

# Pin Control Sybsystem – Building Pins and GPIO from the ground up

**Linus Walleij**  
**Linaro Kernel Workgroup**  
**ST-Ericsson**

---



# Pins, Muxing and Controlling

Two subjects:

- Pin control hardware
- Affected kernel subsystem

Pin control is mainly two things:

- **Pin multiplexing** - Reusing the same pin for different purposes
- **Pin configuration** - Configuring electronic properties of pins

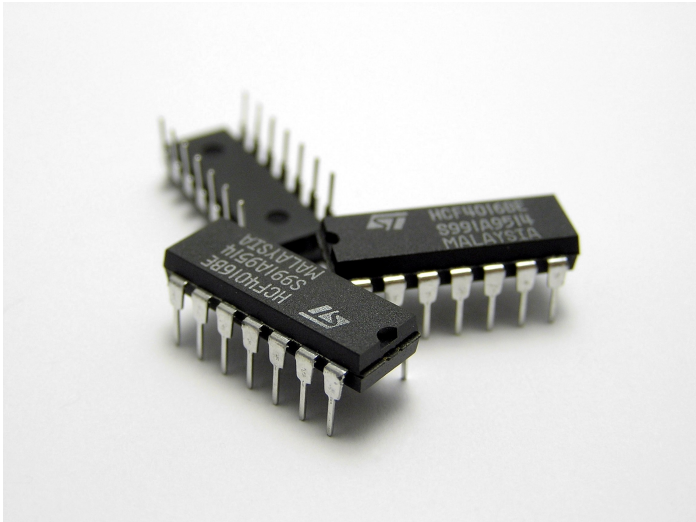
Related but separate concepts:

- GPIO input/output
- Interrupt generation on GPIO pins

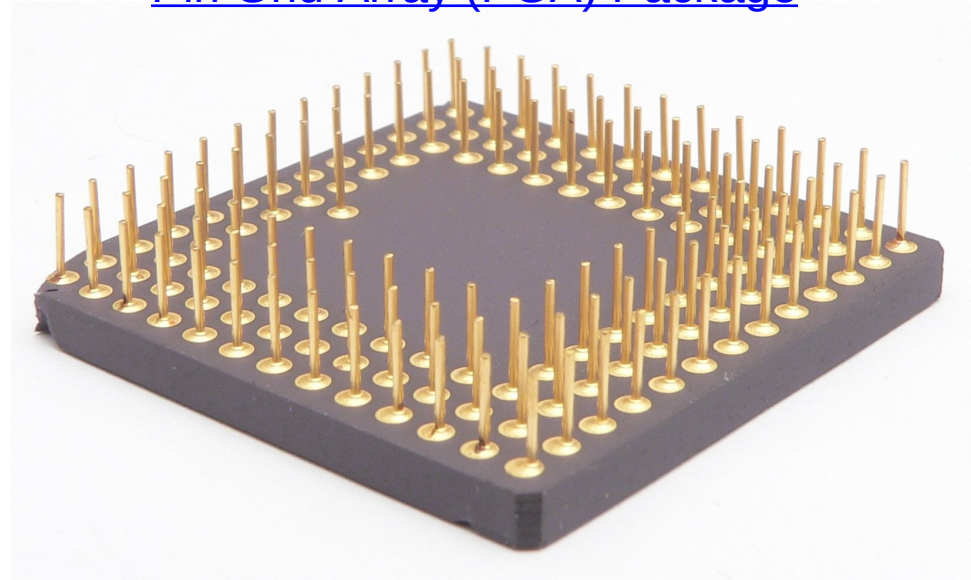
# What is a Pin from a Hardware Engineers Point of View?

- Pins are equivalent to balls found on a BGA etc.
- In an SoC pins are connected to a **pad** (also called finger) on a silicon die (chip)
- Each pad is driven by a specific piece of logic, called a **cell**. These cells come from a **cell library** with a name such as “My IO cells”
- Using a CAD tool, the hardware designer place pads around the edges of the chip, forming a **pad ring**
- Each cell has a number of discrete control signals, for example this simple bidirectional IO cell:
  - OEN - output enable
  - OUT - value of output
  - IN - value of input
  - PUEN - pull-up enable
  - PDEN - pull-down enable
- The cell signals may be hard-coded to certain states or connected to register maps that are controlled by software
- “My IO cell” is used for various things: busses, memories, etc, one specific use case is GPIO (general purpose I/O) which typically group e.g. 32 cells into a GPIO block

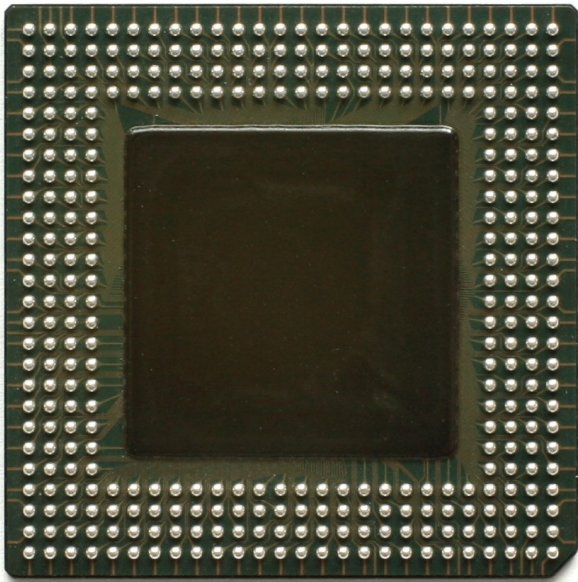
## Dual In-Line (DIL) Packages



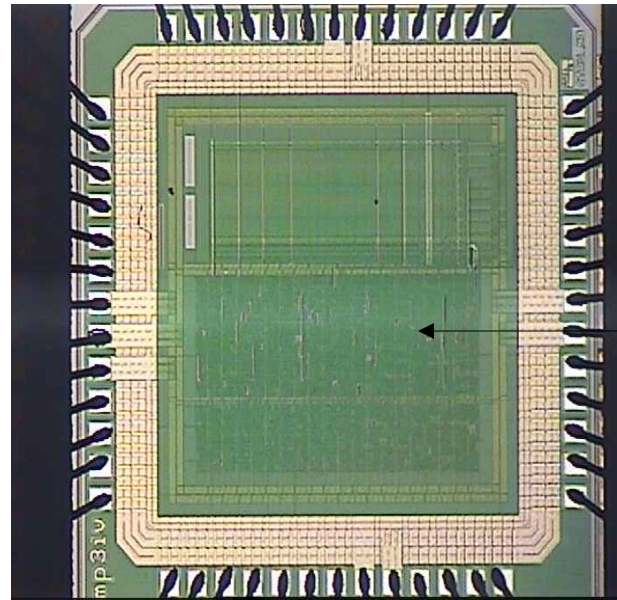
## Pin Grid Array (PGA) Package



## Ball Grid Array (BGA) Package



## Pad ring

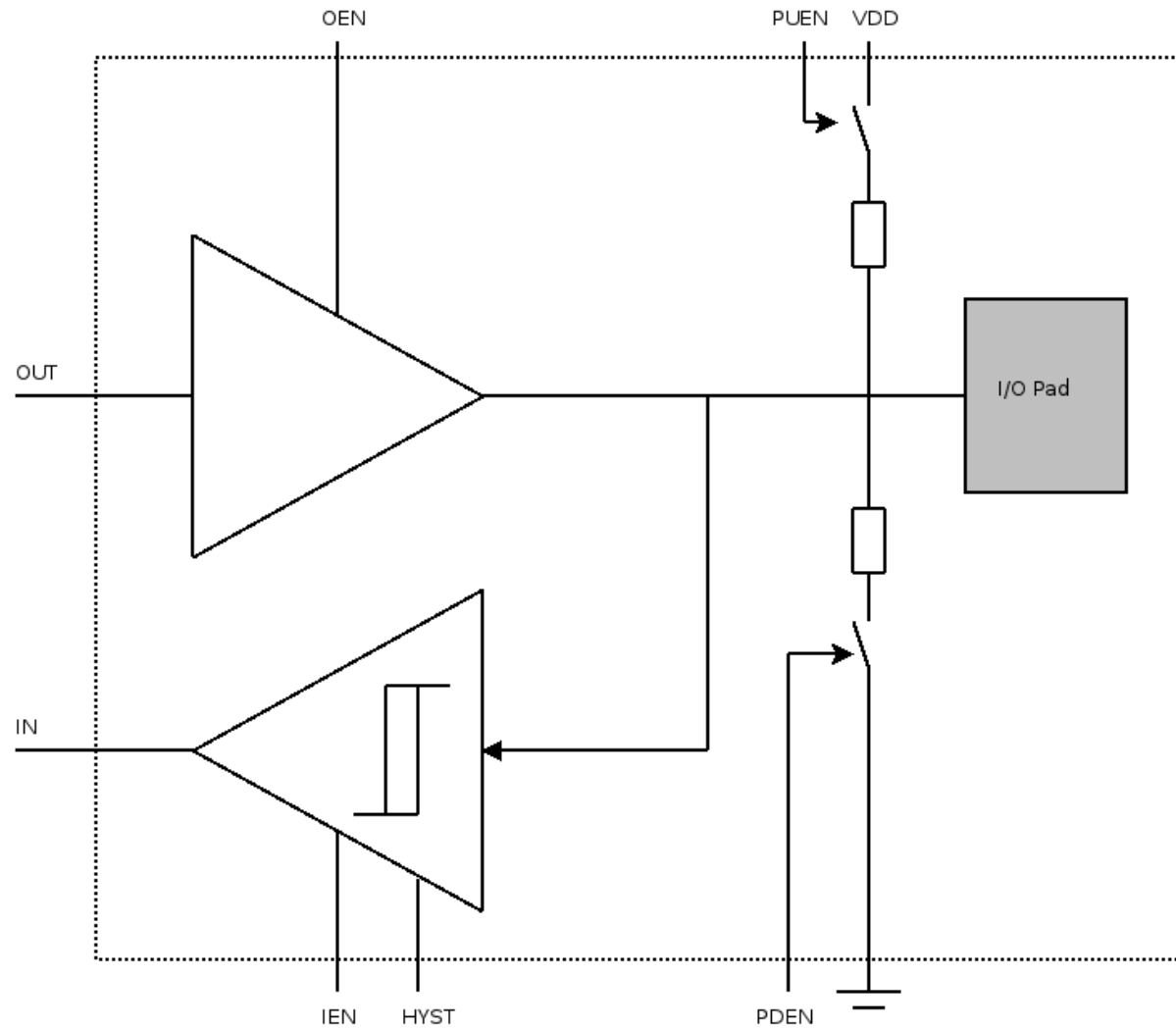


Pad with bonding wire

System on Chip (SoC)

Image sources: the linked Wikipedia entries

# The MyIO Cell from the Library "MyIO Cells"

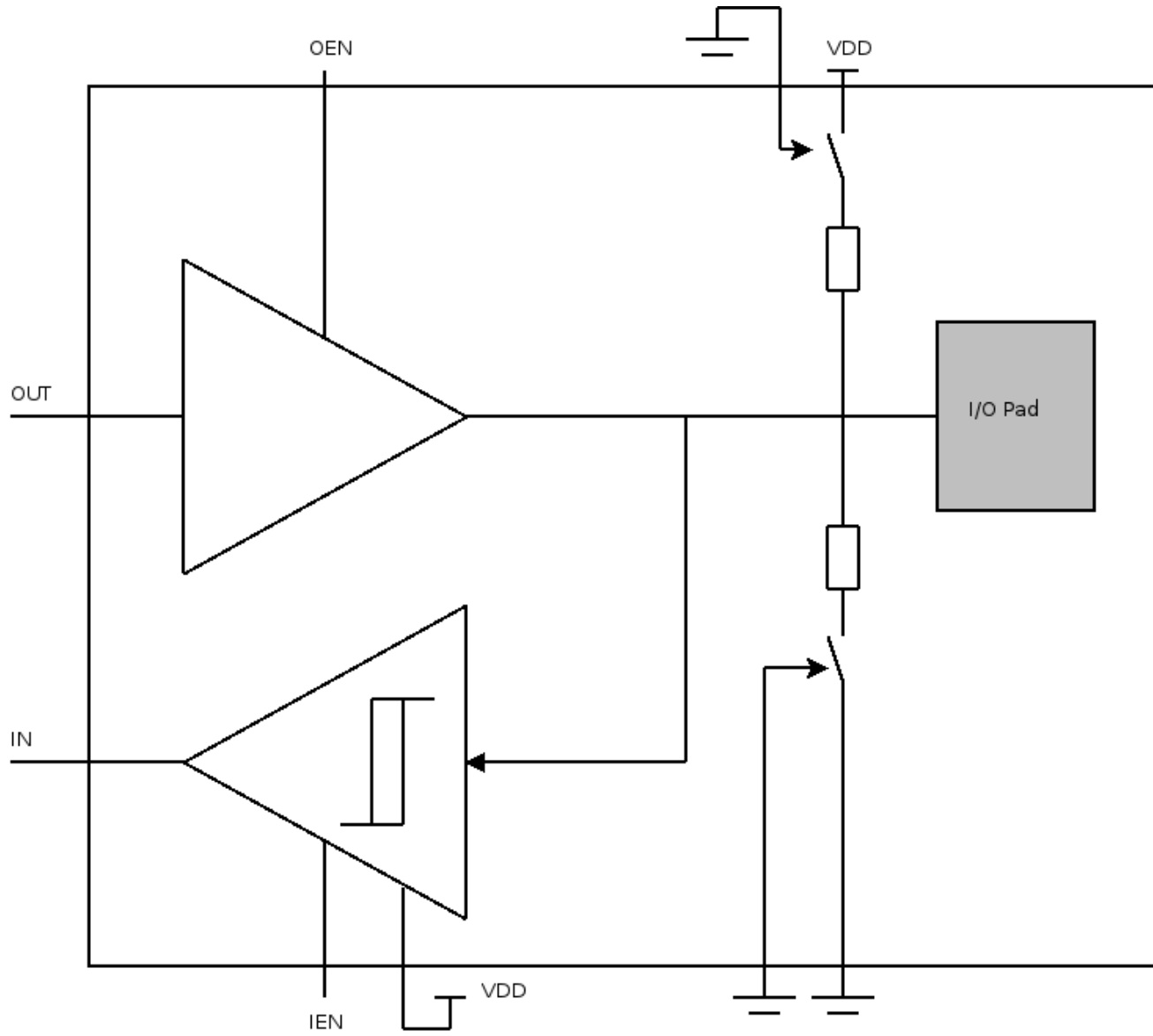


# Complex pin cells

- Pin properties does *not* have to be controlled by signals, so when you choose cell you may select a cell which is:
  - Only input or only output or bidirectional (as in the example)
  - Has a certain number of drive stages
  - Has a certain hard-wired pull-up or pull-down resistance
  - Even if things are controllable using signals, default values like biased to pull-up, pull-down or input (usually called tri-state, high impedance or High-Z)
- Another option is to wire the control signals to VDD or GND when synthesizing (VHDL or Verilog) thus forcing pull-up always on for example
- Pin cells may have **complex configurable** options that can each be selected at synthesis or at runtime using signals:
  - Drive strength, i.e. number of driver stages
  - Schmitt-trigger input (hysteresis)
  - Debounce period selector
  - Slew rate for rise, fall or both edges, reducing EMI (Electromagnetic Interference) from signals
  - Sleep mode – what happens when the cell get a special sleep signal

# How a GPIO block is formed from an IO cell

- A GPIO block is formed by taking a number of IO cells and lumping them into a block
- For example it is convenient to take 8 cells and create a 8 bit GPIO block
- **Control registers** for each cell such as “PUEN” are then typically grouped into a 32 bit register where you can set pull-up/pull-down by setting/clearing bit N in the register, OEN, OUT, IN, PUEN, PDEN are just passed through to one bit each in a register
- The GPIO logic then typically adds asynchronous **edge detection logic** for rising/falling/both edges when the pin is used as input and builds:
  - **IRQ signals** from edges and an **IRQ enable** bit per pin
  - **Wakeup signals** from edges and a **WAKEUP enable** bit per pin
  - Typically both of these are OR:ed into a single IRQ or WAKEUP signal out of the block



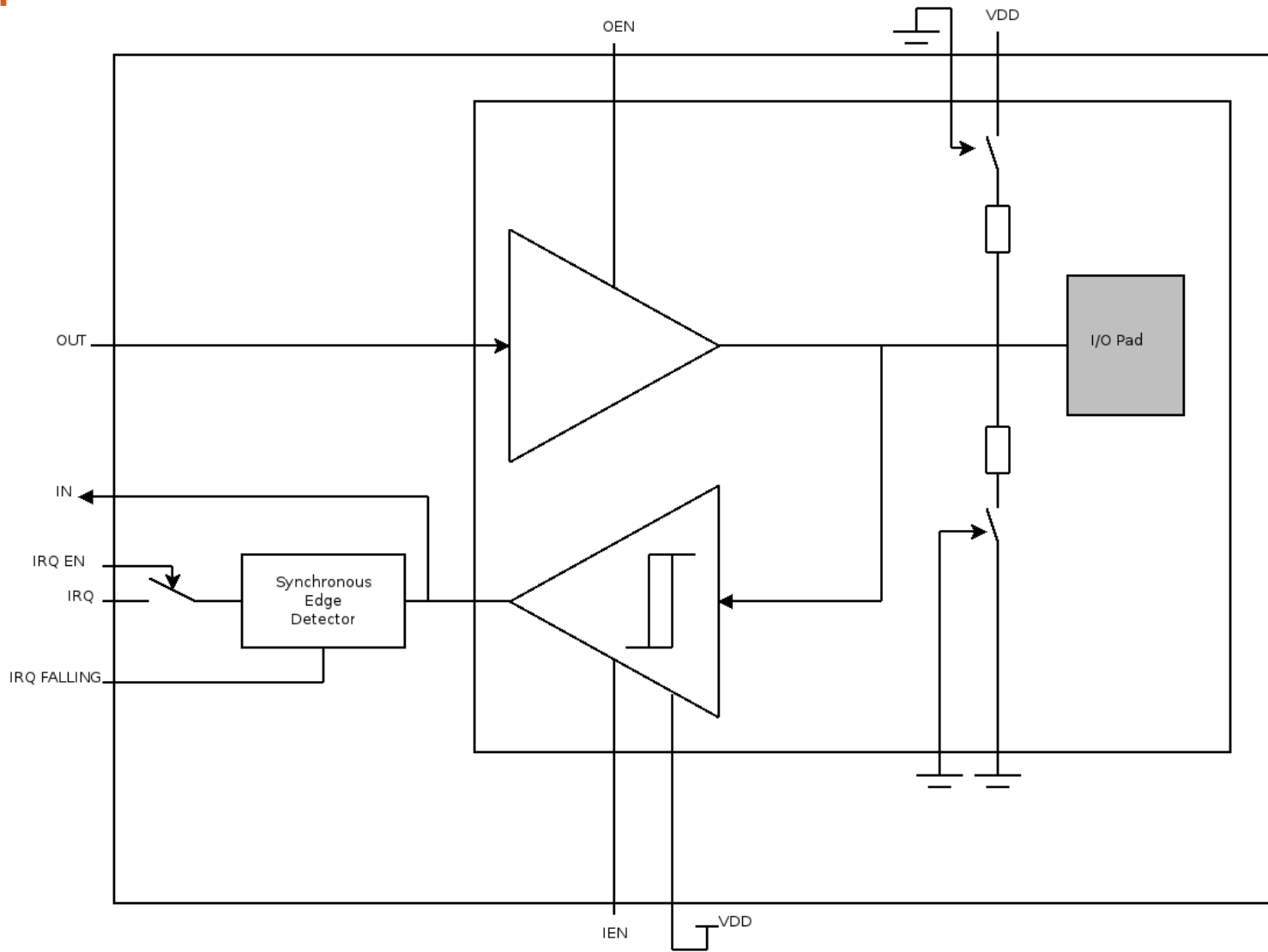


# Simple GPIO block register map after Ctrl+C Ctrl+V x8 of this cell for 8 pins

- 0x00 8 bits MY\_GPIO\_OEN
- 0x01 8 bits MY\_GPIO\_OUT
- 0x02 8 bits MY\_GPIO\_IEN
- 0x03 8 bits MY\_GPIO\_INVALID

The GPIO subsystem in drivers/gpio can handle this 100%

# Now we add IRQ generation on the GPIO pins



# Simple GPIO block register map after Ctrl+C Ctrl+V x8 of this cell for 8 pins

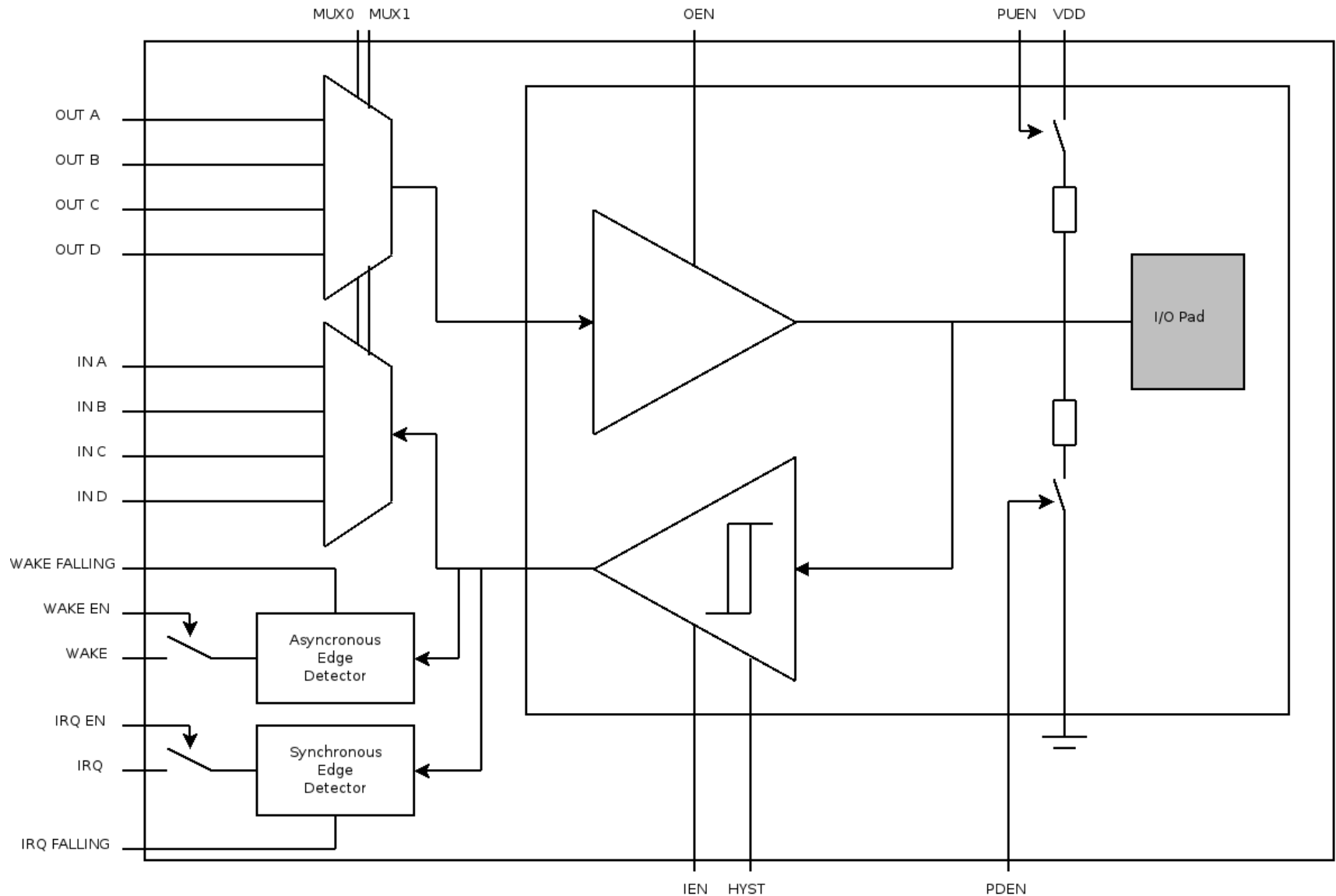
- 0x00 8 bits MY\_GPIO\_OEN
- 0x01 8 bits MY\_GPIO\_OUT
- 0x02 8 bits MY\_GPIO\_IEN
- 0x03 8 bits MY\_GPIO\_INVALID
- 0x04 8 bits MY\_GPIO\_IRQ\_EN
- 0x05 8 bits MY\_GPIO\_IRQ\_FALLING

The GPIO subsystem in drivers/gpio needs help from the IRQ core struct irq\_chip to handle this and register interrupts

# Multiplexing MyIO pins – not just GPIO

- Since we want to be able to use each pin also for other things than GPIO, we add pin multiplexing
- This is also known as “alternate functions”, “mission modes”, etc.
- This can be done in conjunction with individual GPIO control for each pin or separately, for pins that cannot also do GPIO
- Often the pin config attributes are kept – even if we reuse two GPIO pins for an I2C bus, I2C require that its clock (SCL) and data (SDA) lines be pulled-up, so the pin config logic enabling pull-ups can be reused. So basically only the input/output lines are multiplexed.
- Multiplexing two I/O cells on the same pin is an exotic option, this usually just replaces the electrical routing to the pad. When you cannot for example control drive strength other than when synthesizing your hardware, this approach may be chosen.

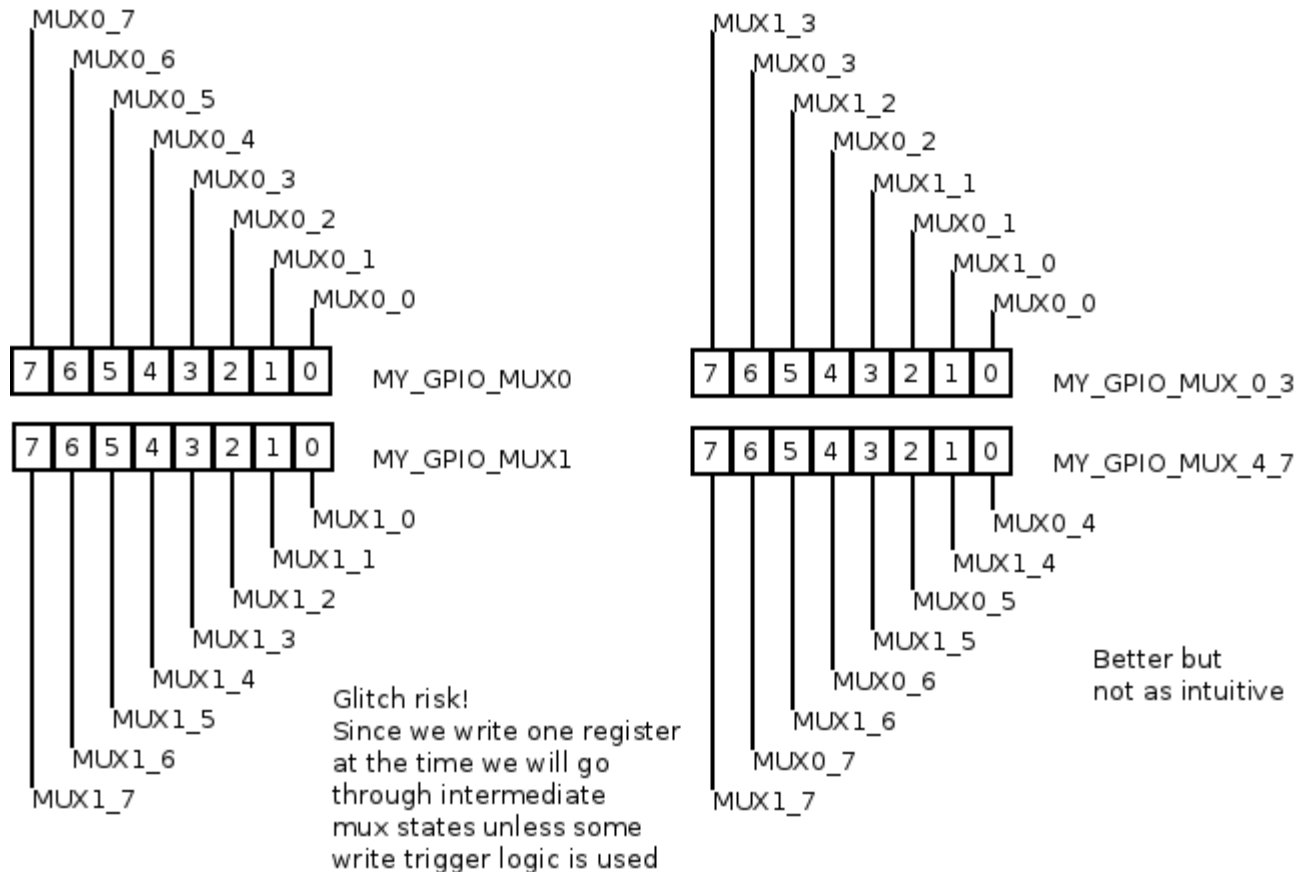
# MyIO GPIO Block



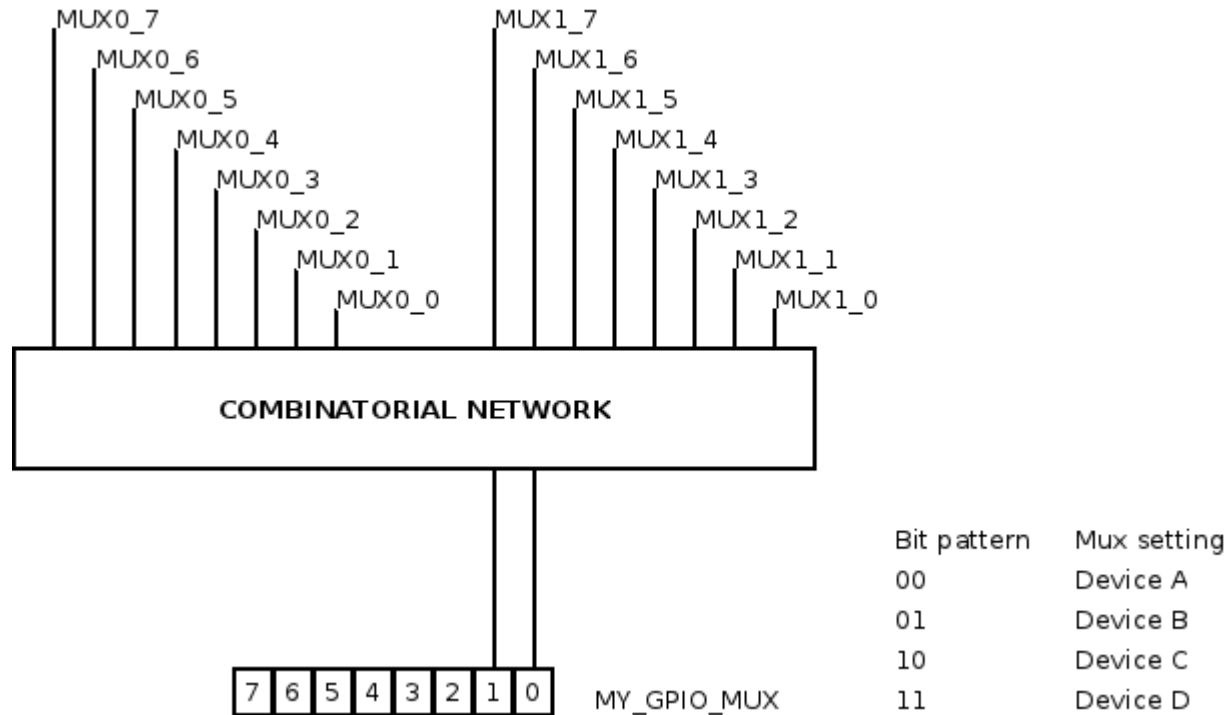
# GPIO block register map after Ctrl+C Ctrl+V x8 of this cell for 8 pins

- 0x00 8 bits MY\_GPIO\_OEN
- 0x01 8 bits MY\_GPIO\_OUT
- 0x02 8 bits MY\_GPIO\_IEN
- 0x03 8 bits MY\_GPIO\_INVALID
- 0x04 8 bits MY\_GPIO\_IRQ\_EN
- 0x05 8 bits MY\_GPIO\_IRQ\_FALLING
- 0x06 8 bits MY\_GPIO\_PUEN
- 0x07 8 bits MY\_GPIO\_PDEN
- 0x08 8 bits MY\_GPIO\_WAKE\_EN
- 0x09 8 bits MY\_GPIO\_WAKE\_FALLING
- 0x0A 8 bits MY\_GPIO\_HYST
- 0x0B 8 bits MY\_GPIO\_MUX0
- 0x0C 8 bits MY\_GPIO\_MUX1

# Different ways of connecting MUX0 and MUX1



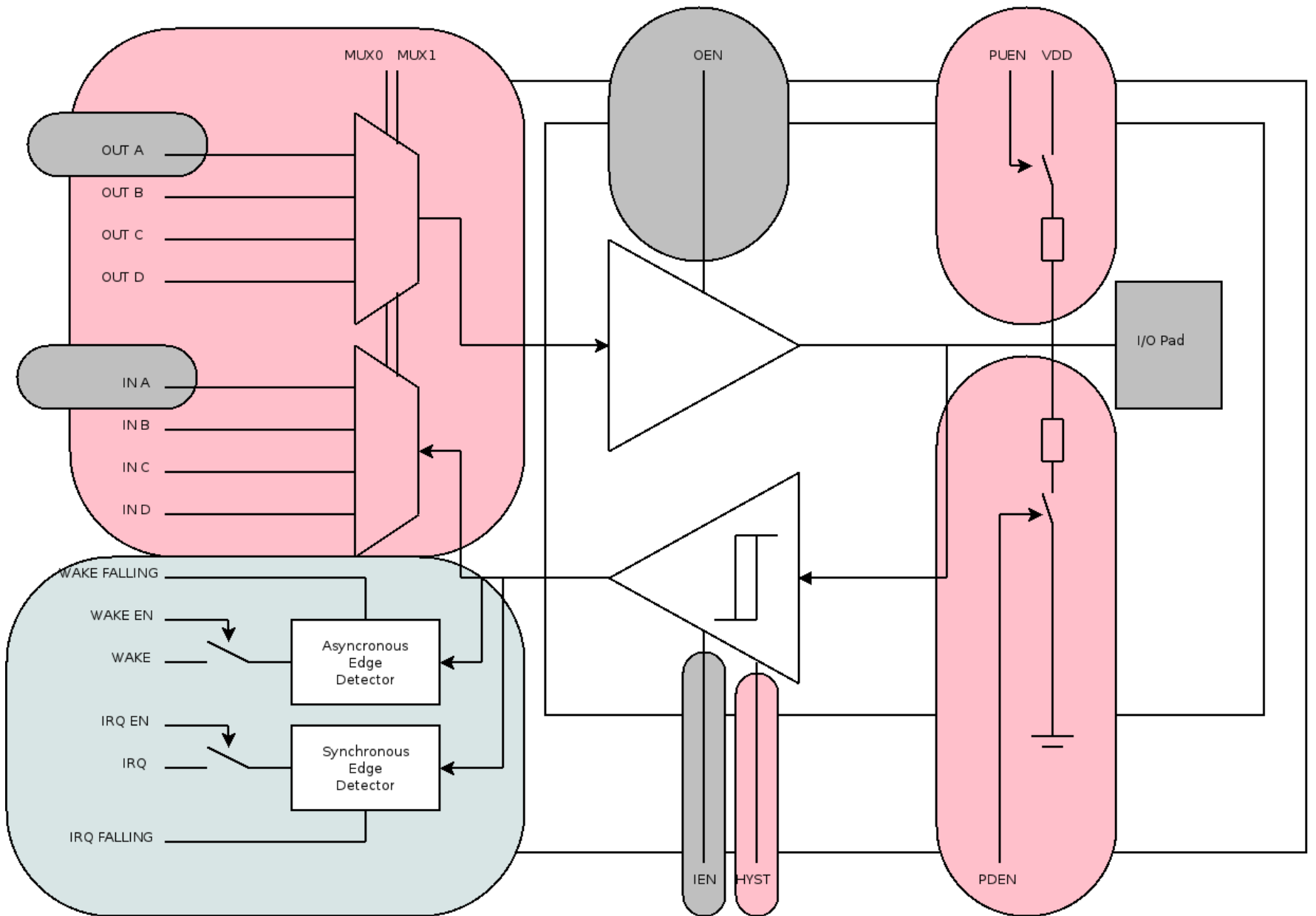
# Group mux control – now the ASIC engineer attempts to be helpful





# Several Involved Subsystems

- Now we just constructed a piece of hardware that needs to involve three different subsystems at once:
  - The GPIOlib subsystem handles the GPIO pin portions – setting pins to input or output (selecting direction) and driving them high or low, it also interacts with IRQ core to map a certain pin to a certain interrupt line (Grant Likely)
  - The IRQ core code (irqchips.h) handles the IRQ enable/disable, edge selection and wakeup enable/disable for the pin (Thomas Gleixner)
  - Pin control and pin multiplexing is the subject addressed by the pin control subsystem (Linus Walleij)



# GPIO subsystem driver interface “gpiolib”

## include/asm-generic/gpio.h

```
struct gpio_chip {
    const char      *label;
    struct device   *dev;
    struct module   *owner;
    int             (*request)(struct gpio_chip *chip,
                               unsigned offset);
    void            (*free)(struct gpio_chip *chip,
                            unsigned offset);
    int             (*direction_input)(struct gpio_chip *chip,
                                       unsigned offset);
    int             (*get)(struct gpio_chip *chip,
                           unsigned offset);
    int             (*direction_output)(struct gpio_chip *chip,
                                       unsigned offset, int value);
    int             (*set_debounce)(struct gpio_chip *chip,
                                    unsigned offset, unsigned debounce);
    void            (*set)(struct gpio_chip *chip,
                           unsigned offset, int value);
    int             (*to_irq)(struct gpio_chip *chip,
                              unsigned offset);
    void            (*dbg_show)(struct seq_file *s,
                                struct gpio_chip *chip);

    int             base;
    u16             ngpio;
    const char      *const *names;
    unsigned        can_sleep:1;
    unsigned        exported:1;
};
```

# struct irq\_chip IRQ subsystem interface

## include/linux/irq.h

```
struct irq_chip {
    const char *name;
    unsigned int (*irq_startup)(struct irq_data *data);
    void (*irq_shutdown)(struct irq_data *data);
    void (*irq_enable)(struct irq_data *data);
    void (*irq_disable)(struct irq_data *data);

    void (*irq_ack)(struct irq_data *data);
    void (*irq_mask)(struct irq_data *data);
    void (*irq_mask_ack)(struct irq_data *data);
    void (*irq_unmask)(struct irq_data *data);
    void (*irq_eoi)(struct irq_data *data);

    int (*irq_set_affinity)(struct irq_data *data, const struct cpumask *dest, bool force);
    int (*irq_retrigger)(struct irq_data *data);
    int (*irq_set_type)(struct irq_data *data, unsigned int flow_type);
    int (*irq_set_wake)(struct irq_data *data, unsigned int on);

    void (*irq_bus_lock)(struct irq_data *data);
    void (*irq_bus_sync_unlock)(struct irq_data *data);

    void (*irq_cpu_online)(struct irq_data *data);
    void (*irq_cpu_offline)(struct irq_data *data);

    void (*irq_suspend)(struct irq_data *data);
    void (*irq_resume)(struct irq_data *data);
    void (*irq_pm_shutdown)(struct irq_data *data);

    void (*irq_print_chip)(struct irq_data *data, struct seq_file *p);

    unsigned long flags;
};
```

# Pin configuration subsystem interface

## include/linux/pinctrl/pinctrl.h

```
struct pinctrl_desc {
    const char *name;
    struct pinctrl_pin_desc const *pins;
    unsigned int npins;
    struct pinctrl_ops *pctlops;
    struct pinmux_ops *pmxops;
    struct pinconf_ops *confops;
    struct module *owner;
};

struct pinctrl_ops {
    int (*list_groups) (struct pinctrl_dev *pctldev, unsigned selector);
    const char *(*get_group_name) (struct pinctrl_dev *pctldev,
                                   unsigned selector);
    int (*get_group_pins) (struct pinctrl_dev *pctldev,
                           unsigned selector,
                           const unsigned **pins,
                           unsigned *num_pins);
    void (*pin_dbg_show) (struct pinctrl_dev *pctldev, struct seq_file *s,
                          unsigned offset);
};
```

# Composite pin controllers

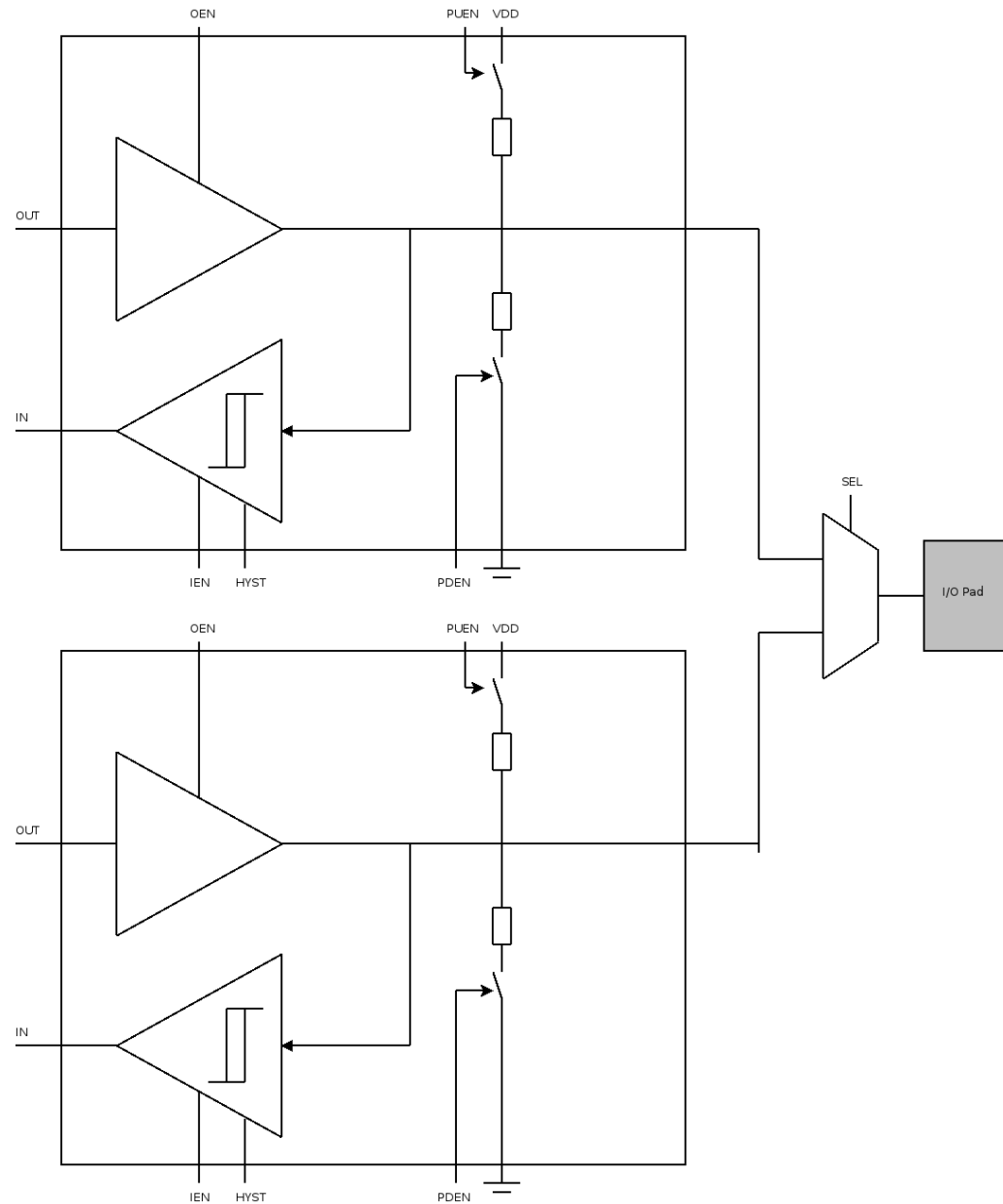
- If your GPIO/pin controller can only do simple GPIO, implement just `struct gpio_chip` in `drivers/gpio/gpio-foo.c` and keep it there. Please do not use generic GPIO and old-style GPIO.
- If your GPIO/pin controller can generate interrupts on top of the GPIO functionality, keep it in `drivers/gpio`, just fill in your `struct irq_chip` and register to the IRQ subsystem.
- If you have pin multiplexing, advanced pin driver strength, complex biasing etc, please put the composite pin controller drivers in `drivers/pinctrl/pinctrl-foo.c`
- Still implement all three interfaces: `struct gpio_chip`, `struct irq_chip`, `struct pinctrl_desc`

# Other common pin config subjects

- **Pin Biasing** - apart from pull up/pull down some controllers can ground pins or connect them to VDD.
- **Pin Driving** - pins may be driven push-pull (the most common mode, and illustrated) or as open source/open drain (MOS) or open collector/open emitter (TTL) - the latter may be exposed in the GPIO subsystem as well.
- **Pin Drive Strength** for output pins. The amplifier typically supports something like a fixed 2mA drive for a capacitive load, by shunting in several driver stages the pin can drive several loads.
- **Pin Debounce** - for keypads connected to GPIO lines this avoids multiple key presses for example. Also exposed in the GPIO subsystem.
- **Pin Slew Rate** - rise and fall time-control for the output signals - if these have too sharp edges they can cause EMI disturbances in neighbor circuitry.

The above appear as yet more control lines to the cells resulting in yet more bits in different control registers that need to be set up.

# Multiplexed pads with hard-coded I/O cells





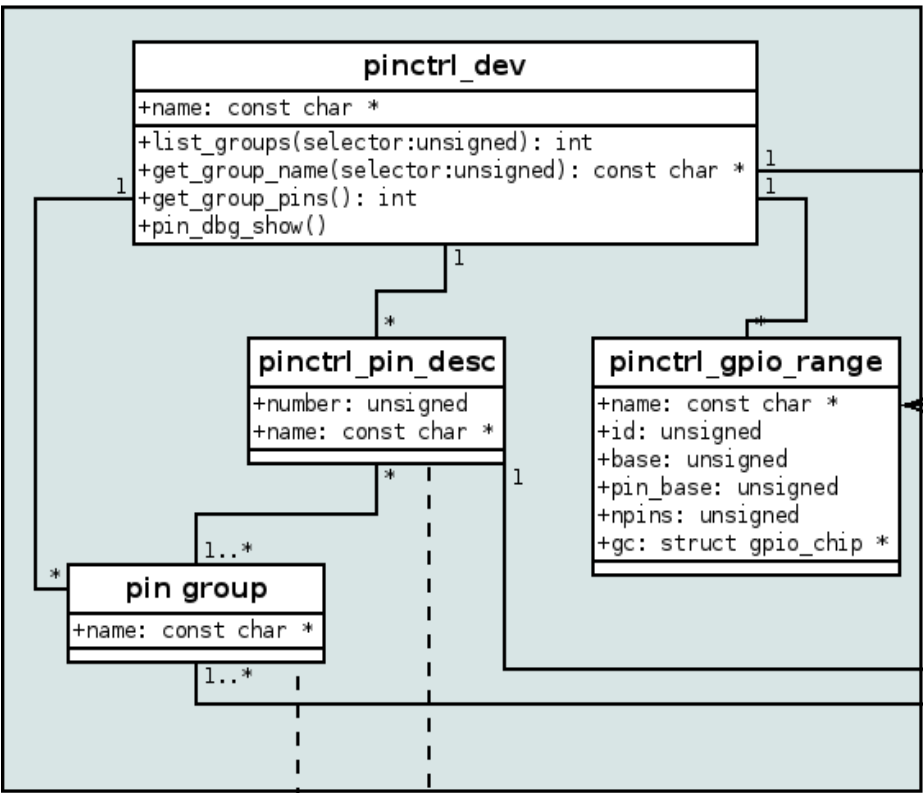
# Pin control has been part of the arch/arm/\* churn

- Pin control, especially pin multiplexing, was seen as one of the “necessarily different” properties of any one ARM SoC.
- Instances of ARM machines starting to copy each others' pin controller implementations hint at a hidden consolidation opportunity.
- In the U300 machine we had modeled our “pad multiplex” code after the regulator subsystem with get/enable/disable/put operations.
- The GPIO subsystem also has problems: several drivers start accumulating custom interfaces that are not part of gpiolib: `foo_set_pinmux()`, `foo_do_pullup()` etc.
- The pin control subsystem may be part of the solution

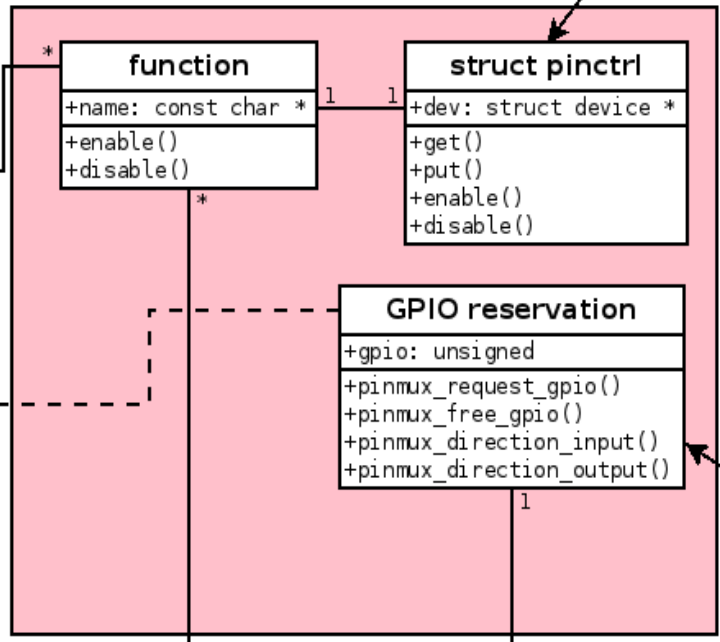
# The Pin Control Subsystem

- The first iteration of the Pin Control subsystem was basically a patch set I evolved from the U300 arch/arm/mach-u300/padmux.[c|h]
- Several people provided extensive feedback
- The tenth iteration of the patch set was eventually pulled into the mainline kernel for kernel 3.2 with support for U300
- Several bug fixes, a basic pin configuration interface and a new driver for Sirf Prima II was integrated into kernel 3.3
- For kernel 3.4 we will likely have atleast pin configuration states and a driver for the PXA/MMP platform which migrates that support from its current form in arch/arm
- Linaro have got involved in the pin control work and are working on board support for a range of contemporary high-end ARM devices.
- Something not-ARM would be very welcome so the generality of the subsystem can be established.

**Pinctrl core**



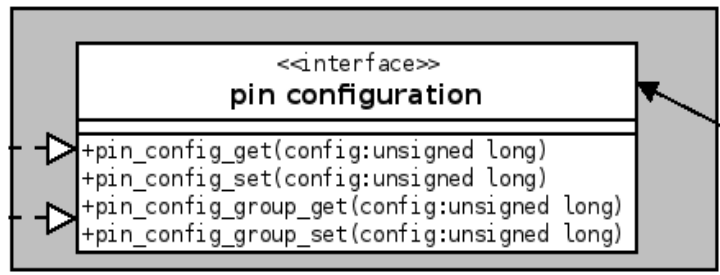
**Pinmux**



Devices can get pinctrl handles here

The GPIO subsystem can get a handle for reserving a GPIO pin here

**Pinconf**



The core board code or drivers can get to re-configure pins here

# The Pin Control Subsystem

- The subsystem lives in drivers/pinctrl/\*
- Documentation in Documentation/pinctrl.txt
- The subsystem will handle some sanity checks like assuring that a pin is not used for two functions at the same time.
- Drivers can select to implement the pin multiplexing interface or the pin configuration interface or both.
- Drivers can interact with the GPIO subsystem so that GPIO pins and pin control pins can be cross-referenced - GPIOs have a global number space.
- A pin controller is registered by filling in a struct pinctrl\_desc and registering it to the subsystem with pinctrl\_register()
- The boards/machines can register a number of pin multiplexing settings to be auto-activated on boot called pinmux hogs.
- Drivers can get/enable/disable/put mux settings at runtime akin to how they get/enable/disable/put clocks or regulators.

# Example: just a pin controller and its pins

```
#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");
}
```

# Example: pin groups

```
#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_list_groups(struct pinctrl_dev *pctldev,
    unsigned selector)
{
    if (selector >= ARRAY_SIZE(foo_groups))
        return -EINVAL;
    return 0;
}

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 = {
    .list_groups = foo_list_groups,
    .get_group_name = foo_get_group_name,
    .get_group_pins = foo_get_group_pins,
};

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

# Example: using pin control from a driver

```
foo_switch()
{
    struct pinctrl *p;

    /* Enable on position A */
    p = pinctrl_get(&device, "spi0-pos-A");
    if IS_ERR(p)
        return PTR_ERR(p);
    pinctrl_enable(p);

    /* This releases the pins again */
    pinctrl_disable(p);
    pinctrl_put(p);

    /* Enable on position B */
    p = pinctrl_get(&device, "spi0-pos-B");
    if IS_ERR(p)
        return PTR_ERR(p);
    pinmux_enable(p);
    ...
}
```

# Pin Control Subsystem – Next Steps

- Pin Config States – currently we only have some open-coded functions to set configuration for individual pins and groups, with config states you can supply initialization settings at boot time or per-device at runtime similar to how pin muxes can currently be handled. (Patch is in v2.)
- Device Tree Bindings – several ideas being discussed. The main issue is that the data collected in the kernel for pin controllers tend to grow quite rapidly and for platforms with large and complex pin configurations we need to get this out of the kernel. The main source of the problem is that ARM systems lack a BIOS (think ACPI, UEFI) so these services need to be open-coded into the kernel.
- Convert over as many in-tree pin configuration drivers as possible.
- One PCI driver in the works (Alessandro Rubini)



THANK YOU

