

---

## The GPIO subsystem (uC)

# The GPIO pins

---

## All microcontrollers have GPIO pins

- GPIO means "General Purpose Input/Output"
- Some manufacturers call them just PIO or Parallel I/O
- Sometimes their are called I/O pins
- .... or other strange names ...

## They are usually divided in groups, called "ports"

- Ports are of varying width
  - ♦ AVR has 8-bit ports (it's an 8-bit CPU)
  - ♦ Some Cortex-M implementations offer 12-bit ports
  - ♦ Most have 16-bit or 32-bit ports
- Some bit may be "not present"
  - ♦ Sometimes ports may have as few as 0 useful bits
  - ♦ Fortunately, vendors use proper "sparse" names

## If you use the concept of ports, they **\*must\*** be 32b wide

- We have so many numbers in one integer
- Portability is paramount

# Alternate Functions

---

## **Most GPIO pins feature alternate functions**

- Each pin has one or several predefined uses
  - ◆ PWM, UART, SPI, I2C, ...
- Most of the device pins are managed by the GPIO subsystem
- Usually, only power and high-speed signals (e.g. USB) live on dedicated pins

## **When using GPIO, programs should be portable**

- We need an API that always works, with any CPU family
- The same program should build and work everywhere
- The applicatin code (and data) should never refer to CPU specifics

## **When using alternate functions, it's a device-specific issue**

- The driver (UART, SPI, whatever) is machine specific,
  - ◆ It must know the device registers
  - ◆ It can know the device pins (and AF) too
- Still, we want a consistent API for GPIO use

## **When offering and alternate function API, GPIO must be 0**

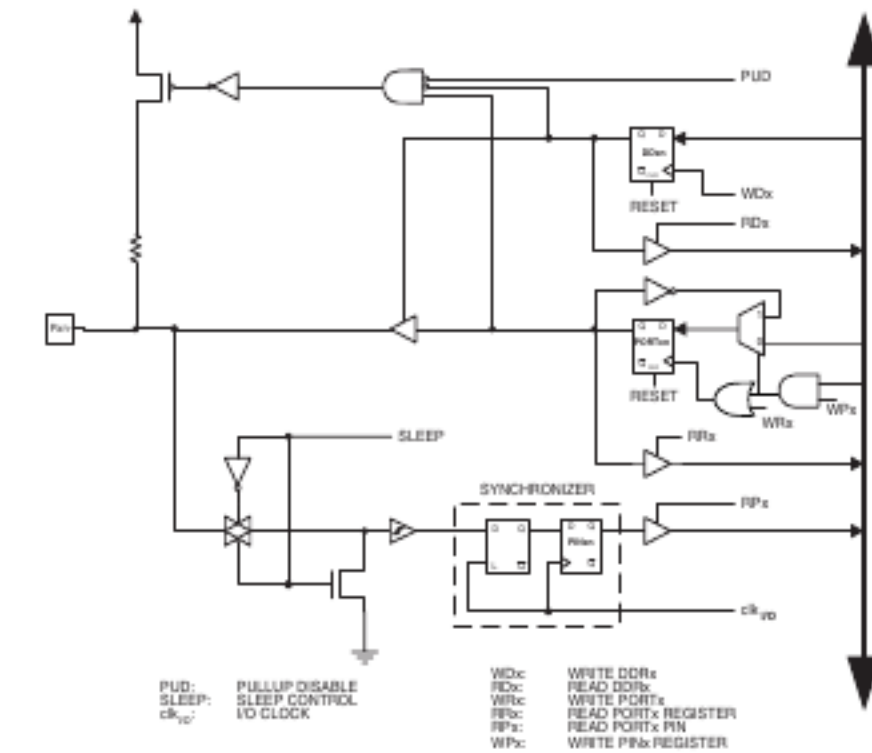
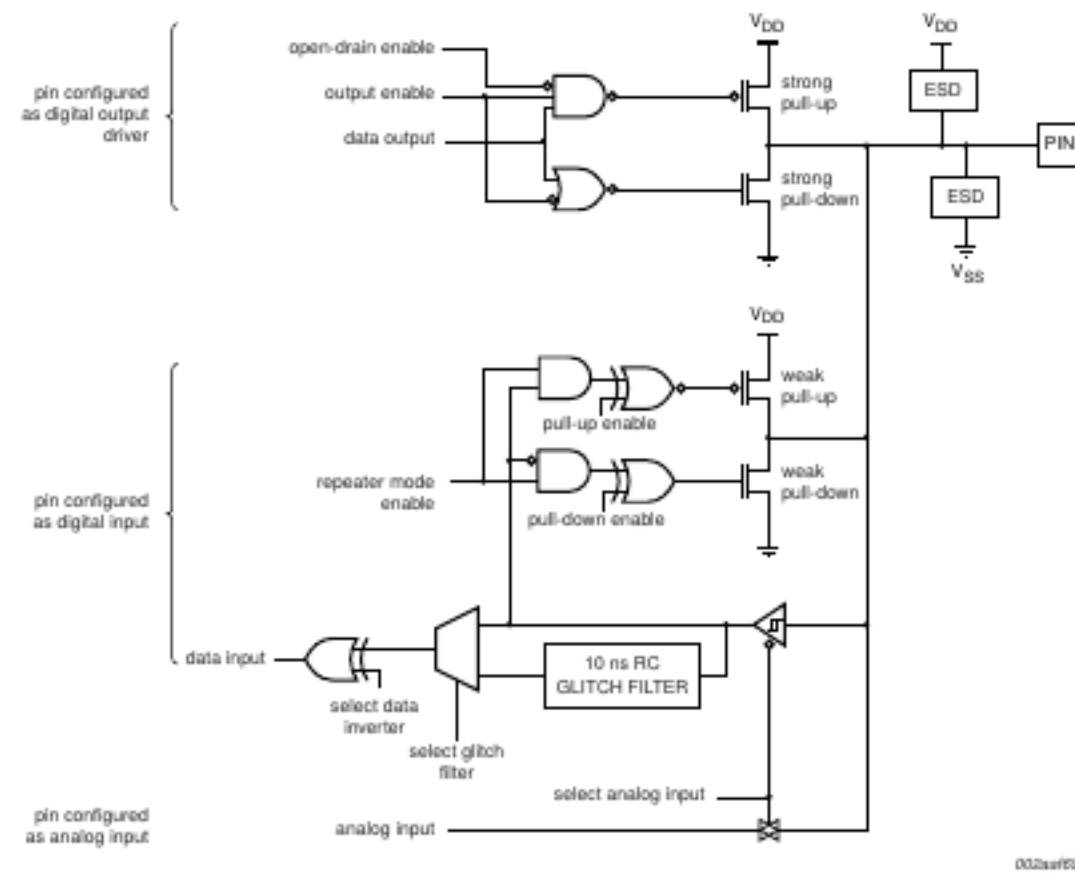
- Portability is paramount

# Electrically

## Every user manual describes the electrical GPIO

- All of them have input and output modes
- Some can feature pull-up and/or pull-down
- Some can have an open-drain mode

## Examples (LPC11 and ATmega):



Still, we want a unified API that covers all of them.

# But GPIO pins are everywhere

---

## **Microcontroller ports are only part of the game**

- You have GPIO extenders over I2C
- Most peripheral chips offer some GPIO pin
- You can have a remote controller, behind pci/usb/whatever

## **We need a flexible API that can be extended over time**

- Really, you are expected to think about it, and we may see it later
- 

## **Vendors solutions are not "usually" up to the task**

- They only offer register names
  - ♦ "IOSET1 = n"
- Or they offer structures
  - ♦ "GPIOC->IDR"
- I'm ready to apologize if you show me good vendor code

## **The Linux approach grew too complex over time**

- If features dynamic allocation of GPIO numbers: it can't work with microcontrollers

# So, this is the API we are going to use

---

## No specific header to include

- The gpio header is included by default by cpu.h

```
GPIO_NR(port, bit)
```

```
GPIO_PORT(nr)
```

```
GPIO_BIT(nr)
```

```
extern void gpio_init(void);
```

```
extern int gpio_dir_af(int gpio, int output, int value, int afnum);
```

```
extern void gpio_dir(int gpio, int output, int value);
```

```
extern int gpio_get(int gpio);
```

```
extern uint32_t __gpio_get(int gpio);
```

```
extern void gpio_set(int gpio, int value);
```

```
extern void __gpio_set(int gpio, uint32_t value);
```

## Then, there are constants to help the caller

- GPIO\_DIR\_IN, GPIO\_DIR\_OUT, ...

## Initialization can be slow (who cares)

## Runtime may need to be fast

- Sometimes, the program may directly act on registers -- API suggestions?

# And now the homework

---

**Please read my headers and C files (include/gpio\* and lib/gpio\*)**

- Understand what they do
- Learn from what is good
- Complain about what is bad

**Try to use the API to turn on and off leds**

**Suggest changes to the API to make it better**