Pin Control and GPIO Update

Linus Walleij
Linaro Kernel Workgroup
ST-Ericsson

What are we talking about here?

GPIO subsystem provides:

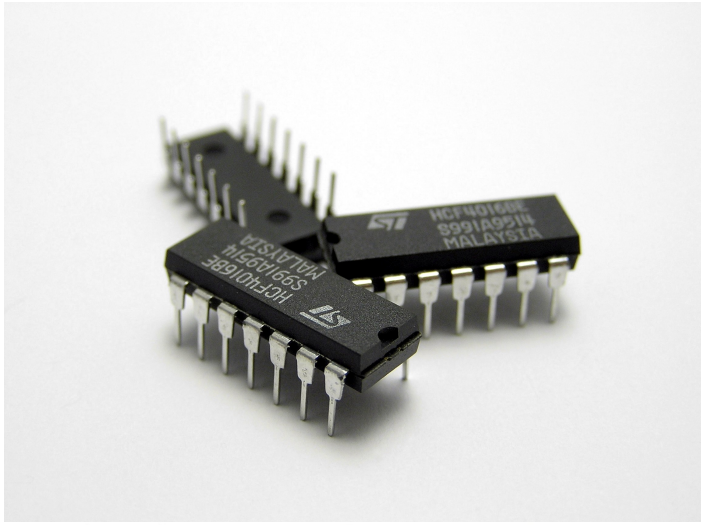
- Reads a **one-bit signal** as high or low – asserted or unasserted.
- Drives a line high or low – assert or de-assert - a one-bit signal.
- May (or may not) be tied in to per-pin IRQ generation, then uses the irqchip subsystem.
- Conceptually GPIOs have very few electronic properties, but can be specified as open drain/open source for e.g. wired-AND, and given a debounce period.

Pin control subsystem provides:

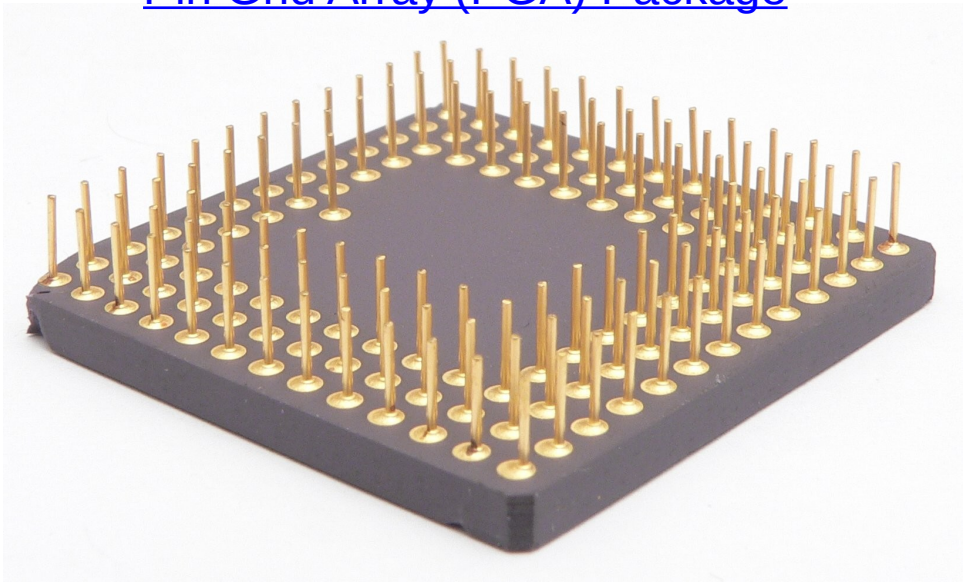
- **Pin multiplexing** – allows for reusing the same pin for different purposes, such as one pin being a UART TX pin, HSI data line or GPIO due to different multiplexing. Multiplexing can affect groups of pins or individual pins.
- **Pin configuration** - configuring electronic properties of pins such as pull-up, pull-down, driver strength etc.

ST ERICSSON

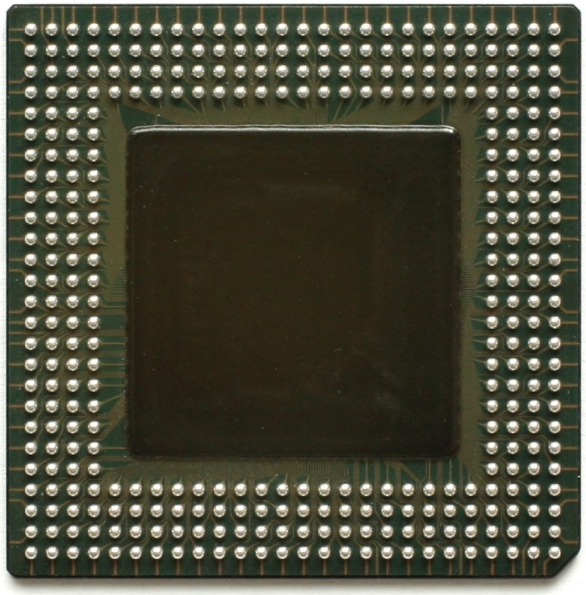
Dual In-Line (DIL) Packages



Pin Grid Array (PGA) Package



Ball Grid Array (BGA) Package



Pad ring

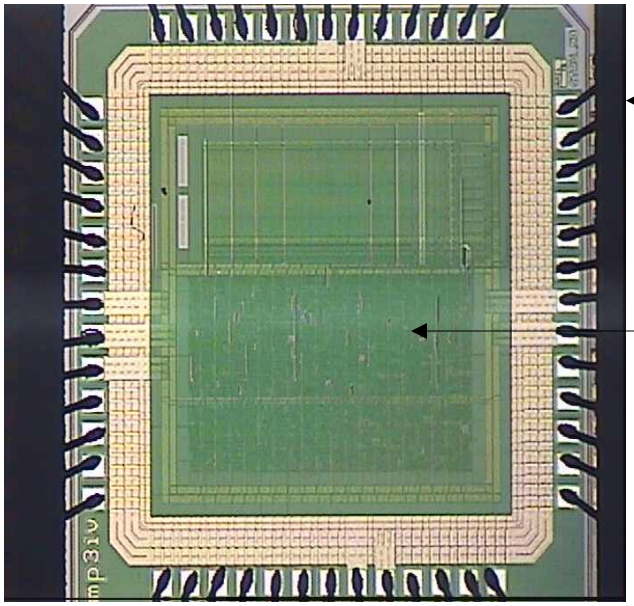
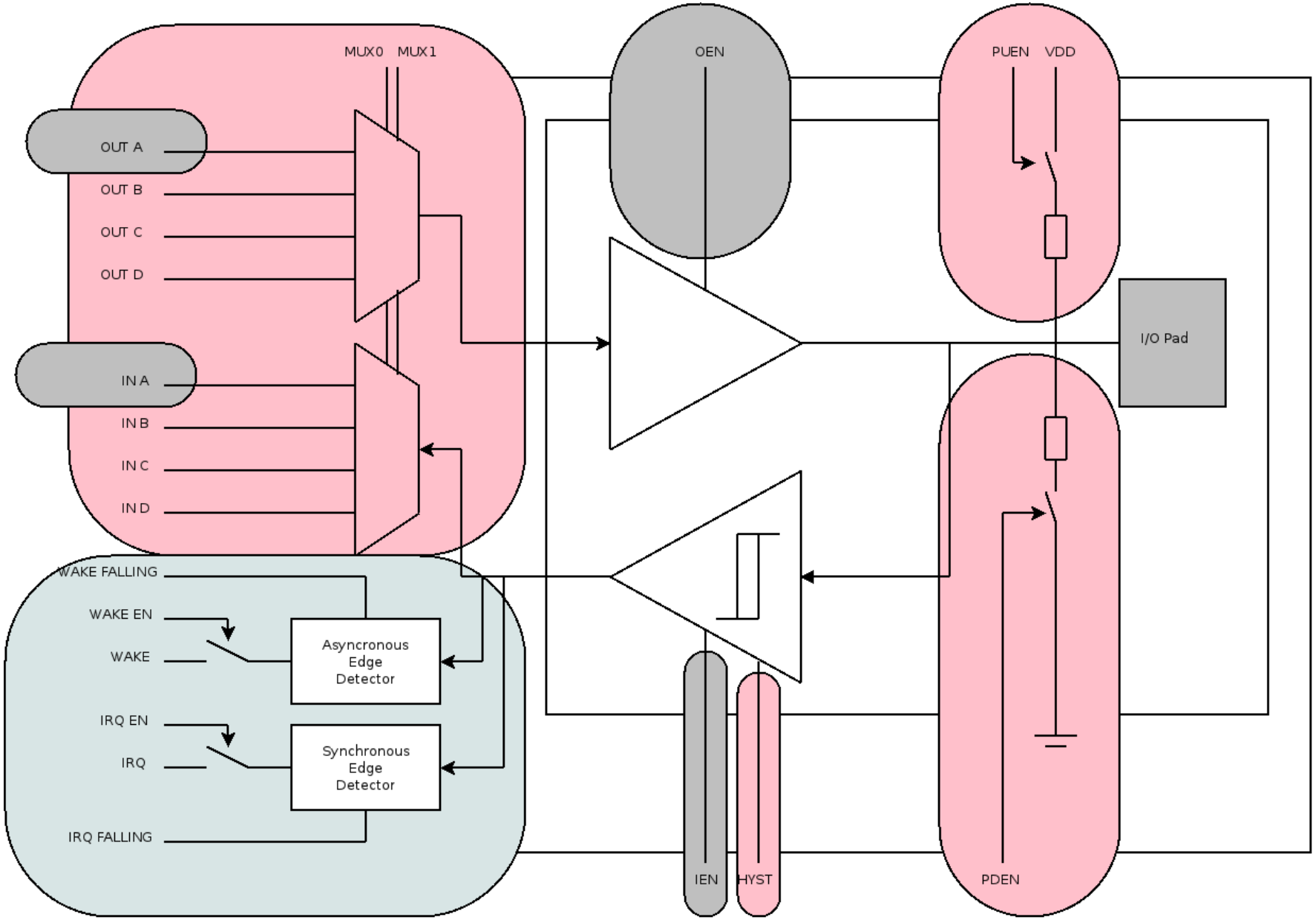


Image sources: the linked Wikipedia entries

ST ERICSSON


MUX0
MUX1
OEN
PUEN
VDD
OUT A
OUT B
OUT C
OUT D
IN A
IN B
IN C
IN D
I/O Pad
WAKE FALLING
WAKE EN
WAKE
Asyncronous Edge Detector
IRQ EN
IRQ
Synchronous Edge Detector
IRQ FALLING
IEN
HYST
PDEN

The GPIO Subsystem

- The subsystem lives in drivers/gpio/*
- Documentation in Documentation/gpio.txt
- All GPIO lines are defined in a linear, typically non-sparse numberspace [0...N]
- Consumers request GPIO lines with [devm_]gpio_request(gpio, label);
- Lines can then be set as input or output
- Inputs can be read as high or low (returning 1 or 0)
- Outputs can be driven high or low (represented by 1 or 0)
- GPIOs can be mapped to Linux IRQs. From this point the irqchip subsystem takes over (determining trig edge etc), but many GPIO drivers also register an irqchip, so you will often see then in the code.
- GPIOs can be exported to userspace via sysfs

GPIO sybsystem 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;
};
```

ST ERICSSON

GPIO – irqdomain refactoring

- The irqdomain <linux/irqdomain.h> was introduced in july 2011 to translate hardware IRQs to Linux IRQs
- With the number of GPIO controllers with IRQ capability rising, the number of irq_chip:s in systems are rising
- This leads to various “interesting” #define hacks to keep track of the global IRQ number space, akin to how the global GPIO number space is managed.
- The basic idea is to make the driver only deal with the hwirq, the IRQ flag offset from zero that the hardware actually produces, then let irqdomain translate that into a Linux IRQ
- In the longer run, Linux IRQ numbers are not necessary. Currently all IRQs have a number, but could just as well be just descriptors.
- The last 2 years we have converted a large number of IRQ-capable GPIO drivers to use the irqdomain, splitting responsibilities between drivers and the irq subsystem.

ST ERICSSON

GPIO – descriptor refactoring

- Has been on my drawing board for a while
- Currently driven by Alexandre Courbot after he observed that we have no gpio_get() function referencing the consuming device: compare clk_get(), regulator_get(), pinctrl_get()...
- Motivation: get rid of the global GPIO numberspace, be more abstract enitities connected to consumers, such as the clocks, regulators or pins
- Motivation: the IRQ numberspace also sucks.
- Just like in the case with IRQs, first transition the core code to use descriptors internally, then expose the new descriptor APIs and modify clients.
- Provide fallbacks to map GPIOs to descriptors and vice versa, akin to irqdomain.
- Some patches merged for v3.9, work will continue.
- Open debate: shall gpiod_get() return a IS_ERR(pointer) like its cousins in other subsystems?

ST ERICSSON

GPIO – blocked GPIO requests

- Basic problem: several GPIO lines need to toggle values at the same time, such as a data and clock line. Alternatively you need to read several GPIO lines with infinitesimal delay between each line.
- Since several GPIO lines are often handily placed in the same register, it is often actually possible to read or write 8, 16 or 32 GPIO lines at the same time. By attaching hardware to lines in the same register, the problem can be solved.
- Has existed in some out-of-tree hacks/forks for a while
- Currently driven by Roland Stigge
- Mostly OK for kernel-internal interface, sketicism around the sysfs API/ABI due to perpetual maintenance requirements.

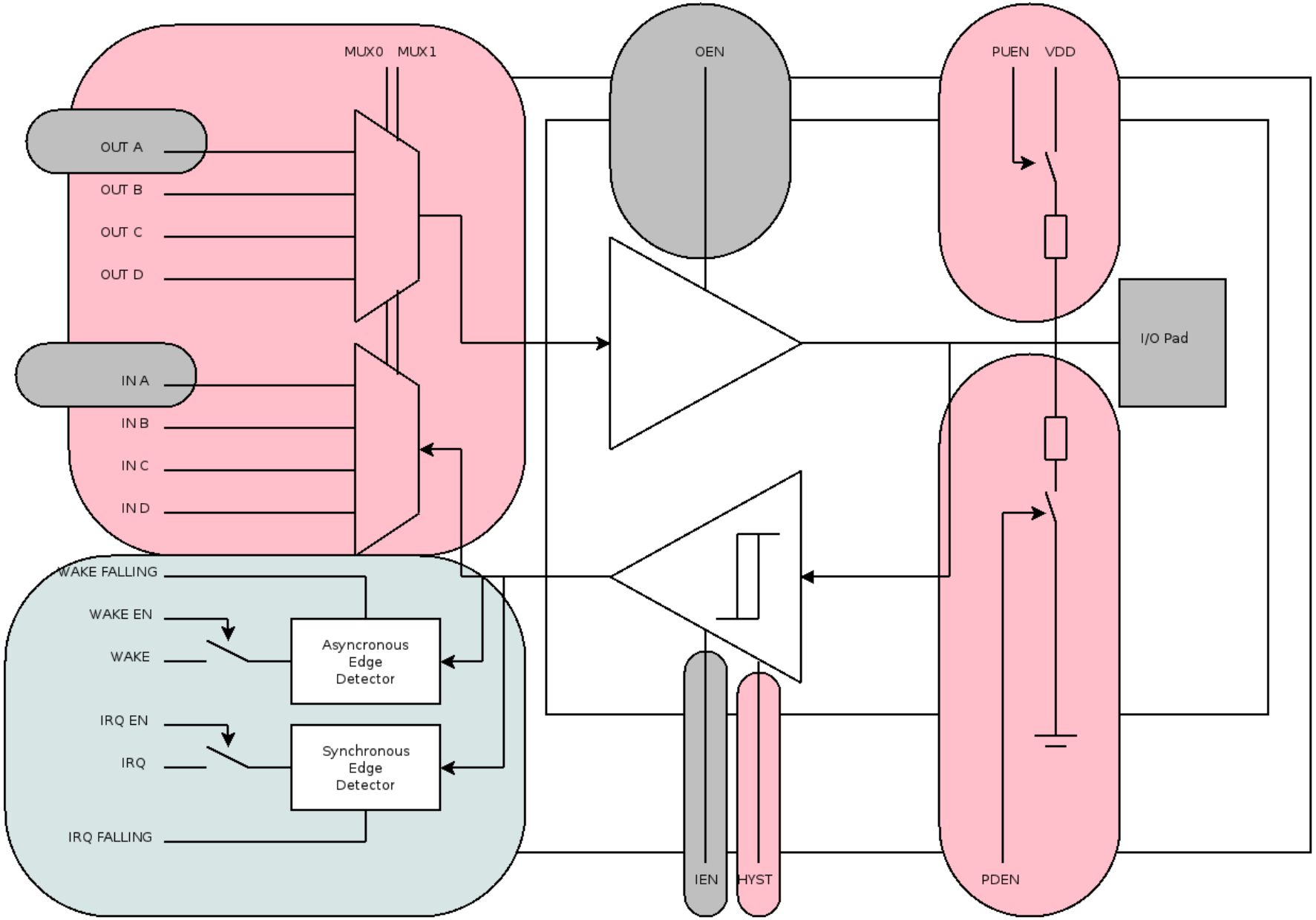
GPIO – next steps

- Install the descriptor API and move consumers over to using the descriptor API
- Make sysfs kobjects represent reality, tie GPIO into the device core properly.
- Evaluate the future of the userspace API/ABI
- Can we simplify interaction with the irqchip subsystem?
- Can we simplify interaction with the pinctrl subsystem?
- GPIO hogs?

ST ERICSSON

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.


MUX0
MUX1
OEN
PUEN
VDD
OUT A
OUT B
OUT C
OUT D
IN A
IN B
IN C
IN D
I/O Pad
WAKE FALLING
WAKE EN
WAKE
Asyncronous Edge Detector
IRQ EN
IRQ
Synchronous Edge Detector
IRQ FALLING
IEN
HYST
PDEN

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);
};
```

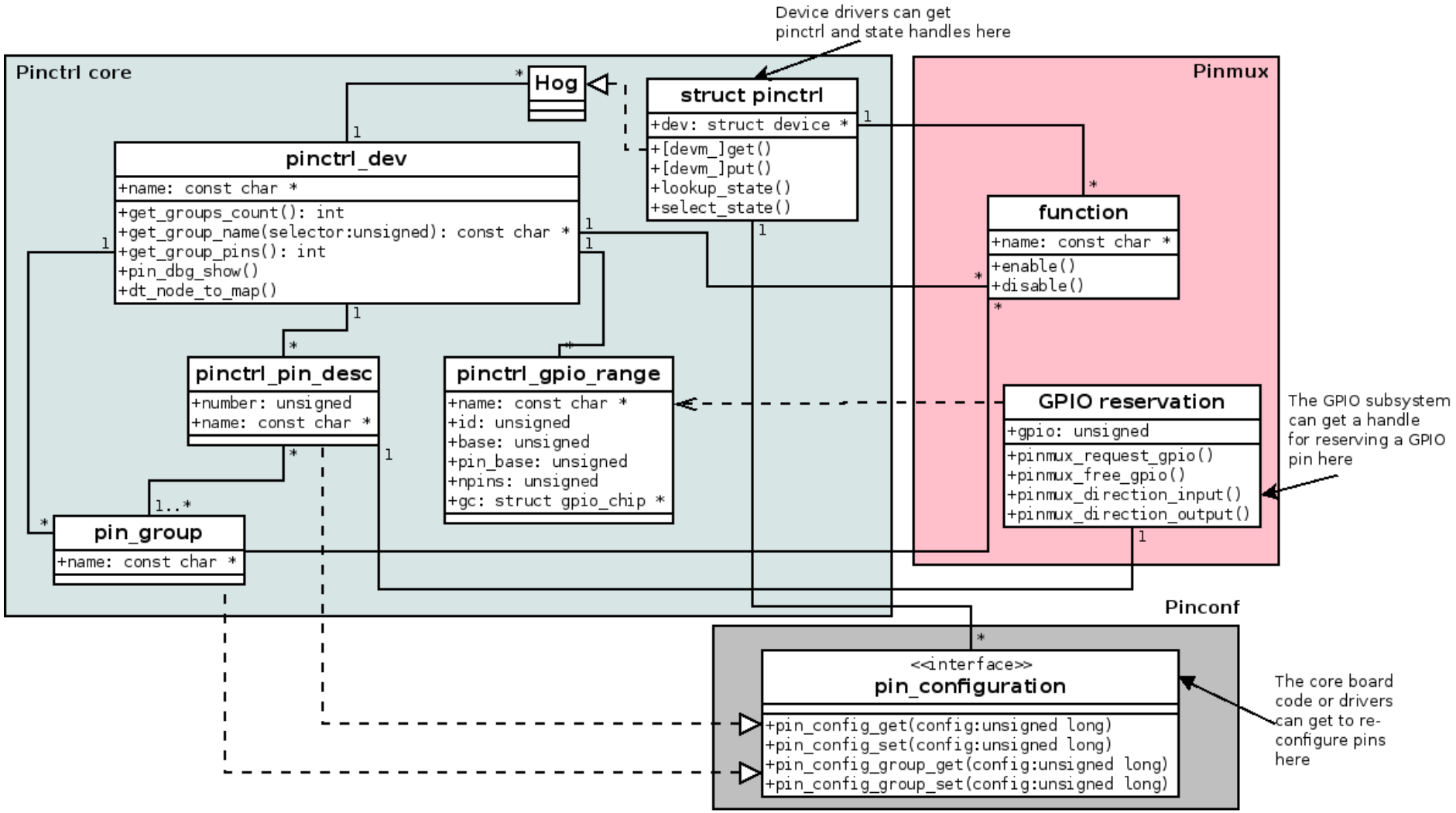
ST ERICSSON

The Pin Control Subsystem History

- Designed to counter the churn in the ARM arch/arm/* architecture. Too many “necessarily different” pin control schemes were running amok in the arch.
- Grant would not let med export gpio_to_chip() in the autumn of 2011. (Ironically it is exported now.)
- 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, notably Stephen Warren from nVidia.
- The tenth iteration of the patch set was eventually pulled into the mainline kernel for v3.2 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
- Kernel 3.4 introduced pin configuration states, and the API that expose a single struct pinctrl * handle to consumers, and define a separate struct pinctrl_state * container to hold a certain state for a consumer.

ST ERICSSON

UML makes everything so much clearer



Who has moved in?

- ST-Ericsson: U300, Nomadik, and NovaThor (Ux500), ABx500 (v3.9)
- CSR: Sirf Prima II
- Nvidia: Tegra 20, 30, 114 (v3.9)
- Intel/Marvell: PXA 168, 910, MMP2
- Marvell: Armada, Kirkwood, Dove
- Freescale: i.MX 23, 28, 35, 51, 53, 6q
- STMicroelectronics: SPEAr
- Atmel: AT91
- Samsung: Exynos 5440
- Broadcom BCM2835
- pinctrl-single: abstract variant intended for OMAPs and HiSilicon
- MIPS Lantiq: Falcon, xway
- Super-H: all variants, also ARM shmobile (v3.9)

Prospects: Samsung S3P, OMAP variants, DaVinci, EP93xx, mach-imx/iomux*, MSM, ..

ST ERICSSON

Pin Control Subsystem – Next Steps

- Grab default handles from the device core (v3.9)
- Possible to handle sleeping hogs (v3.9)
- One PCI driver in the works (Alessandro Rubini)
- Make pinctrl and GPIO and irqchip live closer together?
- Expand and encourage generic pinconfig?
- Propose standardized DT bindings using generic pinconfig for future platforms?
- Create a pin controller with an additional set of operations basically providing all of a struct gpio_chip, and expose a gpio_chip?
- Runtime PM helper inlines.
- Will people need a userspace API/ABI?

ST ERICSSON

THANK YOU