

1. Organisation matérielle du microcontrôleur . . . . .	2
1.1. Vue d'ensemble : exemple du STM32F411 . . . . .	2
1.2. Structure d'un coupleur périphérique . . . . .	6
1.3. Accès logiciel aux registres des coupleurs . . . . .	8
1.4. Configuration des coupleurs du STM32F411 . . . . .	13
2. Interruptions et exceptions . . . . .	17
2.1. Concept d'interruption et d'exception . . . . .	17
2.2. Table des vecteurs d'exception du Cortex-M4 . . . . .	19
2.3. Séquencement des exceptions/interruptions . . . . .	20
2.4. Configuration des interruptions/exceptions . . . . .	25
3. Réflexion sur les interfaces de programmation . . . . .	31
3.1. Organisation logicielle d'une application microcontrôleur . . . . .	31
3.2. Interfaces synchrones . . . . .	32
3.3. Interfaces asynchrones . . . . .	34

## 1. Organisation matérielle du microcontrôleur

### 1.1. Vue d'ensemble : exemple du STM32F411

- Le STM32F411 (ST Microelectronics) est un microcontrôleur ( $\mu$ C). Le processeur Cortex-M4 est intégré dans un boîtier avec :

\* des coupleurs :

- ports parallèles d'entrée/sortie GPIO,
- compteurs, PWM, watchdog, RTC,
- liaisons séries : SPI, I2C, USART,
- convertisseurs analogique-numérique : ADC,
- contrôleur d'interruption,
- ...

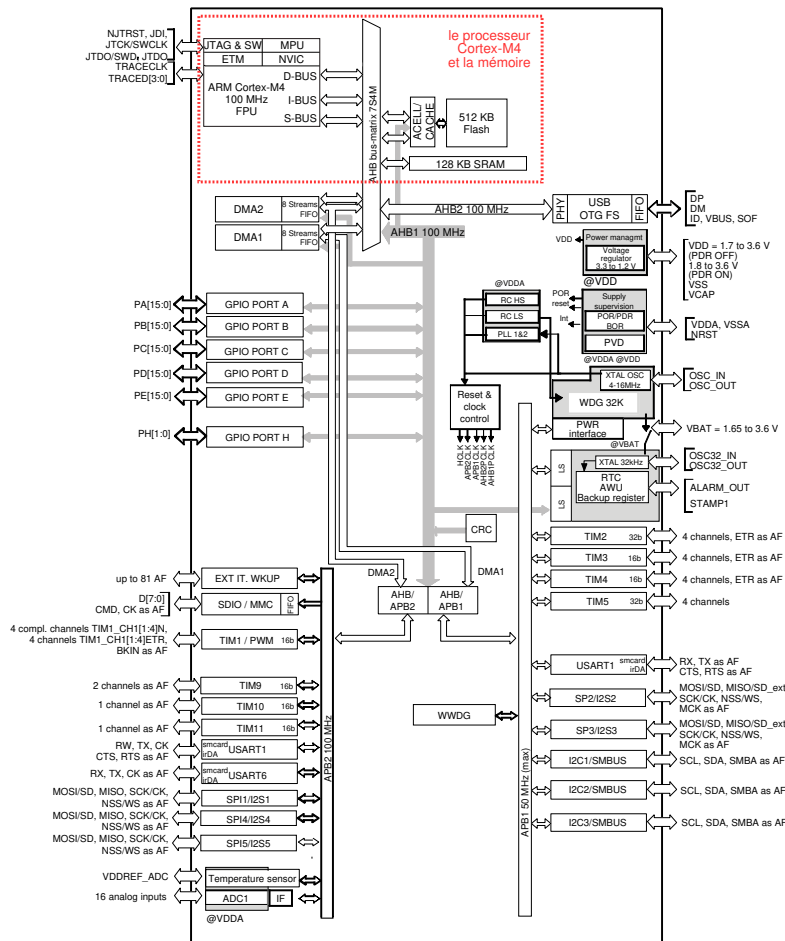
\* de la mémoire :

- SRAM (128 kb) et Flash (512 kb) internes,

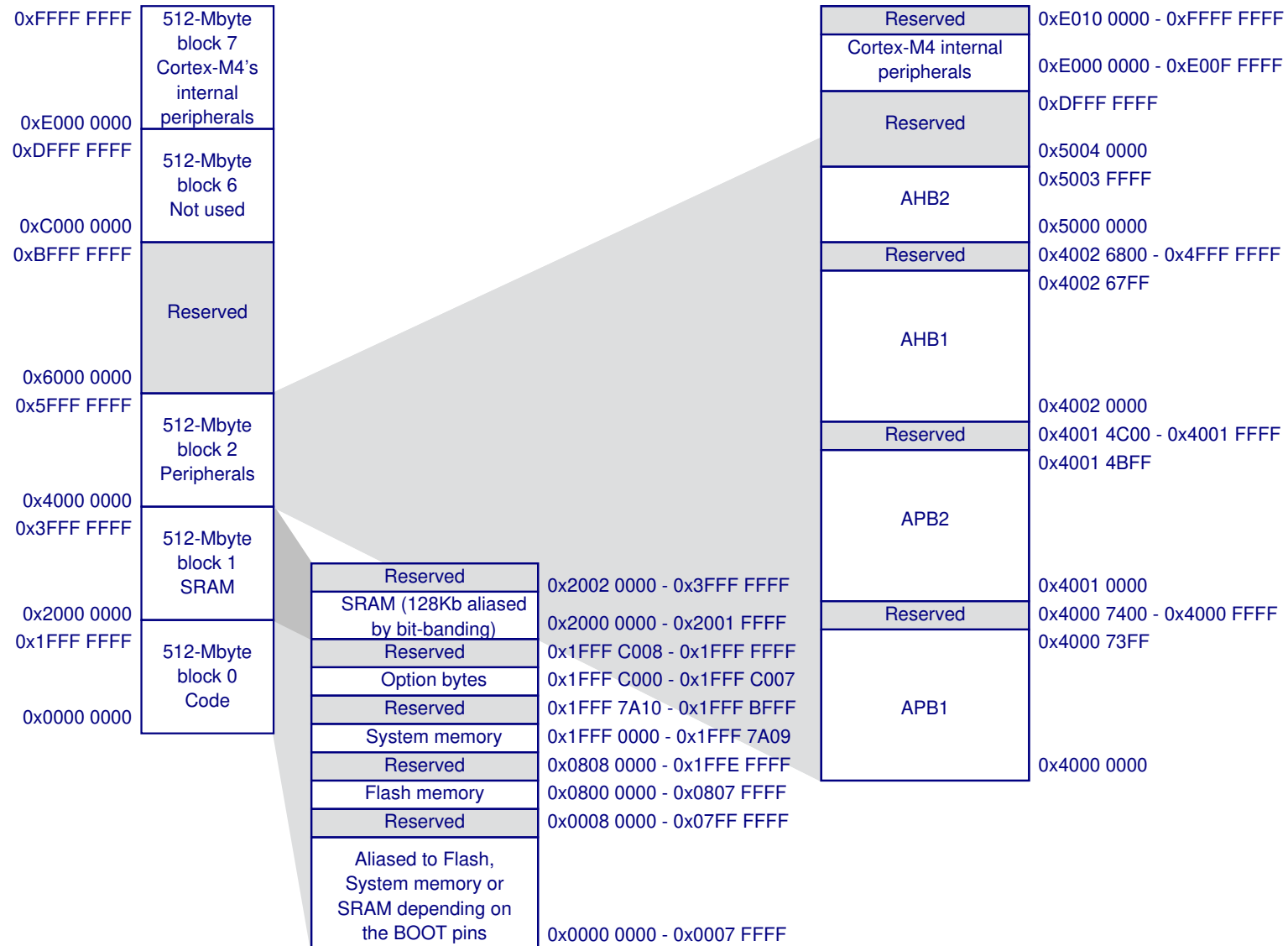
\* des bus d'interconnexion :

- AMBA AHB-Lite, APB,
- JTAG/SWD

Certains coupleurs périphériques peuvent être connectés aux broches du microcontrôleur pour communiquer avec l'extérieur.



### • Mapping mémoire



- Mapping des coupleurs périphériques : document **STM32F411xE datasheet** p. 54-56

Bus	Boundary address	Peripheral
	0xE010 0000 - 0xFFFF FFFF	Reserved
Cortex®-M4	0xE000 0000 - 0xE00F FFFF	Cortex-M4 internal peripherals
	0x5004 0000 - 0xDFFF FFFF	Reserved
AHB2	0x5000 0000 - 0x5003 FFFF	USB OTG FS

Bus	Boundary address	Peripheral
APB1	0x4000 7000 - 0x4000 73FF	PWR
	0x4000 6000 - 0x4000 6FFF	Reserved
	0x4000 5C00 - 0x4000 5FFF	I2C3
	0x4000 5800 - 0x4000 5BFF	I2C2
	0x4000 5400 - 0x4000 57FF	I2C1
	0x4000 4800 - 0x4000 53FF	Reserved
	0x4000 4400 - 0x4000 47FF	USART2
	0x4000 4000 - 0x4000 43FF	I2S3ext
	0x4000 3C00 - 0x4000 3FFF	SPI3 / I2S3
	0x4000 3800 - 0x4000 3BFF	SPI2 / I2S2
	0x4000 3400 - 0x4000 37FF	I2S2ext
	0x4000 3000 - 0x4000 33FF	IWDG
	0x4000 2C00 - 0x4000 2FFF	WWDG
	0x4000 2800 - 0x4000 2BFF	RTC & BKP Registers
	0x4000 1000 - 0x4000 27FF	Reserved
	0x4000 0C00 - 0x4000 0FFF	TIM5
	0x4000 0800 - 0x4000 0BFF	TIM4
	0x4000 0400 - 0x4000 07FF	TIM3
	0x4000 0000 - 0x4000 03FF	TIM2

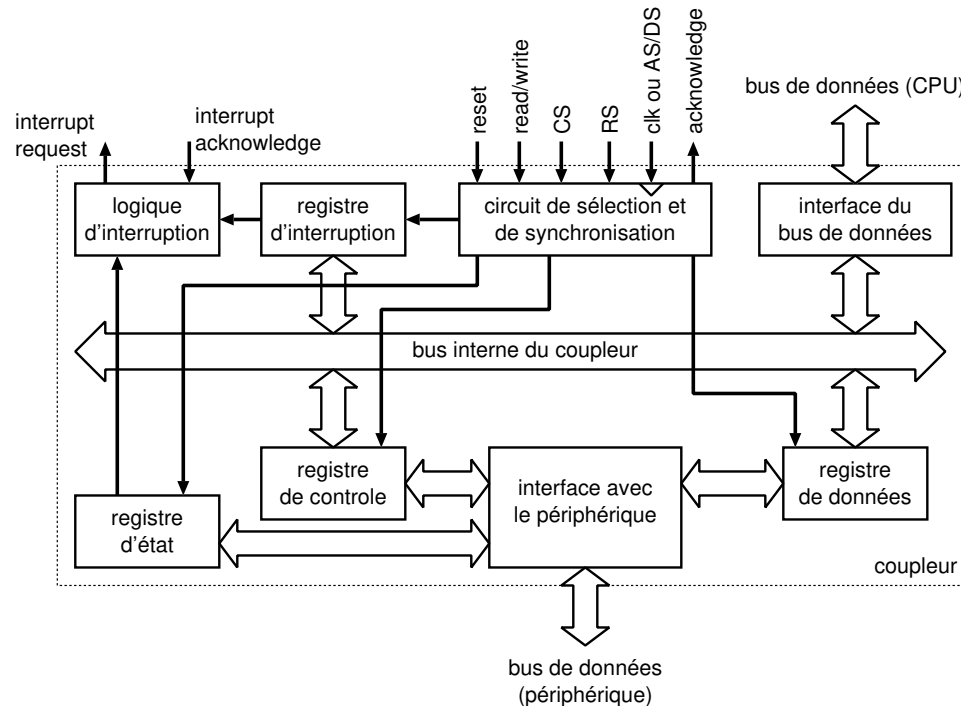
Bus	Boundary address	Peripheral
AHB1	0x4002 6800 - 0x4FFF FFFF	Reserved
	0x4002 6400 - 0x4002 67FF	DMA2
	0x4002 6000 - 0x4002 63FF	DMA1
	0x4002 5000 - 0x4002 4FFF	Reserved
	0x4002 3C00 - 0x4002 3FFF	Flash interface register
	0x4002 3800 - 0x4002 3BFF	RCC
	0x4002 3400 - 0x4002 37FF	Reserved
	0x4002 3000 - 0x4002 33FF	CRC
	0x4002 2000 - 0x4002 2FFF	Reserved
	0x4002 1C00 - 0x4002 1FFF	GPIOH
	0x4002 1400 - 0x4002 1BFF	Reserved
	0x4002 1000 - 0x4002 13FF	GPIOE
	0x4002 0C00 - 0x4002 0FFF	GPIOD
	0x4002 0800 - 0x4002 0BFF	GPIOC
	0x4002 0400 - 0x4002 07FF	GPIOB
	0x4002 0000 - 0x4002 03FF	GPIOA
APB2	0x4001 5400 - 0x4001 FFFF	Reserved
	0x4001 5000 - 0x4001 53FFF	SPI5/I2S5
	0x4001 4800 - 0x4001 4BFF	TIM11
	0x4001 4400 - 0x4001 47FF	TIM10
	0x4001 4000 - 0x4001 43FF	TIM9
	0x4001 3C00 - 0x4001 3FFF	EXTI
	0x4001 3800 - 0x4001 3BFF	SYSCFG
	0x4001 3400 - 0x4001 37FF	SPI4/I2S4
	0x4001 3000 - 0x4001 33FF	SPI1/I2S1
	0x4001 2C00 - 0x4001 2FFF	SDIO
	0x4001 2400 - 0x4001 2BFF	Reserved
	0x4001 2000 - 0x4001 23FF	ADC1
	0x4001 1800 - 0x4001 1FFF	Reserved
	0x4001 1400 - 0x4001 17FF	USART6
	0x4001 1000 - 0x4001 13FF	USART1
	0x4001 0400 - 0x4001 0FFF	Reserved
	0x4001 0000 - 0x4001 03FF	TIM1
	0x4000 7400 - 0x4000 FFFF	Reserved

- Mapping des périphériques système sur le bus périphérique privé (PPB) (interne au processeur Cortex-M4) : document **ARM®v7-M Architecture Reference Manual** p. 591

The System Control Space (SCS) is a memory-mapped 4KB address space that provides 32-bit registers for configuration, status reporting and control. The SCS registers divide into the following groups: System control and identification, The CPUID processor identification space, System configuration and status, Fault reporting, A system timer, SysTick, A Nested Vectored Interrupt Controller (NVIC), A Protected Memory System Architecture (PMSA), System debug.

System Control Space, address range		
0xE000E000 to 0xE000EFFF		
Group	Address range	Notes
System control and ID registers	0xE000E000 -0xE000E00F	Includes the Interrupt Controller Type and Auxiliary Control registers
	0xE000ED00 -0xE000ED8F	System Control Block
	0xE000EDF0 -0xE000EEFF	Debug registers in the SCS
	0xE000EF00 -0xE000EF4F	Includes the SW Trigger Interrupt Register, see <a href="#">Software Triggered Interrupt Register STIR on page B3-615</a>
	0xE000EF50 -0xE000EF8F	Cache and branch predictor maintenance, see <a href="#">Cache and branch predictor maintenance operations on page B2-573</a>
	0xE000EF90 -0xE000EFCF	IMPLEMENTATION DEFINED
	0xE000EFD0 -0xE000EFFF	Microcontroller-specific ID space
SysTick	0xE000E010 -0xE000E0FF	System Timer, see <a href="#">The system timer, SysTick on page B3-616</a>
NVIC	0xE000E100 -0xE000ECFF	External interrupt controller, see <a href="#">Nested Vectored Interrupt Controller, NVIC on page B3-620</a>
MPU	0xE000ED90 -0xE000EDEF	Memory Protection Unit, see <a href="#">Protected Memory System Architecture, PMSAv7 on page B3-628</a>

## 1.2. Structure d'un coupleur périphérique



### • Éléments d'interfaçage matériel

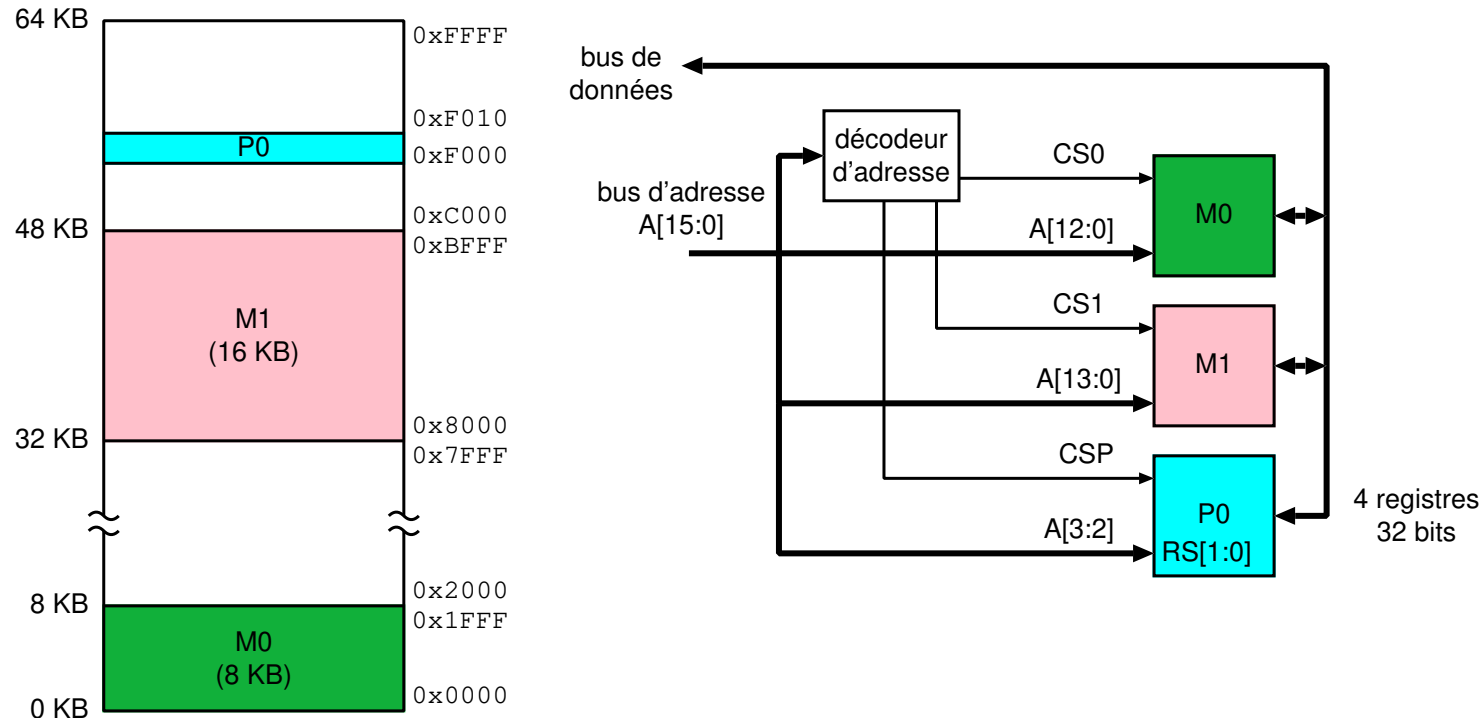
- interne (bus de données, bloc de synchronisation) : échange des données entre le processeur et le périphérique en respectant le protocole utilisé par les bus internes du microcontrôleur.
- externe (interface périphérique) : échange avec le périphérique extérieur en respectant son protocole de communication et ses caractéristiques électriques. Une configuration spécifique pour **connecter l'interface externe du coupleur aux broches du microcontrôleur** est généralement nécessaire.

- Configuration du coupleur périphérique (vitesse de transmission, durée de comptage, ...) via un ou plusieurs **registres de contrôle**. Définition du mode de communication utilisé avec le coupleur (scrutation/interruption/DMA. Obligatoire avant d'utiliser un coupleur.

### • Utilisation du coupleur périphérique

- **registre d'état/d'interruption** : permettre au processeur d'obtenir de l'information sur les événements survenus (nouvelle donnée arrivée, expiration du comptage, ...), de manière logicielle (lecture du registre d'état) ou matérielle (ligne d'interruption).
- **registre de données** : tampon intermédiaire d'échange de données entre le processeur et l'extérieur. Il peut être dupliqué (envoi/réception). Ce peut être une FIFO pour s'accomoder de différences importantes de débit entre le processeur et le périphérique externe.

- Insertion du coupleur dans un l'espace adressable du microcontrôleur : décodage d'adresse



Exemples de signaux CS et RS :

- CS0 = 1 si A[15:13] == "000" (mémoire adressable par octet)
- CS1 = 1 si A[15:14] == "10" (mémoire adressable par octet)
- CSP = 1 si A[15:4] == "1111 0000 0000" (les bits 0 et 1 sont inutilisés ⇒ on n'accède qu'à des adresses multiples de 4).

### 1.3. Accès logiciel aux registres des coupleurs

Les registres internes des coupleurs sont visibles dans l'espace adressable. Ils sont caractérisés par

- leur **adresse**, fixée par le fabricant du microcontrôleur.
- un **type d'accès** possible : lecture et écriture (R/W), lecture seule (RO), écriture seule (WO).

Les registres (data, contrôle, état, interruption) sont regroupés par coupleur pour former une **interface de programmation** cohérente qu'il sera possible d'utiliser pour programmer, puis interagir avec les coupleurs. *Ces informations sont disponibles dans le manuel de programmation du microcontrôleur.*

Différentes structures de registres peuvent être rencontrées.

- Mapping 1 adresse → 1 registre

Exemple d'un port *GPIO* (General Purpose Input Output). La datasheet du microcontrôleur décrit l'interface suivante :


adresse du registre	nom	description	format	mode d'accès	valeur après reset	
0xE0028000	IOPIN	GPIO Pin Value Reg	32 bits	RO	NA	valeur présente sur le port
0xE0028004	IODIR	GPIO Dir control Reg	32 bits	RW	0	choix du sens de la broche : $\begin{cases} '0' = \text{entrée} \\ '1' = \text{sortie} \end{cases}$
0xE0028008	IOSET	GPIO Out Set Register	32 bits	WO	NA	si la broche est en sortie elle est forcée à '1' par l'écriture d'un '1' sur le bit correspondant
0xE002800C	IOCLR	GPIO 1 Out Clear Register	32 bits	WO	NA	si la broche est en sortie elle est forcée à '0' par l'écriture d'un '1' sur le bit correspondant



## Accès en langage C

### \* Définition explicite de chacun des registres

```
#include <stdint.h>
#define BASE_ADDR 0xE0028000
#define _IOPIN (*(volatile uint32_t *) (BASE_ADDRESS))
#define _IODIR (*(volatile uint32_t *) (BASE_ADDRESS + 4))
#define _IOSET (*(volatile uint32_t *) (BASE_ADDRESS + 8))
#define _IOCLR (*(volatile uint32_t *) (BASE_ADDRESS + 12))
...
uint32_t val = _IOPIN;           // lecture du registre
_IODIR = 0x00FF0000;           // écriture du registre
```

 modificateur de type **volatile** pour empêcher le compilateur d'optimiser le code quand on accède aux registres car ils peuvent être modifiés à l'insu du processeur.

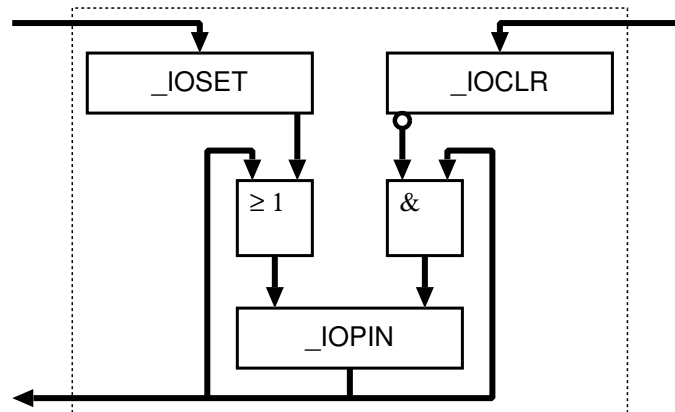
### \* Définition d'une structure superposée ("mappée") sur la zone d'accès des registres dans l'espace adressable

```
typedef volatile struct {           // définition d'un nouveau type
    uint32_t PIN;
    uint32_t DIR;
    uint32_t SET;
    uint32_t CLR;
} IO;
IO * _IO = (IO *)0xE0028000;       // pointeur sur la structure de registres du port 1
...
unsigned int val = _IO->PIN;       // lecture du registre PIN
_IO->DIR = 0x00FF0000;             // écriture du registre DIR
```

\* Modification d'un registre accessible en R/W par masquage

```
// modification des bits 0 et 1 sans altérer les autres bits
_IODIR = _IODIR | 0x3;    // mise à 1 -> lecture, modification, puis écriture de _IODIR
_IODIR = _IODIR & ~0x3;   // mise à 0
_IODIR = _IODIR ^ 0x3;    // inversion
```

\* Modification d'un registre RO (Read Only), associé à des registres SET et CLR



```
uint32_t var = _IOPIN;           // lecture de _IOPIN

_IIOSET = 0x3;                   // mise à 1 des bits 0 et 1 de _IOPIN
_IIOCLR = 0x3;                   // mise à 0 des bits 0 et 1 de _IOPIN
```

- Mapping 1 adresse → accès à deux registres complémentaires RO et WO (Read Only et Write Only).

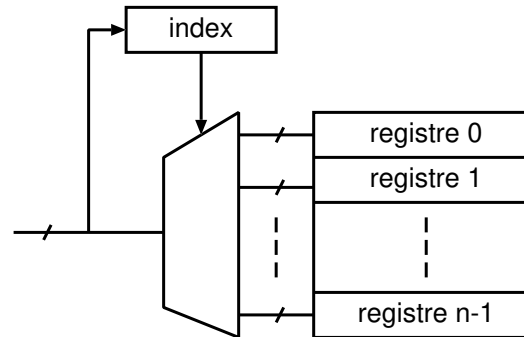
Registre	Fonction	Accès	Adresse
3	Données (depuis le coupleur)	RO	base + 4
2	Données (vers le coupleur)	WO	base + 4
1	Etat	RO	base
0	Contrôle	WO	base

- Mapping 1 adresse → 2 registres dont l'accès est associé à un bit de contrôle.

Registre	Fonction	CRA[i]	Adresse
2	Données port A (DRA)	1	base + 4
1	Direction Données port A (DDRA)	0	base + 4
0	Contrôle port A (CRA)	X	base

1. forcer le bit CRA[i] du registre CRA à '0'.
2. programmer DDRA pour configurer les bits du port en entrée ou en sortie.
3. forcer CRA[i] à '1',
4. accès au registre de données DRA du coupleur.

- Mapping 1 adresse → plusieurs registres dont l'accès est associé à un registre d'index :



- \* Pour accéder à un registre, l'utilisateur doit d'abord charger le registre d'index.
- \* Le registre vu de l'extérieur du coupleur est alors celui qui correspond à l'index. Plusieurs registres sont donc placés à la même adresse, mais un seul est visible à un instant donné. Cette méthode est souvent utilisée pour des périphériques identifiés par une adresse unique (ex : les différents registres d'un capteur accroché à un bus I2C).

Le registre utilisé comme index peut avoir la capacité à s'auto-incrémenter après chaque accès aux registres qu'il sert à adresser. Technique intéressante si les registres doivent être adressés en séquence.

#### 1.4. Configuration des coupleurs du STM32F411

Avant de pouvoir être utilisé, les coupleurs doivent être configurés. Pour cela, il faut :

1. Déterminer le(s) coupleur(s) nécessaire(s) pour l'application envisagée et déterminer à quelle(s) broche(s) extérieure(s) ils seront rattachés.

Exemples :

- \* contrôler un led nécessite une broche associée à une sortie d'un port GPIO.
- \* communiquer sur un port série nécessite deux broches associées respectivement aux fils d'émission et de réception.

2. Voir la **table 9** du document **STM32F411xE datasheet p. 47-53** pour obtenir la liste des signaux en provenance des coupleurs rattachables aux broches extérieures du microcontrôleur.

- \* L'utilisation d'une broche en GPIO est associée à l'“Alternate Function” AF0 (valeur par défaut au reset). Il faut simplement repérer le nom de la broche. Par exemple PB0 → Port B, pin 0.

- \* Pour les entrées/sorties des autres coupleurs, il faut repérer, dans la table 9, le nom de la broche et l'“Alternate Function”. Par exemple : l'USART6 utilise l'alternate function AF08 et la broche PC6 pour l'émission et la broche PC7 pour la réception de données.

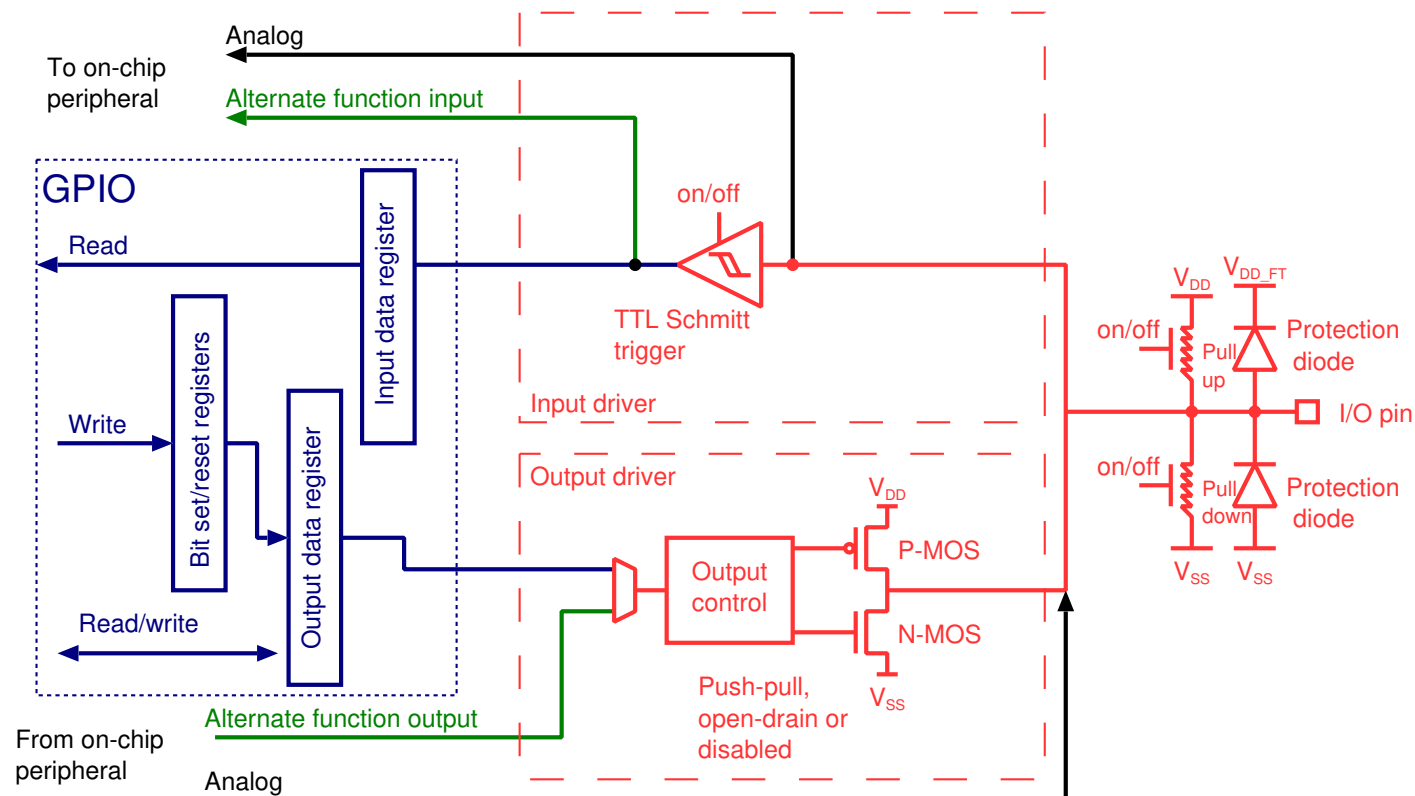
Table 9. Alternate function mapping (continued)

Port		AF00	AF01	AF02	AF03	AF04	AF05	AF06	AF07	AF08	AF09	AF10	AF11	AF12	AF13	AF14	AF15
		SYS_AF	TIM1/TIM2	TIM3/ TIM4/ TIM5	TIM9/ TIM10/ TIM11	I2C1/I2C2/ I2C3	SPI1/I2S1S PI2/ I2S2/SPI3/ I2S3	SPI2/I2S2/ SPI3/ I2S3/SPI4/ I2S4/SPI5/ I2S5	SPI3/I2S3/ USART1/ USART2	USART6	I2C2/ I2C3	OTG1_FS		SDIO			
Port C	PC0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PC1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PC2	-	-	-	-	-	SPI2_MISO	I2S2ext_SD	-	-	-	-	-	-	-	-	EVENT OUT
	PC3	-	-	-	-	-	SPI2_MOSI /I2S2_SD	-	-	-	-	-	-	-	-	-	EVENT OUT
	PC4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PC5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PC6	-	-	TIM3_CH1	-	-	I2S2_MCK	-	-	USART6_ TX	-	-	-	SDIO_ D6	-	-	EVENT OUT
	PC7	-	-	TIM3_CH2	-	-	SPI2_SCK/I 2S2_CK	I2S3_MCK	-	USART6_ RX	-	-	-	SDIO_ D7	-	-	EVENT OUT
	PC8	-	-	TIM3_CH3	-	-	-	-	-	USART6_ CK	-	-	-	SDIO_ D0	-	-	EVENT OUT
Port C	PC9	MCO_2	-	TIM3_CH4	-	I2C3_SDA	I2S2_CKIN	-	-	-	-	-	-	SDIO_ D1	-	-	EVENT OUT
	PC10	-	-	-	-	-	-	SPI3_SCK/I2 S3_CK	-	-	-	-	-	SDIO_ D2	-	-	EVENT OUT
	PC11	-	-	-	-	-	I2S3ext_SD	SPI3_MISO	-	-	-	-	-	SDIO_ D3	-	-	EVENT OUT
	PC12	-	-	-	-	-	-	SPI3_MOSI/I 2S3_SD	-	-	-	-	-	SDIO_ CK	-	-	EVENT OUT
	PC13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	PC14	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Pinouts and pin description

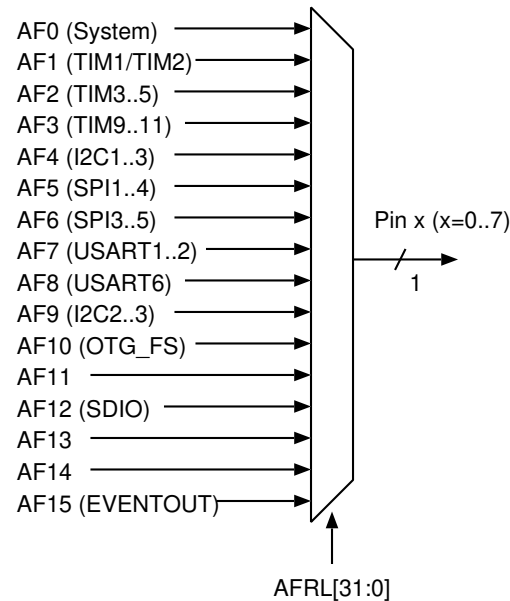
STM32F411xC STM32F411xE

3. Repérer sur quels bus sont accrochés le port associé à la broche utilisée (AHB1), ainsi que le coupleur. Autoriser l'horloge pour le port et le périphérique : registres **peripheral clock enable register** RCC\_busENR.
4. Pour chaque broche utilisée, réaliser la configuration de la broche en entrée ou en sortie, en définissant les types de sortie (push-pull / open-drain), la vitesse de commutation, et la présence ou pas de résistances de pull-up / pull-down.

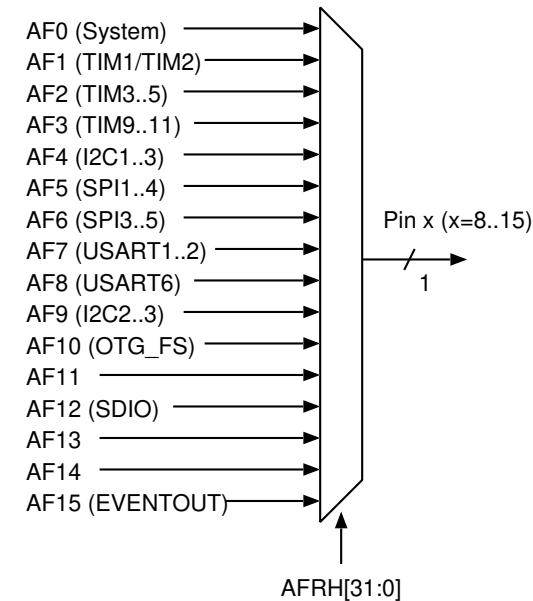


## 5. Configurer l'Alternate Function sélectionnée dans les registres GPIOxAFRL / GPIOx\_AFRH.

For pins 0 to 7, the GPIOx\_AFRL[31:0] register selects the dedicated alternate function



For pins 8 to 15, the GPIOx\_AFRH[31:0] register selects the dedicated alternate function



Le paramètre AFxx est la commande d'un multiplexeur qui permet d'associer les entrées/sorties d'un coupleur à certaines broches. Les broches sont multiplexées et il n'est pas possible d'avoir tous les coupleurs associés aux broches. Il est nécessaire d'opérer un choix en fonction de l'application envisagée.

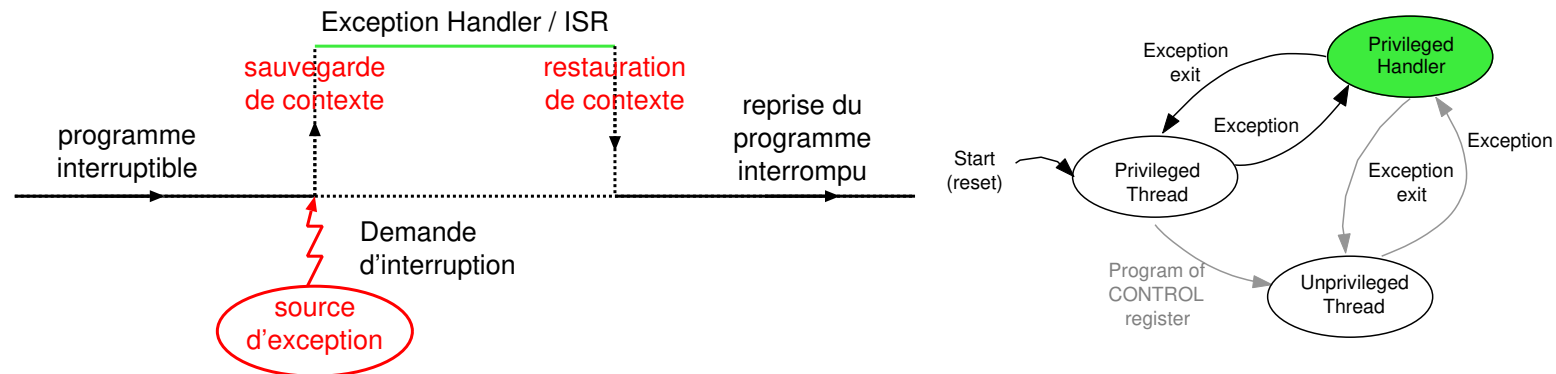
6. Configurer le périphérique (voir la partie spécifique de la datasheet).
7. Voilà ! Le coupleur est prêt à être utilisé :)



## 2. Interruptions et exceptions

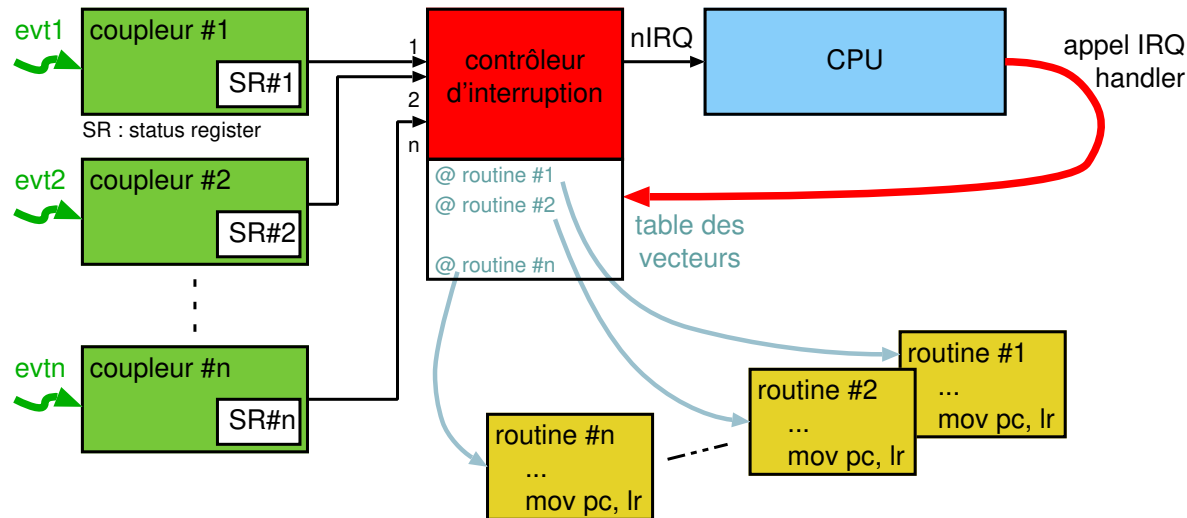
### 2.1. Concept d'interruption et d'exception

Du point de vue de l'exécution du code, une **exception** ou **interruption**, est un événement qui arrête l'exécution d'un programme en cours pour exécuter un traitement considéré comme *plus prioritaire* à ce moment-là. Le traitement est réalisé dans le **mode privilégié Handler** avec le pointeur de pile **MSP**.



- Le terme **exception** fait référence à une erreur qui intervient au niveau de l'exécution du code par le processeur et qui nécessite un traitement spécifique réalisé par l'**exception handler** (accès mémoire ou instruction incorrects, division par 0).
- Le terme **interruption** fait référence à un événement matériel issu d'un coupleur périphérique. Il fait intervenir une requête (**Interrupt ReQuest = IRQ**) du coupleur périphérique auprès du processeur pour qu'il procède à un traitement spécifique réalisé par le **Interrupt ReQuest Handler = IRQ Handler / Interrupt Service Routine = ISR** (suite à l'appui sur un bouton poussoir, à un choix d'un menu, à la prise en compte d'une fin de course, à la réception d'un caractère, à la fin d'une temporisation, ...).
- Lorsqu'un événement matériel intervient, il est signalé par un **drapeau (flag)** mémorisé dans le registre d'état du coupleur. Il appartient à la routine de traitement de la demande d'interruption d'indiquer au coupleur que l'évènement a été traité.

- Connexion des sources d'interruption via un contrôleur d'interruption (**NVIC**) et une table des vecteurs



Une entrée du contrôleur d'interruption (demande d'exception/interruption) peut être dans l'état :

- «**inactive**» : aucune demande faite
- «**pending**» : une exception/IRQ a été générée. Elle n'est pas encore traitée.
- «**active**» : l'exception/IRQ est en cours de traitement.
- «**active & pending**» : l'exception/IRQ est en cours de traitement. Une autre demande du même type a été générée.

Le contrôleur d'interruption a pour rôle de garder une trace de la demande des coupleurs, de hiérarchiser différentes demandes simultanées et d'associer à une entrée un traitement.

- hiérarchisation / masquage possibles au niveau du contrôleur d'interruption,
  - vectorisation des interruptions : adresse de la routine d'interruption est placée dans la table des vecteurs à une adresse fonction du numéro de l'entrée sollicitée, ce qui permet de fournir automatiquement l'adresse de la routine à exécuter en réaction à une demande d'interruption,
  - latence faible (délai entre le moment où la demande intervient, et le moment où le traitement démarre).
- Les **traitements réalisés par les routines d'interruption doivent être courts**. Une partie du travail peut éventuellement être déléguée à une zone du programme principal contrôlée par un drapeau (flag) positionné par l'ISR.

## 2.2. Table des vecteurs d'exception du Cortex-M4

Exception number 16+n	IRQ number n	Offset 0x0040+4n	Vector table
			Interrupt#n vector
			.
			.
			.
18	2	0x004C	Interrupt#2 vector
17	1	0x0048	Interrupt#1 vector
16	0	0x0044	Interrupt#0 vector
15	-1	0x0040	Systick vector
14	-2	0x003C	PendSV vector
13		0x0038	Reserved
12			Debug Monitor vector
11	-5	0x0030	SVCall vector
10		0x002C	Reserved
9			
8			
7			
6	-10	0x0018	Usage fault vector
5	-11	0x0014	Bus fault vector
4	-12	0x0010	Memory manage fault vector
3	-13	0x000C	Hard fault vector
2	-14	0x0008	NMI vector
1		0x0004	Reset vector
		0x0000	Initial SP value

- La table des vecteurs est définie dans le fichier  
`startup/startup_stm32f411xe.s`
- Elle est composée, à l'adresse `0x00000000`<sup>1</sup> de la valeur d'initialisation du pointeur de pile `mnp` (main stack pointer), et des vecteurs d'exception et d'interruption.
- Chaque vecteur d'exception/interruption stocke l'**adresse** du gestionnaire correspondant (System/Interrupt Handler).
- La table des vecteurs du Cortex-M4 permet de gérer jusqu'à 240 sources d'interruption hormis les 16 emplacements réservés de manière fixe pour les exceptions système. Pour le microcontrôleur STM32F411 seules 52 sources sont utilisables.
- Chaque coupleur source d'IRQ est associé à un **IRQ number défini par le fabricant du microcontrôleur**. Voir, dans le projet, la liste dans les fichiers `include/stm32f411xe.h` et `startup/startup_stm32f411xe.s`
- Le **Reset vector** contient l'adresse de la fonction exécutée après un "reset" du système.

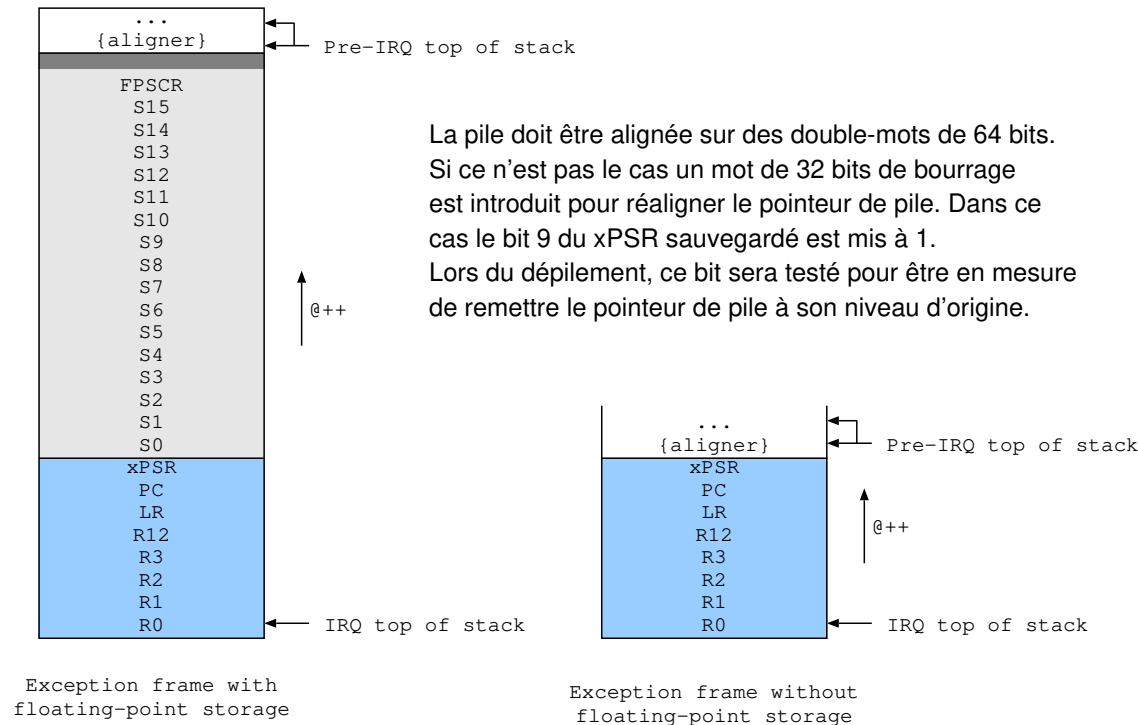
<sup>1</sup>La position de la table des vecteurs peut être modifiée via le registre **Vectored Table Offset Register**

```
_SCB->VTOR = offset
```

offset est tel que les 9 bits de poids faible doivent être nul (alignement minimum sur une frontière multiple de 128 mots de 32 bits). La table peut être en mémoire FLASH ou en RAM. Sur le STM32F411, la table est par défaut en mémoire FLASH : `VTOR=0x08000000`.

## 2.3. Séquencement des exceptions/interruptions

- Entrée dans la routine d'exception



Dans le cas où les registres FPU sont empilés (25 registres), un mot de 32 bits de bourrage est inséré en premier pour conserver l'alignement sur une adresse multiple de 64 bits

- **Sauvegarde automatique** des registres R0-R3, R12, LR, PC, xPSR (8 registres) et éventuellement des registres de la FPU<sup>1</sup> S0-S15 et FPSCR **sur la pile courante** du programme interrompu.
- Le registre LR se voit ensuite affecté un **code qui indique comment devra se faire le retour**.
- **Saut à l'adresse du handler de l'exception/interruption obtenue via la table des vecteurs** et passage du processeur au **mode de fonctionnement Handler**.
- Modification des registres d'état et de contrôle :

```
IPSR.ISR_NUMBER = 'Exception Number' ; EPSR.ICI/IT=0
```

```
CONTROL.FPCA=0 (Lazy Stacking)
```

```
CONTROL.SPSEL=0 (utilisation du pointeur de pile MSP)
```

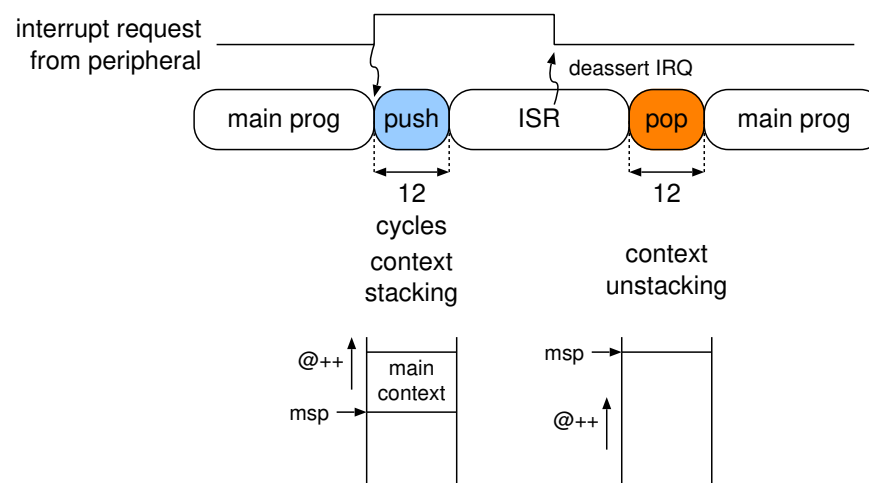
<sup>1</sup>Lazy stacking : l'espace est réservé sur la pile, mais les registres de la FPU ne sont empilés que si une instruction FPU est utilisée, auquel cas, on aura aussi CONTROL.FPCA=1.

- Retour de la routine d'exception : il s'effectue lorsqu'on copie le code placé dans le registre `LR` à l'entrée de la routine d'exception dans le `PC` (Lors du retour de la routine de traitement de l'exception/interruption). Le code est alors interprété de la manière suivante :

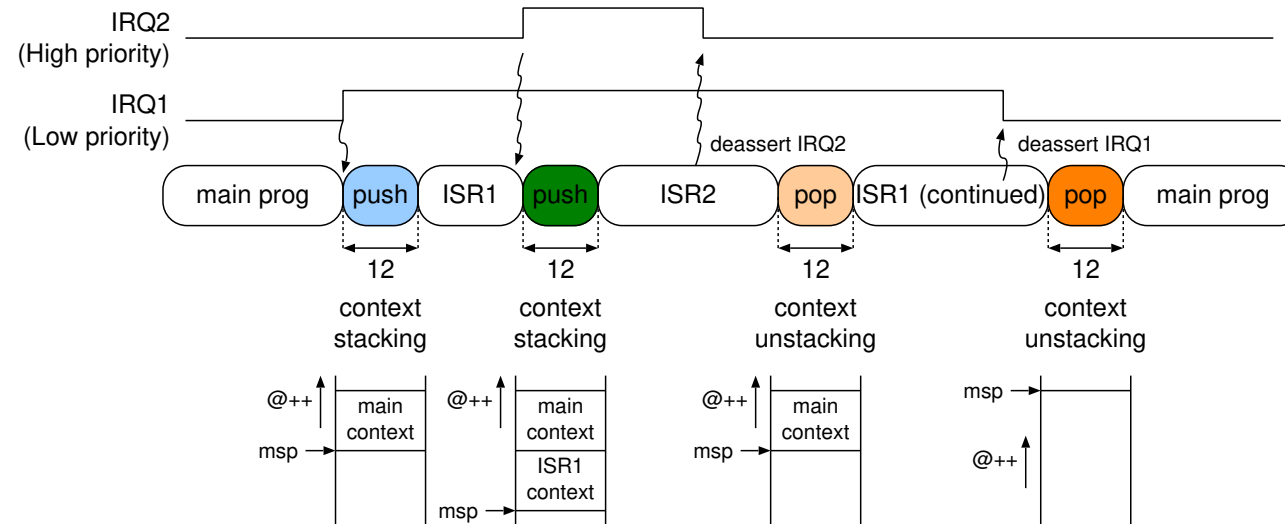
FPU utilisée avant l'exception	FPU non utilisée avant l'exception	Actions à réaliser en sortie d'exception
0xFFFFFEE1	0xFFFFFFF1	retour au mode Handler, SP = MSP
0xFFFFFEE9	0xFFFFFFF9	retour au mode Thread, SP = MSP
0xFFFFFEEF	0xFFFFFFF7	retour au mode Thread, SP = PSP

Les actions nécessaires sont effectuées pour dépiler les valeurs et rétablir le registre `CONTROL` à sa valeur initiale.

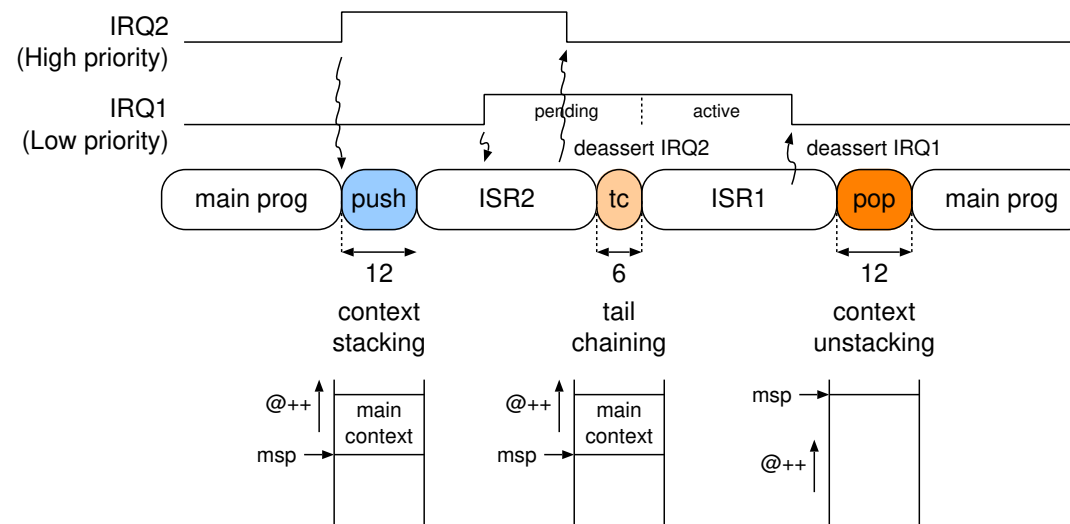
- Latence (avec une mémoire 0 wait-states)
  - Configuration standard



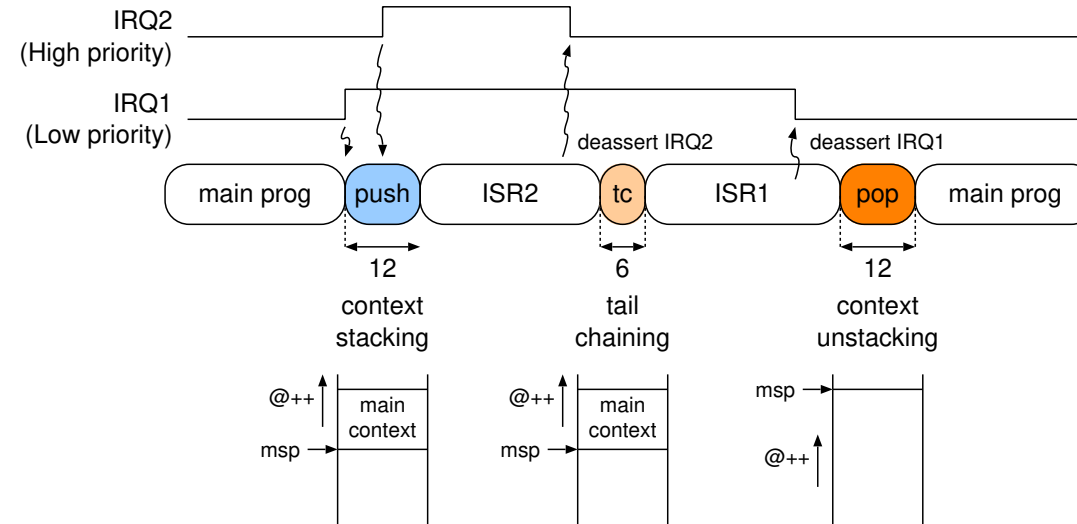
### – Prémption (ex : main stack = msp)



### – Tail chaining (ex : main stack = msp)



- Late Arrival : si une demande d'interruption IRQ2 intervient pendant l'empilement du contexte destiné à servir une demande IRQ1 moins prioritaire, c'est l'ISR2 qui prend la main, profitant de l'empilement démarré pour servir l'IRQ1.



- Pop preemption : si une demande d'interruption intervient pendant le dépilement, celui-ci est abandonné et le pointeur de pile est remis à sa position précédente. On n'a donc pas besoin de ré-empiler le contexte. Il coûte 6 cycle à partir du moment où le dépilement a été interrompu.

- Priorité/préemption/caractère synchrone ou asynchrone

Exception number <sup>a</sup>	IRQ number <sup>a</sup>	Exception type	Priority	Vector address or offset <sup>b</sup>	Activation
1	-	Reset	-3, the highest	0x00000004	Asynchronous
2	-14	NMI	-2	0x00000008	Asynchronous
3	-13	HardFault	-1	0x0000000C	-
4	-12	MemManage	Configurable <sup>c</sup>	0x00000010	Synchronous
5	-11	BusFault	Configurable <sup>c</sup>	0x00000014	Synchronous when precise, asynchronous when imprecise
6	-10	UsageFault	Configurable <sup>c</sup>	0x00000018	Synchronous
7-10	-	Reserved	-	-	-
11	-5	SVCall	Configurable <sup>c</sup>	0x0000002C	Synchronous
12-13	-	Reserved	-	-	-
14	-2	PendSV	Configurable <sup>c</sup>	0x00000038	Asynchronous
15	-1	SysTick	Configurable <sup>c</sup>	0x0000003C	Asynchronous
16	0	Interrupt (IRQ)	Configurable <sup>d</sup>	0x00000040 <sup>e</sup>	Asynchronous

a. To simplify the software layer, the CMSIS only uses IRQ numbers and therefore uses negative values for exceptions other than interrupts. The IPSR returns the Exception number, see *Interrupt Program Status Register*

b. See *Vector table* for more information.

c. See *System Handler Priority Registers*

d. See *Interrupt Priority Registers*

e. Increasing in steps of 4.

\* Le niveau de priorité est configurable pour tous les types d'exception sauf pour les exceptions HardFault, NMI et Reset pour lesquelles il est fixe.

**Plus la valeur de priorité est faible, plus l'exception est prioritaire.**

\* Les erreurs d'exécution sont reportées de manière **synchrone** à la tentative d'exécution de l'instruction précise. Il est possible de remonter à l'instruction et l'adresse qui ont causé l'erreur. On parle d'exception **precise**.

L'exception peut devenir **imprécise** si le gestionnaire d'erreur n'est pas en mesure de s'exécuter immédiatement car préempté par un handler de plus grande priorité. On perd alors les informations relatives à l'instruction et l'adresse qui ont causé l'erreur. L'exécution du gestionnaire d'erreur devient alors **asynchrone** par rapport à l'exécution de l'instruction qui a causé l'erreur.

\* L'activation d'une ISR (Interrupt) est toujours **asynchrone** vis à vis du code qui s'exécutait précédemment. Un programme ne sait pas quand il va être interrompu par un événement matériel.

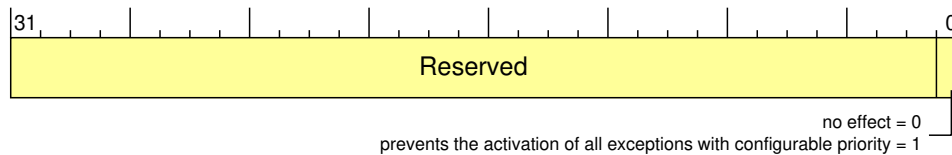


## 2.4. Configuration des interruptions/exceptions

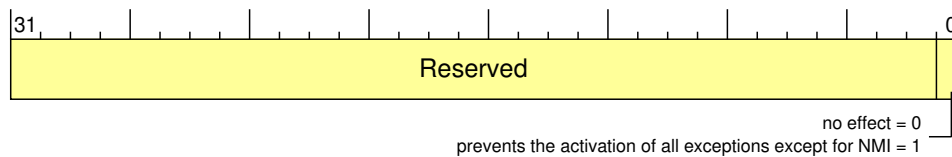
### • Autorisations

#### \* Cœur Cortex-M

PRIMASK (Priority Mask Register)



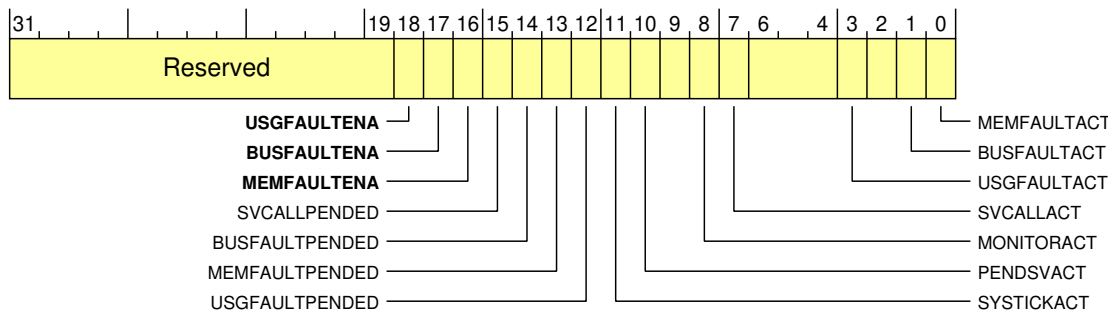
FAULTMASK (Fault Mask Register)



Name	Type	Required privilege	Reset value
PRIMASK	RW	privileged	0x00000000
FAULTMASK	RW	privileged	0x00000000

#### \* Autorisation des exceptions système : registre `_SCB->SHCSR`

SHCSR (System Handler Control and State Register)



Bits	Name	Function
18	USGFAULTENA	UsageFault enable, 1 to enable
17	BUSFAULTENA	BusFault enable, 1 to enable
16	MEMFAULTENA	MemManage enable, 1 to enable

La capture des HardFaults est toujours autorisée.

Exemple : autorisation de la capture de BusFaults : `_SCB->SHCSR |= 1<<17;`

- \* Autorisation des demandes d'interruption externes au niveau du NVIC :

Les registres du NVIC Interrupt Set Enable Registers `ISER0-ISER7` et Interrupt Clear Enable Registers `ICER0-ICER7` apparaissent comme un tableau de 256 bits, chaque bit correspondant à une entrée d'interruption (IRQ) en provenance d'un coupleur. Chaque coupleur est caractérisé par le **IRQ Number** défini dans le fichier `include/stm32f411xe.h`.

- En écriture, ces registres sont modifiés par un masque appliqué soit au registres “set”, soit au registres “clear” pour autoriser / inhiber la prise en compte des IRQ, ce qui, dans le principe, revient à (mais pas dans la pratique, puisqu'il faut casser l'accès à un tableau de 256 bits en accès par paquets de 32 bits)

$$ISER \mid = 1 \ll IRQn$$

La couche CMSIS-Core offre des fonctions pour modifier les autorisations :

---

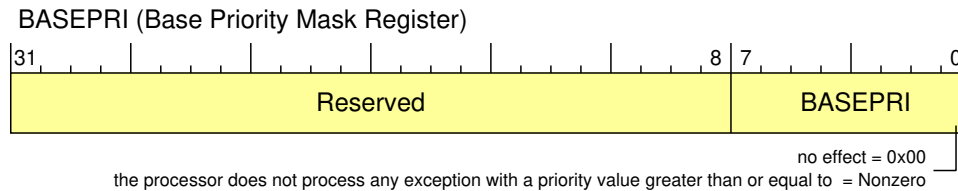
```
void NVIC_EnableIRQ(IRQn_Type IRQn);  
void NVIC_DisableIRQ(IRQn_Type IRQn);
```

---

- En lecture, ces registres fournissent une liste donnant l'état des 256 entrées possibles du NVIC.
- \* Au niveau des coupleurs périphériques : l'interface de registres des coupleurs comprend un registre qui permet d'autoriser ou inhiber les différentes sources d'interruption possibles au niveau du coupleur.

## • Définition des priorités configurables

### • Cœur Cortex-M



Name	Type	Required privilege	Reset value
BASEPRI	RW	privileged	0x00000000

- Exceptions Système : la priorité est codée dans le tableau d'octets System Handler Priority `_SCB->SHP[]`, chaque octet permettant d'encoder la priorité pour une exception de priorité configurable (à partir de l'IRQ Number = -12 → MemManage Handler). **Sur le STM32F4xx la priorité est codée sur les 4 bits de poids fort de l'octet (16 niveaux de priorités).**

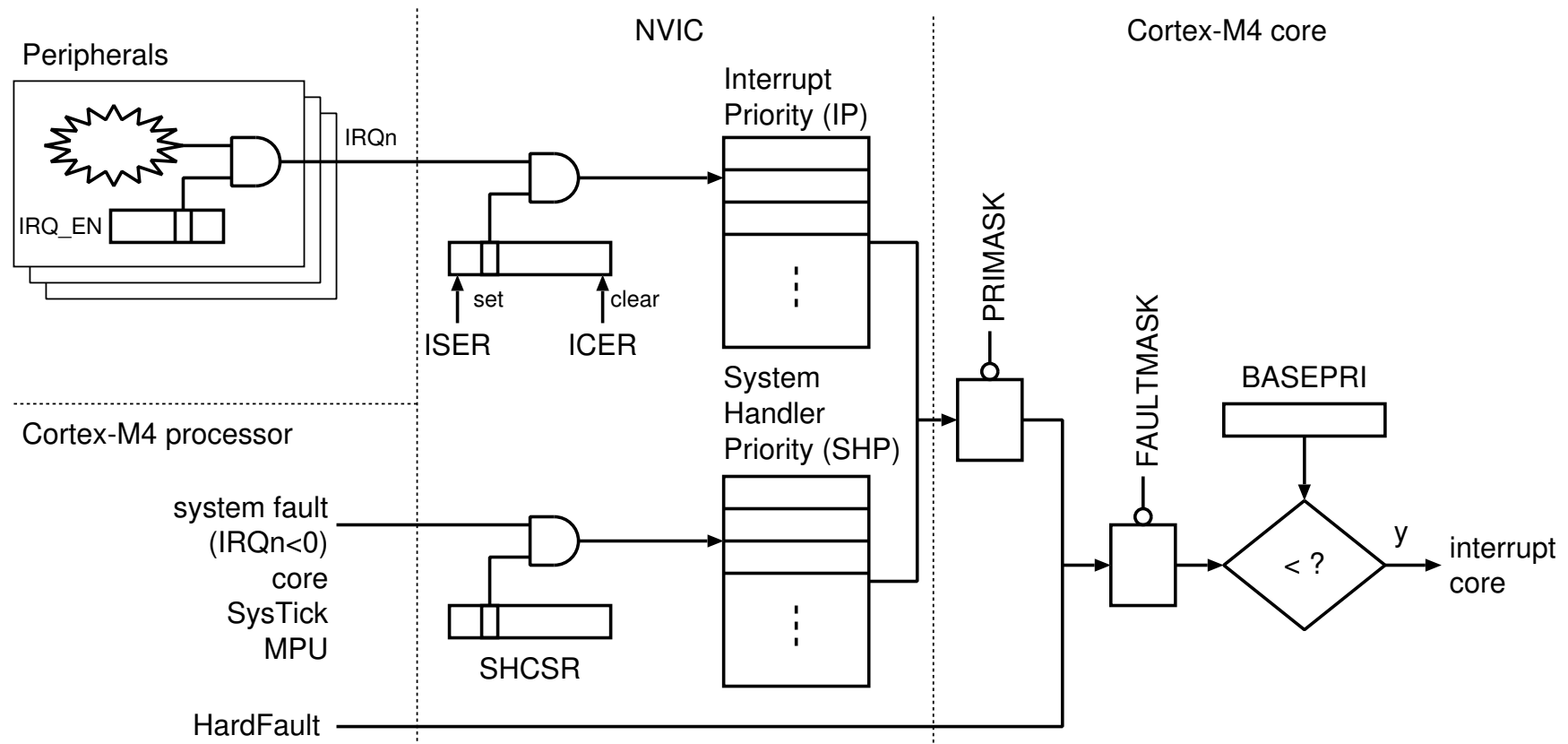
```
_SCB->SHP[(IRQn & 0xF) - 4] = priority<<4;
```

- NVIC (IRQ des coupleurs périphériques) : la priorité est codée dans le tableau d'octets Interrupt Priority `_NVIC->IP[]`, chaque octet permettant d'encoder la priorité pour une source d'interruption (à partir de l'IRQ Number  $\geq 0$ ). **Sur le STM32F4xx la priorité est codée sur les 4 bits de poids fort de l'octet (16 niveaux de priorités).**

```
_NVIC->IP[IRQn] = priority<<4;
```

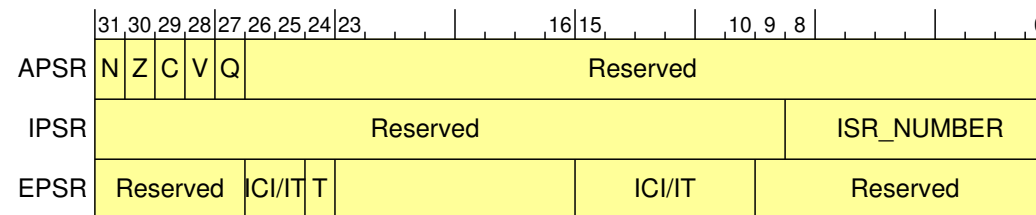
- La couche CMSIS-Core offre les fonctions suivantes pour manipuler la priorité des **exceptions système et des IRQ venant des périphériques**.

```
void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority);
uint32_t NVIC_GetPriority(IRQn_Type IRQn);
```



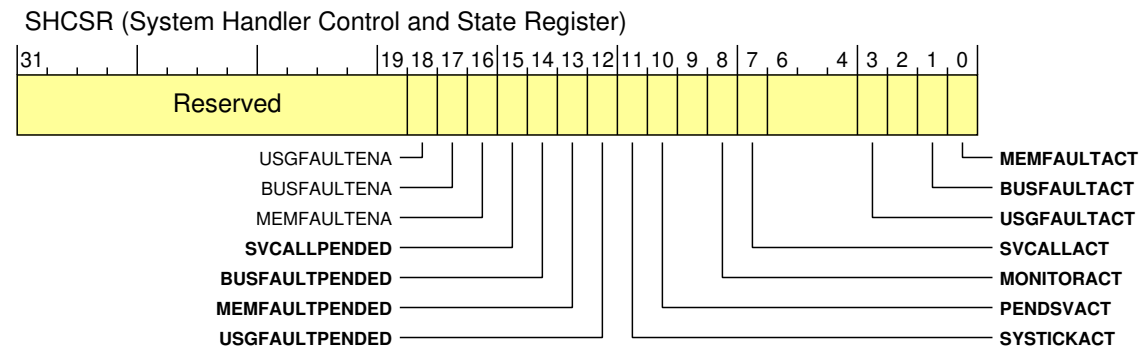
## • Informations d'état

- \* Cœur Cortex-M : lorsque le processeur exécute un gestionnaire d'exception/IRQ, on trouve dans le **registre d'état xPSR** l'ISR Number (IRQ Number + 16) de l'exception traitée.



## \* Exceptions système

- Le registre System Handler Control and State Register `_SCB->SHCSR` renseigne sur l'état **pending/active** de l'exception. L'état des bits peut être modifié manuellement.



- Le registre Configurable Fault Status Register `_SCB->CFSR` associé aux registres `_SCB->MMFAR` (adresse erreur MPU) et `_SCB->BFAR` (adresse erreur d'adressage) renseigne sur les raisons qui ont conduit à une exception système.
- Le registre HardFault Status Register `_SCB->HFSR` renseigne sur les raisons qui ont conduit à une exception HardFault.

- \* IRQ NVIC : état **pending** de l'IRQ par lecture des tableaux de bits `_NVIC->ISPR[]` / `_NVIC->ICPR[]`, ou **active** par lecture de `NVIC->IABR[]`. Possibilité de modification manuelle de l'état pending par écriture d'un masque dans les tableau Set ou Clear.

- *Remarque :*

- le bit 'pending' est automatiquement remis à 0 lorsque l'état de l'IRQ devient active,
- le bit 'active' est automatiquement remis à 0 lorsque la routine de traitement de l'IRQ se termine.

- *Interface CMSIS-Core*

---

```
uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn);  
void NVIC_SetPendingIRQ(IRQn_Type IRQn);  
void NVIC_ClearPendingIRQ(IRQn_Type IRQn);  
  
uint32_t NVIC_GetActive(IRQn_Type IRQn)
```

---

- \* Au niveau coupleur périphérique, le registre d'état permet de déterminer la source de la demande d'interruption.

## 3. Réflexion sur les interfaces de programmation

### 3.1. Organisation logicielle d'une application microcontrôleur

- Le système à microprocesseur comprenant plusieurs coupleurs, susceptibles d'être utilisés dans des contextes différents, il est utile de structurer le code pour le rendre aussi générique que possible (modularisation).
- Définition d'une API [Application Programming Interface] simple (qui ne nécessite pas la relecture de la datasheet pour pouvoir l'utiliser) pour manipuler les coupleurs.
  - Abstraction de la couche matérielle,
  - Réutilisation du code facilitée,
  - Debug plus simple car distribué sur les différents modules.
- Code spécifique à chaque coupleur décomposé entre les fichiers : `coupleur.c` et `coupleur.h` : informations relative au coupleur dans le *STM32 Reference Manual* - v2 dont l'emplacement de l'adresse de l'ISR dans la table des vecteurs et l'`irq_number` correspondant page 200, Table 37
- Définition des broches pour l'ensemble des périphériques et niveaux de priorité NVIC utilisés dans un fichier de configuration global

```
include/config.h
```

```
#define COUPLEUR_IRQ_PRIORITY 4
```

```
#define COUPLEUR_GPIO_PORT    _GPIOA
```

```
#define COUPLEUR_GPIO_PINS    PIN_2|PIN_3
```

```
#define COUPLEUR_GPIO_CFG     PIN_MODE_ALTFUNC | PIN_OPT_AFx /* config des broches */
```

### 3.2. Interfaces synchrones

- Les fonctions qui encapsulent l'accès au coupleur périphérique sont bloquantes et se terminent lorsque l'action auprès du périphérique est totalement réalisée.
- Synchronisation du code et des actions matérielles assurée,
- Simplicité d'écriture dans le cas où l'accès au périphérique est réalisé par scrutation du registre d'état.
- Méthode intéressante lorsqu'on échange des données.
- Exemple de structuration du code

lib/coupleur.h

```
#ifndef _COUPLEUR_H_
#define _COUPLEUR_H_

#include "include/board.h"           // Paramètres de la carte

void coupleur_init(COUPLEUR *c, <PE>); // prototype de la fonction d'initialisation d'un coupleur

int coupleur_read(COUPLEUR *c, uint8_t *buffer, size_t n); // lecture synchrone
int coupleur_write(COUPLEUR *c, uint8_t *buffer, size_t n); // écriture synchrone

#endif
```



lib/coupleur.c

```
#include "coupleur.h"
#include "include/io.h"                // pour io_configure

void coupleur_init(COUPLEUR *c, <PE>)
{
    // Configuration des broches si le coupleur a besoin d'un accès externe
    io_configure(COUPLEUR_GPIO_PORT, COUPLEUR_GPIO_PINS, COUPLEUR_GPIO_CFG, NULL);

    c-> ....                          // Configuration du coupleur (cf datasheet)
                                      // c : pointeur passé en paramètre
}

int coupleur_read(COUPLEUR *c, uint8_t *buffer, size_t n)    // lecture synchrone
{
    for (in i=0; i<n; i++) {
        while (c->SR & COUPLEUR_DATA_REG_EMPTY); // wait until we receive something
        buffer[i] = c->DR;                       // read data reg
    }
    return n;
}

/* ... */
```

### 3.3. Interfaces asynchrones

- Les fonctions qui encapsulent l'accès au coupleur périphérique se contentent de configurer l'action à réaliser au niveau du coupleur périphérique, mais n'attendent pas sa réalisation complète. Le coupleur matérielle réalise les actions matérielles demandées en parallèle avec l'exécution du code principal. Eventuellement une interruption peut signaler lorsque les actions ont été menées à leur terme.
- Communication par interruption et DMA libère le processeur pour qu'il puisse continuer à exécuter le code principal.
- Programmation événementielle, échange plus fluide avec l'ensemble des périphériques.
- Exemple de structuration du code

lib/coupleur.h

```
#ifndef _COUPLEUR_H_
#define _COUPLEUR_H_

#include "include/board.h"                // Paramètres de la carte

typedef void (* coupleur_cb_t)(void);      // pointeur sur fonction de rappel

void coupleur_init(COUPLEUR *c, <PE>, coupleur_cb_t cb); // prototype de la fonction d'initialisation d'un coupleur

// Autres prototypes des fonctions liées au coupleur
#endif
```

lib/coupleur.c

```
#include "coupleur.h"
#include "include/io.h"                                // pour io_configure

static coupleur_cb_t callback = 0;                    // par défaut, pas de fonction de rappel (callback) attachée
                                                        // au coupleur

void coupleur_isr(void)
{
    // Désarmement de la demande d'interruption
    _Coupleur-> ....                                //cf mécanisme de désarmement du coupleur

    if (callback) callback();                          // appel si fonction de rappel si définie
}

void coupleur_init(COUPLEUR *pC, <PE>, coupleur_cb_t cb)
{
    // Configuration des broches si le coupleur a besoin d'un accès externe
    io_configure(COUPLEUR_GPIO_PORT, COUPLEUR_GPIO_PINS, COUPLEUR_GPIO_CFG, NULL);

    pC-> ....                                          // Configuration du coupleur (cf datasheet)
                                                        // pC : pointeur passé en paramètre

    if (cb) {
        callback = cb;                                // lien fonction de rappel <-> fonction à appeler
        NVIC_SetPriority(<irq_number>, COUPLEUR_IRQ_PRIORITY); // <irq_number> -> stm32f411xe.h
        NVIC_EnableIRQ(<irq_number>);                 // <irq_number> -> stm32f411xe.h
    }
}
```

`src/main.c`

```
#include "include/board.h"
#include "lib/coupleur.h"

static void coupleur_cb(void)                // fonction à appeler (callback) suite à une irq
{
    // code de la fonction de traitement de l'interruption d'un coupleur
}

int main(void)
{
    // Initialisation d'un coupleur
    coupleur_init(_Coupleur, <PE>, coupleur_cb); // PE : liste des paramètres d'appel
                                                // _Coupleur : pointeur global sur une structure COUPLEUR
                                                // IRQ autorisées par défaut

    // code de l'application
    while (1) { /* ... */ }
}
```

Le pointeur global `_Coupleur` est référencé de la manière suivante dans `include/board.h`

```
extern COUPLEUR * const _Coupleur;
```

Il est défini de la manière suivante (dans `startup/stm32f411_periph.c`)

```
COUPLEUR * const _Coupleur = (COUPLEUR*)0x....;
```

### programme principal

```
main() [mode Thread]
{
    ...
    initialisation coupleur#n
    ...

    code de l'application
    while(1) {
        /* ... */
    }

    callback() [mode Handler]
    {
        réaction à l'évènement
        matériel
    }
}
```

### coupleur#n.c

```
coupleur#n_init(...) [mode Thread]
{
    configuration du coupleur#n
    ...
    callback <- fonction à appeler
    installation de l'irq
}

coupleur#n_isr() [mode Handler]
{
    désarmement de l'irq
    appel fonction callback
}

...
Autres fonctions du coupleur #n
```

CMSIS  
include/cmsis/core\_cm4.h

