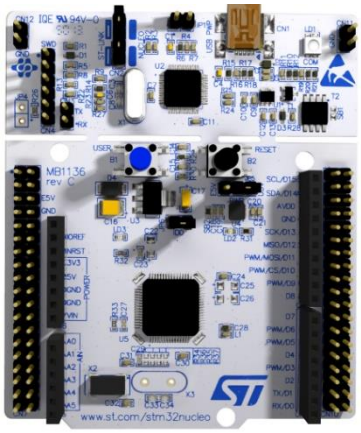
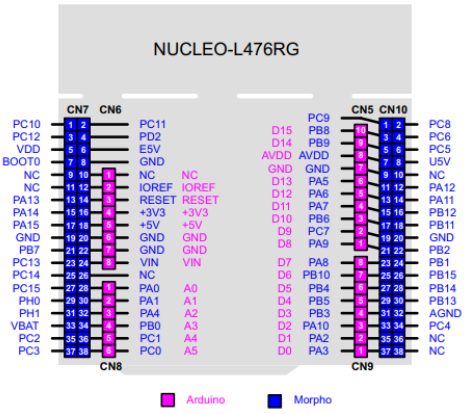
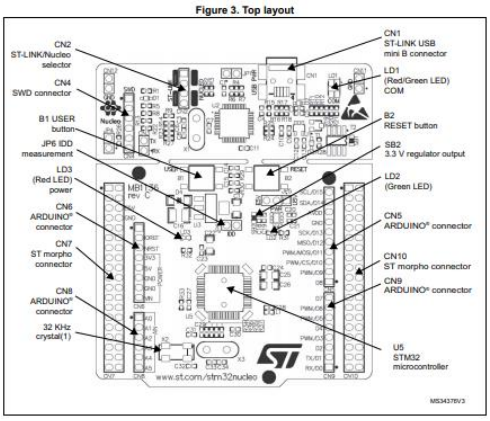
Initiation et découverte du stm32







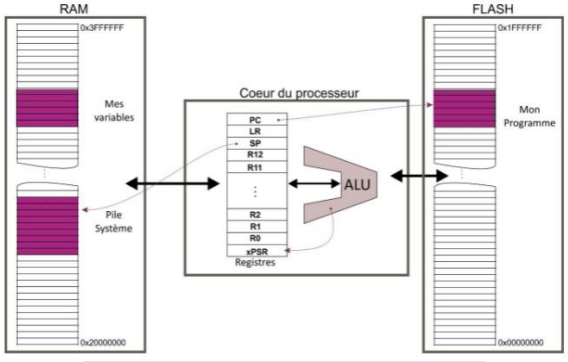
Sur carte NucleoL476RG (for 1 Mbyte) basé sur un cortex M4

Introduction du microcontrôleur

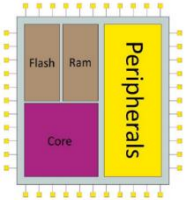
ALU (unité arithmétique et logique) c'est la partie opérative du système la seule capable de faire des calculs et communique avec la mémoire via des registres.
L'ALU pour gagner en rapidité et efficacité, elle est connectée à des registres dont l'accès peut se faire en un seul cycle d'horloge, ces registres sont :
- soit à usage général (stocks des informations alimentant les calculs).
- soit spécifiques, dédiés à une tâche très précise.

Architecture ARM repose sur 17 registres :
Registres généraux : R0 à R12.
Registre SP (pointeur de pile) R13. C'est la PILE SYSTEME, elle contient l'adresse accessible en mémoire vive pour stocker des informations temporaires mais vitales pour le bon déroulement de l'ensemble.
Registre LR (de lien) R14.
Registre PC (pointeur de programme) R15. Contient l'@ courante du programme en cours et progresse dans l'exécution du programme.
Registre xPSR (status) se décline (APSR, EPSR,IPSR).
Et se complète avec 4 registres spéciaux
PRIM, FAULTMASK, BASEPRI (gestion d'exception), CONTROL (gère le niveau de privilège)

La zone mémoire RAM (sans mémoire après une coupure d'alimentation) est prédéfinie au départ par le programme. Cette zone est pointée par le registre SP et évolue au gré des écritures/lectures dans une structure LIFO (pile d'assiette).



L'architecture ARM de type Harvard sépare physiquement les accès à mémoire code (stockage du programme) des accès à la mémoire données (variables programme, accessible en lecture/écriture).



Basé sur un jeu d'instruction RISC (Reduced Instruction Set Computer)
L'ALU peut mettre à jour les flag (nombre de 5) dans xPSR .
Indicateur C (Carry) retenue lors de calcul débordement de la représentation non signée lors de l'instruction précédente donc le calcul est partiellement faux (C=1) comme 8 bits la somme 200+60 dépasse le max 255.
Indicateur Z (Zero) = 1 si le résultat de l'instruction donne zéro.
Indicateur N (Négative) contient le bit de poids fort celui qui contient la plus grande valeur comme celui de gauche pour une écriture de gauche à droite. Pour les valeurs signé N=1 donc valeur négative.
Indicateur V(Overflow) =1 s'il y a un débordement de la représentation signée et dons le résultat signé est faux (8bits : 140+40 en signé on peut juste entre -128 et 127)
Indicateur Q (saturation flag) pour USAT et SSAT =1 si ces instructions sont saturé le registre traité.

Le microcontrôleur à la capacité de n'activer qu'en fonction des besoins certaines parties du microcontrôleur ou d'endormir le circuit complètement s'il n'est pas utilisé.

Les périphériques

Pour dialoguer avec l'extérieur nous avons des E/S GPIO.
Le **timer** pour compter/décompter les changements d'état (front montant/descendant) d'un signal binaire, on parle de counter (compteur).
Le système voit une horloge synchrone dont la fréquence est fixe et connue.
Il peut être overflow ou underflow car le registre a une capacité finie comme un registre de 16 bits on compte 0xFFF et un front il bascule à 0 et les informations accumulées sont perdues, mais un tel évènement est détectable car l'unité émet une requête d'interruption pour que le programme puisse réagir en conséquence.
Sinon plutôt que repartir à zéro on peut repartir d'une valeur prédéfinie dans le registre ad-hoc (reload).
Le temps entre 2 débordements est alors modifié. Comme la base de temps est ajustable (diviseur de fréquence) et le nombre de front est programmable, on peut intervenir entre 2 requêtes d'interruptions successives (reload) il est possible de mettre en place des évènements logiciels synchrones de façon précise à l'aide de ces timers. On trouve le watchdog qui s'assure que le système ne reste pas bloqué dans le traitement qu'il exécute (boucle infinie), c'est un sablier qu'il faut venir retourner avant qu'il ne soit vide. Sinon une alerte interruption est levée.

Les **Capture** permet de mesurer le temps qu'un bouton poussoir a été appuyé. L'unité capture est associé à un signal de type binaire sur une broche, à un timer qui mesure le temps qui passe et à un registre pour stocker la valeur capturée du timer lorsque l'évènement apparait.

Les **Compare** (symétrique) toujours un signal binaire mais le timer est remplacé par un compteur qui compte les évènement (les fronts) sur une broche. Lorsque le compteur égale la valeur il déclenche une interruption.

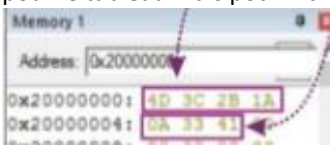
L'**ADC** (température, pression, distance, accélération…) l'électronicien équipe le process d'un capteur qui transforme cette grandeur en volt de type analogique compatible avec notre microcontrôleur [0-5V].
Brancher sur un multiplexeur et que logiciellement on sélectionne la voies. L'ADC a une résolution la quantité de valeurs entières que peut prendre l'ADC après conversion.

PWM la symétrique d'un ADC est un DAC (digital analog converter) permet de transformer une valeur numérique en analogique, mais peu courante car cela consomme beaucoup est peut être remplacé par un PWM.

Les **bus** pour envoyer des données sur un simple fil avec des 0 et 1.
Le protocole le plus simple est la liaison asynchrone (Universal Asynchronous Receiver Transmitter). Un fil pour émettre et un pour recevoir et une masse commune. Les données transitent octet par octet à une vitesse fixée à l'avance.
Il est complexifié avec une horloge commune qui permet de synchroniser les données, on obtient une Universal Synchronous/Asynchronous Receiver Transmitter (USART) comme rs232, mais insuffisant il faut augmenter le débit, sécuriser les données pour s'assurer que ce qui est envoyé est correctement reçu. Et il faut aussi plus de 2 systèmes ensemble comme un réseau.

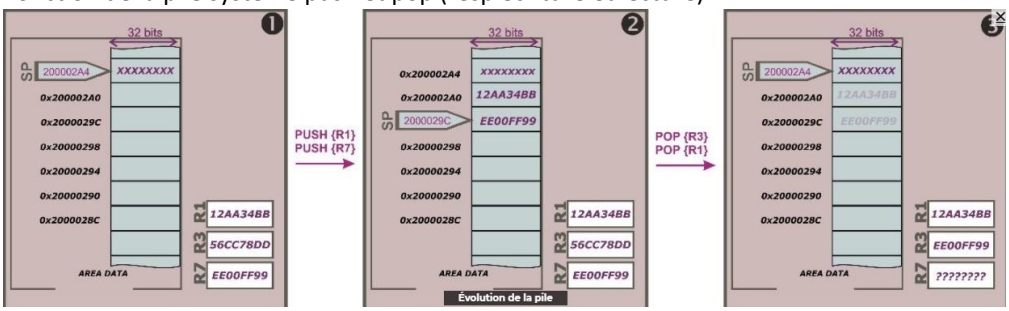
Pour répondre à ces besoins on a les protocoles industriels bus industriel ou terrain : I2C, SPIn CAN…
On peut aussi prendre un microcontrôleur simple et ajouter le périphérique manquant à travers un bus industriel si ce protocole équipe le circuit comme un DAC absent mais on peut ajouter une extension électronique et le piloter avec une interface I2C ou SPI pour avoir accès à cette conversion numérique analogique.

ARM utilise la convention little endian c'est-à-dire que pour une variable locale de 4 octets et une variable Carac de 3 octets nous avons pour ranger la mémoire : une valeur de 32 bits occupe 4 adresse consécutives : la première stock le poids faible, la seconde le poids suivant ainsi de suite mais dans Carac rien de tel (0X0A correspond à 10 en hexa et le code ASCII le chiffre 3 (char Carac[3] = {10, '3', 'A'} ; 3 est la valeur 0X33 et 'A' est 0x41 chaque case du tableau correspond à un seul octet l'inversion de poids faible et fort n'est pas nécessaire pour le tableau mais pour int Locale = 0X1A2B3C4D c'est bien little endian.

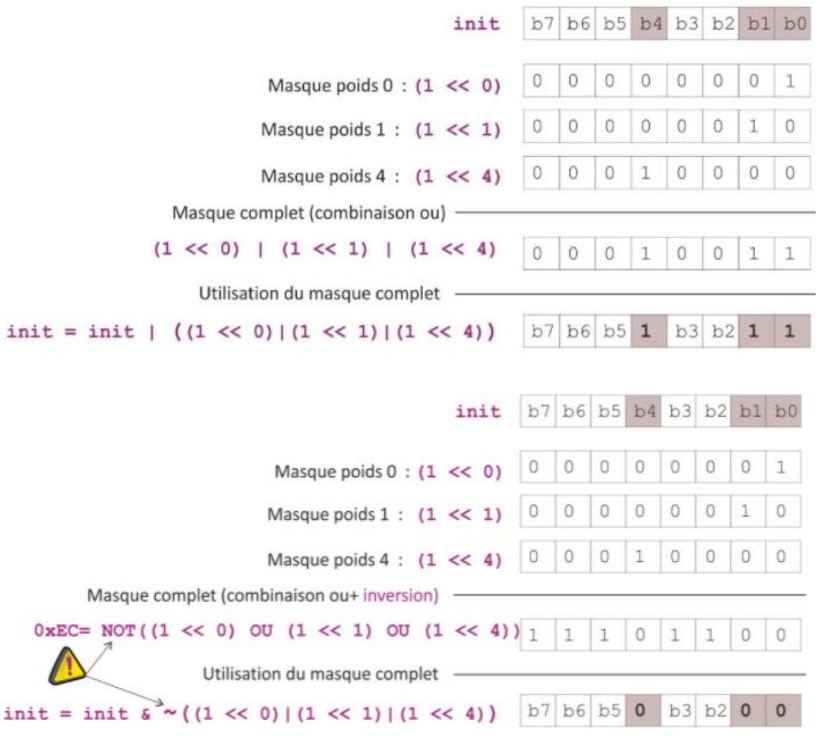


!! pour les valeurs codées sur 16 bits (short int) l'adresse la plus basse (contient les 8 bits de poids faible) doit être paire (termine par un bit à 0) donc 32 bits comme Locale l'adresse la plus basse doit être doublement paire (2 dernier bits de l'adresse sont à 0) ici ok 0X20000000. Mais pour 64bits le problème ne se propage pas puisqu'il lit par 2 lectures successives de 32bits.

Le literal pool c'est la lecture du processeur sur une case mémoire créée au début.
ce sont des valeurs immédiates.
Le compilateur utilise systématiquement R0 pour retourner un argument (il est associé au return)
Le processeur stocke dans LR l'adresse de retour d'appel à la procédure.
Fonction de la pile système push et pop (resp écriture et lecture)



Action sur les registres



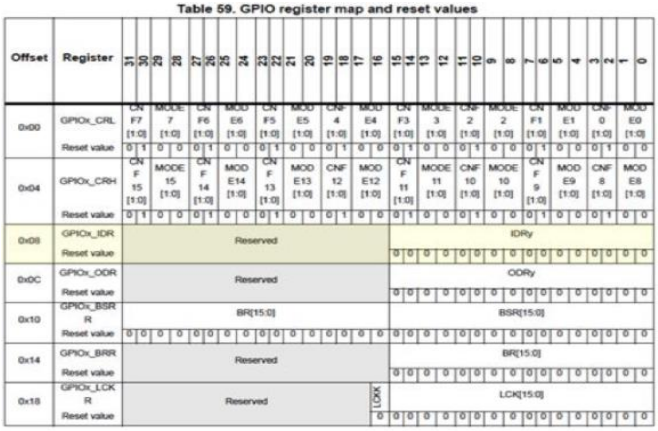
Exemple d'adresse GPIO

Table 3. Register boundary addresses (continued)

Boundary address	Peripheral	Bus	Register map
0x4001 5800 - 0x4001 7FFF	Reserved	APB2	-
0x4001 5400 - 0x4001 57FF	TIM11 timer		Section 16.5.11 on page 469
0x4001 5000 - 0x4001 53FF	TIM10 timer		Section 16.5.11 on page 469
0x4001 4C00 - 0x4001 4FFF	TIM9 timer		Section 16.4.13 on page 459
0x4001 4000 - 0x4001 4BFF	Reserved		-
0x4001 3C00 - 0x4001 3FFF	ADC3		Section 11.12.15 on page 252
0x4001 3800 - 0x4001 3BFF	USART1		Section 27.6.8 on page 832
0x4001 3400 - 0x4001 37FF	TIM8 timer		Section 14.4.21 on page 362
0x4001 3000 - 0x4001 33FF	SPI1		Section 25.5 on page 745
0x4001 2C00 - 0x4001 2FFF	TIM1 timer		Section 14.4.21 on page 362
0x4001 2800 - 0x4001 2BFF	ADC2		Section 11.12.15 on page 252
0x4001 2400 - 0x4001 27FF	ADC1		
0x4001 2000 - 0x4001 23FF	GPIO Port G		
0x4001 1C00 - 0x4001 1FFF	GPIO Port F		
0x4001 1800 - 0x4001 1BFF	GPIO Port E		
0x4001 1400 - 0x4001 17FF	GPIO Port D		Section 9.5 on page 193
0x4001 1000 - 0x4001 13FF	GPIO Port C		
0x4001 0C00 - 0x4001 0FFF	GPIO Port B		
0x4001 0800 - 0x4001 0BFF	GPIO Port A		
0x4001 0400 - 0x4001 07FF	EXTI		Section 10.3.7 on page 213
0x4001 0000 - 0x4001 03FF	AFIO		Section 9.5 on page 193

Registres du port B

Table 59. GPIO register map and reset values



Page 50 l'ensemble des registres du port sont fixés sur la plage mémoire de 0x4001 0C00 à 0x401 0FFF	Comment trouver les différents registres d'un port ? doc chap GPIO (chp9, p158). Au début une description générique du périphérique et à la fin un tableau de ses registres (p193) une description bit-à-bit une section est dédiées.
	
	Pour lire la valeur du registre IDR du port B il est possible en C de faire
	valueIDR = *(int *)0x40010C08; // lecture de la valeur du registre IDR.
	La valeur contiendra la valeur logique des broches du port B.

Pour résumer si on souhaite manipuler le registre GPIOx_IDR il faut chercher l'adresse de base du périphérique, puis décaler cette adresse de la valeur d'offset propre au registre (très fastidieux)

C'est pourquoi ST-MicroElectronics fournissent des fichiers de configuration ici on a le fichier STM32f10x.h qui contient l'ensemble de définitions qui va aider.
On trouve la structure C prédéfinies pour accéder plus rapidement aux registres ainsi qu'un ensemble de variables ayant pour valeur les adresses des périphériques.
Pour les ports on trouve la structure GPIO_TypeDef

```
Typedef struct
{
  __IO uint32_t CRL;
  __IO uint32_t CRH;
  __IO uint32_t IDR;
  __IO uint32_t ODR;
  __IO uint32_t BSRR;
  __IO uint32_t BRR;
  __IO uint32_t LCKR;
} GPIO_TypeDef;
```

Où __IO uint32_t est un type pour des valeurs volatiles signées en 32 bits et nous y retrouvons la liste de tous les registres d'un port ordonnés en fonction de leur offset.
Pour le port B
On trouve une adresse pour GPIOB définie par

```
#define GPIOB    ((GPIO_TypeDef *) GPIOB_BASE)
```

Puis continuons à explorer le fichier de définition

```
#define GPIOB_BASE  (APB2PERIPH_BASE + 0x0C00)
...
#define APB2PERIPH_BASE (PERIPH_BASE + 0x10000)
...
#define PERIPH_BASE ((uint32_t)0x40000000)
```

Donc en remontant la chaîne nous avons GPIOB qui prend la valeur 0x40000000+0x10000+0x0C00 soit 0x4001 0C00 Qui est bien l'adresse de base du GPIOB fournie par la documentation.
Pour l'accès au registre il suffit pour lire le registre IDR du port B à écrire
value = GPIOB->IDR ; // value prend la valeur stockée à l'adresse 0x40010C08
Vous savez comment trouver et manipuler l'adresse d'un registre pour le STM32,

Direction de port

On a 8 configurations possibles. Les possibilités sont :

direction	mode	Description	Valeur du champ de bits
input	Analog	mode d'entrée permettant une acquisition directe de la tension (l'état binaire de la broche n'est plus connue sur le registre)	0000
input	Floating input	mode de base pour une entrée, si aucune tension n'est appliquée, la valeur n'est pas connue	0100
input	With pull-up / pull-down	mode d'entrée pour lequel la valeur il est possible de fixer l'état par défaut à 0 (pull down) ou à 1 (pull-up)	1000
input	Analog	Non utilisé	1100
Output(*)	Push-pull	mode de sortie dans lequel la tension appliquée à la broche est toujours forcée à 0 ou 1	0001 0010 0011
Output(*)	Open-drain	mode de sortie dans lequel le niveau de tension haut est fixé par le circuit connecté à la broche	0101 0110 0111
Output(*)	Alternate function push-pull	mode de sortie identique au mode push-pull, mais l'état de la broche est contrôlé par un autre périphérique que le port	1001 1010 1011
Output(*)	Alternate function open-drain	mode de sortie identique au mode open-drain, mais l'état de la broche est contrôlé par un autre périphérique que le port	1101 1110 1111

Il y a 16 broches sur un port, il faut 4 bits par broche pour configurer e/s. soit 64 bits pour configurer l'ensemble des broches d'un port. Or un registre fait 32 bits. Il faut donc 2 registres pour configurer l'ensemble des broches du même port d'où l'existence de 2 registres : CRL et CRH.
CRL (L pour low) permet de configurer les broches de 0 à 7. CRH (H pour high) de 8 à 15.

On veut configurer la broche 5 du port A en entrée (input floating), il faut affecter la valeur binaire 0b0100 aux bits b23 b22 b21 b20 dur registre CRL on écrit alors :
GPIOA->CRL = GPIOA->CRL & ~(0xF << 20); // Mise à 0 des bits b23 b22 b21 b20
GPIOA->CRL = GPIOA->CRL | (0x1 << 22); // Mise à 1 du bit b22

Dans la même idée en sortie (output push pull) on fixe 0b0001 aux bits b11 b10 b9 b8 dans le CRH
GPIOA->CRH = GPIOA->CRH & ~(0xF << 8); // Mise à 0 des bits b11 b10 b9 b8
GPIOA->CRH = GPIOA->CRH | (0 << 8); // Mise à 1 du bit b8

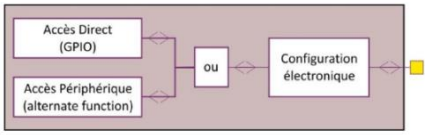
Lire les valeurs en entrée
Le registre IDR contient les 16 premiers bits
On veut lire la valeur de la broche 7 du port C on écrit
valueb7 = (GPIOC->IDR & (0x1 << 7)) >> 7;
rappelons que si on utilise l'état de la broche pour faire un test le dernier décalcage à droite est inutile on écrit :
if (GPIOC->IDR & (0x1 << 7)) { si le bit 7 de IDR est à 1 et fausse si le bit est à 0

Ecrire sur un port en sortie
Contrôler l'état d'une broche ? le registre ODR dont les 16 premiers bits fixent l'état de chaque broche en sortie d'une port.
Pour mettre à 1 la broche 8 du port B nous fixons à 1 le bit 8 de ODR
GPIOB->ODR = GPIOB->ODR | (1 << 8);

Remarque
On remarquera qu'étant donné que le STM32 n'a pas de mémoire « bit accessible » l'affectation d'un bit nécessite la lecture d'un registre, le masquage adéquat de la valeur ainsi lue et enfin l'affectation du résultat sur le registre ; soit au moins 3 opérations.

Il existe cependant dans l'architecture du Cortex une alternative qui permet d'éviter cela (le « bit-banding »). Sans entrer dans le détail l'idée est de mettre en correspondance un bit avec un mot. STMicro a utilisé ce principe pour créer des registres de SET et de RESET des GPIO. Ce sont les registres BRR (Bit Reset Register) et BSRR (Bit Set and Reset Register). Chaque bit de ces registres correspond à un bit des GPIO et seul le niveau logique 1 a un effet. Le programmeur peut donc directement affecter le valeur 1 en mettant à 1 le bit correspondant du registre BSRR (action SET) et affecter le valeur 0 en mettant 1 le bit correspondant du registre BRR (action RESET : on agit en mettant 1 pour forcer la mise à 0, ce qui peut être déroutant en première lecture).

Les gpio sont des broches regroupées pas paquet de 8(octet) ou 16(mot) et forme des ports e/s (I/O ports). Une broche peut être pilotée soit directement par le biais des registres dédiés (General Purpose Input Output, GPIO) soit indirectement par le biais d'un périphérique (Alternate Function), dans tous les cas il sera nécessaire de configurer l'étage électronique qui permet d'interfacer le matériel et le logiciel. Les broches sont configurables indépendamment des autres broches avec sa direction (entrée ou sortie).

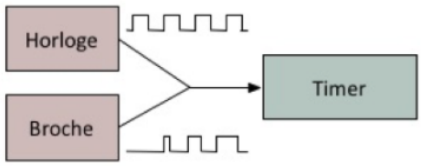


Entrée : en fonction de sa configuration, nous pouvons directement exploiter la valeur physique du signal en le convertissant en une valeur numérique (ADC) ou sous forme binaire logique (0 ou 1) fixé sur la tension d'entrée.
Sortie : pour le cas de alternate functions cette tension est fixée par un autre périphérique (UART, convertisseur numérique-analogique, PWM…) sinon en accès direct un registre propre au port impose une tension haute et basse au circuit qui lui est connecté.
En entrée lire la valeur : value = (GPIOC->IDR & (0x1<<7))>>7 ; tester : if(GPIOC->IDR & (0X1 <<7))
En sortie : GPIOB->ODR = GPIOB->ODR | (1<<8) ;

LES TIMERS

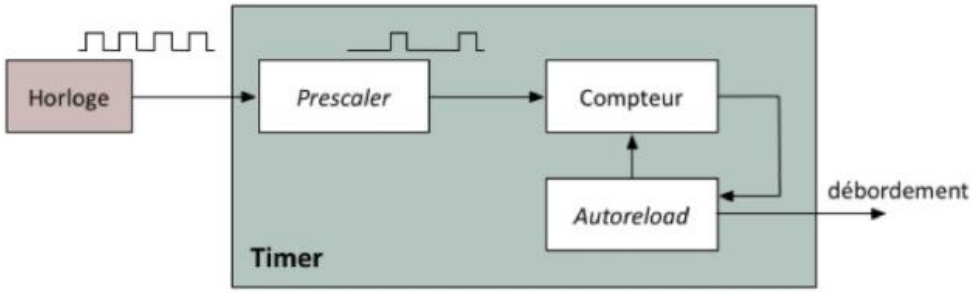
Un timer est un compteur électronique. Il a une résolution qui dépend de son architecture et contraint l'amplitude des valeurs sur lesquelles il peut compter.
Un timer avec une résolution de 8 bits pourra compter entre 0 à 255.
Un timer de 16 bits entre 0 à 65 535 et 32 bits de 0 à 4 294 967 295.
Dans des architectures on peut mettre en série 2 times du processeur ainsi avec 2 times 16 bits en sérialisant on peut fabriquer un timer de 32 bits.

L'incrémentation d'un timer se fait sur des évènements et sont générés par une horloge avec une fréquence fixée. (le timer compte le temps)
Il peut aussi compter des évènements non périodiques comme un signal venant d'une broche représentant des fronts montants et descendants on parle plutôt de compteur.

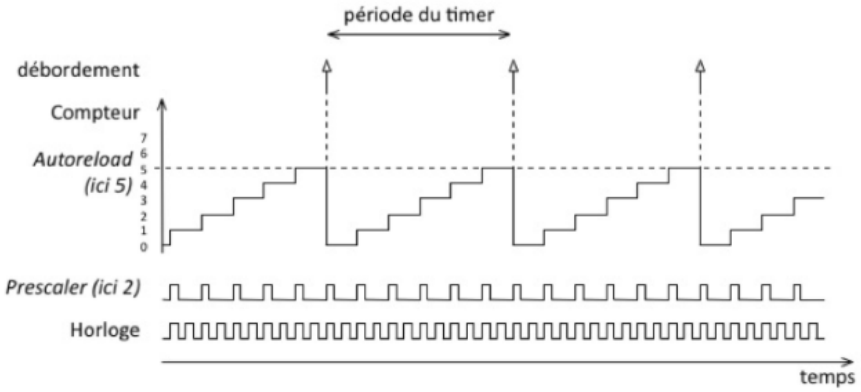


L'entrée d'horloge d'un timer est souvent précédée d'un diviseur (prescaler). Son rôle est d'opérer une première division de la fréquence de l'horloge pour ralentir la fréquence de comptage du timer (son intérêt dans la pratique).

L'utilisation classique d'un timer va consister à réagir au débordement (overflow) de son compteur.
En effet la résolution du timer étant bornée, son incrémentation va inévitablement conduire à un débordement, c'est-à-dire que l'incrément suivant dépasse la capacité du compteur. S'il survient le compteur est remis à zéro et un évènement interne est généré pour le signaler cette rupture dans la séquence de comptage.



Afin de mieux contrôler le débordement le compteur est associé à un registre dit autoreload. Celui-ci contient la valeur à laquelle le compteur va déborder, même s'il n'a pas atteint sa capacité maximale. Ainsi le compteur va avoir le comportement décrit par la figure ci-dessus :



On voit que le timer est incrémenté à chaque impulsion de l'horloge et que lorsque son compteur atteint la valeur de l'autoreload il déborde.

Calculer les paramètres d'un timer
En connaissant la fréquence de l'horloge entrante du timer et en réglant le *prescaler* et l'*autoreload* il est possible de régler finement la fréquence à laquelle le timer déborde.

Formule pour obtenir la période à laquelle le timer déborde :

$$T_{timer} = T_{horloge} \times prescaler \times (autoreload + 1)$$

Horloge à 72Mhz, comment régler le prescaler et l'autoroload pour obtenir un débordement du timer toutes les secondes ? : si le prescaler vaut 1 donc :

$$autoreload = \frac{T_{timer}}{T_{horloge}} - 1 = \frac{1}{\frac{1}{72.10^6}} - 1 = 72.10^6 - 1$$

Ce qui est possible pour un timer comptant sur 32 bits mais impossible pour un timer en 16bits puisqu'il ne peut compter que jusqu'à 65 535

Pour configurer un timer il faut donc connaître sa résolution et faire attention à l'amplitude des valeurs disponibles.
Pour trouver une configuration faisable il faut aussi utiliser le prescaler.
Comme dans notre exemple en fixant le prescaler à 7 200 nous avons :

$$autoreload = \frac{T_{timer}}{(prescaler \times T_{horloge})} - 1 = \frac{1}{\frac{7200}{72.10^6}} - 1 = 1.10^4 - 1$$

Qui est acceptable pour un compteur 16 bits.
Au final il est possible de configurer le prescaler à 7 200 et l'autoreload à 9 999 pour avoir un timer qui expire toutes les secondes (dans le cas d'un compteur en 16bits) d'autre valeurs sont évidemment possibles.

Remarquez qu'il peut ne pas y avoir de solution entière pour les valeurs prescaler et autoreload. Dans ce cas il vaut mieux chercher à régler ces valeurs de manière à minimiser l'erreur induite par les arrondis. En prenant un prescaler le plus petit possible on a tendance à minimiser les erreurs puisque l'unité de comptage (période de comptage du timer après la division par le prescaler) sera la plus petite.

PRATIQUES sur le timer STM32F103
On a 2 espèces différentes une est constituée des timers 1 à 8 Advanced-control et l'autre timer 2 à 5 General-purpose. On ne verra pas la différence on prendra timers 2 à 5. Il existe un timer spécifique au cœur du cortex appelée Systick plus simple et a pour vocation d'être utilisé par un système d'exploitation.

Le STM32F103RB contient uniquement l'advanced timer 1 et les general purpose timers 2, 3, 4.
On suppose que les timer sont branchés sur une horloge à 72MHz (par défaut de la carte nucleo)

Les registre de configuration nous trouvons ARR pour l'autoreload et PSC pour le prescaler (divise par 1 pour la valeur 0, 2 pour la valeur1 …). Donc la formule précédente on a donc :

$$T_{timer} = T_{horloge} \times (PSC + 1) \times (ARR + 1)$$

Le timer 2 désigné par TIM2 dans le fichier stm32f10x.h et commençons avec un main vide :

```c
#include <stm32f10x.h>
int main (void)
{
    RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;
    while(1)
    {
    }
    return 0;
}
```

1. Configure les horloges

Pour configurer le timer 2 on ajoute avant la boucle while les lignes

```c
TIM2->ARR = 9999;
TIM2->PSC = 7199;
```

Il reste à démarrer le timer (doc p.404) indique que le bit 0 CEN du registre CR1 permet de lancer et arrêter le compteur ajouter donc :

```c
TIM2->CR1 = TIM2->CR1 | ( 1 << 0);
```

Pour lancer le compteur remarquez que le masque du bit CEN est défini dans le fichier stm32f10x.h et il est possible d'écrire directement :

```c
TIM2->CR1 = TIM2->CR1 | TIM1_CR1_CEN;
```

Passer en debug avec un point d'arrêt sur cette ligne et ouvrer la fenêtre liée au timer 2 dans Periphéral.
On doit voir que les registre ARR et PSC prennent les valeurs 0x270F (pour 9999) et 0x1C1F (7199)
Avancez le simulateur pour lancer le comptage et observez que le compteur représenté par le registre CNT s'incrémente bien.

Scruter le débordement d'un timer
On parle de scruter l'évènement provoqué par ce débordement, il existe un registre contenant un bit indiquant si un débordement a eu lieu ou non.
Pour le stm32f103 il est indiqué par la valeur du bit 0 nommé UIF, du registre SR (doc p.409) il passe à 1 suite à un débordement. Pour scruter le bit UIF.
Test :
Faire clignoter la led USER toutes les secondes (port A broche 5). Afinde la contrôler il faut la configurer en sortie soit ajouter avant la boucle le code :

```c
RCC->APB2ENR |= RCC_APB2ENR_IOPAEN;
GPIOA->CRL &= ~(0xF << 20); // PA.5 output push-pull
GPIOA->CRL |= (0x1 << 20); // PA.5 output push-pull
```

Nous allons ajouter dans la boucle le code pour scruter l'état du timer :

```c
if (TIM2->SR & TIM_SR_UIF) { // Scrutation de UIF
    TIM2->SR = TIM2->SR & ~TIM_SR_UIF; // Remise à zéro de UIF
}
```

Le masque TIM_SR_UIF existe et vaut 0x1 et d'autre part nous avons dû remettre à zéro le bit UIF. Cela est nécessaire sinon il reste dans l'état indiquant un débordement et la prochaine évaluation dans la boucle le timer sera toujours considéré comme ayant débordé.

Pour clignoter la led ajouter la ligne de code l.3:

```
1 if (TIM2->SR & TIM_SR_UIF) {
2     TIM2->SR = TIM2->SR & ~TIM_SR_UIF;
3     GPIOA->ODR = GPIOA->ODR ^ (1 << 5);
4 }
```

Dans le chapitre suivant nous allons voi comment guetter un timer efficacement. Mais on va voir les horloges :

Principe des arbres d'horloge
L'ajout du code : `RCC->APB2ENR |= RCC_APB2ENR_IOPAEN`
Cela active les horloges reliées aux différents périphériques sinon ils n'ont pas de signal pour faire évoluer leurs circuits logiques et donc non actifs. Par défaut les horloges ne sont pas actives pour économiser l'énergie consommée même si elles ne font rien.

Il faut absolument allumer les horloges pour les périphériques que l'on utiliser avant de commencer à les configurer

Le code :

```
1 GPIOA->CRL &= ~(0xF << 4*5); // PA.5 output push-pull
2 GPIOA->CRL |= (0x1 << 4*5); // PA.5 output push-pull
3 RCC->APB2ENR |= RCC_APB2ENR_IOPAEN;
```

Ne permet pas de configurer le port car l'horloge a été activée après et donc les registres du périphériques ne sont pas mis à jour.

Un autre élément qu'il faut appréhender pour bien maîtriser l'écoulement du temps sur un microcontrôleur est la source d'horloge du système. Dans le cas du STM32, mais qui est aussi commun à d'autres familles de processeurs, il est possible d'avoir diverses sources pour l'horloge globale du système.

Par défaut, le microcontrôleur dispose d'un circuit RC qui oscille à 8MHz, lui permettant ainsi d'avoir sa propre horloge et donc de fonctionner par défaut. L'inconvénient d'un tel circuit est qu'il est très sensible aux conditions externes (vous pouvez par exemple utiliser une bombe à froid pour perturber l'oscillation du circuit).

Il est possible de rendre plus robuste cette horloge en ajoutant un quartz externe au microcontrôleur. Dans le cas de la Nucleo, il existe un quartz externe cadencé à 8MHz qui permet, une fois l'arbre d'horloge correctement configuré de faire tourner le cœur à une fréquence de 72MHz (l'horloge du système est nommée SYSCLCK).

A cela s'ajoute un grand nombre de circuits permettant de régler finement les horloges dans le microcontrôleur. La documentation page 92 donne un aperçu de ce qui s'appelle l'arbre des horloges. On remarque sur cette figure que l'entrée principale SYSCLK est configurée par défaut à 72Mhz.

Ce signal peut ensuite être divisé à l'aide du *prescaler* AHB qui servira ensuite d'entrée pour les deux sous-arbres APB1 et APB2 qui alimentent les périphériques tels que les timers.

Il est donc possible de configurer les registres des *prescaler* de AHB, APB1 et APB2 pour faire battre les périphériques à des rythmes plus faibles et ainsi économiser l'énergie globalement consommée par le microcontrôleur. Pour la suite, nous ne toucherons à rien de tout cela et garderons les configurations initiales. Par contre, n'oubliez pas d'activer les horloges des périphériques avant de les utiliser.

Vous êtes maintenant capable de :

- d'expliquer le fonctionnement d'un timer,
- de calculer les principaux paramètres permettant de configurer la période du timer,
- de configurer les registres d'un timer,
- de mettre en place un mécanisme de scrutation pour détecter le débordement d'un timer,
- de comprendre le principe de l'arbre des horloges sur le STM32.

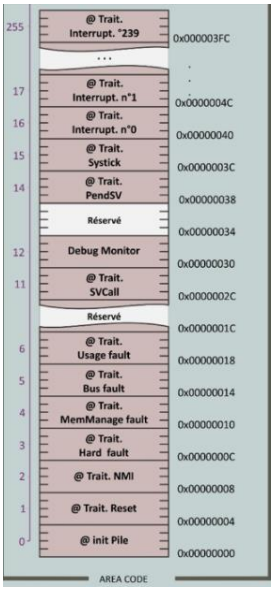
Maitrisez les Exceptions et les Interruptions

Le déroulement séquentiel d'un programme peut être perturbé pour trois raisons essentielles :

1. Il se passe un événement matériel grave qui l'empêche de continuer. Typiquement un boitier mémoire ne répond pas (ou plus).
2. On demande au processeur de faire quelque chose qu'il ne parvient pas à faire comme aller lire un mot de 32 bits à une adresse impaire.
3. Une unité périphérique a besoin de la CPU pour effectuer une tâche essentielle. Par exemple la liaison série (USART) vient de recevoir un caractère et il serait opportun de le stocker en mémoire avant qu'une autre donnée n'arrive et ne l'écrase.

Les deux premiers cas relèvent de l'exception et le troisième est plus communément appelée interruption, mais leur traitement est similaire. Elles viennent des demandes de requêtes formulées par un périphérique (timer, ADC, USART …).

Le principe de fonctionnement repose sur la table des Vecteurs d'interruptions (TVI) placé aux adresses les plus basses de la mémoire 0X00000000



Les données stockées en position 0 et 1 de cette table sont des valeurs obligatoires comme initialiser le pointeur SP et en 04 le pointeur de PC. Cette tables est initialisé par défaut dans le fichier startup_stm32..md.s comme le traitement d'une erreur de bus BusFault. On pourra redéfinir les fonctions d'interruption pour un traitement moins définitif.

Traitement

- terminer l'instruction en cours,
- sauvegarder sur la pile et dans cet ordre **R0,R1,R2,R3, R12**, l'@ de retour, **xPSR** et **LR**,
- mettre dans le registre **LR** un code spécifique (0xFFFFFFFx),
- récupérer le n° de la demande d'interruption et lire l'entrée correspondante dans la *TVI*,
- affecter **PC** avec l'@ trouvée dans la table au n° correspondant.

Mais en C le compilateur est au courant et prévoira les sauvegardes nécessaires.

Rôle du NVIC (Nested Vectored Interrupt Controller)

Unité propre du cortex ARM, son rôle est essentiel. En effet, la table des vecteurs d'interruption peut faire jusque 255 entrées. Cela veut dire que potentiellement 255 événements différents peuvent demander l'intervention de la CPU pour résoudre un problème. Que se passe-t-il si plusieurs demandes arrivent simultanément ? Idem si une nouvelle demande est levée alors qu'une autre est en cours de traitement. Il faut donc un chef d'orchestre pour organiser cela et c'est le rôle du NVIC. Il a pour tâche

- de recevoir les demandes d'interruption,
- de contenir les priorités relatives de chacun des vecteurs de la table,
- de contenir les autorisations de transmette à la CPU les demandes d'interruption,
- de transmettre ou de mettre en attente les demandes en cours selon les autorisations et les priorités respectives

Dans le cahier des charges du Cortex-M3, il existe jusque 240 vecteurs d'interruption dont la priorité peut varier de 0 à 255 sachant que plus le niveau est faible plus priorité est grande. Les 3 premières entrées de la table (*Reset*, *NMI* et *Hard_Fault*) possède un niveau figé (respectivement en -3,-2 et -1) les autres peuvent être programmées (le niveau -3 du *Reset* est donc le plus élevé).

Si l'on considère les choix fait par STMicro pour ses STM32, il n'y a au plus (au maximum) 67 vecteurs d'interruption reconfigurables sur 16 niveaux de priorité (de 0 à 15) plus 6 vecteurs d'exception.

La redéfinition des niveaux de priorité des exceptions (*Bus_Fault,Usage_Fault,*…) se fait à travers deux registres spécifiques. Comme nous n'aurons a priori pas à nous préoccuper de ces exceptions, concentrons-nous sur les vecteurs liés aux périphériques. Leur niveau de priorité se fixe en affectant le contenu des registres **IPx** (x allant de 0 à 17) du NVIC. Ces registres sont organisés comme suit :

	32 24	23 16	15 8	7 0
IPR17	reserved	IP[67]	IP[66]	IP[65]
⋮	⋮	⋮	⋮	⋮
IPRm	IP[4m+3]	IP[4m+2]	IP[4m+1]	IP[4m+0]
⋮	⋮	⋮	⋮	⋮
IPR1	IP[7]	IP[6]	IP[5]	IP[5]
IPR0	IP[3]	IP[2]	IP[1]	IP[0]

Organisation des registres

STMicro pour ces STM32 ne définit le niveau de priorité que sur 4 bits soit donc une valeur entre 0 et 15. Ces 4 bits correspondent aux 4 bits de poids forts des différent octets (notés **IP[k]**) de ces 21 registres.

Concrètement pour affecter la priorité 11 au timer n°2 comment faire ?

Connaitre le numéro de l'interruption lié au périphérique (timer 2) dans la table des vecteurs d'interruption. Reference manual p204 table 63 il s'agit du numéro 28. La priorité se règle en fixant le bit de 4 à 7 du registre 32 bits IPR[7] pour faire cette affectation le fichier core_cm2.h définit la structure logicielle du NVIC suivante :

```
typedef struct
{
  __IO uint32_t ISER[8];                /*!< Offset: 0x000 (R/W)
  Interrupt Set Enable Register          */
  uint32_t RESERVED0[24];
  __IO uint32_t ICER[8];                /*!< Offset: 0x080 (R/W)
  Interrupt Clear Enable Register        */
  uint32_t RSERVED1[24];
  __IO uint32_t ISPR[8];                /*!< Offset: 0x100 (R/W)
  Interrupt Set Pending Register         */
  uint32_t RESERVED2[24];
  __IO uint32_t ICPR[8];                /*!< Offset: 0x180 (R/W)
  Interrupt Clear Pending Register       */
  uint32_t RESERVED3[24];
  __IO uint32_t IABR[8];                /*!< Offset: 0x200 (R/W)
  Interrupt Active bit Register          */
  uint32_t RESERVED4[56];
  __IO uint8_t  IP[240];                /*!< Offset: 0x300 (R/W)
  Interrupt Priority Register (8Bit wide) */
  uint32_t RESERVED5[644];
  __O  uint32_t STIR;                   /*!< Offset: 0xE00 ( /W)
  Software Trigger Interrupt Register    */
} NVIC_Type;
```

Les pseudo registres de 8 bits sont définis dans cette structure inutile d'aller calculer le registre IPR la programmation des différents niveaux est simple :
NVIC->IP[28] = 11<<4
11 est la priorité à imposer et le décalage de 4 bits est fait pour déposer cette valeur dans les 4 bits de poids fort de l'octet IP.

Maintenant il faut autoriser à transmettre l'interruption à la CPU pour chaque vecteurs d'interruptions il existe 5 bits qui gère les autorisation et l'état courant des interruptions. 5 bits répartis dans 5 familles de registre sur le modèle suivant :

	SET ENABLE	CLEAR ENABLE	SET PENDING	CLEAR PENDING	ACTIVE BIT read only
0-31	ISER[0]	ICER[0]	ISPR[0]	ICPR[0]	IABR[0]
32-63	ISER[1]	ICER[1]	ISPR[1]	ICPR[1]	IABR[1]
64-67	ISER[2]	ICER[2]	ISPR[2]	ICPR[2]	IABR[2]

Dans ce tableau la première colonne fait correspondre chaque numéro d'interruption avec un des 32 bits du registre concerné.
Les registres **IABR[x]** contiennent l'état actif de chaque interruption. Si elle a été détectée et servie le bit correspondant passe à un. Ces registres ne sont accessibles qu'en lecture puisque qu'il est mis à 1 suite à la mise en place d'un détournement.
Les 4 autres registres fonctionnent par paire *SET/RESET*. C'est une technique classique qui permet d'éviter certaines erreurs d'écriture. Pour activer la fonction il faut mettre à 1 le bit *SET* mais sa mise à 0 ne le désactive pas. Pour désactiver il faut imposer un reset et donc mettre à 1 le bit de *RESET*.
La paire **ISER[x]/ICER[x]** permet ainsi de gérer l'autorisation de déclenchement de l'interruption concernée.

La prise en compte d'une interruption est vue ici du côté du gestionnaire NVIC. Il faut savoir qu'au niveau du périphérique il y a également des bits qui valident en local la remontée de l'interruption vers le NVIC. C'est donc en quelque sorte un circuit avec deux interrupteurs en série.

La paire **ISPR[x]/ICPR[x]** gère quant à elle la mise en attente. En effet quand une interruption intervient et qu'elle n'est pas prioritaire elle est mise dans cet état dit d'attente afin d'être servie au moment où le niveau de priorité courant le permet. Jouer sur l'état de mise en attente à travers **ISPR** est un moyen logiciel pour déclencher une interruption. Cela peut être utile dans la phase de test des procédures de traitement par exemple.

Reprenons la mise en place de l'interruption du timer n°2. Pour l'autoriser il faut affecter le bit 28 du registre ISER[0] comme :

```c_cpp
NVIC->ISER[0] = NVIC->ISER[0]  | (0x01 << 28);
```

On peut également utiliser les masques prédéfinis dans le fichier *stm32f10x.h* soit comme un véritable masque :

```c_cpp
NVIC->ISER[0]  = NVIC->ISER[0]   | NVIC_ISER_SETENA_28;
```

Soit en se rappelant que ce sont des registres de *SET/RESET*, donc que le zéro n'a pas d'effet. Par conséquent une affectation directe du masque est dans ce cas possible (faisant économiser l'opération de OU logique :

```c_cpp
NVIC->ISER[0]  = NVIC_ISER_SETENA_28;
```

Ecriture d'un handler

Maintenant que nous savons autoriser le déclenchement d'une interruption en programmant le NVIC, reste à déterminer la technique qui permette de « remplir » la table des vecteurs d'interruption avec l'adresse de nos propres routines.
Tout d'abord oublions que cela puisse se faire dynamiquement, c'est à dire que le programme vienne lui-même réécrire (dans une phase d'initialisation ou pour faire évoluer dynamiquement le contexte de l'application) dans cette table. C'est de prime abord impossible puisque par défaut la table est en mémoire 0x0000000 c'est-à-dire en mémoire morte non accessible en écriture. Cela reste cependant techniquement possible : il faut alors déplacer la table et reconfigurer le Cortex pour lui indiquer le nouvel emplacement. Nous ne développerons pas ce point ici.
Il faut donc que la table soit construite par le duo compilateur/linker afin d'être ensuite chargée en même temps que le chargement de l'application
Comme nous l'avons déjà indiqué, la table est créée par défaut avec des procédures « puits » de type boucle infinie. Pour les interruptions liées aux périphériques, il s'agit même que d'une seule procédure appelée *Default_Handler*. Cette procédure par défaut est ensuite renommée avec les noms qui seront ensuite utilisés pour fabriquer la table de vecteurs d'interruption, ce qui donne (en version raccourcie) :

```
Default_Handler PROC

                EXPORT  WWDG_IRQHandler           [WEAK]
                EXPORT  PVD_IRQHandler            [WEAK]
                EXPORT  TAMPER_IRQHandler         [WEAK]
                ...
                EXPORT  TIM2_IRQHandler           [WEAK]
                ...
                EXPORT  USBWakeUp_IRQHandler      [WEAK]

WWDG_IRQHandler
PVD_IRQHandler
TAMPER_IRQHandler
...
TIM2_IRQHandler
...
USBWakeUp_IRQHandler

                B       .
                ENDP
```

Tous ces nouveaux noms recopient la valeur associée au symbole *Default_Handler*. Cela signifie qu'au départ toutes les entrées de la table d'interruption seront affectées avec la même valeur, à savoir l'adresse de cette routine par défaut.

Il faut remarquer que la redéfinition du nom est faite avec l'option **[WEAK]**, c'est-à-dire faible. Cela rend possible ce qui est communément appelé en informatique une surcharge. Si quelque part dans votre projet vous déclarez une nouvelle procédure dont le nom est un des noms prédéfinis pour cette table, le linker va prendre l'adresse de votre procédure pour remplacer l'ancienne. Le doublon de définition de procédure qui normalement devrait mener à une erreur du linker, est ici accepté par le simple fait qu'une des définitions a été déclarée comme faible.

Concrètement pour vous cela signifie qu'il vous faut connaitre le nom générique de la routine (**handler**) de traitement d'interruption qui va être déclenchée par l'usage d'un périphérique. En utilisant le même nom vous pouvez écrire la routine de traitement qui fera le travail à faire lors de la survenue de l'interruption.

Illustrons cela toujours avec le timer n°2. On suppose que ce timer a été correctement configuré pour lever une interruption toutes les secondes et que le NVIC a été programmé pour autoriser la prise en compte. Pour que la variable globale MaSeconde s'incrémente à chaque interruption, il suffit d'écrire en langage C :

```
1 void TIM2_IRQHandler(void)
2 {
3     MaSeconde = MaSeconde +1;
4 }
```

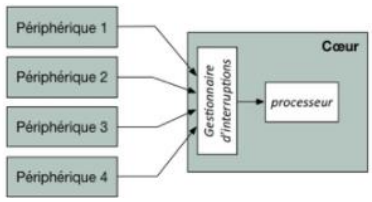
Il est impossible passer des arguments (en entrée ou en sortie) dans une procédure d'interruption ; elles seront donc obligatoirement de type void Fonc(void). En effet par essence, ces fonctions ne sont pas appelées par une autre procédure qui pourrait lui donner ses arguments d'appel. L'emploi de variables globales est une solution pour partager des ressources entre les routines d'interruption et le reste de l'application.

On peut maintenant

- Expliquer le cheminement d'une exécution lors d'une interruption,
- Identifier les éléments nécessaires à la configuration d'une interruption et à son traitement.

INTERRUPTIONS

Le processeur dispose d'un gestionnaire d'interruption (NVIC) dont le rôle est d'associer à chaque interruption sa fonction de traitement. Les interruptions proviennent des périphériques.



Mécanisme d'interruption :

Autoriser le périphérique à lancer une interruption.

Configurer le processeur pour gérer l'interruption

Ecrire le code associé à l'interruption.

On utilise la LED pour réaliser l'interruption comme le code suivant lequel le timer 2 est configuré pour déborder toutes les secondes et la broche 5 du port A est configurée en sortie pour contrôler la LED USER.

```c
#include "stm32f10x.h"
int main (void)
{
    // Configuration de la broche en sortie
    RCC->APB2ENR |= RCC_APB2ENR_IOPAEN;
    GPIOA->CRL &= ~(0xF << 4*5);
    GPIOA->CRL |= (0x1 << 4*5);

    // Configuration du timer
    RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;
    TIM2->ARR = 9999;
    TIM2->PSC = 7199;

    // Lancement du timer
    TIM2->CR1 |= TIM1_CR1_CEN;
    while(1)
    {
    }
    return 0;
}
```

Autorisez l'interruption sur un périphérique

Le débordement d'un timer peut être la source d'une interruption.

(doc p.409) nous indique que le bit 0, noté UIE, du registre DIER, permet de contrôler le lancement de l'interruption d'un timer il faut le passer à 1.

On ajoute la ligne avant de lancer le timer

```c
...
TIM2->DIER = TIM2->DIER | (1 << 0);
TIM2->CR1 |= TIM1_CR1_CEN;
...
```

Ou équivalent

```c
TIM2->DIER |= TIM_DIER_UIE;
```

Une demande d'interruption sera transmise au NVIC à chaque fois que le timer déborde.

Configurez l'interruption dans le cœur

Chaque source est identifiée par un numéro, nécessaire pour configurer la gestion des interruptions au niveau du processeur. Le tableau p.197 à 199 indique pour chaque périphérique le numéro de l'interruption associé. L'interruption du timer 2 est ainsi identifié par le numéro 28.

Les informations de configuration des interruptions au niveau du cœur ne sont pas disponibles dans le document TM0008 mais dans Programming manuel (PM0056) qui décrit les informations liées au Cortex M3.

https://www.st.com/content/ccc/resource/technical/document/programming_manual/5b/ca/8d/83/56/7f/40/08/CD00228163.pdf/files/CD00228163.pdf/jcr:content/translations/en.CD00228163.pdf

chapitre Nested vectored interrupt controller (NVIC informations de configuration des interruptions) p.118. il comprend un ensemble de registre au même titre que ceux des périphériques pour seul différence qu'ils appartiennent au cœur.

La déclaration des emplacements mémoires de ces registres est spécifiée dans le fichier core_cm3.h inclus dans le fichier stm32f10x.h
La documentation sur le cortex M3 confirme que le groupe de registres ISER a pour fonction d'activer et de désactiver l'autorisation du traitement des différentes interruptions. Chaque bit de ces registres est associé à une interruption ce cortex M3 est de 81, ce groupe de registre est décomposé en 3 éléments décrits dans la structure NVIC par un tableau.
Pour activer l'interruption 28 il faut mettre le bit 28 du registre ISER[0] comme
NVIC->ISER[0] = NVIC->ISER[0] | (1 << 28); OU NVIC->ISER[0] = NVIC->ISER[0] | NVIC_ISER_28;
Ça suffit pour activer une interruption au niveau du cœur et chaque irq a une priorité c'est au niveau du registre IPR du NVIC. Il existe le fichier core_cm3.h une union nommée IP permettant d'accéder directement à l'emplacement des bits de configuration des priorités. Exemple : NVIC -> IP[28] fixe la priorité de l'interruption 28. Les priorités sont codées sur 8 bits mais seul les bits 7 à 4 sont pris en considération. Donc si on veut fixer la priorité à 7 de l'interruption 28 il faut ajouter le code : NVIC->IP[28] = NVIC->IP[28] | (7 << 4);

Associez du code à l'interruption

```c
#include <stm32f10x.h>
int main (void)
{
    // Configuration de la broche en sortie
    RCC->APB2ENR |= RCC_APB2ENR_IOPAEN;
    GPIOA->CRL &= ~(0xF << 4*5);
    GPIOA->CRL |= (0x1 << 4*5);

    // Configuration du timer
    RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;
    TIM2->ARR = 9999;
    TIM2->PSC = 7199;
    TIM2->DIER |= TIM_DIER_UIE;

    //Configuration de l'interruption sur le coeur
    NVIC->ISER[0] |= NVIC_ISER_SETENA_28;
    NVIC->IP[28] |= (7 << 4);

    // Lancement du timer
    TIM2->CR1 |= TIM_CR1_CEN;
    while(1)
    {
    }
    return 0;
}
```

Le programme permet au timer 2 de lancer une interruption et au cœur de la traiter. Il faut à cela ajouter le code qui sera réalisé pendant cette interruption.

La fonction d'interruption du timer 2 est prototypée par : void TIM2_IRQHandler(void);
Le nom `TIM2_IRQHandler` est l'étiquette associée à l'adresse mémoire de l'interruption (déclarée dans le fichier *stm32f10x.s*). Lors de la compilation, cette étiquette prend la valeur de l'adresse mémoire de la fonction `TIM2_IRQHandler`. Ainsi, quand l'interruption survient, le processeur lit l'adresse contenue dans `TIM2_IRQHandler` et saute à cette adresse, permettant ainsi l'exécution de la fonction.

Revenons à la déclaration de la fonction d'interruption et ajoutons le code nécessaire au traitement que l'on souhaite réaliser. Le but est de faire clignoter la LED branchée sur la broche 5 du port A, soit :

```c
void TIM2_IRQHandler(void) {
    GPIOA->ODR = GPIOA->ODR ^ (1 << 5);
}
```

Ce code n'est cependant pas opérationnel car l'interruption est toujours valide au niveau du périphérique et donc une nouvelle interruption va immédiatement être relancée quand le traitement du *handler* sera terminé, bouclant ainsi indéfiniment dans la fonction d'interruption.
Il faut donc valider le traitement dans la fonction d'interruption en mettant à 0 le bit UIF du timer 2, soit :

```
void TIM2_IRQHandler(void) {
    TIM2->SR &= ~TIM_SR_UIF;
    GPIOA->ODR = GPIOA->ODR ^ (1 << 5);
}
```

Vous pouvez maintenant compiler et exécuter le code en debug pour observer le clignotement de la LED, ainsi que l'activation de l'interruption.

Scruter ou interrompre un évènement

Mais pourquoi s'embêter avec une interruption alors que mon code fait exactement la même chose qu'avant en scrutant l'état du timer ?

C'est vrai, mais la scrutation est globalement inefficace car le processeur va passer la plupart de son temps à attendre qu'un état change, alors qu'il pourrait faire autre chose !

De plus, l'interruption permet un traitement plus réactif car le traitement va survenir dès que l'interruption se produit. Alors que dans le cas de la scrutation, il faut attendre que toute la boucle soit parcourue avant de détecter le changement (dans notre cas la boucle ne fait pas grand-chose, mais le traitement pourrait être beaucoup plus long). Si le signal à observer change très vite, il se pourrait donc que la scrutation ne permette pas d'observer ce changement.

Donc, essayez, dans la mesure du possible, de favoriser une implémentation avec des interruptions.

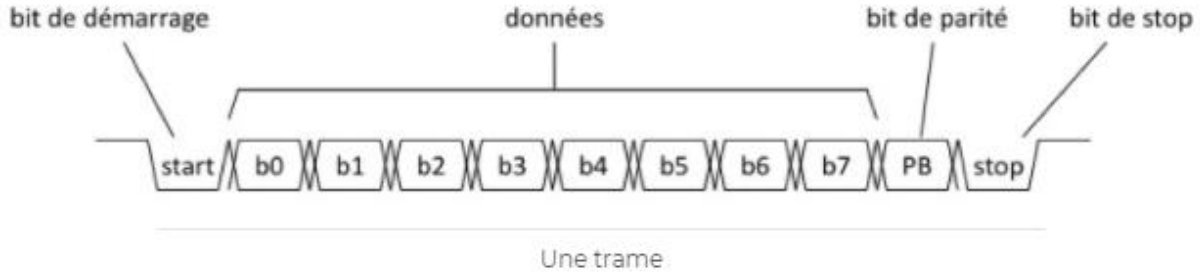
Maintenant on sait :

- Expliquer le fonctionnement d'une interruption,
- Configurer une interruption sur un STM32 et d'y associer une fonction de traitement.

COMMUNICATION

Communication en série
On trouve une liaison série (UART) et des bus industriels comme l'I2C, SPI, CAN et d'autres plus communs en informatique sont aussi disponibles tels que l'USB, Ethernet.

(UART)
Une liaison série consiste à envoyer une information bit après bit avec un délai entre chacun.
Le bus Universal Asynchronous Receiver Transmitter est une implémentation de ce principe.
La communication se fait par l'envoi d'une trame, un paquet de bits que l'on ne peut pas subdiviser.
Le protocole RS232 définit une trame comme étant constituée de ;
- un bit de démarrage (start bit) dont le niveau logique est zéro.
- entre 7 et 8 bits de données.
- Eventuellement un bit de parité paire ou impaire (parity)
- et un ou deux bit de stop avec le niveau logique de zéro (stop bits)



Le bit de parité peut être utilisé pour détecter des erreurs.
La parité est générée par l'émetteur et vérifiée par le receveur.
Pour une parité paire, le nombre de 1 dans la donnée plus le bit de parité est pair,
alors que pour
une parité impaire le nombre de 1 de la donnée plus celui de la parité est impaire.

Utiliser une parité implique un surcoût temporel à la communication, ainsi, si la probabilité d'apparition d'une erreur est faible et qu'elle n'entraîne pas de conséquence grave pour le système, ne pas utiliser de parité est à privilégier.

Caractéristiques d'une liaison série

Dans une communication UART, les horloges entre le receveur et l'émetteur ne sont pas communes (Asynchrone).
Ainsi, afin d'identifier les différents bits d'une trame, le récepteur se base sur la durée de ses bits. Cette durée doit donc être fixée et identique pour l'émetteur et le récepteur.
La **durée d'un bit** est le temps entre chaque bit.
Ainsi l'émetteur émet un bit, attend la durée d'un bit puis émet un nouveau bit.
Le receveur attend jusqu'à ce qu'il détecte un bit de démarrage puis attend la moitié d'une durée d'un bit pour se caler sur la lecture des bits qu'il reçoit.

En général, la caractéristique associée à la durée d'un bit est décrite en terme de *baud rate* (unité bps), c'est-à-dire le nombre de symboles (ici des bits) par seconde qui est transmis par la communication série. On a donc

$$\text{baud rate} = \frac{1}{\text{durée d'un bit}}$$

La bande passante *(bandwith)* d'une communication quantifie l'information transmise sans prendre en considération les surcoûts induits par les bits de démarrage, de parité et de stop, soit

$$\text{Bande passante} = \frac{\text{nombre de bits de la donnée par trame}}{\text{nombre de bits total de la trame}} \text{baud rate}$$

Une autre caractéristique de la communication est de savoir si le système est **full duplex**, c'est-à-dire si des informations peuvent être transmises dans les deux sens simultanément, ou **half duplex**, c'est-à-dire si l'information ne peut transiter que dans un sens à un même moment et qu'en cas de collision (les deux systèmes envoient des informations en même temps) un bus doit assurer la retransmission.

Une communication UART est par définition **asynchrone**, c'est-à-dire que l'émetteur et le récepteur ne partagent pas la même horloge et donc qu'elles peuvent dérivées l'une par rapport à l'autre. Cela ne pose normalement pas de problème si les horloges ne diffèrent pas trop car une synchronisation à lieu à chaque bit de démarrage. Par contre cela limite la bande passante.

Un mode de communication dit **synchrone**, ou parle alors d'USART pour *Universal Synchronous Asynchronous Receiver Transmitter*, permet d'avoir des baud rates élevés sans l'apparition de problèmes dus aux dérives d'horloge. Dans ce mode, une horloge commune est partagée par les deux systèmes communicants. Cela permet aussi de réduire les surcoûts dus au bit de démarrage et de stop car les données peuvent être plus longues. Par contre cela nécessite de partager physiquement le signal de l'horloge entre l'émetteur et le récepteur.

Configuration une liaison série sur STM32F103

Le microcontrôleur STM32F103 dispose d'un périphérique USART full-duplex avec la possibilité de sélectionner un baud rate pouvant aller jusqu'à 4.5 Mbps.

Une communication bidirectionnelle via l'USART nécessite au moins deux broches :

- une broche, notée RX, pour recevoir les bits,
- une broche, notée TX, pour transmettre les bits. Quand l'émetteur est désactivé, la broche prend l'état imposée par la configuration du port, alors que si l'émetteur est activé et que rien n'est à transmettre, la broche TX est dans l'état haut.

Par la suite, vous allez configurer un périphérique USART en émetteur avec 1 bit de démarrage, 8 bits de données et 1 bit de stop (pas de bit de parité) avec un *baud rate* de 9600 bps. Pour cela la documentation indique page 797 qu'il faut :

- activer l'USART en mettant à 1 le bit UE du registre CR1,
- choisir la taille des données (8 ou 9 bits) à l'aide du bit M du registre CR1,
- indiquer le nombre de bits stop dans le champ de bits STOP du registre CR2,
- sélectionner le baud rate à l'aide du registre BRR,
- mettre le bit TE de CR1 à 1 pour envoyer la première trame d'attente.

Pour régler le *baud rate*, la documentation page 803 montre qu'il faut diviser la fréquence d'entrée du périphérique, 72 MHz dans notre cas, par une valeur représentée par un nombre à virgule fixe. Le tableau 192 page 804 donne comme diviseur la valeur 468,75 pour obtenir un baud rate de 9600bps. Cette valeur doit être saisie dans le registre BRR avec les 4 premier bits utilisé pour représenter la partie fractionnaire et les 12 suivants la partie entière.

Tout cela peut donc se traduire par la fonction :

```
void configure_usart1_9600bps(){
    RCC->APB2ENR |= RCC_APB2ENR_USART1EN; // validation horloge USART1
    USART1->CR1 |= USART_CR1_UE; // Activation de l'USART
    USART1->CR1 &= ~USART_CR1_M; // Choix d'une taille de 8 bits de données
    USART1->CR2 &= USART_CR2_STOP; // Choix d'un seul bit de stop
    USART1->BRR |= 468 << 4; // Fixe le baud rate à 9600bps
    USART1->BRR |= 75; // Fixe le baud rate à 9600bps
    USART1->CR1 |= USART_CR1_TE; // Envoi de la première trame d'attente
}
```

Pour envoyer une donnée, il faut ensuite :

- écrire la donnée dans le registre DR,
- attendre que le bit TC du registre SR passe à 1 ce qui indique que la dernière trame a été transmise.

Cela s'écrit :

```
void send(char data){
    USART1->DR |= data; // Ecriture de la donnée dans le registre DR
    while(!(USART1->SR & USART_SR_TC)) {} // Attente de la fin de transmission
}
```

Si vous voulez transmettre des données, il ne faut pas non plus oublier de configurer la broche connectée à l'USART, ici la broche PA.9 en mode *alternate output push-pull*. Pour cela réutilisez la fonction écrite dans le chapitre sur le PWM, mais en la rendant plus générique afin de configurer n'importe quelle broche de n'importe quel port en *alternate output push-pull*, soit :

```
int configure_gpio_alternate_push_pull(GPIO_typdef *gpio, int pin){
    If (gpio == GPIOA) {
        RCC->APB2ENR |= RCC_APB2ENR_IOPAEN;
    } else if (gpio == GPIOB){
        RCC->APB2ENR |= RCC_APB2ENR_IOPBEN;
    } else if (gpio == GPIOC){
        RCC->APB2ENR |= RCC_APB2ENR_IOPCEN;
    } else {
        return -1;
    }
    if (pin < 8){
        gpio->CRL &= ~((0xF << 4*pin)| (0xF << 4*pin));
        gpio ->CRL |= (0xA << 4*pin);
    } else if (pin < 15){
        pin = pin - 8;
        gpio ->CRH &= ~((0xF << 4*pin) | (0xF << 4*pin));
        gpio ->CRH |= (0xA << 4*pin);
    } else {
        return -1;
    }
    return 0;
}
```

Maintenant si vous voulez transmettre les lettres de l'alphabet, il suffit d'écrire le main suivant :

```c
#include "stm32f10x.h"
int main (void)
{
    char data = 'A';
    int cmpt = 0;
    configure_gpio_alternate_push_pull (GPIOA, 9);
    configure_usart1_9600bps ();
    while (cmpt < 26)   {
        send(data+cmpt);
        cmpt ++;
    }
    while (1);
    return 0;
}
```

Autres bus de communication :

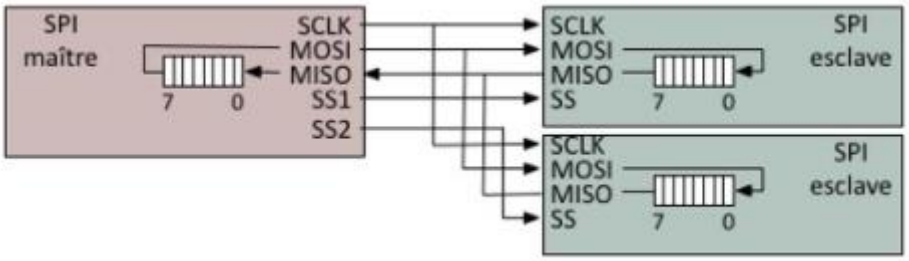
Le bus SPI

Le bus SPI *(Synchronous Peripheral Interface)* est un bus série assurant une communication en full-duplex synchrone. Il est multipoint avec un maître communiquant avec plusieurs esclaves. La communication est réalisée entre le maître et un esclave, le choix de l'esclave se faisant par une sélection directe (donc sans adressage). La communication est unidirectionnelle dans le sens où c'est toujours le maître qui initie la communication.

La ligne entre le maître et les esclaves est constituée de quatre fils :

- SCLK (Serial Clock): l'horloge du bus qui est produit par le maître,
- MOSI (Master Out Slave In) : la ligne de donnée du maître vers l'esclave,
- MISO (Master In Slave Out) : la ligne de donnée de l'esclave vers le maître,
- SSn (Slave Select n) : la ligne permettant de sélectionner l'esclave destinataire de la communication.

Le principe de communication du SPI repose sur des registres à décalage présents côté maître et côté esclave. La taille du registre peut être configurée (dans le cas du STM32 par exemple ce registre peut-être de 8 ou 16 bits). À chaque période d'horloge, un bit est transféré du maître vers l'esclave et réciproquement de l'esclave vers le maître. Dans le cas d'un registre de 16 bits, il faut donc 16 périodes d'horloge pour effectuer le transfert des données.



La synchronisation se faisant sur l'horloge, la communication est initiée par les front de l'horloge. Quatre modes de transmission existent en fonction de l'état au repos de l'horloge et du front de l'horloge sur lequel se fait la transmission. Pour qu'une communication fonctionne, il faut que le maître et l'esclave partage le même mode.

Bien qu'ayant des débits élevés, ce bus présente l'inconvénient de ne fonctionner que sur des faibles distances (en général il est utilisé pour de la communication entre des éléments sur la même carte) et qu'il n'y pas d'acquittement, donc le maître ne sait pas s'il est écouté.

Au niveau d'un STM32F103, un périphérique est dédié au SPI et est facilement configurable à l'aide de quelques registres. La gestion et la lecture des données est ensuite à gérer au niveau de l'applicatif, par exemple en les traitant dans l'interruption de fin de transmission ou à l'aide de la DMA pour des fréquences élevées.

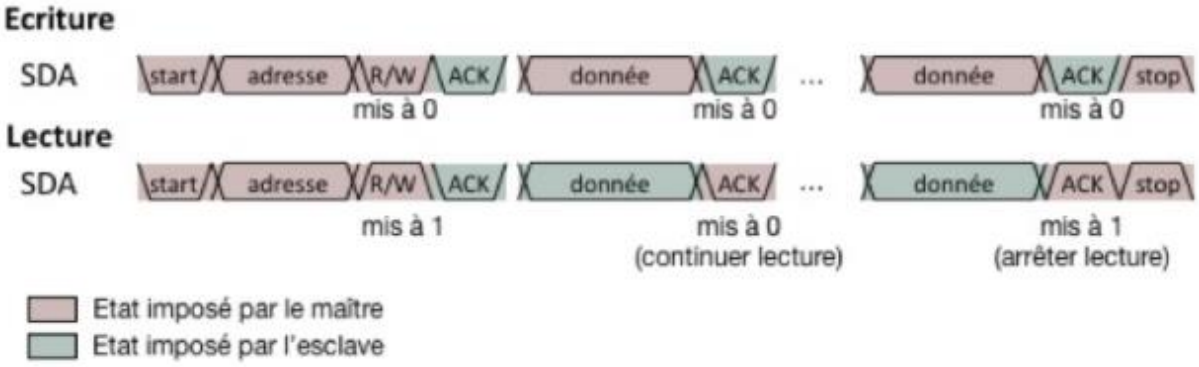
Le bus I2C

Le bus I2C *(Inter-Intergrated Circuit)* est un bus série synchrone half-duplex. Il est multi-maître et multi-esclaves et n'utilise que deux lignes (et une masse commune) :

- SCL (Serial Clock) : l'horloge commune à tous les équipements,
- SDA (Serial Data) : la ligne portant les données.

Les lignes étant au niveau logique haut quand il n'y a pas de communication, les broches connectées aux lignes DCL et SDA doivent être en open-drain.

Lorsqu'un maître émet sur la ligne SDA, la trame qu'il envoie est décomposée en un bit de démarrage, suivi de l'adresse de l'esclave (7 ou 10 bits) puis d'un bit indiquant si le maître souhaite envoyer une donnée ou en recevoir de l'esclave. À chaque envoi d'un octet de donnée (maître ou esclave) un bit d'acquittement est produit par le récepteur. La communication se termine par un bit de stop.



Dans le cas où il y a plusieurs maîtres, un mécanisme d'arbitrage est ajouté. Ainsi lorsqu'un maître commence à transmettre un message, les autres maîtres ne peuvent pas émettre tant que le bit stop n'est pas arrivé. Les maîtres monitorent aussi ce qu'ils émettent afin d'éviter le cas où deux maîtres accèdent simultanément au bus. Ainsi si un maître constate que le niveau est bas alors qu'il a envoyé un bit haut, c'est qu'un autre maître émet. Le maître constatant ce cas interrompt alors immédiatement sa transmission.

Le STM32 dispose d'un périphérique I2C et peut être configuré comme maître et esclave. Le fonctionnement du bus est entièrement pris en considération par le circuit, et ST fournit un ensemble de librairies permettant de facilement le configurer.

Le bus CAN

Le bus CAN (Controller Area Network) est un bus série half-duplex couramment utilisé dans l'industrie automobile et avionique. La transmission suit le principe de transmission en paire différentielle et possèdent donc deux lignes CAN L (CAN LOW) et CAN H (CAN HIGH). Tous les équipements, appelés nœuds, souhaitant communiquer via le bus sont connectés et peuvent échanger des informations.

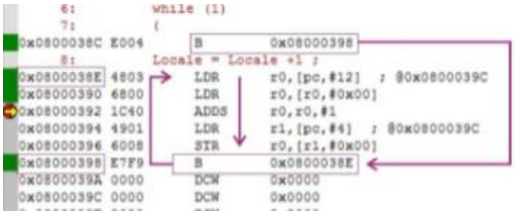
Lors de la communication, un nœud envoi son message à l'ensemble des autres nœuds. Ces derniers filtrent ensuite le message et le traitent ou non. Lors d'un envoi de message, les nœuds pouvant demander l'accès simultané au bus, un principe d'arbitrage basé sur la priorité est mis en place. Ainsi, chaque trame transmise porte un champ d'arbitrage de 12 ou 30 bits. Remarquez que ce champ doit être unique pour chaque message du réseau. Si plusieurs nœuds tentent de communiquer en même temps, le message ayant la plus petite valeur est le plus prioritaire. Une fois qu'un message est envoyé plus aucun nœud ne peut transmettre de donnée avant la fin du message. Si un nœud perd l'arbitrage, son message est de nouveau envoyé dès que le bus est de nouveau disponible.

La taille des données d'un message varie de 0 à 64 bits. À cela s'ajoute dans une trame, des bits CRC pour vérifier l'intégrité des données transmises ainsi que d'autres bits pour indiquer la taille de la donnée, pour valider la transmission et pour indiquer la fin.

De nombreux autres mécanismes sont aussi utilisés afin d'assurer la fiabilité du bus et de maîtriser les cas d'erreur. Tous ces mécanismes assurent une grande disponibilité du bus et il peut ainsi être utilisé pour des applications industrielles ayant besoin d'être sûr, ce qui est par exemple le cas pour des applications dans le domaine de l'automobile ou de l'avionique.

Programmation C ET SURTOUT DEBOGAGE

Programmation sur Registre



ADD addition

ADDS incrémente un registre

En C : locale = locale +1 ;

ARM conçoit des architectures de type Load/Store

Les accès mémoire ne se font qu'à travers 2 instructions pour l'accès mémoire :

LDR load register

LDR R0,[R0,#0] L'ALU va lire à l'adresse contenu dans R0 décalée de 0 octet (donc l'adresse elle m^me, le décalage ici est inutile) une donnée 32bits qu'il stockera dans R0 (écrasant l'ancien contenu)

STR store register

STR R0,[R1,#0] l'adresse de destination est ici préalablement stockée dans R1 et le contenu de R0va être transféré à cette adresse

LDR R0,[PC,#12] et LDR R1,[PC #4] permet de pré-charger les registres (pointeurs) avec l'adresse de la variable Locale.

Celle-ci a été déposée par l'assembleur avec le code lors de la création de l'exécutable respectivement à 12 octets et à 4 octets de l'emplacement de l'instruction en elle-même.

L'architecture load/store et la limitation à des accès par indirection est une spécificité des processeurs ARM. Dans d'autres « L.A. » il n'en est pas de même et il est alors possible de trouver des instructions du type ADD R1, MaVar pour additionner le contenu de la variable MaVar avec **R1**. L'ALU est ainsi capable de faire une opération arithmétique en accédant directement à la case mémoire.

Ce n'est pas le cas ici, nous aurons donc en permanence une décomposition en trois temps pour ce genre d'opération : récupération de l'adresse de la variable, lecture de la variable dans un registre « miroir », exécution de l'opération sur la copie.

C'est fastidieux mais c'est incontournable !!!!!!!!!!

Exploration de mémoire d'une architecture ARM

(Opérandes et accès mémoire)

Opérandes (3 types)

Des Registres (la plupart du temps les généraux R0 à R12)

```
SUBS R4,R2,R5   //R4 = R2-R5
```

Les Valeurs Immédiates (#)

```
ORR R11,#0x3C   //R11 = R11 OU 0x3C  (OU Logique)
```

Des adresse mémoires se rencontrent que pour LDR et STR (indirection l'adresse est dans un registre)

Syntaxe le registre est encapsulé dans des crochets pour indiquer l'indirection.

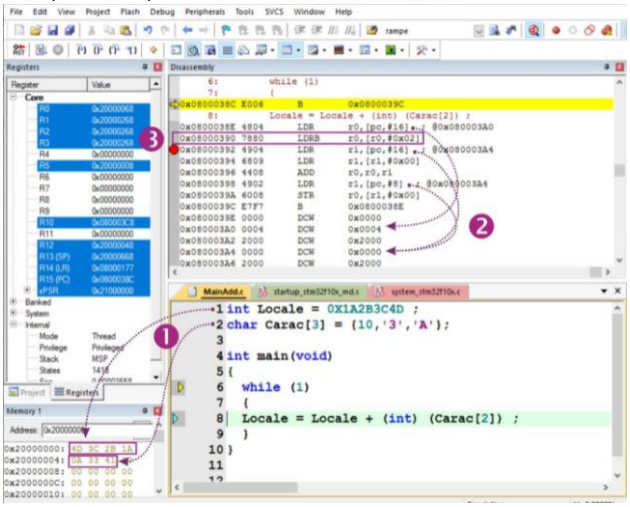
```
LDR R7,[SP] // R7 sera chargé avec la valeur pointée par le pointeur de pile SP.
```

Opérandes immédiat, la valeur est stockée directement dans l'instruction elle-même, pour 16 bits mais pour les valeurs codées 32bits il n'est plus possible d'utiliser des opérandes immédiats même en codage d'instruction sur 32bits si tous les bits sont occupés pour contenir les données immédiates, il ne reste plus de place pour mettre le code propre de l'instruction ni le numéro du registre du second opérande.

L'assembleur va décider de stocker la valeur à récupérer en fin du code zone appelée literal pool.

Le processeur va accéder à la valeur non plus en décodant l'instruction, mais en réalisant une lecture mémoire dans l'espace réservé au code. L'information est par conséquent toujours stockée dans le code mais elle ne se trouve plus directement imbriquée dans le code de l'instruction. Par contre, un accès mémoire supplémentaire est nécessaire pour la récupérer.
Opérandes mémoire, l'accès est en indirection donc une instruction *doit récupérer l'adresse de la variable dans un registre* elle se fait avec de l'adressage de type immédiat (l'assembleur connait les adresses où sont stockées les variables) et donc elle se traduit par une lecture en literal pool.
Maintenant que l'adresse est dans un registre les accès mémoire se font sans souci (via LDR et STR) LDR R6,[R3]
Entre crochet c'est le second registre opérande contenant l'adresse.

Exemple de compréhension



A partir de l'initialisation on peut voir que les variables ont été créées en tout début de la mémoire RAM du STM32 (1) (logique l'entier locale occupe 4 octets et le tableau Carac 3 octet.
!! surprenant les différents octets composant la valeur initiale sont rangés à l'envers (convient little endian qu'utilise ARM pour ranger les valeurs mémoire.
Une valeur codée sur 32bits occupe 4 adresses consécutive :
La première utilisée pour stocker le poids faible, la seconde pour l'octet de poids suivant et ainsi de suite, par contre dans le cas du tableau Carac rien de tel (0X0A correspond à 10 en hexa et le code ASCII de lza chiffre 3 est la valeur 0X33 celui de la lettre A est (0X41)chaque case du tableau correspond à un seul octet l'inversion de poids faible et de poids for n'est pas nécessaire.

A savoir : **la mémoire est alignée**, cela signifie que pour accéder à une valeur codée sur 16bits (short int) l'adresse la plus basse (celle qui contient les 8 bits de poids faible) doit être paire (donc se termine par un bit 0). Pour une valeur 32bits (Locale) l'adresse la plus basse doit être doublement paire (2 derniers bits de l'adresse sont à 0) c'est le cas pour Locale qui occupe l'adresse 0X20000000. Pour le 64 bits le problème ne se propage pas (double par exemple) car le processeur ne peut lire que 32 bits à la fois. Lire 64 bits se fait par 2 lecture successives de 32 bits il suffit que les 64 bits soient rangés à une adresse doublement paire pour que cela fonctionne.
Le point ❸ montre comment on peut accéder à un octet (char) d'un tableau. Déjà en utilisant l'instruction **LDRB**. En effet **LRD** se décline en :

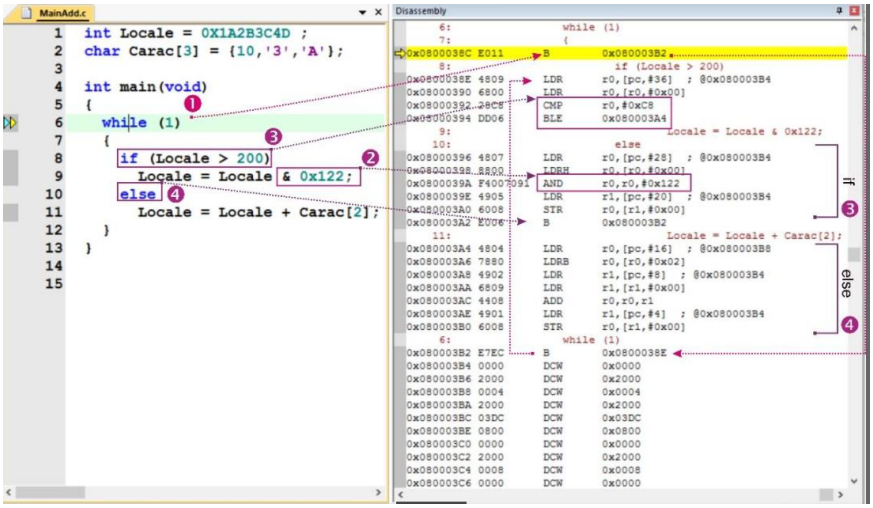
- **LDRB** pour lire un octet non signé (les bits 8 à 31 du registre seront mis à 0),
- **LDRSB** pour lire un octet signé (les bits 8 à 31 recopieront le bit de poids fort de l'octet lu —c'est-à-dire le 8ème bit de l'octet ce qui correspond à son signe—, pour que la valeur signée sur 32 bits soit la même que celle sur 8 bits),
- **LDRH** pour lire un 16 bits non signés (les bits 16 à 31 du registre seront mis à 0),
- **LDRSH** pour lire un 16 bits signé (les bits 16 à 31 recopieront le bit de poids fort — qui est le signe des 16 bits lus— pour que la valeur signée sur 32 bits soit la même que celle sur 16 bits),
- **LDR** pour lire un 32 bits (signé ou pas car dans ce cas les 32 bits sont tous directement affectés par le transfert).

Pour l'instruction **STR** on trouve les mêmes déclinaisons sauf qu'il n'y a pas l'aspect signé ou pas. L'écriture de 8 (resp. 16) bits en mémoire prend les 8 (resp. 16) bits de poids faible pour les écrire dans 1 (resp 2) adresse(s) mémoire. Il n'y a donc pas d'interprétation de signe à faire.

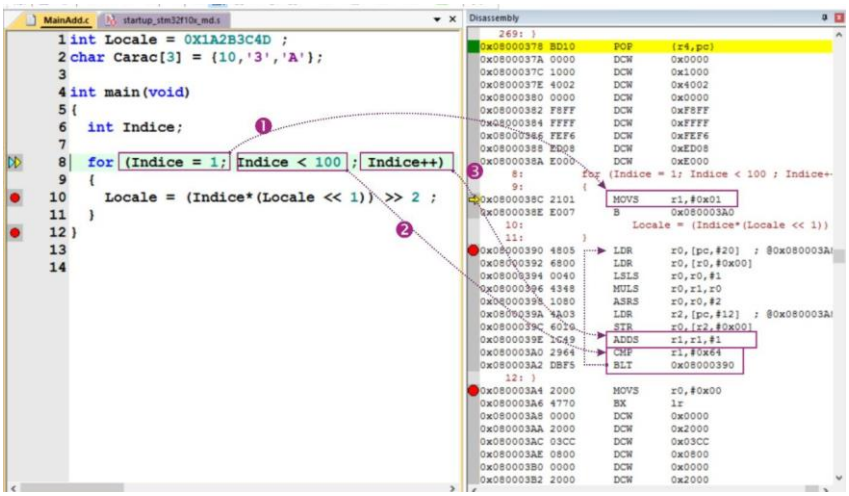
Accès mémoire le mode d'adressage de l'architecture ARM LDR et STR

- **LDR Rt,[Rn]** : Transfert dans Rt du contenu de l'adresse pointée par Rn. C'est la notion de pointeur classique.
- **LDR Rt,[Rn,#±imm8]** : Transfert dans Rt du contenu de l'adresse contenue dans Rn décalée de ±imm8. Rn n'est pas modifié par ce transfert. Cela permet d'adresser des structures de données où l'on connaît l'adresse de base de la structure et où les différents éléments qui la composent sont rangés en position relative par rapport à cette adresse de base.
- **LDR Rt,[Rn,Rm]** : Transfert dans Rt du contenu de l'adresse pointée par Rn décalée du contenu de Rm. Ni Rn ni Rm ne sont modifiés par ce transfert. Ce mode d'adresse est idéal pour la gestion de tableaux où l'indice du tableau est géré par le registre Rm.
- **LDR Rt,[Rn],#±imm8** : Transfert dans Rt du contenu de l'adresse pointée par Rn puis ajout de ±imm8 à Rn. Le registre pointeur Rn est post-déplacé, c'est-à-dire déplacé après avoir transféré le contenu de l'adresse pointée par Rn Cela correspond aux pointeurs d'incrémentation **(Ptr++)* du langage C. Ce mode d'adresse est parfait pour parcourir une zone mémoire avec un pointeur.
- **LDR Rt,[Rn,#±imm8] !** : Ajout de ±imm8 à Rn puis transfert dans Rt du contenu de l'adresse nouvellement pointée par Rn. Le registre pointeur Rn est pré-déplacé, c'est-à-dire déplacé avant de transférer le contenu de la nouvelle adresse adresse pointée par Rn. Grande similitude avec le cas précédent. Ce mode d'adresse est notamment utile pour la gestion de piles LIFO à pré-décrémentation.

Exemple de structure

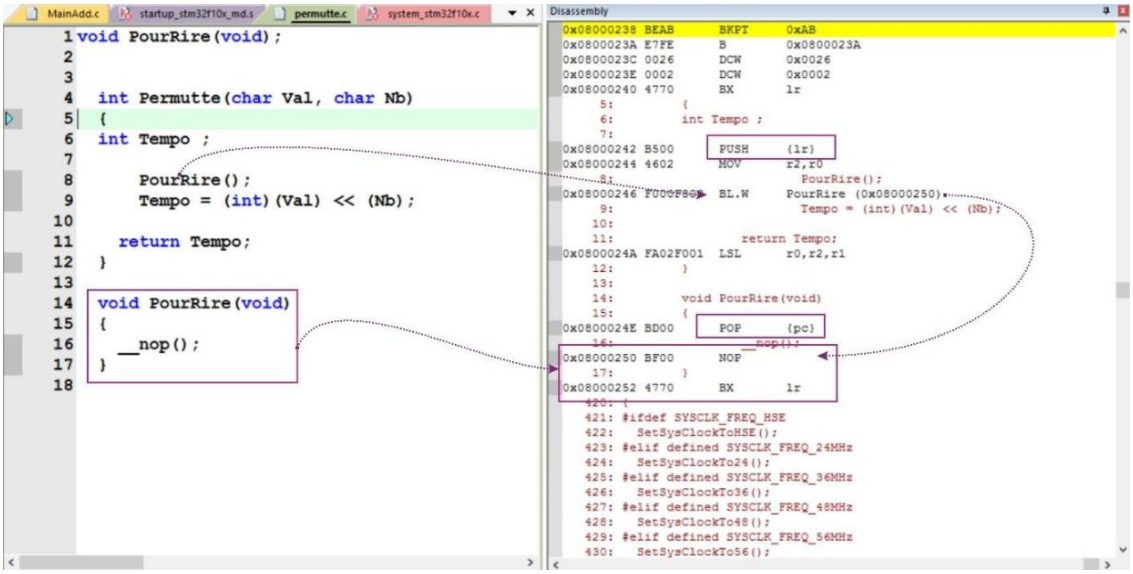


La condition est liée à l'état des fanions *(Z,C,N,V)*. Quand la condition est vraie le saut est réalisé et le processeur se branche à l'adresse indiquée. Si la condition est fausse le saut ne se fait pas et le processeur réalise l'instruction qui suit, sans rupture de séquence. Cela permet donc d'éviter une partie de code. Les conditions correspondent à un suffixe qui s'ajoute à **B**.
Dans notre cas nous avons un **BLE** pour *Branch if Less or Equal* qui correspond à un certain état des fanions. Ces fanions ont été mis à jour par l'instruction précédente, soit **CMP R0, #0xC8**. `0xC8` est l'écriture hexadécimale de la valeur décimale `200` et **R0** est le registre qui contient une copie de la valeur de `Locale`. **CMP** effectue la comparaison entre les deux, nous retrouvons bien notre test *if(Locale > 200)*. Un **CMP** est une instruction de soustraction qui ne retourne pas le résultat mais qui affecte les fanions. L'ALU réalise *Locale-200*. Donc notre saut conditionné sera réalisé si *Locale ≤ 200* ; c'est-à-dire que l'on va éviter de faire les instructions ❸ si l'on ne vérifie pas que *Locale >200*. Le « L.A. » oblige la plupart du temps à raisonner en logique inversée.



UTILISEZ LES PROCEDURES ET LA PILE SYSTEME

On ne fait pas la différence entre fonctions, procédures, sous-programme (c'est un saut d'adresse) mais le processeur doit mémoriser l'adresse de départ vers sa procédure car il devra y revenir.



Fonctionnement de la pile système (push, pop, resp écriture, lecture)

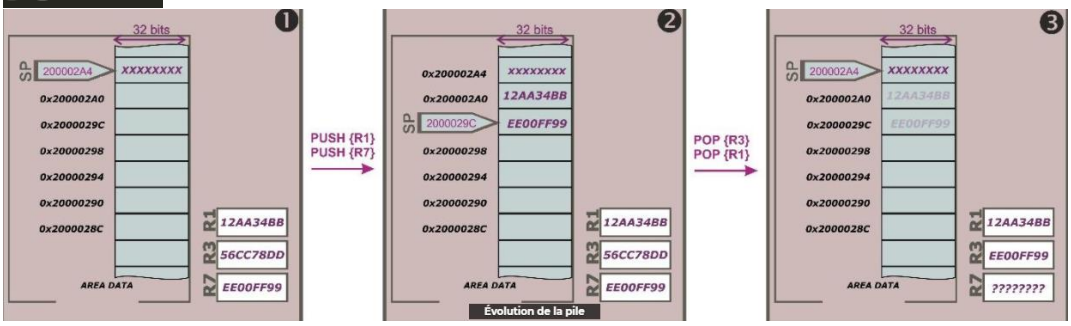
Push {Rx} le pointeur SP est décrémenté de 4 adresses puis le contenu des 32 bits de Rx est déposé dans ces 4 adresses

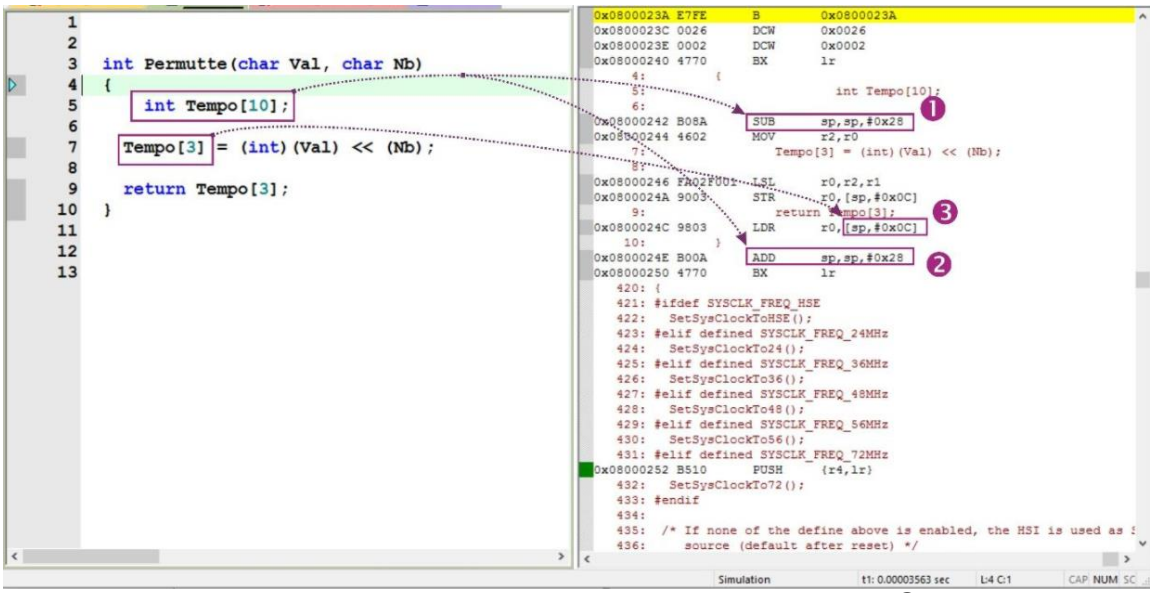
Pop {Rx} lit les 32 bits qu'il voit à l'adresse contenue dans SP puis incrémente le pointeur SP de 4.

Pile d'assiette donc quand on dépose elle est au sommet (écriture) et en lecture c'est celle qui est sur le haut de la pile qui est prise.

Elle ne correspond pas à une zone mémoire figée commune prédéterminée dans ldu Cortex M mais une réservation spécifique faite au début du fichier startup_stm32f10x_md.s

```
1 ❶
2 PUSH {R1}
3 PUSH {R7}
4 ❷
5 …
6 POP {R3}
7 POP{R1}
8 ❸
```





À l'entrée dans la procédure, un « trou » est créé sur la pile en décalant le pointeur de pile vers le bas (❶). Ce trou de 40 octets (0x28) est la zone de stockage pour les 10 entiers du tableau `Tempo`. À la fin de la procédure, ce « trou » sera rebouché en remontant le pointeur de pile **SP** de 40 valeurs(❷). Dans la procédure, lorsqu'il y a nécessité de lire ou d'écrire cette variable locale, cet accès se fait en relatif par rapport au pointeur de pile. Ici (❸) on accède au 4e élément du tableau avec un décalage de 12 octets (le premier élément est l'élément 0 qui se trouve à un décalage nul). Attention toutefois, si des **PUSH**/**POP** sont effectués entre temps, les décalages ne seront plus les mêmes car le pointeur de pile aura évolué. Ce n'est pas forcément facile à suivre mais le compilateur sait normalement très bien le faire.