

Ministère de l'Enseignement Supérieur  
et de la Recherche Scientifique

Université de Carthage

Ecole Nationale d'Ingénieurs de Carthage



المدرسة الوطنية للمهندسين بقرطاج

Ecole Nationale d'Ingénieurs de Carthage

وزارة التعليم العالي و البحث العلمي

جامعة قرطاج

المدرسة الوطنية للمهندسين بقرطاج

# **Systèmes embarqués**

## **Niveau: 2 ING INFO**

**Année Universitaire**  
**2020/2021**



# Plan du module



1

Introduction aux systèmes embarqués

2

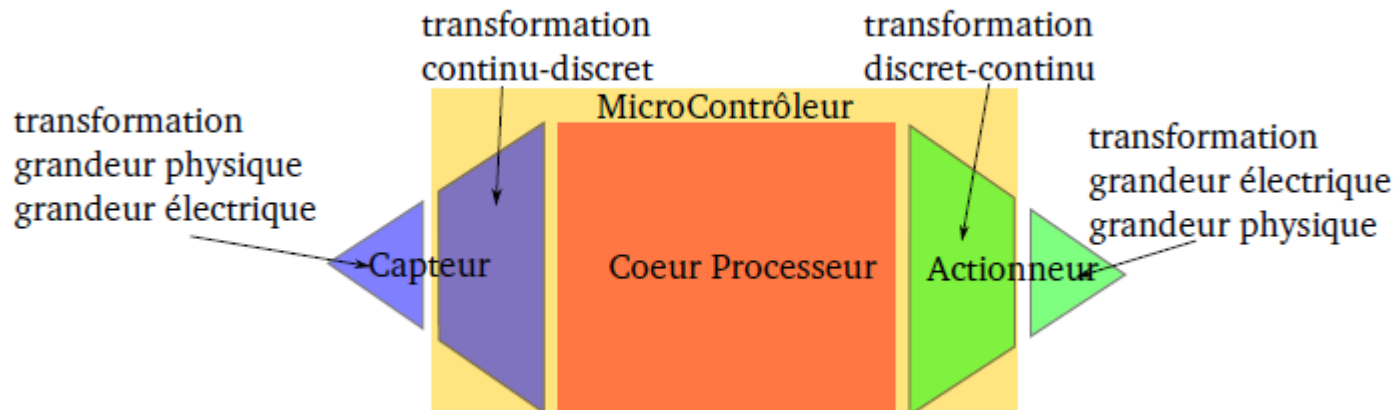
**Cible logicielle: Les microcontrôleurs STM32**

3

Cible matérielle: Langage VHDL et carte FPGA

# Les microcontrôleurs

- ❑ Interaction forte avec l'environnement
- ❑ Concevoir un composant qui associe 3 aspects
  - ✓ Acquisition
  - ✓ Traitement
  - ✓ Commande
- ❑ Premier SoC (System on Chip ou système sur puce)
- ❑ Dans la pratique : faible capacité de traitement
- ❑ Dédié plutôt contrôle



# Les microcontrôleurs STM32

- ARM ne fabrique pas de puce. Elle vend des architectures sous licence de propriété intellectuelle aux concepteurs
- ST Microelectronics développe la famille STM32 intégrant des cœurs CORTEX M.
- ST est leader mondial dans le domaine des microcontrôleurs 32 bits destinés aux systèmes embarqués.



## Carte STM32F4 Discovery

- La carte STM32F4 Discovery permet aux utilisateurs de développer des applications avec un microcontrôleur STM32F4 muni d'un processeur ARM Cortex-M4 32 bits
- Processeur RISC
- Architecture Harvard : bus instructions et données séparées
- Supporte uniquement le jeu d'instruction Thumb2
- consommation réduite de l'énergie
- Fournit une vaste gamme de périphériques allant du simple GPIO (port d'entrée-sortie généraliste) et interface de communication série synchrone (SPI) ou asynchrone (RS232) aux interfaces aussi complexes que l'USB, Ethernet ou HDMI



# STM32F4-Discovery



releasing your creativity

STM32 F4

STMicroelectronics

Everything you need to  
discover our STM32F4  
32-bit ARM Cortex™-M4  
based MCUs featuring:

- Evaluation board
- Embedded ST-LINK/V2
- USB interface for debugging and programming
- Numerous examples available on [www.st.com](http://www.st.com)

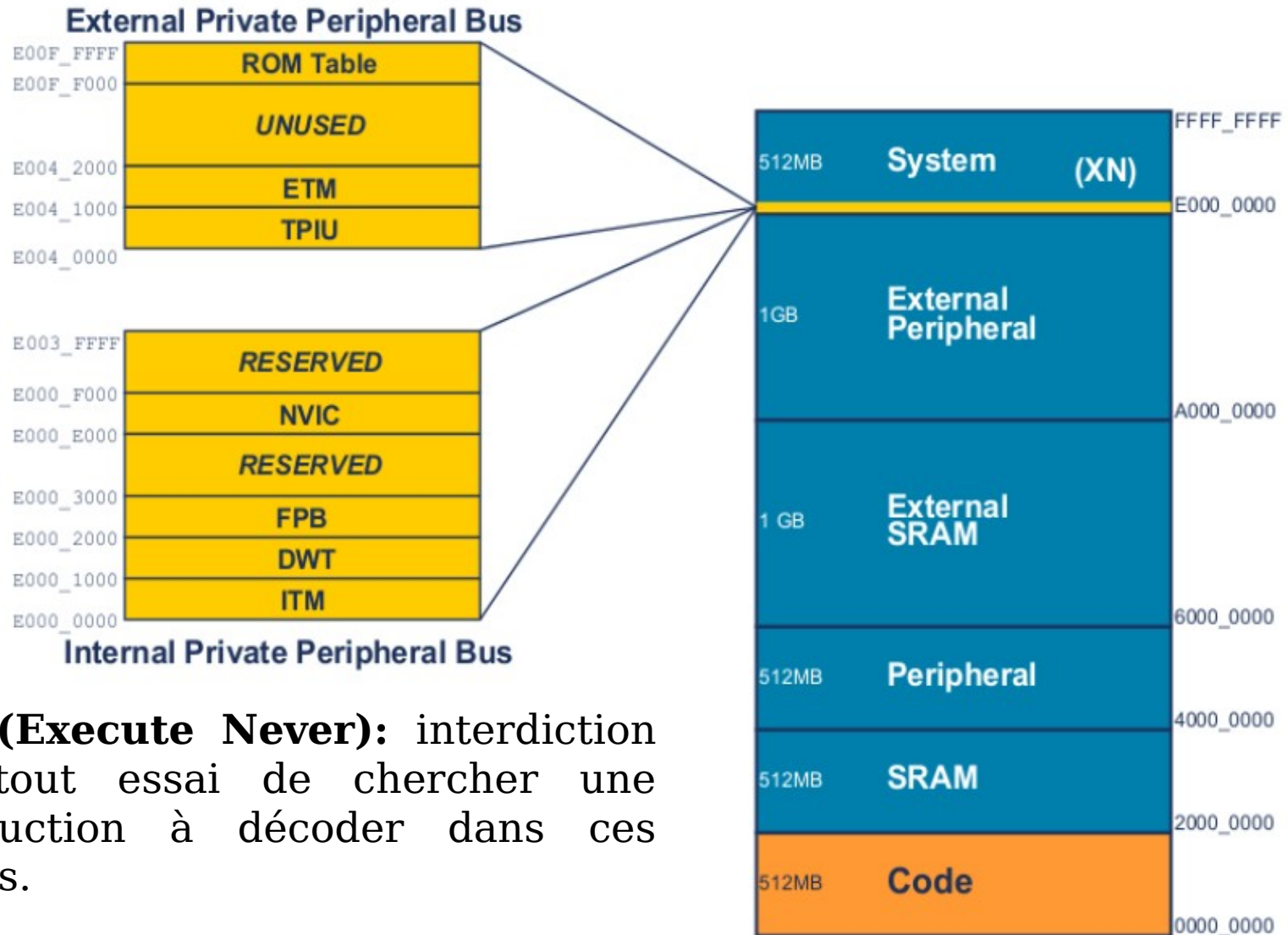


[www.st.com/stm32f4-discovery](http://www.st.com/stm32f4-discovery)

## Rappel: Cortex M3 et M4

- ❖ Les cœurs Cortex M3 et Cortex M4 sont semblables seulement que Cortex M4 ajoute des instructions DSP (Digital Signal Processing)
- ❖ Les deux processeurs sont utilisés dans des applications embarquées qui nécessitent une réponse rapide aux interruptions.
- ❖ Ils peuvent adresser 32 bits de mémoire totale (4 Go) selon une Architecture de Harvard .
  - Le 4 Go adressables par un cœur ARM Cortex M4 sont réparties entre la mémoire non-volatile (FLASH), la mémoire volatile interne et externe (RAM), les périphériques, des adresses réservées aux périphériques du vendeur et plus.
- ❖ Par défaut, un mot de mémoire est sur 8 bits.
- ❖ Ce cœur est un cœur Memory Mapped Input Output: les

# Memory Mapping



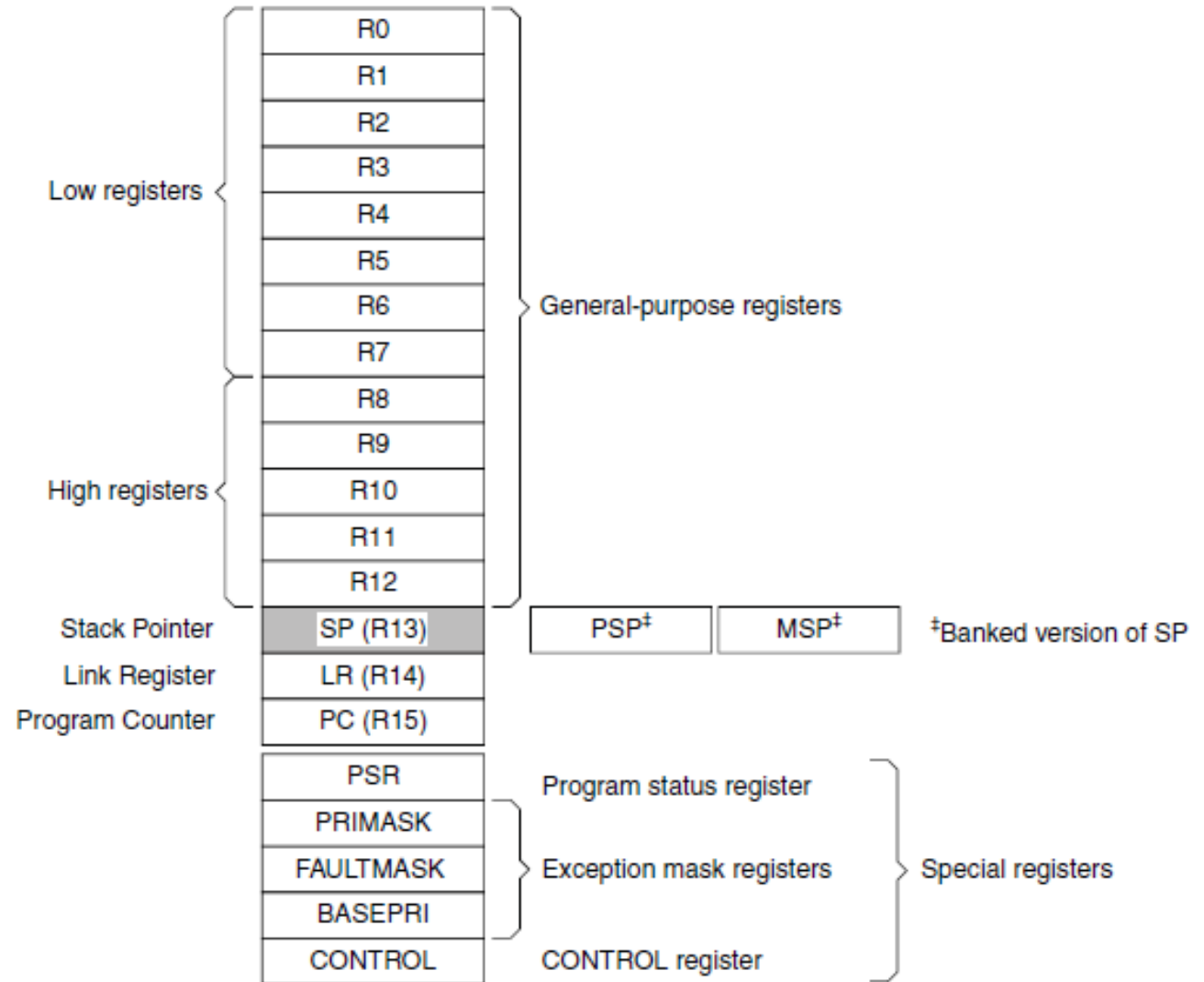
**XN (Execute Never):** interdiction de tout essai de chercher une instruction à décoder dans ces zones.



# Banc de registres

## Architecture 32 bits:

- Bus de données sur 32 bits
- Registres internes sur 32 bits



## ➤ **R0-R12 : registres généraux**

- ✓ Les registres bas sont des registres disponibles pour toutes les instructions, encodées sur 16 ou 32 bits.
- ✓ Les registres hauts disponibles uniquement pour les instructions 32 bits.
- ✓ Il faut 3 bits pour désigner un registre bas et 4 bits pour désigner tous les registres de R0 à R15.

## ➤ **R13 : pointeur de pile (SP)**

- ✓ Il indique l'adresse de la mémoire étant le haut de la pile
- ✓ Registre doublé (banked) soit MSP (SP\_main) soit PSP (SP\_Process)
- ✓ Choix automatique du microprocesseur de l'un des registres de la banque en fonction du mode d'opération et des registres de

Processor mode	Used to execute	Privilege level for software execution	Stack used
Thread	Applications	Privileged or unprivileged <sup>1</sup>	Main stack or process stack
Handler	Exception handlers	Always privileged	Main stack

## ➤ **R14 : Link register (LR)**

il est automatiquement mis à jour lors d'appel de fonction. Il contient l'adresse de retour de la fonction (l'adresse de la fonction à être exécutée lorsque l'exécution de la fonction sera terminée)

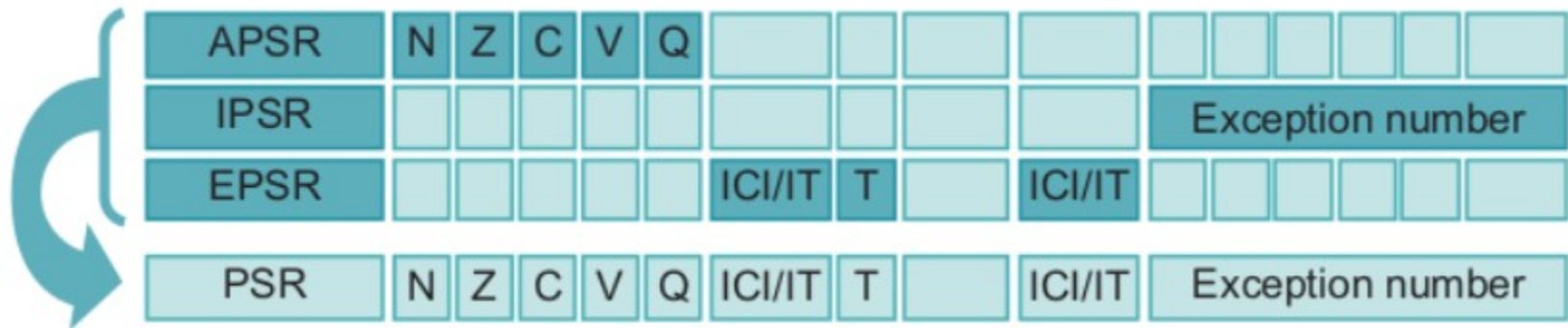
➤ **R15: registre PC ou compteur de programme**

Il indique l'adresse des instructions à être exécutées. Changer R15 équivaut à faire un saut à travers le code du programme.


➤ **Registre d'état xPSR (Program Status Register)**

✓ Il indique l'état d'exécution d'un programme: la valeur des drapeaux de l'ALU (APSR), l'état d'exécution (EPSR) ou le numéro de l'interruption en cours (IPSR)

✓ Peut être vu comme un tout (PSR) ou comme trois registres séparés (virtuellement), chaque registre met en relief une partie des bits

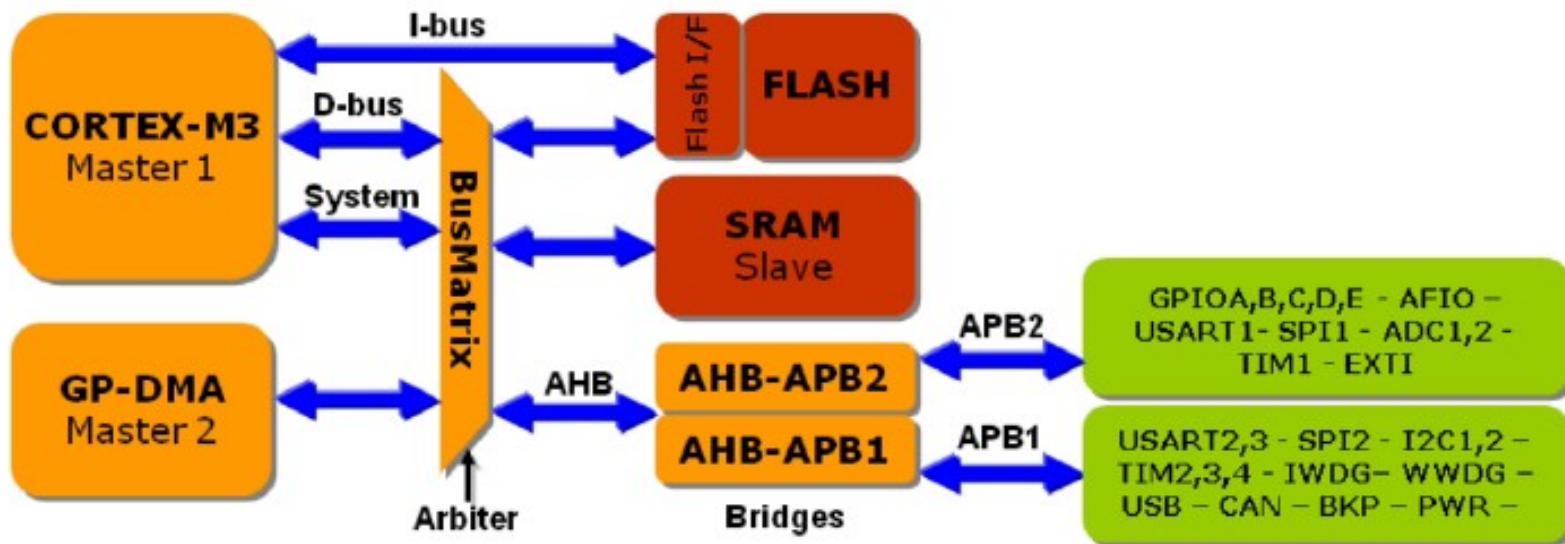


➤ xPSR ne fait pas partie des 16 registres pouvant être adressés par les instructions arithmétiques du microprocesseur. Il faut utiliser des instructions spéciales pour lire et écrire ce registre.

- 
- APSR contient 5 drapeaux (flags):
    - ✓ N indique si le résultat de l'ALU est négatif,
    - ✓ Z indique si le résultat est nul,
    - ✓ C (Carry) indique si l'ALU a produit une retenue lors d'une addition,
    - ✓ V (Overflow) indique que le résultat déborde des valeurs permises (dépassement)
    - ✓ Q (Sticky Saturation) indique que le résultat a saturé (exemple: minimums et maximums atteints)
  - EPSR indique si le microprocesseur exécute actuellement une instruction If-Then ou une instruction Load/Store Multiple Registre.
  - ISPR indique le numéro de l'interruption en cours.

## STM32: Architecture Système

- Architecture Harvard ( bus instructions + bus données)
- Deux maitres : CPU et DMA
- Bus hiérarchique pour une meilleure



et



## Les entrées/sorties

- Les fonctions d'entrée/sortie sont implémentées (comme tout système basé sur ARM) en utilisant une combinaison de registres mappés en mémoire et des entrées d'interruptions. Certaines fonctions peuvent aussi utiliser l'accès direct à la mémoire via le DMA
- Un périphérique tel que le GPIO, USART, Timer, ... contient un nombre de registres internes. Les registres peuvent être classés comme
  - ✓ En lecture seule (R)
  - ✓ En écriture seule (W)
  - ✓ En lecture/ écriture (R/W)

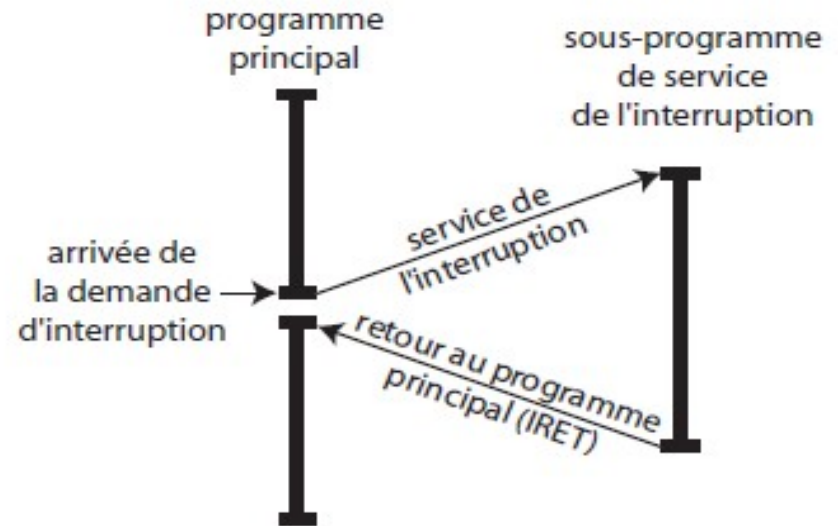
# Les interruptions

## ❖ Exemples

- ✓ appui sur un bouton
- ✓ déclenchement d'un capteur
- ✓ arrivée d'un message sur un canal de communication
- ✓ horloge temps réel (timer) informant qu'un temps est écoulé; par exemple "1 ms s'est écoulé"

## interruption:

Mécanisme qui permet d'interrompre l'exécution d'un programme en cours à l'arrivée d'un événement, d'exécuter du code correspondant puis de reprendre le programme interrompu



## Les interruptions sur les cœurs ARM

Pour les cœurs ARM:

✓ **une *interruption*** est spécifiquement liée à un événement matériel

ex : arrivée d'un message du bloc UART, front montant sur le port GPIO, signal du Timer...

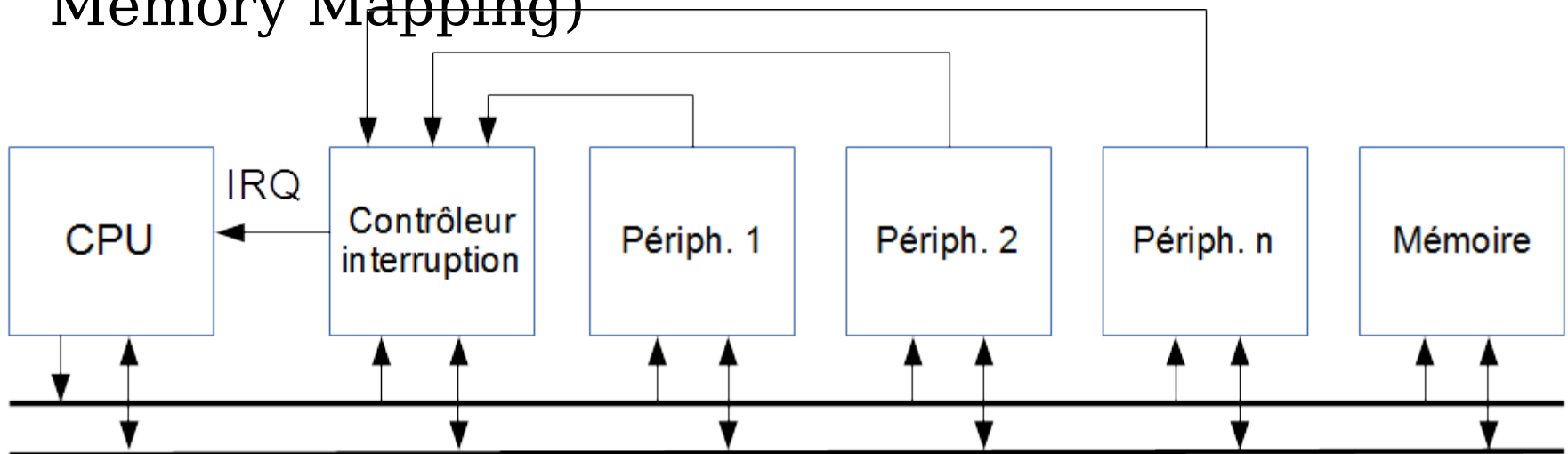
✓ **une *exception*** peut être liée à un événement logiciel


ex : division par zéro, erreur d'accès mémoire

Pour certains concepteurs ils considèrent les exceptions comme interruptions logicielles

## Les interruptions sur Cortex M4

- de 32 à 256 sources possibles d'interruption (dépend des périphériques installés)
- Il existe un module contrôleur d'interruption appelé le Nested Vectored Interrupt Controller (NVIC) qui est un périphérique adressé entre les adresses 0xE000E000 et 0xE000EFFF (voir Memory Mapping)



- 
- ❖ **Le rôle du Contrôleur d'interruption NVIC** est de:
    - ✓ Recevoir les requêtes d'interruptions **matérielles** venant des périphériques
    - ✓ Gérer les séquences d'interruptions, la priorité de chaque interruption...
    - ✓ Émettre une requête d'interruption au processeur IRQ
  - ❖ Le processeur utilise cette IRQ qui contient la nature de l'interruption, sa priorité (...) pour rechercher dans une table en mémoire l'adresse du sous-programme d'interruption à exécuter appelé routine d'interruption (Interrupt service routine ISR)
  - ❖ La table est appelée **vecteur d'interruption**



## Vecteur d'interruption

➤ Table en mémoire à une adresse bien définie

➤ Chaque entrée dans la table correspond à une interruption. Elle contient l'adresse de la routine d'interruption (Interrupt service routine ISR)

➤ Dans les **Cortex-M4**, cette table est stocké à l'adresse 0x4 de la mémoire FLASH

- l'interruption 0 s'appelle Reset (appui sur bouton, reset software, allumage)

- table[0] contient l'adresse du code où sauter en cas de reset

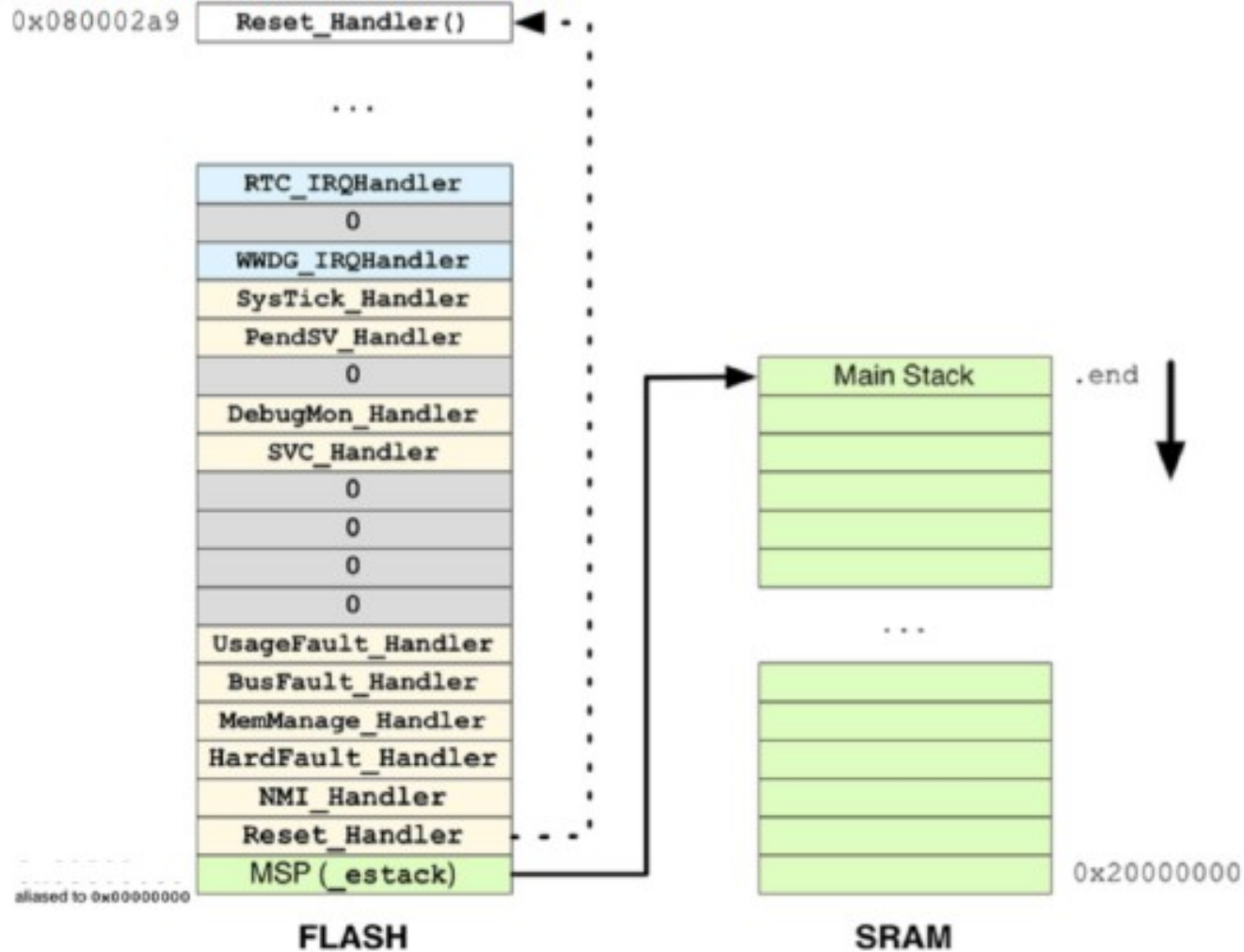
**RQ:** la case 0x0 contient l'adresse de début de la pile, copié dans R13 (ou SP) au démarrage.

Address		Vector #
0x40 + 4*N	External N	16 + N
...	...	...
0x40	External 0	16
0x3C	SysTick	15
0x38	PendSV	14
0x34	Reserved	13
0x30	Debug Monitor	12
0x2C	SVC	11
0x1C to 0x28	Reserved (x4)	7-10
0x18	Usage Fault	6
0x14	Bus Fault	5
0x10	Mem Manage Fault	4
0x0C	Hard Fault	3
0x08	NMI	2
0x04	Reset	1
0x00	Initial Main SP	N/A

## Modèle d'exécution en cas d'interruption

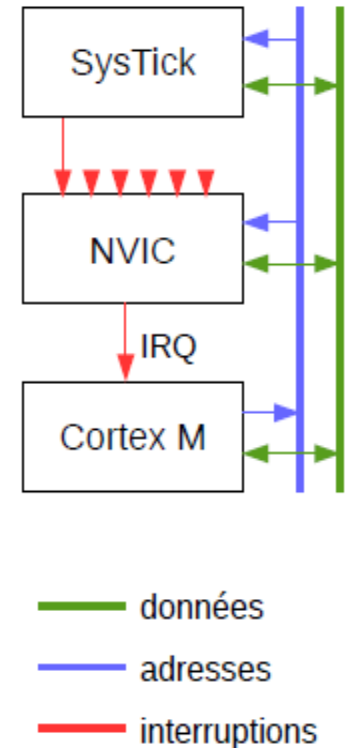
- ❖ si une interruption *i* est levée par le NVIC :
  - 1) Enregistrer certains registres internes ainsi que le PC dans une zone mémoire SRAM appelée pile et dont l'adresse se trouve dans SP
  - 2) lire dans le *vecteur d'interruption* l'adresse stockée à la case
  - 3) écrire dans PC cette adresse
  - 4) exécuter le code correspondant par avancement du PC
  
- ❖ À la fin de l'exécution du code d'une interruption:
  - 1) Reprendre les valeurs des registres et du PC à partir de la pile
  - 2) Exécution de la continuité du programme initial20 par avancement du PC

## exemple: interruption Reset

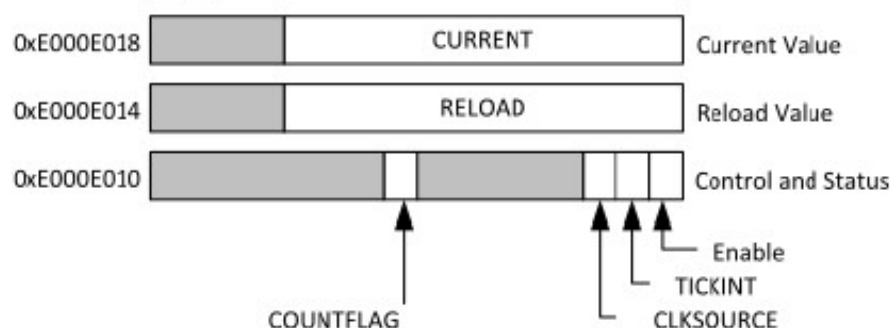


# Programmation d'un périphérique interne: SysTick

- Etroitement lié au processeur Cortex M (tout comme le contrôleur d'interruption NVIC)
- Contient un (dé)compteur sur 24 bits
  - ✓ Peut être activé ou désactivé
  - ✓ Part d'une valeur initiale programmable
  - ✓ Cadencé par l'horloge du processeur ou une horloge externe
  - ✓ Lorsque le compteur atteint 0, le SysTick peut générer une interruption au processeur
  - ✓ A tout moment, on peut lire la valeur courante du compteur



# Registres du SysTick



## • Registres

- **CURRENT** : Contient la valeur courante du compteur. À chaque coups d'horloge, le registre est décrémenté de 1
- **RELOAD** : contient la valeur initiale du compteur. Lorsque CURRENT arrive à 0, il est réinitialisé à la valeur de RELOAD
- **CONFIG** : registre de configuration/Status. Contient :
  - Un bit pour activer/désactiver le TIMER
  - Un bit pour activer/désactiver la génération d'interruption
  - Un bit pour indiquer la nature de l'horloge (interne/externe)
  - Un bit qui indique le passage à zéro

Registre	Adresse	Type
CONTROL	0xE000E010	R/W
RELOAD	0xE000E014	R/W
CURRENT	0xE000E018	R/W



# Registres du SysTick

## RELOAD

Bits	Name	Type	Reset Value	Description
23:0	RELOAD	R/W	0	Reload value when timer reaches 0

## CURRENT

Bits	Name	Type	Reset Value	Description
23:0	CURRENT	R/Wc	0	Read to return current value of the timer. Write to clear counter to 0. Clearing of current value also clears COUNTFLAG in SYSTICK Control and Status Register

## CONTROL

Bits	Name	Type	Reset Value	Description
16	COUNTFLAG	R	0	Read as 1 if counter reaches 0 since last time this register is read; clear to 0 automatically when read or when current counter value is cleared
2	CLKSOURCE	R/W	0	0 = External reference clock (STCLK) 1 = Use core clock
1	TICKINT	R/W	0	1 = Enable SYSTICK interrupt generation when SYSTICK timer reaches 0 0 = Do not generate interrupt
0	ENABLE	R/W	0	SYSTICK timer enable

- **Séquence typique de configuration:**
  - ✓ Désactiver le périphérique (écrire 0 dans CONTROL)
  - ✓ Initialiser RELOAD avec la valeur souhaitée (nombre de cycles -1)
  - ✓ Ecrire une valeur quelconque a CURRENT pour le mettre a zéro ainsi que le bit COUNTFLAG du registre CONTROL
  - ✓ Ecrire dans le registre CONTROL pour activer le SysTick

**Exercice** : écrire un programme assembleur qui permet de configurer et lancer le SysTick. Le programme doit ensuite attendre que le compteur passe à zéro pour continuer.

- Utiliser l'horloge interne
- Désactiver l'interruption du systick
- Initialiser le compteur à la valeur 1000

```

LDR R1, =0
LDR R0, =0xE000E010
STR R1, [R0]
LDR R1, =1000
LDR R0, =0xE000E014
STR R1, [R0]
LDR R1, =0
LDR R0, =0xE000E018
STR R1, [R0]
LDR R1, =5
LDR R0, =0xE000E010
STR R1, [R0]

```

Test

```

LDR R1, [R0]
LDR R2, =0x00010000
ANDS R1, R2
BE Test

```

Test

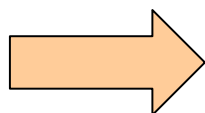
```

LDR R1, =0
LDR R0, =0xE000E010
STR R1, [R0], #4
LDR R1, =1000
STR R1, [R0], #4
LDR R1, =0
STR R1, [R0]
LDR R1, =5
SUB R0, R0, #8
STR R1, [R0]

LDR R1, [R0]
LDR R2, =0x00010000
ANDS R1, R2
BE Test

```

**Version plus optimisée**



Programme en langage

C

## Avantages du langage C

- **universel** : C n'est pas orienté vers un domaine d'applications spéciales comme SQL par exemple...
- **compact** : C est basé sur un noyau de fonctions et d'opérateurs limités qui permet la formulation d'expressions simples mais efficaces.
- **moderne** : C est un langage structuré, déclaratif et récursif. Il offre des structures de contrôle et de déclaration comme le langage Pascal.
- **près de la machine** : C offre des opérateurs qui sont très proches de ceux du langage machine (manipulations de bits, pointeurs...). C'est un atout essentiel pour la programmation des systèmes embarqués.
- **rapide** : C permet de développer des programmes concis et rapides.

- **indépendant de la machine** : C est un langage près de la machine (microprocesseur) mais il peut être utilisé sur n'importe quel système ayant un compilateur C.
- **portable** : En respectant le standard ANSI-C, il est possible d'utiliser (théoriquement) le même programme sur tout autre système (autre hardware, autre système d'exploitation), simplement en le recompilant. Il convient néanmoins de faire attention dans le cas d'un système embarqué qui n'inclut pas toujours un système d'exploitation...

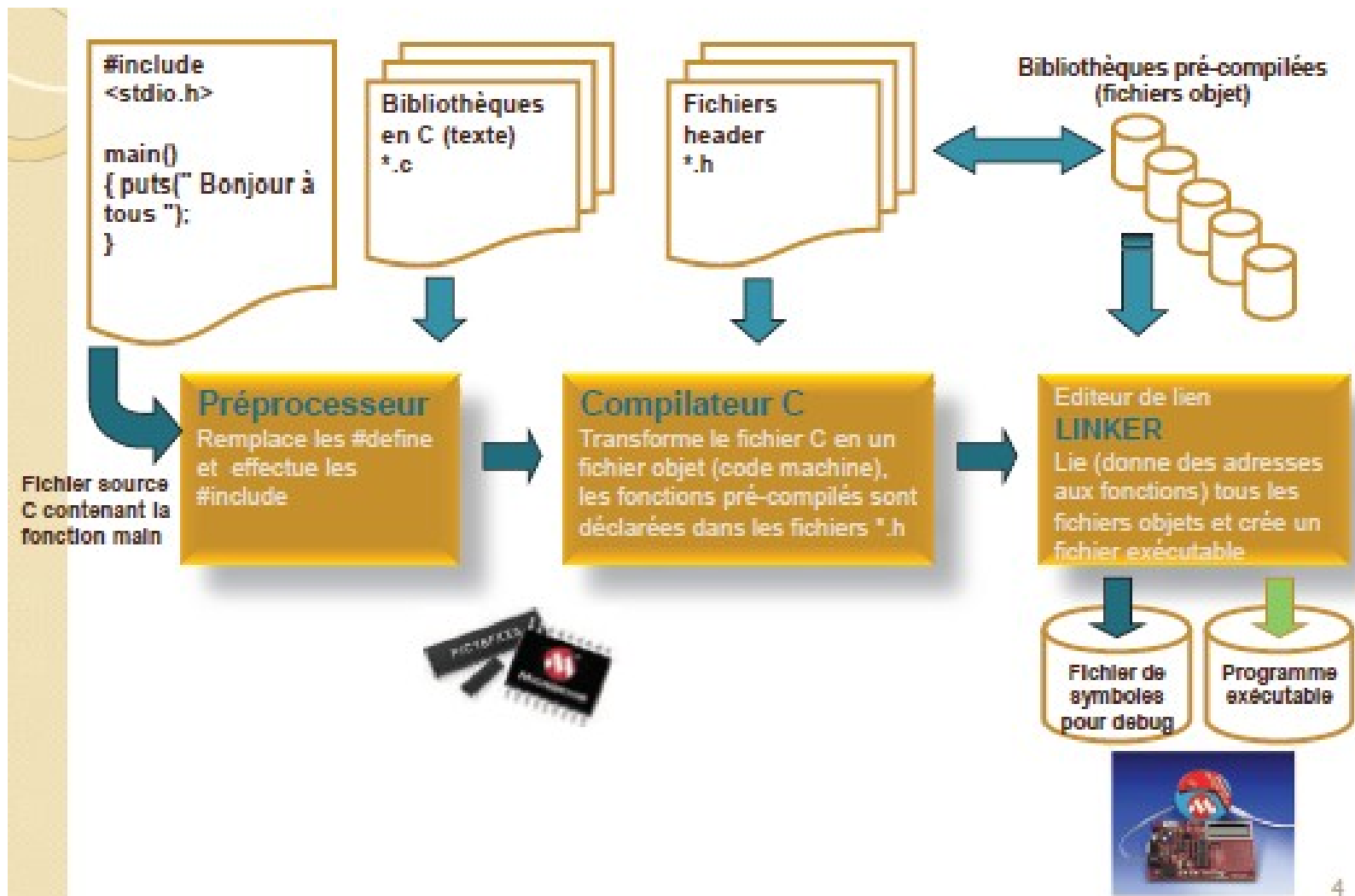


## Généralités langages

- Le langage naturel de la machine n'utilise que deux symboles (0 et 1) : c'est le langage machine. Le langage machine ne comprend et n'exécute qu'une suite de 0 et 1 structurée sous forme d'octets (8 bits).
- Le langage assembleur permet à l'humain de générer cette suite de 0 et 1 à partir d'un fichier source compréhensible qui utilise des instructions assembleur. Le programme assembleur traduit le source assembleur en **code objet** (suite de 0 et 1) exécutable par la machine.
- Le langage C va opérer la même chose que précédemment mais avec plus de lisibilité encore.
- Un programme C est un texte écrit avec un éditeur de texte respectant une certaine syntaxe et stocké sous forme d'un ou plusieurs fichiers (généralement avec l'extension .c).

- A l'opposé du langage assembleur, les instructions du langage C sont obligatoirement encapsulées dans des fonctions
- il existe une fonction privilégiée appelée *main()* qui est le point d'entrée de tout programme C .
- Dans le cas d'un système embarqué sans système d'exploitation, la fonction spécifique *main()* perd son sens. Le point d'entrée du programme sera celui spécifié (en mémoire ROM) dans la table des vecteurs pour l'initialisation du compteur programme au RESET...

# FLOT DE DONNÉES D'UN COMPILATEUR C



# Premier programme

```
#define pi 3.14
#include <stdio.h>
float d,c;

int main()
{
    d=2.0 ;
    c=pi*d ;
    puts("bonjour à tous\n");
    printf("la circonférence est %f m\n",c);
}
```

Equivalence : Le pre-processeur remplacera tous les pi par 3.14

Header de la bibliothèque standard in/out. (pour printf)

Déclaration de deux variables réelles

Entrée du programme principal

Envoie une chaîne de caractères sur le périphérique de sortie

Printf affiche des chaînes formatées, ici c est affiché sous format réel.

# COMPILATION ET ÉDITION DES LIENS

- Chaque fichier source (ou texte) est appelé *module et est composé* :
  - ✓ des définitions de fonctions
  - ✓ des définitions de variables
  - ✓ des déclarations de variables et fonctions externes
  - ✓ des directives pour le préprocesseur (lignes commençant par #) de définitions de types ou de constantes (struct, union...).

**C'est la base de la programmation modulaire**

- Pour compiler correctement un fichier, le compilateur a besoin d'informations concernant les déclarations des structures de données et de variables externes ainsi que de l'aspect (on dira *prototype* ou *signature*) des *fonctions prédéfinies*.
- Un programme se compose donc :
  - ✓ De structures de données.
  - ✓ D'algorithmes utilisant ces structures de données.
- Ces informations sont contenues dans des fichiers *header* avec l'extension *.h*

# TYPES DE DONNÉES

**int** pour tout ce qui est comptable (compteurs de boucle, variables, évènements...)


**char** pour les caractères et les chaînes

**float** pour tout ce qui est mesurable (secondes, distance, température)

**uint32\_t** pour les manipulations de bits, spécialement les registres 32 bits

Les types **stdint.h** sont utilisés pour l'enregistrement et le fonctionnement des données explicitement au niveau bit





<b>C type</b>	<b>stdint. h type</b>	<b>Bits</b>	<b>Sign</b>	<b>Range</b>
char	uint8_t	8	Unsigned	0 .. 255
signed char	int8_t	8	Signed	-128 .. 127
unsigned short	uint16_t	16	Unsigned	0 .. 65,535
short	int16_t	16	Signed	-32,768 .. 32,767
unsigned int	uint32_t	32	Unsigned	0 .. 4,294,967,295
int	int32_t	32	Signed	-2,147,483,648 .. 2,147,483,647
unsigned long long	uint64_t	64	Unsigned	0 .. 18,446,744,073,709,551,615
long long	int64_t	64	Signed	-9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807

## Qualificatif volatile

- En qualifiant par volatile le type d'une variable, le programmeur prévient le compilateur que cette variable peut être modifiée par un moyen extérieur au programme.
- Ceci se produit lorsqu'on interagit avec des parties matérielles de la machine : entrées-sorties généralement.
- Lorsqu'une variable est de type volatile le compilateur ne doit pas procéder aux optimisations qu'il réalise sur les variables normales.

## Les constantes et les variables

- « **const** » définit une constante; donc à une adresse fixe
- Les variables globales déclarées en dehors d'une fonction et sont toujours statiques (adresse fixe). Au démarrage du programme, le système alloue l'espace nécessaire et effectue les initialisations. Tout cet espace sera rendu au système lors de la terminaison du programme.
- Les variables locales déclarées dans une fonction, soit à une adresse statique (exemple: `static char c`), ou dans une pile LIFO donc dynamique. L'espace nécessaire est alloué lors de l'activation de la fonction ou du bloc correspondant. Il est restitué au système à la fin du bloc ou de la fonction.
- Le qualificateur **extern** placé devant la déclaration d'une variable ou d'une fonction informe que celui-ci sera défini plus loin dans le même module ou dans un autre module.

# MANIPULATIONS DES BITS

- Opérateurs binaires:

- Et bit par bit :  $\&$
- OU bit par bit :  $|$
- XOR bit par bit :  $\wedge$

- Opérateur unaire

- Inversion de bits (complément à 1) :  $\sim$

- Opérateurs de décalage

- Décalage à gauche :  $\ll$
- Décalage à droite :  $\gg$

- Génération de nombres particuliers :

- bit  $n$  à 1, le reste à 0 :  $1 \ll n$ 
  - Exemple  $10000000 = 1 \ll 7$  ;
- bit  $n$  à 0, le reste à 1 (à faire)

- Tester la valeur du bit  $n$

$$a \& (1 \ll n)$$

- Mettre à 1 le bit  $n$

$$a = a | (1 \ll n)$$

- Mettre à 0 le bit  $n$

$$a = a \& \sim(1 \ll n)$$

- Mettre à  $x$  le bit  $n$  ?

## Les pointeurs

- Un pointeur est une variable qui contient l'adresse d'une autre variable. L'opérateur unaire & donne l'adresse mémoire d'un objet.
- L'instruction : `p = &c;`  
affecte l'adresse de `c` à la variable `p`.
- Un pointeur est déclaré par `*` de type de donnée

`char *p ;` déclare un pointeur `p` sur un caractère

`float *f ;` déclare un pointeur sur un réel.

`Char *fonction(void)` déclare une fonction qui retourne un  
pointeur sur un caractère

`void(*fonction) (void)` déclare un pointeur sur une fonction

`void(*fonction) (void)=0x8000` crée un pointeur sur une  
fonction en 8000

```
int a=1,b=2,c; /*trois entiers */
```

```
int *p1,*p2 ; /*deux pointeurs sur des entiers*/
```

```
p1=&a ; /*p1 contient l'adresse de a*/
```

```
p2=p1 ; /*p2 contient l'adresse de a*/
```

```
c=*p1 ; /*c égale le contenu de l'adresse pointé  
par p1 donc c=a*/
```

```
p2=&b ; /*p2 pointe b*/
```

```
*p2=*p1
```

/\*la donnée à l'adresse pointé par p2 est placée dans l'adresse pointé  
par p1, cela revient à donc recopier a dans b\*/

```
#define PORTA *(unsigned char *) (0x1000)
```



PORTA est le contenu de cette adresse

## Les structures

- Une structure est composée de variables de types différents:

```
struct personne { char  
nom[20]; char prenom[20];  
int num_employe; };  
/* personne est une étiquette  
de structure*/
```

```
struct personne p1,p2;  
struct personne *po1,*po2;
```

### **// combinaison**

```
struct personne { char nom[20];  
char prenom[20];  
int num_employe; } p1,p2;
```

```
struct personne pers1, pers2,  
pers3;
```

```
struct { char nom[20];  
char prenom[20];  
int num_employe; }  
p1,p2;
```

```
struct { char nom[20];  
char prenom[20];  
int num_employe; } p3;  
// pas même type  
// il faut tjrs redéclarer
```



- Une structure peut être initialisée par une liste d'expressions constantes à la manière des initialisations de tableau.
- Exemple : `struct personne p = {"Jean", "Dupond", 7845};`
- **typedef** de faciliter l'écriture des programmes, et d'en augmenter la lisibilité

```
typedef struct { ... } PERSONNE;
PERSONNE p1, p2;
PERSONNE *po1, *po2;
```

```
typedef PERSONNE *P_PERSONNE;
/* P_PERSONNE type pointeur vers struct */
P_PERSONNE po1, po2;
/* po1 et po2 pointeurs vers des struct */
```

## Programmation C du périphérique: SysTick

- Lire ou écrire dans un registre correspond à l'accès à une adresse bien déterminée
- En C, une adresse correspond à un pointeur
  - Exemple : pour écrire la valeur 0 dans le registre CONTROL du SysTick qui se trouve à l'adresse 0xE000E010, on peut écrire en C

```
* ( ( int* ) 0xE000E010 ) = 0 ;
```

- Ou encore

```
#define SYSTICK_CONTROL 0xE000E010
```

```
* ( ( int* ) SYSTICK_CONTROL ) = 0 ;
```

- Utilisation des structures de données

```
typedef struct{
    volatile int CONTROL
    volatile int RELOAD
    volatile int CURRENT
}SysTick_Type
```



```
#define SYSTICK_BASE 0xE000E010
#define SysTick ((SysTick_Type *)SYSTICK_BASE)
```

- sizeof(SysTick) ?
- Comment accéder à CONTROL à partir de SysTick ?

- Accès aux registres

```
SysTick->CONTROL = 0 ;  
SysTick->RELOAD = 100 ;  
if (SysTick->CONTROL == 0)  
...
```

- Remarques

- **volatile** empêche le compilateur d'optimiser l'utilisation des variable CONTROL, ...
- Il n'y a aucune réservation de mémoire pour la structure de données SysTick\_Type

**Exercice** : refaire en C le même exercice fait précédemment en assembleur

```
SysTick->CONTROL = 0 ;  
SysTick->RELOAD = 1000 ;  
SysTick->CURRENT = 0 ;  
SysTick->CONTROL = 5 ;  
While (! (SysTick->CONTROL & 0x10000)) ;
```

## Utilisation de l'interruption SysTick

Souvent le systick est exploité via son interruption (SysTick). On configure l'activation de l'interruption et on implémente la fonction `sysTick_Handler()` qui sera exécutée chaque cycle d'interruption (= temps entre valeur initiale et valeur nulle)

```
/*routine de gestion d'interruption. Automatiquement  
exécutée lorsque l'interruption SYSTICK est déclenchée*/
```

```
void SysTick_Handler(){
```

```
    myCounter ++ ;
```

```
    ...
```

```
}
```

```
...
```

```
int main(){
```

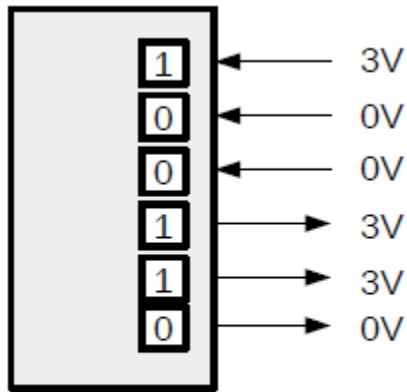
```
    /* configure et active le Timer SysTick en autorisant  
    l'interruption SYSTICK*/
```

```
    SysTick_Config(SystemCoreClock / 1000)
```

```
    ...
```

## Les GPIO

- Entrées/sorties a usage general (General Purpose Input Output)
  - ☾ Le plus simple des périphériques est un simple registre. Chaque bit du registre correspond a un signal (PIN) qui peut être configure en logiciel soit comme une entrée ou une sortie



Port GPIO

- Les pins configurés comme entrées permettent de lire l'état d'un signal (exemple bouton)
- Les pins configurés comme sorties permettent de commander l'état d'un signal (exemple LED)



## Les GPIO dans STM32

- Le STM32F4 a 5 ports GPIO de 16 bits chacun Port A (PA0 → PA15), Port B, Port C, Port D, Port E.
- Chaque port dispose de 16 pins ☾ 78 pins sur les deux connecteurs (PA11 et PA12 non disponibles)



- Chaque port est associé à cet ensemble de registres:
  - ✓ GPIO port mode register (GPIOx\_MODER)
  - ✓ GPIO port output type register (GPIOx\_OTYPER)
  - ✓ GPIO port output speed register (GPIOx\_OSPEEDR)
  - ✓ GPIO port pull-up/pull-down register (GPIOx\_PUPDR)
  - ✓ GPIO port input data register (GPIOx\_IDR)
  - ✓ GPIO port output data register (GPIOx\_ODR)
  - ✓ GPIO port bit set/reset register (GPIOx\_BSRR)
  - ✓ GPIO port configuration lock register (GPIOx\_LCKR)
  - ✓ GPIO alternate function low register (GPIOx\_AFRL)
  - ✓ GPIO alternate function high register (GPIOx\_AFRH)
  - ✓ GPIO Port bit reset register (GPIOx\_BRR)

x = A, B, C, D ou E

- **GPIO port mode register (GPIOx\_MODER)** : 32 bits désignant le mode de chaque I/O pin:
- '00' -> input mode, which allows the GPIO pin to be used as an input pin,
- '01' -> Output mode, which allows the GPIO pin to be used as an output pin,
- '11' -> Analog mode, which allows the GPIO pin to be used as an Analog input pin and finally,
- '10' -> Alternate function mode which allow the GPIO pins to be used

by peripherals

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 port output type register (GPIOx\_OTYPER) :** 32 bits désignant le type de chaque Output pin

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res
	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

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

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

0: Output push-pull (reset state)

1: Output open-drain

Configuration de l'amplificateur de sortie:

- ✓ Push/Pull: valeurs possibles en sortie sont 0V et  $+V_{DD}$
- ✓ Open drain: valeurs possibles en sorties sont 0V et H (haute impédance)

- **GPIO port output speed register (GPIOx\_OSPEEDR) : 32 bits** désignant la vitesse de chaque output pin

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OSPEEDR15[1:0]		OSPEEDR14[1:0]		OSPEEDR13[1:0]		OSPEEDR12[1:0]		OSPEEDR11[1:0]		OSPEEDR10[1:0]		OSPEEDR9[1:0]		OSPEEDR8[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
OSPEEDR7[1:0]		OSPEEDR6[1:0]		OSPEEDR5[1:0]		OSPEEDR4[1:0]		OSPEEDR3[1:0]		OSPEEDR2[1:0]		OSPEEDR1[1:0]		OSPEEDR0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

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

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

x0: 2 MHz Low speed

01: 10 MHz Medium speed

11: 50 MHz High speed

- **GPIO port pull-up/pull-down register (GPIOx\_PUPDR) :** 32 bits désignant la configuration de la résistance de rappel de chaque input pin

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[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
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

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

These bits are written by software to configure the I/O pull-up or pull-down

00: No pull-up, pull-down

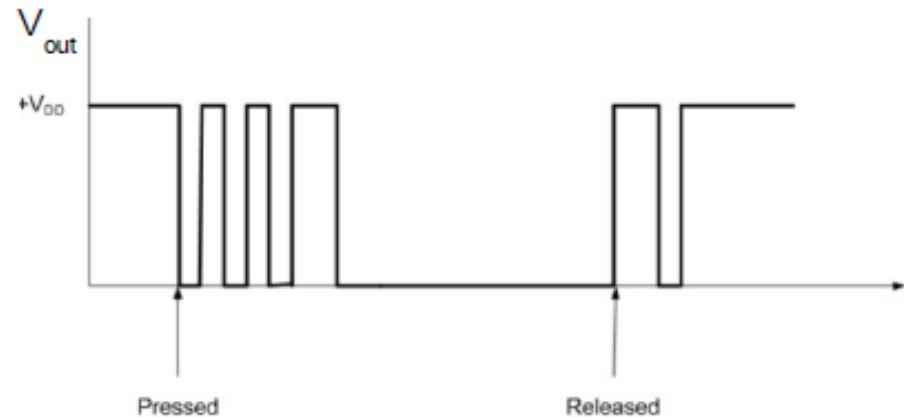
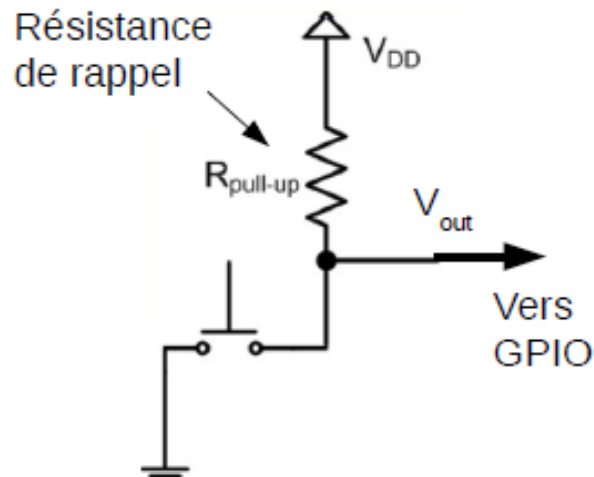
01: Pull-up

10: Pull-down

11: Reserved

- Pour tirer vers le haut (VDD) : Pull Up
- Pour tirer vers le bas (GND) : Pull Down
- Aucune résistance de rappel

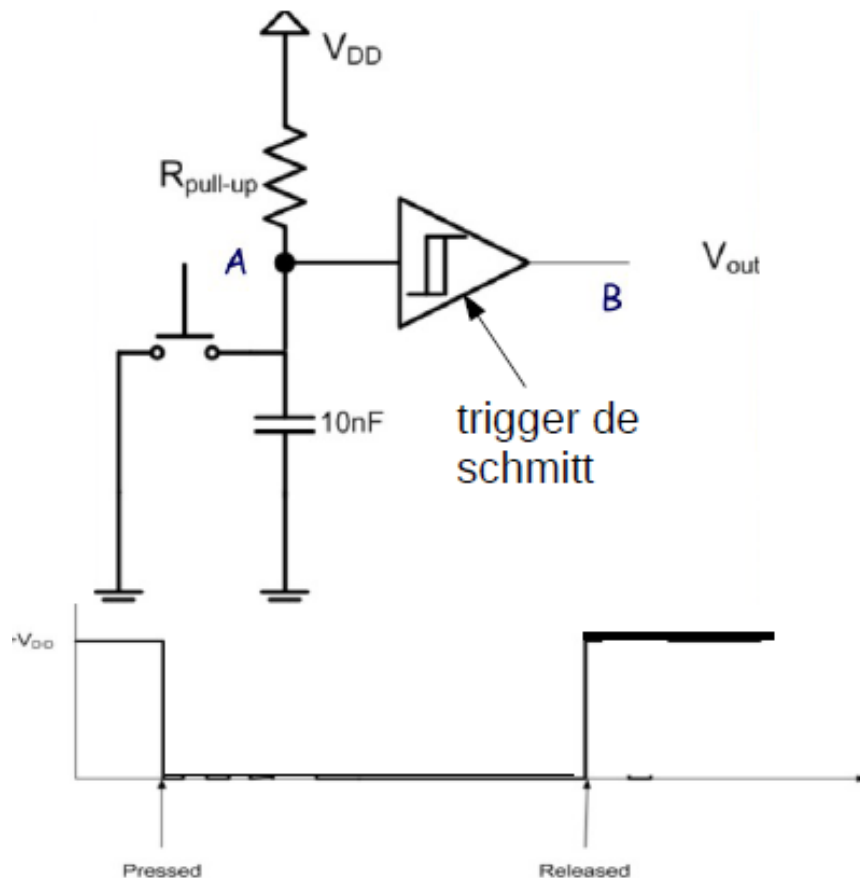
# Exemple : Interface avec un bouton



- Problème : les propriétés mécanique du bouton font apparaître des **rebonds**
- Solution
  - Matérielle : ajouter un circuit anti-rebond
  - Logicielle

# Exemple : Interface avec un bouton

- Solution matérielle



- Solution logicielle

- Faire des lectures périodique (exemple chaque 1 ms) en utilisant l'interruption du SysTick
- La variable globale représentant l'état du bouton ne changera que lorsque N lectures successives donnent la même valeur



- **GPIO port input data register (GPIOx\_IDR):** 16-bit read-only register; chaque bit représente la valeur d'entrée du pin correspondant

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **IDR[15:0]**: Port input data

These bits are read-only. They contain the input value of the corresponding I/O port.

- **GPIO port output data register (GPIOx\_ODR):** 16-bit read/write register; chaque bit représente la valeur de sortie du pin correspondant