NSY107
Cours 2 Les entrées/sorties génériques (GPIO)

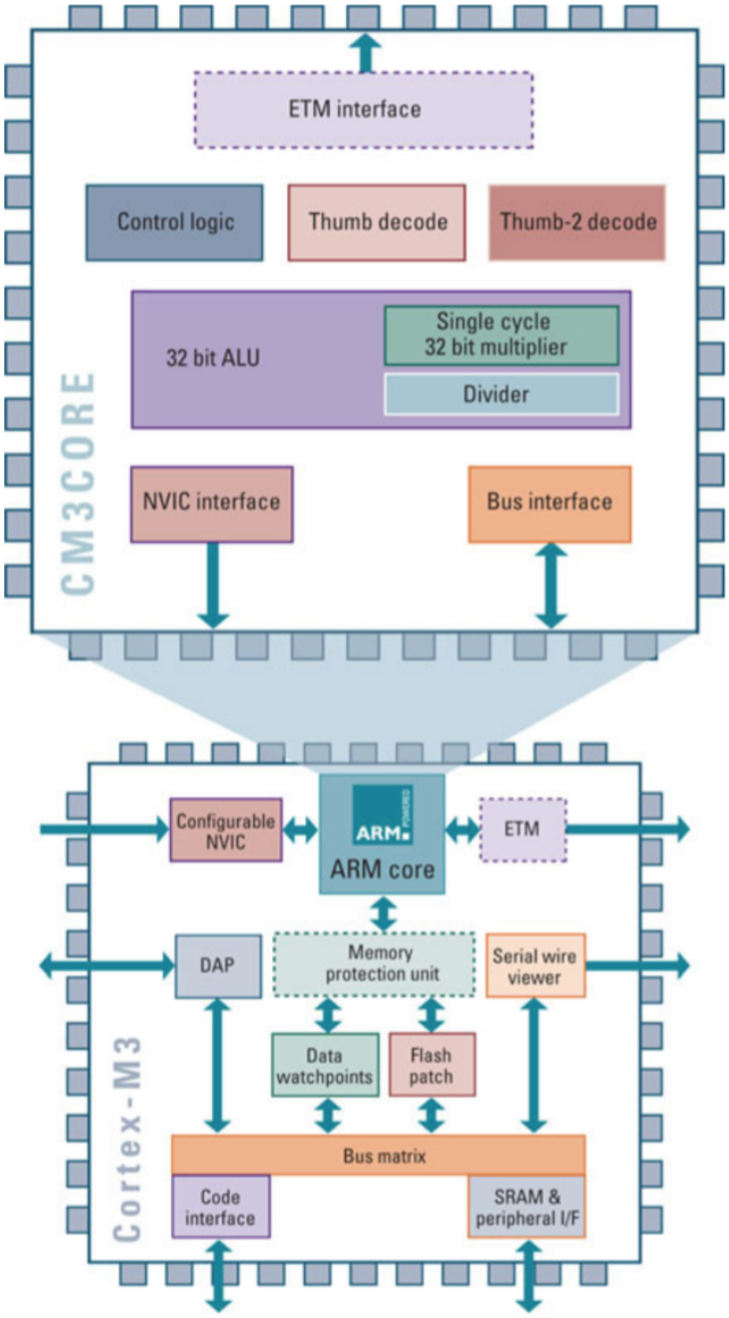
Matthias Puech

Master 1 SEMS — Cnam

Résumé des épisodes précédents

- MCU = CPU + SRAM + périphériques
- ARM ⊃ Cortex ⊃ Cortex M4 ⊃ STM32 ⊃ STM32F303
- modèle d'execution : machine load/store
- interaction avec périphériques par écriture dans registres
- notion d'espace d'adressage
- les bibliothèques C : CMSIS et HAL

Résumé des épisodes précédents



Aujourd'hui

Interagir avec l'extérieur grâce aux pins génériques

GPIO General Purpose Inputs/Outputs.

- configuration des horloges de périphériques
- configuration des GPIOs
- lecture/écriture sur les les GPIOs

Rappels Syntaxe des littéraux numériques en C

Avec `gcc` on a la possibilité de noter les constantes numériques dans diverses bases et dans divers codages :

Le préfixe = la base

(comment *nous* écrivons le nombre)

décimal	pas de préfixe (ex : `42`)
octal	préfixe `0` ou `0o` (ex : `052`)
hexadécimal	préfix `0x` (ex : `0x2A`)
binaire	préfix `0b` (ex : `0b101010`)

Exercice

Complétez : `0xF0F` = `0b`...

Rappels Syntaxe des littéraux numériques en C

Avec `gcc` on a la possibilité de noter les constantes numériques dans diverses bases et dans divers codages :

Le suffixe = le codage machine

(comment la machine le stocke)

non signé	suffixe `u` (ex `42u`)	32 bits
signé	pas de suffixe (ex : `42`)	complément à deux 32 bits
long	suffixe `l` (ex `42u`)	complément à deux 64 bits
float	suffixe `f` (ex `42.0f`)	IEEE 754 sur 32 bits
double	suffixe `d` (ex `42.0d`)	indisponible sur STM32F3

Rappels Opérations bits-à-bits

Permettent d'opérer sur la représentation binaire des entiers

décalages décale un certain nombre de fois la représentation binaire d'un cran à droite/gauche et comble les vides par des zéros
(ex : `0b100101 << 3` = `0b100101000`)
(ex : `0b101101 >> 2` = `0b1011`)

Exercices

Quel est l'effet du décalage gauche/droite sur un `int` ?

Comment écrire de façon concise 2^n ?

Rappels Opérations bits-à-bits

Permettent d'opérer sur la représentation binaire des entiers

conjonction "&" le n-ème bit du résultat est le "et" logique des n-èmes bits des deux opérandes
(ex : `0b1010 & 0b1100` = `0b1000`)

disjonction "|" le n-ème bit du résultat est le "ou" logique des n-èmes bits des deux opérandes
(ex : `0b1010 | 0b1100` = `0b1110`)

Exercices

Comment mettre le n-ième bit d'un entier m à 1 ?

Comment tester si le n-ième bit d'un entier m est 1 ?

Rappels Opérations bits-à-bits

Permettent d'opérer sur la représentation binaire des entiers

xor "^" le n-ème bit du résultat est le "ou exclusif" des n-èmes bits des deux opérandes
(ex : `0b1010 | 0b1100` = `0b0110`)

négation "~" le n-ème bit du résultat est la négation du n-ème bit de l'opérande
(ex : `0b1010` = `0b0101`)

Exercices

Comment mettre le n-ième bit d'un entier m à 0 ?

Comment tester si le n-ième bit d'un entier m est 0 ?

Astuce : les *masques*

Un *masque* est une valeur constante que l'on utilise pour isoler certains bits d'une autre valeur, en utilisant le *et* bit-à-bit.

Example

Soit `n` une variable de type `char` :

- `n & 1`

Astuce : les *masques*

Un *masque* est une valeur constante que l'on utilise pour isoler certains bits d'une autre valeur, en utilisant le *et* bit-à-bit.

Example

Soit `n` une variable de type `char` :

- `n & 1` isole le bit de poids faible de `n`

Astuce : les *masques*

Un *masque* est une valeur constante que l'on utilise pour isoler certains bits d'une autre valeur, en utilisant le *et* bit-à-bit.

Example

Soit `n` une variable de type `char` :

- `n & 1` isole le bit de poids faible de `n`
- `n & (1 << 7)`

Astuce : les *masques*

Un *masque* est une valeur constante que l'on utilise pour isoler certains bits d'une autre valeur, en utilisant le *et* bit-à-bit.

Example

Soit `n` une variable de type `char` :

- `n & 1` isole le bit de poids faible de `n`
- `n & (1 << 7)` isole le 7ème bit de `n`

Astuce : les *masques*

Un *masque* est une valeur constante que l'on utilise pour isoler certains bits d'une autre valeur, en utilisant le *et* bit-à-bit.

Example

Soit `n` une variable de type `char` :

- `n & 1` isole le bit de poids faible de `n`
- `n & (1 << 7)` isole le 7ème bit de `n`
- `n & ((1 << 7) - 1)`

Astuce : les *masques*

Un *masque* est une valeur constante que l'on utilise pour isoler certains bits d'une autre valeur, en utilisant le *et* bit-à-bit.

Example

Soit `n` une variable de type `char` :

- `n & 1` isole le bit de poids faible de `n`
- `n & (1 << 7)` isole le 7ème bit de `n`
- `n & ((1 << 7) - 1)` isole les 6 premiers bits de `n`

Les GPIOs

Chaque pins des STM32 est multiplexée : on peut choisir en software sa fonction parmi :

- entrée numérique (0V ou 3.3V) lisible en soft
- sortie numérique (0V ou 3.3V) contrôlable en soft
- fonction analogique (ADC/DAC)
- fonction alternative (E/S des périphériques)
  (UART, I2C, déclenchement d'un timer, ... *cf.* Tableau 12. de la specsheet)

Les GPIOs

Chaque pins des STM32 est multiplexée : on peut choisir en software sa fonction parmi :

- entrée numérique (0V ou 3.3V) lisible en soft
- sortie numérique (0V ou 3.3V) contrôlable en soft
- fonction analogique (ADC/DAC)
- fonction alternative (E/S des périphériques)
  (UART, I2C, déclenchement d'un timer, ... *cf.* Tableau 12. de la specsheet)

Vocabulaire

En électronique digitale, on parle des deux états d'une pin :

$$\text{set} = 1$$
$$\text{reset/clear} = 0$$

Les GPIOs

- Les entrées/sorties sont organisées en *ports* de 16 pins : GPIOA, GPIOB, GPIOC etc.
- Chaque pin a un nombre (dépendant du package) et un nom : PA1–PA16, PB1–PB16 etc.
- On peut activer/désactiver l'horloge de chacun des ports séparément, i.e. allumer/éteindre le port
  (au boot, presque tous sont éteints pour économiser du courant)

Example (extrait de la specsheet)

Pin number				Pin name (function after reset)	Pin type	I/O structure	Notes	Pin functions	
WLCSP100	LQFP100	LQFP64	LQFP48					Alternate functions	Additional functions
J9	24	15	11	PA1	I/O	TTa	(4)	USART2_RTS_DE, TIM2_CH2, TSC_G1_IO2, TIM15_CH1N, RTC_REFIN, EVENTOUT	ADC1_IN2, COMP1_INP, OPAMP1_VINP, OPAMP3_VINP

Les GPIOs

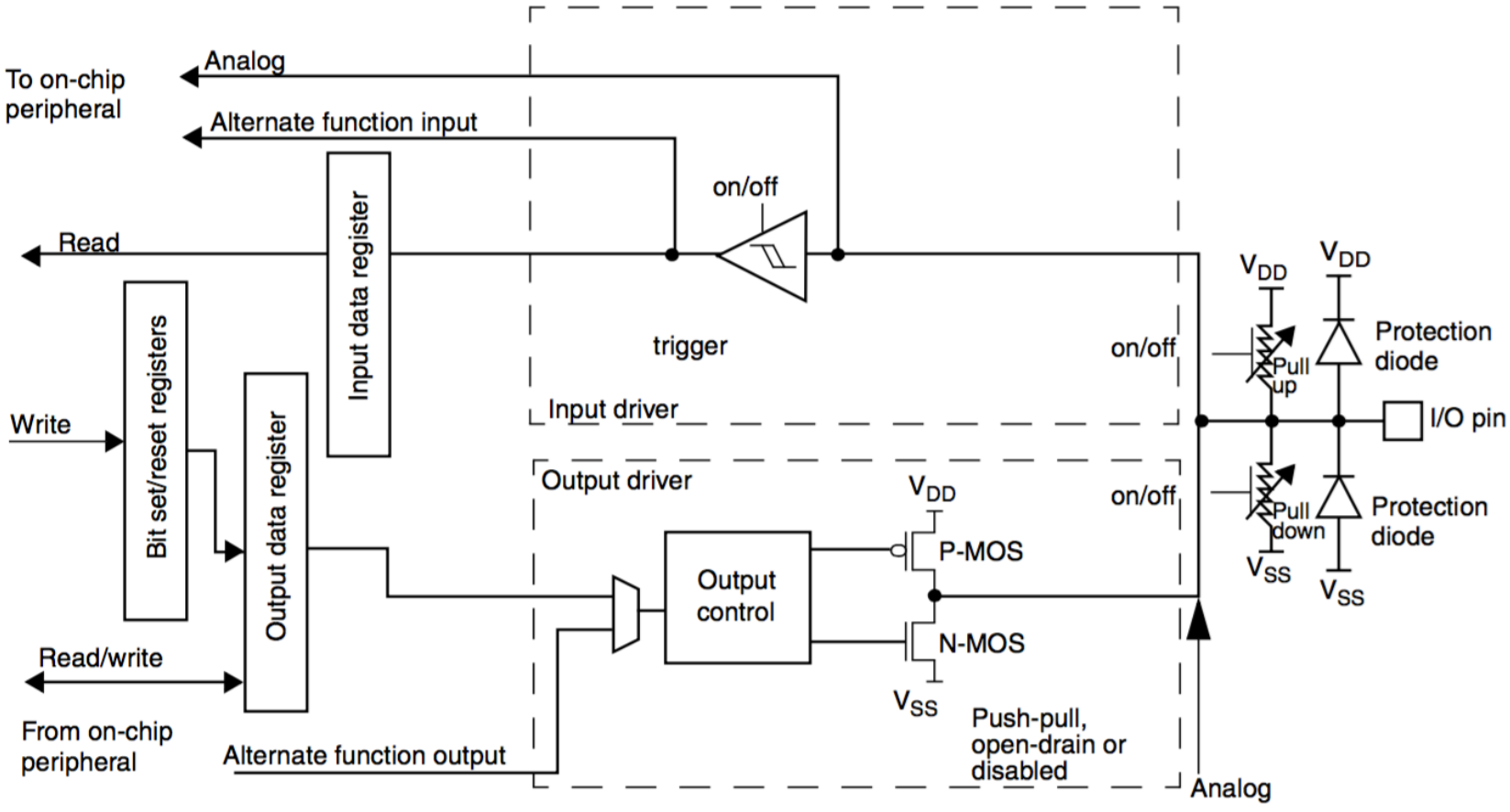
Remarques

- Il y a des exception : certaines pins ne peuvent pas/pas complètement être reprogrammées
  (ex : VREF+, OSC32_IN/OUT)
- Sur votre carte, certaines pins sont directement connectées :
  - aux LEDs
  - aux boutons
  - aux puces MEMS (accel, gyro)
  - au debugger…

  (*cf.* manuel utilisateur STM32F3Discovery)
- Attention : ne pas reprogrammer les pins qui servent au debugging, sinon vous ne pourrez plus parler à votre carte !
  (SWCLK, SWDIO, NRST, SWO)
- pour chaque IO, on choisit le contrôle d'entrée/sortie
  (pull-up/-down, drain sur les sorties, vitesse)

Les GPIOs

Schéma de principe d'une entrée/sortie

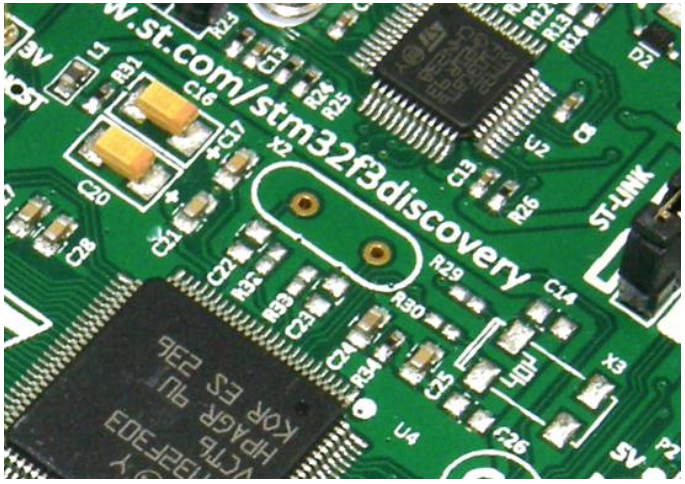


Configurer une pin pour une fonction donnée

1. activer les horloges correspondantes
   (si besoin les configurer)
2. sélectionner les bons paramètres pour chaque pin
   (vitesse, pull-up/pull-down…)
3. router le bon signal sur chaque pin
   (GPIO/fonction alternative/analogique)
4. le cas échéant, configurer le périphérique lui-même

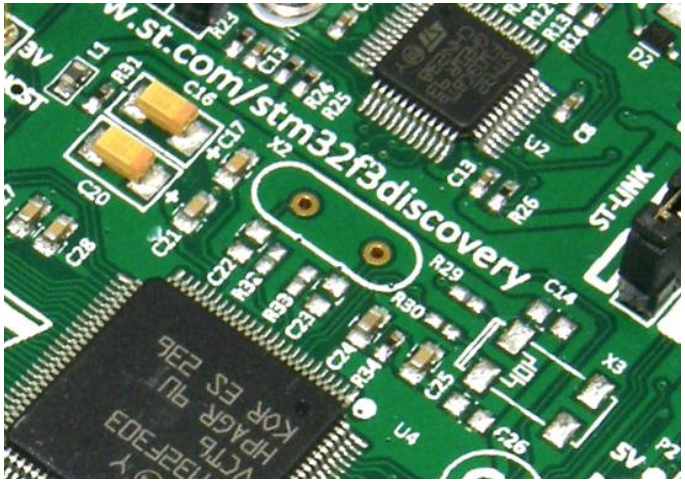
La hiérarchie des horloges STM32

- La source principale de temps est un oscillateur externe (HSE, un quartz) ou interne (HSI, un oscillateur RC)
- Toutes les horloges internes de tous les périphériques sont dérivées de celles-ci par des circuits PLL : *système harmonique*
- Sur notre carte l'oscillateur externe n'est pas soudé, donc on utilise l'oscillateur RC intégré



La hiérarchie des horloges STM32

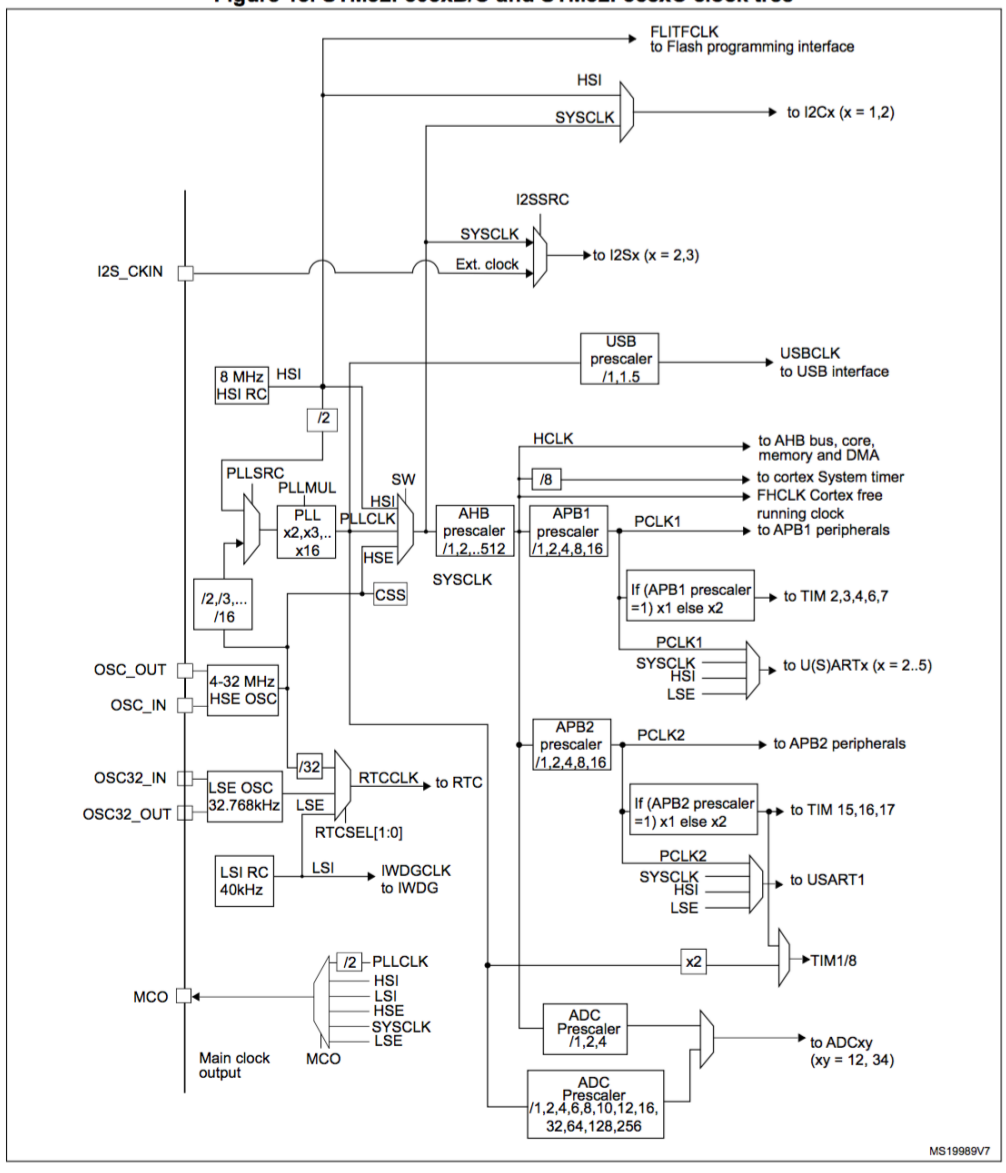
- La source principale de temps est un oscillateur externe (HSE, un quartz) ou interne (HSI, un oscillateur RC)
- Toutes les horloges internes de tous les périphériques sont dérivées de celles-ci par des circuits PLL : *système harmonique*
- Sur notre carte l'oscillateur externe n'est pas soudé, donc on utilise l'oscillateur RC intégré



⇝ le timing ne sera jamais très précis !

La hiérarchie des horloges STM32

Figure 13. STM32F303xB/C and STM32F358xC clock tree



On en reparlera…

GPIO : Configuration des horloges

- les ports GPIO doivent être allumés en y routant une horloge.
- Chaque périphérique est connecté au coeur par un bus
- Les ports GPIO sont sur le bus AHB
  (on en reparlera)
- le registre qui contrôle le bus AHB s'appelle RCC_AHBENR

cf. chapitre RCC du refman (p. 148).

9.4.6 AHB peripheral clock enable register (RCC_AHBENR)

Address offset: 0x14

Reset value: 0x0000 0014

Access: no wait state, word, half-word and byte access

Note: When the peripheral clock is not active, the peripheral register values may not be readable by software and the returned value is always 0x0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	ADC34 EN	ADC12EN	Res	Res	Res	TSCEN	IOPG EN(1)	IOPF EN	IOPE EN	IOPD EN	IOPC EN	IOPB EN	IOPA EN	IOPH EN(1)
		rw	rw				rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res	Res	Res	Res	Res	Res	Res	Res	Res	CRC EN	FMC EN(1)	FLITF EN	Res	SRAM EN	DMA2 EN	DMA1 EN
									rw	rw	rw		rw	rw	rw

1. Only on STM32F303xDxE.

GPIO : Configuration des horloges

CMSIS

cf. dans le code :

- le pointeur `RCC`
- la structure `RCC_TypeDef`
- Bit definition for `RCC_AHBENR` register

GPIO : Configuration des horloges

CMSIS

cf. dans le code :

- le pointeur `RCC`
- la structure `RCC_TypeDef`
- Bit definition for `RCC_AHBENR` register

Example

```
/* enable only GPIOA and GPIOB on AHB */
RCC->AHBENR = RCC_AHBENR_GPIOAEN
                    | RCC_AHBENR_GPIOBEN;
```

GPIO : Les registres de configuration

La configuration de chaque port GPIO est définie par les registres :

MODER	mode de chaque pin : input/output/alternate/analog
OSPEEDR	vitesse de switch
OTYPER	type push-pull ou open-drain
PUPDR	pull-up/pull-down

(cf. refman pp. 237–238)

GPIO : Les registres de configuration

11.4.1 GPIO port mode register (GPIOx_MODER) (x =A..H)

Address offset:0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y+1:2y **MODERy[1:0]:** Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O mode.

00: Input mode (reset state)

01: General purpose output mode

10: Alternate function mode

11: Analog mode

GPIO : Les registres de configuration

CMSIS

cf. dans le code :

- le pointeur `GPIOx`
- la structure `GPIO_TypeDef`
- Bit definition for `GPIO_MODER` register

GPIO : Les registres d'état

Chaque GPIO peut maintenant servir d'entrée sortie générique :

IDR	état des entrées (lecture seule)
ODR	état des sorties (écriture seule)
BRR, BSRR	écriture bit-à-bit de l'état des sorties (voir plus loin)

(*cf.* refman pp. 237–240)

CMSIS

Pour GPIOA, ces registres s'appellent :

```
GPIOA->IDR
GPIOA->ODR
GPIOA->BRR
GPIOA->BSRR
```

Le problème du Read/Modify/Write

Exercice

Écrire une instruction C qui allume PE8 (une des LEDs de notre carte). Attention à ne pas modifier l'état des autres LEDs !

Le problème du Read/Modify/Write

Exercice

Écrire une instruction C qui allume PE8 (une des LEDs de notre carte). Attention à ne pas modifier l'état des autres LEDs !

⇝ 3 instructions assembleur pour changer 1 bit :

- lecture de l'état du registre
- modification par bitmask
- écriture de l'état modifié dans le registre

Le problème du Read/Modify/Write

Une solution

Des registres spécialisés BRR/BSRR en écriture seule qui changent uniquement l'état de certains bits.
(*cf.* refman pp. 240,242.)

Example

Pour GPIOA :

BSRR y écrire `(1 << n)` mettra l'état de la pin PAn à 1 *(set)*

BRR y écrire `(1 << n)` mettra l'état de la pin PAn à 0 *(reset)*