
     .owner = THIS_MODULE,
};

int __init foo_probe(void)
{
     struct pinctrl_dev *pctl;

     pctl = pinctrl_register(&foo_desc, <PARENT>, NULL);
     if (IS_ERR(pctl))
          pr_err("could not register foo pin driver\n");
}
```

Pin Multiplexer “pinmux”

- Also known as “alternate functions” and “mission modes”
- Takes a pin group and maps it to a *function*
- A function is a certain way to configure a set of pins for a particular usecase
- By associating a pin group to a function a named map entry is created
- This map entry name matches the pin control states that can be activated from the handlers

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

static const struct foo_group foo_groups[] = {
      {
            .name = "spi0_grp",
            .pins = spi0_pins,
            .num_pins = ARRAY_SIZE(spi0_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,
                        unsigned ** const pins,
                        unsigned * const 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,
};
```

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

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 struct foo_pmx_func foo_functions[] = {
        {
                .name = "spi0",
                .groups = spi0_groups,
                .num_groups = ARRAY_SIZE(spi0_groups),
        },
};

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

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

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

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

void foo_disable(struct pinctrl_dev *pctldev, unsigned selector,
               unsigned group)
{
        u8 regbit = (1 << selector + group);

        writeb((readb(MUX) & ~(regbit)), MUX)
        return 0;
}

struct pinmux_ops foo_pmxops = {
        .get_functions_count = foo_get_functions_count,
        .get_function_name = foo_get_fname,
        .get_function_groups = foo_get_groups,
        .enable = foo_enable,
        .disable = foo_disable,
};

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

Pin Configuration “pinconf”

- Configures individual pins for a certain named setup
- The configuration is an opaque unsigned long to be defined and interpreted by the driver
- Optionally generic pin config definitions can be used
- The configurations are stored in named entries in a table and looked up and activated by pinctrl handler states, just like pinmux function-to-group map entries

```
#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,
};
```

Pin Multiplexing + Pin Configuration

- You can multiplex a group of pins to a certain function *and* configure them at the same time
- The core will just look up a state and activate it
- This activation can include pin multiplexing and pin configuration alike

Pin Control Hogs

- All pinmux and pinconf maps are tied to a certain pin controller and a certain device
- If you map the table entry to the pin controller itself, you create a *hog*
- The hog will be activated and put into the default state when the pin controller is loaded
- For a simple embedded system a table with hogs may be all that is ever needed, no drivers need any pin control handles

Simple Hog Table
arch/arm/mach-u300/core.c

```
static unsigned long pin_pullup_conf[] = {
     PIN_CONF_PACKED(PIN_CONFIG_BIAS_PULL_UP, 1),
};

static unsigned long pin_highz_conf[] = {
     PIN_CONF_PACKED(PIN_CONFIG_BIAS_HIGH_IMPEDANCE, 0),
};

/* Pin control settings */
static struct pinctrl_map __initdata u300_pinmux_map[] = {
     /* anonymous maps for chip power and EMIFs */
     PIN_MAP_MUX_GROUP_HOG_DEFAULT("pinctrl-u300", NULL, "power"),
     PIN_MAP_MUX_GROUP_HOG_DEFAULT("pinctrl-u300", NULL, "emif0"),
     PIN_MAP_MUX_GROUP_HOG_DEFAULT("pinctrl-u300", NULL, "emif1"),
     /* per-device maps for MMC/SD, SPI and UART */
     PIN_MAP_MUX_GROUP_DEFAULT("mmci",  "pinctrl-u300", NULL, "mmc0"),
     PIN_MAP_MUX_GROUP_DEFAULT("pl022", "pinctrl-u300", NULL, "spi0"),
     PIN_MAP_MUX_GROUP_DEFAULT("uart0", "pinctrl-u300", NULL, "uart0"),
     /* This pin is used for clock return rather than GPIO */
     PIN_MAP_CONFIGS_PIN_DEFAULT("mmci", "pinctrl-u300", "PIO APP GPIO 11",
                         pin_pullup_conf),
     /* This pin is used for card detect */
     PIN_MAP_CONFIGS_PIN_DEFAULT("mmci", "pinctrl-u300", "PIO MS INS",
                         pin_highz_conf),
};

/* Initialize pinmuxing */
pinctrl_register_mappings(u300_pinmux_map, ARRAY_SIZE(u300_pinmux_map));
```

Questions?