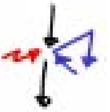
CHAPITRE 1

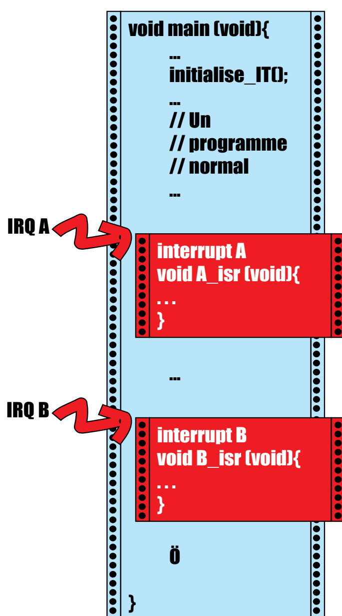
INTERRUPTIONS



1. PRÉSENTATION

En l'absence d'interruptions, les périphériques n'ont jamais la parole. Il ne peuvent rien signaler au *MCU* qui doit les interroger (on parle de scrutation ou polling) pour détecter un état ou un évènement particulier. Grâce aux interruptions, les périphériques ont la parole. Ils peuvent réclamer l'attention du *MCU*.

Le mécanisme des interruptions permet à un processeur de réagir à un ensemble d'événements sans pour autant surveiller par programme leur apparition. Une interruption est un événement capable d'interrompre le déroulement d'un programme pour exécuter immédiatement et automatiquement une routine de service en réponse à cet événement. La routine de service est une routine assembleur ou une fonction C qui diffère d'une routine ou d'une fonction classique par le fait qu'elle n'est jamais appelée par une autre fonction ou routine mais déclenchée par un évènement généralement matériel. L'exécution d'un programme en cours est interrompu par la routine de service. Cette dernière doit donc commencer par sauvegarder l'état du processeur dans la pile et se terminer en restaurant cet état avant de reprendre le déroulement du programme interrompu.



2. ABRÉVIATIONS

2.1 Liées aux interruptions

IT : *Interrupt* : Interruption
ISR : *Interrupt Service Routine* : Routine de service d'interruption.
IRQ : *Interrupt Request* : Requête (demande) d'interruption.
IACK : *Interrupt Acknowledge* : Accusé de réception d'une IRQ.
NMI : *Non-Maskable Interrupt* : Interruption non masquable
IF : *Interrupt Flag* : Bit signalant une requête d'interruption.
IE : *Interrupt Enable* : Bit programmé pour autoriser/inhiber une interruption.

2.2 Autres abréviations utilisées

CPU : *Central Processing Unit* : Processeur, le coeur du microprocesseur ou microcontroleur.
MCU : *Microcontroller*: Microcontrôleur
MPU : *Microprocessing Unit* : Microprocesseur

RTOS : *Real Time Operating System* : Noyau temps réel multitâche.
BIOS : *Basic Input / Output System* : Couche logicielle pour les périphériques d'entrée/sortie.
UART : *Universal Asychronous Receiver Transmitter* : Liaison série asynchrone.
BUS **SPI** : *Serial Peripheral Interface* : Liaison série synchrone, inventé par Motorola
BUS **IIC** ou **I2C** : *Inter Integrated Circuit* : Bus série avec adressage, inventé par Phillips
CAN : *Car Area Network* : Réseau de terrain industriel dédié à l'automobile, inventé par Bosh.
ADC : *Analog to Digital Converter* : Convertisseur Analogique Numérique (CAN[1] en français)
DAC : *Digital to Analog Converter* : Convertisseur Numérique Analogique (CNA en français)
FIFO : *First In Fisrt Out* : File d'attente (le contraire d'une pile).

3. Trois types d'interruption

On peut distinguer les trois classes d'interruption suivantes, présentées dans l'ordre de priorité décroissante : les erreurs, les interruptions matérielles et les interruptions logicielles.

3.1 Erreurs

Au minimum, tous les *MCU* ou MPU doivent répondre à l'interruption RESET qui est bien sûr la plus prioritaire. En ce sens elle peut-être considérée comme une erreur. Son *ISR* est le programme de démarrage.

Au cours de l'exécution d'un programme, différentes erreurs matérielles ou logicielles peuvent survenir. L'apparition de certaines erreurs peut être considérée comme une interruption et déclencher l'exécution d'une routine de service.
Les erreurs sont matérielles telles que l'accès à une adresse invalide ou logicielles telles que la division par zéro ou la tentative d'exécution d'un code machine ne correspondant à aucune instruction du microprocesseur.

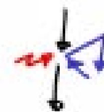
3.2 Interruptions matérielles

L'essentiel des interruptions provient des périphériques.
Les périphériques transmettent au microprocesseur des informations de synchronisation de façon asynchrones. Ces événements peuvent être détectés par scrutation ou déclencher l'exécution d'une routine de service par interruption.
Parmi tous les périphériques des microcontrôleurs, la plupart sont susceptibles de déclencher une interruption. Les périphériques de communication *(RS232, SPI, CAN, I2C,...)* signalent soit l'arrivée d'une donnée, c'est à dire un registre de réception plein qu'une *ISR* doit venir lire, soit la fin de l'émission d'une donnée en sortie, c'est à dire un registre d'émission vide qu'une *ISR* peut à nouveau écrire. Ils peuvent aussi signaler une erreur, par exemple la perte d'une donnée ou une erreur de parité lors d'une transmis-

1. A ne pas confondre avec le réseau CAN cité plus haut.

sion. De même, les convertisseurs analogique-numérique déclenchent des interruptions de fin de conversion pour signaler qu'une donnée convertie peut être lue dans son registre de donnée. Certains ports d'entrée peuvent déclencher des interruptions sur détection d'un niveau ou d'un front. Ces ports peuvent être associés à des circuits périphériques externes au *MCU* ou tout simplement à un capteur binaire ou un bouton poussoir. Enfin, les compteurs peuvent déclencher des interruptions périodiquement lorsqu'un compteur atteint une valeur de référence ou quand il déborde.

3.3 Interruptions logicielles



Lorsqu'un programme déclenche de manière volontaire l'exécution d'une routine de service on parle d'une interruption logicielle. Le déclenchement de ces routines passent par des instructions assembleurs particulières (traps) qui déclenchent l'exécution des routines de service comme s'il s'agissait d'une interruption matérielle.
Cette possibilité est principalement utilisée pour les appels système, c'est à dire au noyau du système d'exploitation. Une routine d'interruption s'exécute généralement dans un mode particulier, appelé superviseur qui autorise l'accès à toutes les ressources du microprocesseur. L'ensemble des routines d'interruptions forme l'interface entre le logiciel et le matériel. Sur un PC, cette couche logicielle s'appelle le *BIOS* (Basic Input/Output System). Pour un système à base de *MCU*, souvent sans OS, les interruptions logicielles sont peu utilisées.

4. Vecteurs d'interruption

Dans chaque microcontrôleur, il existe un contrôleur d'interruption qui permet d'inhiber ou d'autoriser chaque interruption individuellement ou collectivement. Il doit également exister un mécanisme permettant d'associer une *ISR* à chaque source d'interruption. Suivant le type de microcontrôleur, le programmeur à plus ou moins de liberté sur la place que peut occuper une *ISR* en mémoire. L'adresse à laquelle le *CPU* se déroute lors d'une interruption est appelée le *vecteur d'interruption*.

4.1 Vecteurs à adresses figés ou table de vecteur d'interruption

Dans les microprocesseurs simples, le constructeur fixe pour chaque source d'interruption le vecteur d'interruption. Comme il est impossible au constructeur de prévoir la taille de la routine de service, la place laissée pour une interruption est au moins suffisante pour y coder une **instruction de saut vers le corps de l'*ISR*** qui pourra être n'importe où dans la mémoire. Cette technique est utilisée par les microcontrôleurs *PIC* de *Microchip* et la famille *8051*. Comme nous le verrons plus loin, ce n'est pas le cas de Motorola qui a choisi d'utiliser une partie de la mémoire comme table de vecteurs d'interruption où le programmeur viendra écrire ses vecteurs.

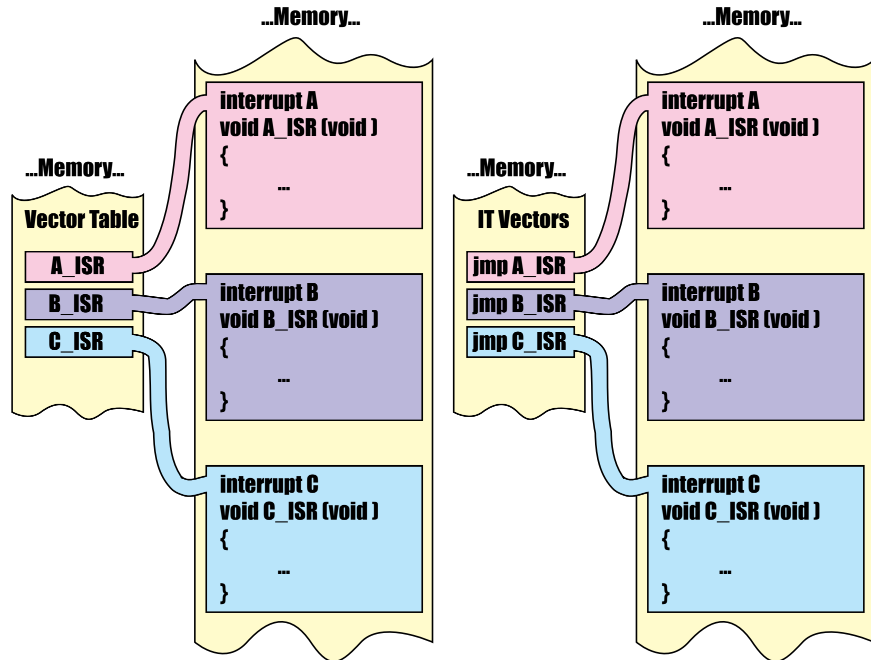


Figure 1 : A gauche, un MCU utilisant une table de vecteur d'interruption contenant les pointeurs sur les ISR, à droite un MCU sans table de vecteurs contenant à la place des instruction de saut (JUMP) vers les ISR.

4.1.1 Le cas du PIC

En dehors du vecteur *RESET* à l'adresse 0x0000, les microcontrôleurs *Microchip* de la famille *PIC 16F8XX* ont 15 sources d'interruptions mais n'ont qu'un seul vecteur d'interruption, qui est l'adresse 0x0004, pour répondre à toutes ces 15 interruptions.

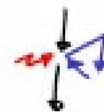
TABLEAU 1. Les 15 sources d'interruption du PIC 16F8xx

N°	Source d'interruption
1	Front sur le port B
2	externe
3	timer0 overflow
4	timer1 overflow
5	timer2 overflow
6	fin de conversion A/N
7	évènement capture / compare 1
8	évènement capture / compare 2
9	Fin d'écriture sur l'EEPROM
10	Port parallèle esclave
11	SPI
12	conflit de bus
13	Rx UART

TABLEAU 1. Les 15 sources d'interruption du PIC 16F8xx

N°	Source d'interruption
14	Tx UART
15	Alimentation faible

Dans ce cas, l'*ISR* doit faire le tri. Si plusieurs interruptions sont autorisées, la routine d'interruption doit déterminer la source qui a déclenché l'interruption pour y répondre. On peut noter que les outils de développement en langage C (*PCW* de *CCS* par exemple) peuvent masquer le manque de vecteurs en générant automatiquement le code nécessaire à la détection de la source d'interruptions (cf. listing 1 et 2). Cela peut faire croire aux développeurs qu'ils existent plusieurs vecteurs, ce qui n'est pas le cas. Le compilateur génère également les instructions de sauvegarde et restauration du contexte, ce qui n'est pas trivial sur ce microcontrôleur sans pile et dont la mémoire est paginée. Le revers de la médaille est qu'une telle interruption exécute au minimum 60 instructions, ce qui peut durer 60 µs sur un *PIC* cadencé à 4 MHz. C'est au moins 30 fois plus long que la plupart des autres *MCU*.



Le compilateur permet de définir plusieurs *ISR* ce qui donne l'illusion qu'ils existent plusieurs vecteurs. Ce n'est pas le cas.

Listing 1: Programme d'interruption avec les outils *PCW* de *CCS* pour les *PIC 16F877* de *Microchip*

```
1       #int_timer1
2       timer1_isr() {
3               ...
4       }
5       #int_lowvolt
6       lowvolt_isr() {
7               ...
8       }
9       void main() {
10          . . .
11          . . .
12          enable_interrupts(INT_TIMER1);
13          enable_interrupts(INT_LOWVOLT);
14          enable_interrupts(global);
15          while(1);
16      }
```

Voici le code assembleur généré :

.

La simplicité du PIC qui n'a qu'un vecteur coûte cher. Environ 60 instructions sont exécutées dans une *ISR* minimale et l'*ISR* en assembleur est un vrai dédale.

Listing 2: Extrait du code assembleur généré

```
1       0000:  MOVLW   00    <- VECTEUR RESET ...
2       0001:  MOVWF   0A
3       0002:  GOTO    056 ... aller au MAIN )
4       0003:  NOP
5       0004:  MOVWF   7F      <- VECTEUR D'INTERRUPTION ...
6       0005:  SWAPF   03,W    ...  Sauvegarder le contexte ...
                  ...          [... 13 instructions ...]
```

Listing 2: Extrait du code assembleur généré

```
7       001C:  BTFSC  0B.2  ... Si c'est une IT RTCC ...
8       001D:  GOTO   037   ... aller vers péparation  RTCC_isr ...
                  ...              [... 4 instructions ...]
9       0022:  BTFSC  0D.7 ... Si c'est une IT LOWVOLT ...
10      0023:  GOTO   03A  ... Aller vers préparation LOWVOLT_isr
...
11      0024:  MOVF   22,W  <- restauration du contexte
                 ...             ...  17 instructions ...
12      0036:  RETFIE         ... RETOUR D'INTERRUPTION
13      0037:  BCF    0A.3
14      0038:  BCF    0A.4
15      0039:  GOTO   03D      ... aller en RTCC_isr
16      003A:  BCF    0A.3
17      003B:  BCF    0A.4
18      003C:  GOTO   052      ... aller en LOWVOLT_isr
19
20      .................... #int_RTCC
21      .................... RTCC_isr() {
22      003D:  MOVLW  00
                [ ... ]           ...  ...
23      0051:  GOTO   024       ... Aller à restauration du contexte
24      .................... #int_LOWVOLT
25      .................... LOWVOLT_isr() {
26      ....................    /* ... */
27      0052:  BCF    0D.7
                [ ... ]           ...  ...
28      0055:  GOTO   024   ... Aller à restauration du contexte
29      .................... }
30
31      .................... void main() {
32      ...
```

4.1.2 Le cas du 8051 d'Intel

Dans le *8051* il y a 5 (6 pour le *8052*) sources d'interruption et autant de vecteurs situés sur les adresses basses. Les outils de développement en C tel que le compilateur *KEIL* génèrent automatiquement l'instruction de saut vers la routine de service *LJMP* et la sauvegarde/restauration du contexte.

Le *8051* possède plusieurs bancs de registres et peut basculer d'un banc à l'autre pour les interruptions. Cela se traduit par le mot clé using pour ce compilateur.

Listing 3: Programmation des interruption avec les outils KEIL pour les 8051

```
1       #include <reg51.h>   /* define 8051 registers */
2       #include <stdio.h>   /* define I/O functions */
3       sbit P3_0=P3^0;
4       void isr_timer(void) interrupt 1 using 2{
5         P3_0=~P3_0;     /* Toggle PORT 3 bit 0 */
6       }
7
8       void main (void)  {
9        while (1)  ;
10      }
```

Le programme du listing 3 produit le code assembleur du listing 4.

On retrouve le saut à l'adresse de l'*ISR* (ligne 2). Le corps de l'*ISR* elle même (ligne 4 à 7) est limpide comparée à celle du PIC.

Listing 4: Programme assembleur généré.

```
1       CSEG    AT      0000BH
2               LJMP    isr_timer
3       ...
4               USING   2
5       isr_timer:
6               CPL     P3_0
7               RETI
```

4.2 Table de vecteurs d'interruption

Dans les microcontrôleur plus évolués, le constructeur prévoit une table dans laquelle le programmeur doit placer les adresses (les vecteurs) de ses routines de service. En fonction de la source d'interruption, le microcontrôleur vient automatiquement lire dans cette table pour se brancher sur la bonne *ISR*. La ***table des vecteurs d'interruption*** correspond en C à une **table de pointeurs sur fonction**. Les éléments de cette table sont les vecteurs (pointeurs sur fonction) et l'indice dans la table s'appelle un ***numéro de vecteur***.

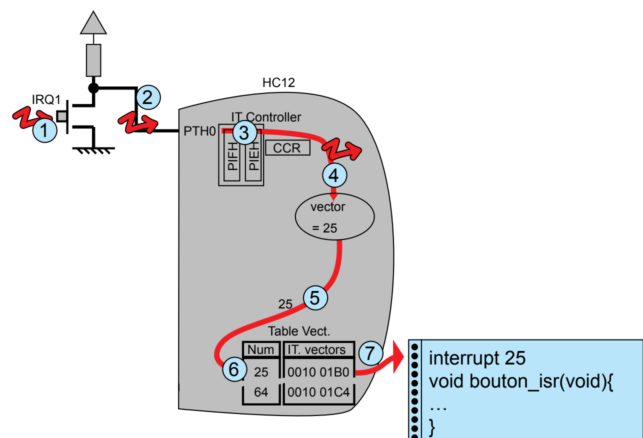


Figure 2 : : Principe d'une interruptions vectorisées sur HC12.

Lorsque le bouton est enfoncé (1), l'entrée PTH0 est amenée à au niveau L (2) qui est son niveau actif. Ceci signale une requête d'interruption. Elle est enregistrée par le contrôleur d'interruption (3). Elle devient une interruption en attente (pending interrupt), ce qui se traduit par un indicateur IF (interrupt Flag) qui passe à l'état 1, ici dans le registre PIFH. Si cette interruption est individuellement autorisée par un bit IE (Interrupt Enable) dans le registre PIEH et que les interruptions ne sont pas globalement inhibées par le bit I du registre d'état, la requête devient une interruption (4). En fonction de la source de l'interruption (ici la broche PTH0 par exemple) le MCU calcul un numéro de vecteur (5), ici 25. Ce numéro sert d'index dans la table des vecteurs (6). Le MCU en extrait le vecteur d'interruption (7) qui est l'adresse de l'ISR qu'il commence à exécuter.

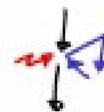
La table des vecteurs d'interruption existe dans tous les microcontrôleurs Motorola dont le *68HC12*. La table des vecteurs d'interruption du *9S12DP256* utilise l'espace mémoire situé entre les adresses FF80 à FFFF. Il s'agit d'une zone de mémoire **non paginée**. De même, les vecteurs étant codés sur 16 bit, **les routines d'interruptions doivent être située dans une zone mémoire non paginée**.

TABLEAU 2. Table des vecteurs d'interruption du HC12

n°	CODE	adresse	source	autorisée localement par
0	RESET_ISR	FFFE-FFFF	RESET	-
1	CLKMONFAIL_ISR	FFFC-FFFD	CLOCK MONITOR FAIL RESET	PLLCTL (CME, SCME)
2	COP_ISR	FFFA-FFFB	COP RESET (WATCHDOG TIMER)	COP RATE SELECT
3	UNIMPLTRAP_ISR	FFF8-FFF9	UNIMPLEMENTED INSTRUCTION TRAP	-
4	SWI_ISR	FFF6-FFF7	SWI	-
5	XIRQ_ISR	FFF4-FFF5	XIRQ	-
6	IRQ_ISR	FFF2-FFF3	IRQ	-
7	RTI_ISR	FFF0-FFF1	REAL TIME INTERRUPT (RTI)	IRQCR (IRQEN)
8	ECT0_ISR	FFEE-FFEF	TIMER 0	TIE (C0I)
9	ECT1_ISR	FFEC-FFED	TIMER 1	TIE (C1I)
10	ECT2_ISR	FFEA-FFEB	TIMER 2	TIE (C2I)
11	ECT3_ISR	FFE8-FFE9	TIMER 3	TIE (C3I)
12	ECT4_ISR	FFE6-FFE7	TIMER 4	TIE (C4I)
13	ECT5_ISR	FFE4-FFE5	TIMER 5	TIE (C5I)
14	ECT6_ISR	FFE2-FFE3	TIMER 6	TIE (C6I)
15	ECT7_ISR	FFE0-FFE1	TIMER 7	TIE (C7I)
16	TOF_ISR	FFDE-FFDF	TIMER OVERFLOW	TSCR2 (TOF)
17	PAOV_ISR	FFDC-FFDD	PULSE ACCUMULATOR A OVERFLOW	PACTL (PAOVI)
18	PA_ISR	FFDA-FFDB	PULSE ACCUMULATOR A INPUT EDGE	PACTL (PAI)
19	SPI0_ISR	FFD8-FFD9	SPI0	SP0CR1 (SPIE,SPTIE)
20	SCI0_ISR	FFD6-FFD7	SCI0	SC0CR2 (TIE,TCIE,RIE,ILIE)
21	SCI1_ISR	FFD4-FFD5	SCI1	SC1CR2 (TIE,TCIE,RIE,ILIE)
22	ATD0_ISR	FFD2-FFD3	ATD0	ATD0CTL2 (ASCIE)
23	ATD1_ISR	FFD0-FFD1	ATD1	ATD1CTL2 (ASCIE)
24	PORTJ_ISR	FFCE-FFCF	PORT J	PTJIF (PTJIE)
25	PORTH_ISR	FFCC-FFCD	PORT H	PTHIF (PTHIE)
26	MCZ_ISR	FFCA-FFCB	MODULUS DOWN COUNTER UNDERFLOW	MCCTL (MCZI)
27	PBOV_ISR	FFC8-FFC9	PULSE ACCUMULATOR B OVERFOW	PBCTL (PBOVI)
28	PLLLOCK_ISR	FFC6-FFC7	CRG PLL LOCK	CRGINT (LOCKIE)
29	PLLSCM_ISR	FFC4-FFC5	CRG SELF CLOCK MODE	CRGINT (SCMIE)
30	BDLC_ISR	FFC2-FFC3	BDLC	BLCBCR1 (IE)
31	I2C_ISR	FFC0-FFC1	IIC BUS	IBCR (IBIE)
32	SPI1_ISR	FFBE-FFBF	SPI1	SP1CR1 (SPIE,SPTIE)
33	SPI2_ISR	FFBC-FFBD	SPI2	SP2CR1 (SPIE,SPTIE)

TABLEAU 2. Table des vecteurs d'interruption du HC12

n°	CODE	adresse	source	autorisée localement par
34	EEPROM_ISR	FFBA-FFBB	EEPROM	EECTL (CCIE,CBEIE)
35	FLASH_ISR	FFB8-FFB9	FLASH	FCTL (CCIE,CBEIE)
36	CAN0WU_ISR	FFB6-FFB7	CAN 0 WAKE UP	CAN0RIER (WUPIE)
37	CAN0ERR_ISR	FFB4-FFB5	CAN 0 ERROR	CAN0RIER (CSIE,OVRIE)
38	CAN0RX_ISR	FFB2-FFB3	CAN 0 RECEIVE	CAN0RIER (RXFIE)
39	CAN0TX_ISR	FFB0-FFB1	CAN 0 TRANSMIT	CAN0TIER (TXEIE2-TXEIE0)
40	CAN1WU_ISR	FFAE-FFAF	CAN 1 WAKE UP	CAN1RIER (WUPIE)
41	CAN1ERR_ISR	FFAC-FFAD	CAN 1 ERROR	CAN1RIER (CSIE,OVRIE)
42	CAN1RX_ISR	FFAA-FFAB	CAN 1 RECEIVE	CAN1RIER (RXFIE)
43	CAN1TX_ISR	FFA8-FFA9	CAN 1 TRANSMIT	CAN1TIER (TXEIE2-TXEIE0)
44	CAN2WU_ISR	FFA6-FFA7	CAN 2 WAKE UP	CAN2RIER (WUPIE)
45	CAN2ERR_ISR	FFA4-FFA5	CAN 2 ERROR	CAN2RIER (CSIE,OVRIE)
46	CAN2RX_ISR	FFA2-FFA3	CAN 2 RECEIVE	CAN2RIER (RXFIE)
47	CAN2TX_ISR	FFA0-FFA1	CAN 2 TRANSMIT	CAN2TIER (TXEIE2-TXEIE0)
48	CAN3WU_ISR	FF9E-FF9F	CAN 3 WAKE UP	CAN3RIER (WUPIE)
49	CAN3ERR_ISR	FF9C-FF9D	CAN 3 ERROR	CAN3RIER (CSIE,OVRIE)
50	CAN3RX_ISR	FF9A-FF9B	CAN 3 RECEIVE	CAN3RIER (RXFIE)
51	CAN3TX_ISR	FF98-FF99	CAN 3 TRANSMIT	CAN3TIER (TXEIE2-TXEIE0)
52	CAN4WU_ISR	FF96-FF97	CAN 4 WAKE UP	CAN4RIER (WUPIE)
53	CAN4ERR_ISR	FF94-FF95	CAN 4 ERROR	CAN4RIER (CSIE,OVRIE)
54	CAN4RX_ISR	FF92-FF93	CAN 4 RECEIVE	CAN4RIER (RXFIE)
55	CAN4TX_ISR	FF90-FF91	CAN 4 TRANSMIT	CAN4TIER (TXEIE2-TXEIE0)
56	PORTP_ISR	FF8E-FF8F	PORT P	PTPIF (PTPIE)
57	PWMEMSH_ISR	FF8C-FF8D	PWM EMERGENCY SHUT-DOWN	PWMSDN (PWMIE)
58-63		FF80-FF8B	réservé	-



4.3 Interruptions externes vectorisées

Finalement, dans les microprocesseurs où les périphériques sont essentiellement des composants externes, le programmeur a même le choix du numéro de vecteur associé à chaque circuit périphérique. Ce numéro se programme dans un registre du circuit périphérique et le microprocesseur interrompu doit venir lire ce numéro par un cycle bus particulier appelé cycle d'acquittement d'interruption (*IACK*).

Si leur programmation n'est pas immédiate les interruptions externes vectorisées du 68k/ ColdFire présentent l'avantage de la souplesse. Tout est programmable y compris les numéros de vecteurs associés aux interruptions.

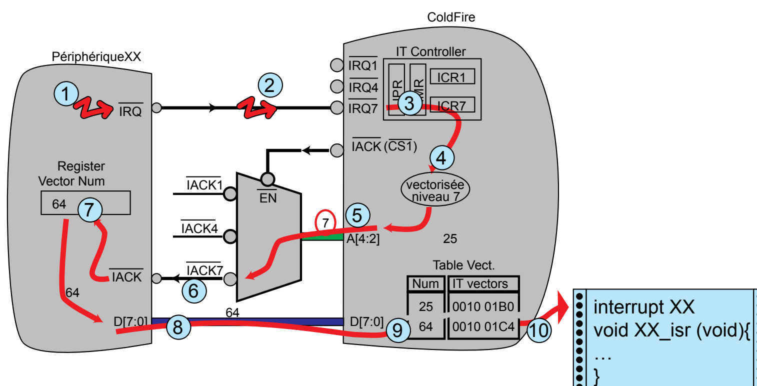


Figure 3 : Cycle IACK pour obtenir un numéro de vecteur sur un 68k/ColdFire

Au commencement, un périphérique "XX " détecte un évènement qu'il souhaite signaler au MCU par une interruption (1). Il émet donc une requête sur sa sortie $\overline{\text{IRQ}}$ qui est reçue sur l'entrée, ici $\overline{\text{IRQ7}}$, du MCU (2). Le requête, est enregistrée par le contrôleur d'interruption (3) dans un bit du registre IPR (Interrupt Pending Register). Si cette interruption est individuellement autorisée dans le registre IMR (Interrupt Mask Register) et que la priorité de l'interruptions est supérieure à la priorité du programme en cours, la requête devient une interruption (4). En fonction de la source de l'interruption (ici la broche $\overline{\text{IRQ7}}$ par exemple) le MCU effectue un cycle de lecture particulier appelé cycle d'acquittement d'interruption (IACK pour Interrupt Acknowledge) avec le niveau de priorité comme adresse, ici 7 (5). Lors de ce cycle, un signal particulier ($\overline{\text{IACK}}$) indique qu'il ne s'agit pas d'un cycle de lecture mémoire traditionnel. Un décodeur externe active l'un des signaux $\overline{\text{IACKN}}$, ici $\overline{\text{IACK7}}$ (6). Le périphérique concerné reçoit ainsi l'ordre de transmettre son numéro de vecteur. Ce numéro a été préalablement programmé dans un registre (7). Il est émis sur le bus de données (8). Ce numéro sert d'index dans la table de vecteurs (9). Le MCU en extrait le vecteur d'interruption (10) qui est l'adresse de l'ISR qu'il commence à exécuter.

5. La pile lors d'une interruption

Lors d'une interruption sur le *HC12*, l'état du processeur est sauvegardé dans la pile. D'abord, l'adresse de retour, c'est à dire l'adresse de l'instruction qui aurait été exécutée si l'interruption n'avait pas eut lieu, est empilée. Ensuite, les registres *Y, X D* sont empilée. Enfin le registre d'état *CCR* est empilé. L'état de la pile après une interruption est ainsi décrit par la figure 4. Lorsque l'*ISR* exécute l'instruction *RTI* (retour d'interruption), l'interruption se termine et le *CPU* restaure l'état du processeur qu'il avait sauvegardé dans la pile.

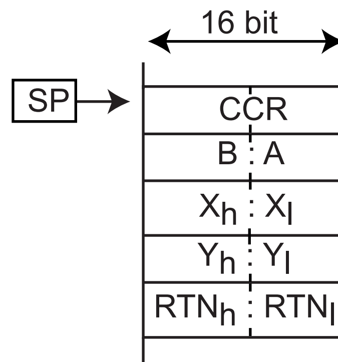


Figure 4 : La pile du 68HC12 lors d'une interruption.

6. Différentes stratégies de programmation

6.1 Scrutation, interruption ou RTOS

La plupart des événements susceptibles de déclencher une interruption peuvent être détectés par scrutation d'un registre d'état. C'est d'ailleurs la solution la plus simple à programmer sous réserve qu'un seul événement soit attendu. Dans ce cas, le processeur est entièrement utilisé à scruter les registres d'état et ne peut pas être utilisé à autre chose. Il devient intéressant d'utiliser les interruptions lorsque plusieurs événements peuvent survenir et qu'il est gênant de bloquer le processeur dans une boucle d'attente. L'utilisation des interruptions permet ainsi de mettre un programme en attente d'un événement sans pour autant bloquer le programme dans une boucle d'attente. Le fonctionnement du système est alors basé sur un programme d'arrière plan, qui communique par l'intermédiaire de tampons d'entrée et de sortie (*FIFO*). Les *ISR* vident ou remplissent ces tampons (cf. figure. 5).

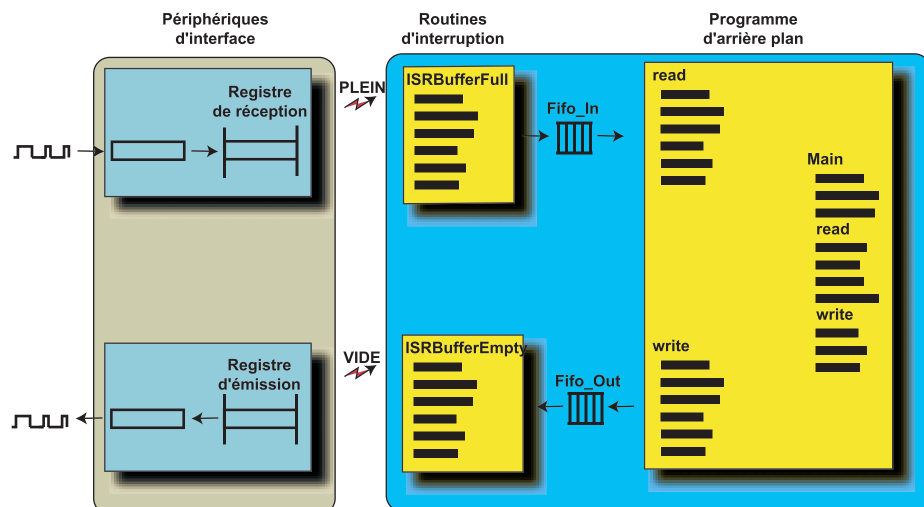


Figure 5 : Principe d'une application gérant les entrées-sor-

ties par interruption avec des tampons d'entrée et de sortie.

Il existe une troisième stratégie qui consiste à utiliser un noyau temps-réel multitâche (*RTOS*). Dans ce cas, le processeur est partagé par plusieurs taches concurrentes. Les appels au système *bloquants*, correspondant à l'attente d'un évènement, provoquent la sauvegarde de la tâche en cours d'exécution et la reprise d'exécution d'une tâche prête, c'est-à-dire qui n'attend plus d'évènement. Les détails concernant l'utilisation et le fonctionnement d'un *RTOS* dépasse le cadre de ce chapitre. Cependant un *RTOS* ne peut pas se passer d'interruptions. Les noyaux temps réel sont conçus pour répondre le plus rapidement possible aux interruptions tout en bloquant le moins possible le *CPU* dans les *ISR*.

6.2 Interruptions imbriquées

Tous les microcontrôleurs et microprocesseurs mettent en jeu des **priorités** sur les interruptions afin de définir quelle interruption sera prise en compte lorsque plusieurs surviennent simultanément. Les priorités sont plus ou moins programmables suivant les *MCU*. Sur le *HC12*, les priorités sont figées par le constructeur. Les sources d'interruption les plus prioritaires sont celles correspondant aux plus grandes adresses dans la table des vecteurs. Il est seulement possible d'élever la priorité de l'une des interruptions en utilisant le registre *HPRIO*. Vous trouverez pour cela les détails dans la documentation *Motorola*.

De plus, suivant les microprocesseurs, les routines de service peuvent être, elles-mêmes, interrompues ou ne pas être *interruptibles*. Dans le premier cas, les interruptions de fortes priorités peuvent interrompre des *ISR* de priorité plus faible. L'avantage d'une routine interruptible est de **minimiser la latence** (le temps de réaction) d'une *ISR* prioritaire puisqu'elle n'a pas besoin d'attendre la fin de l'exécution des interruptions de plus faibles priorités. A contrario, cette solution est plus coûteuse en *RAM* puisque les interruptions imbriquées sont susceptibles d'**utiliser simultanément la pile**. Comme la *RAM* est une ressource rare dans les petits microcontrôleurs, les interruptions imbriquées y sont, en générale, évitées.

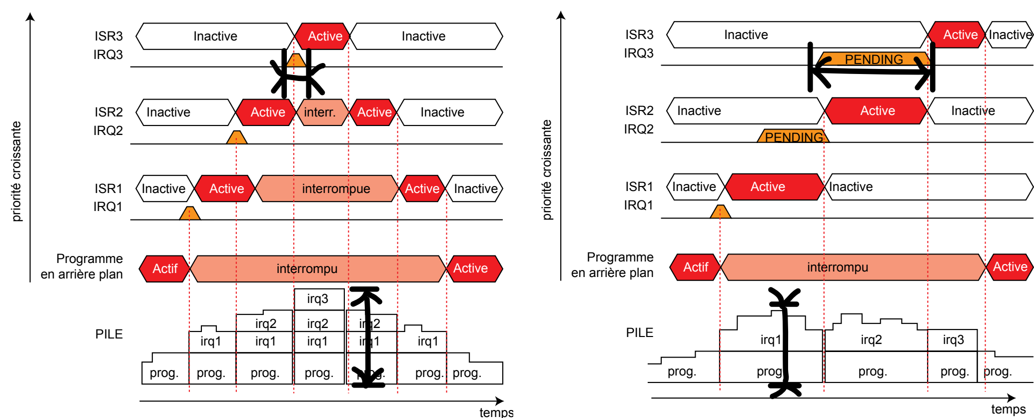
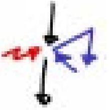


Figure 6 : Interruption imbriquées (à gauche) versus non-imbriquées (à droite).

Lorsque les interruptions sont elles-mêmes interruptibles, il

faut dimensionner l'espace de la pile pour qu'elle puisse contenir les contextes de toutes les interruptions imbriquées. Avec un RTOS multitâche préemptif, il y a autant de pile que de tâches et cet espace doit alors être prévu dans chaque pile de chaque tâche.

A l'inverse, lorsque les interruptions ne sont pas interruptibles, les requêtes d'interruption, même prioritaires, peuvent rester en attente («pending») assez longtemps, le temps qu'une interruption moins prioritaire se termine. En revanche, les ISR n'occupent las pile que l'une après l'autre.



7. Exemple de programmes d'interruption sur HC12

7.1 Avec CodeWarrior

Le programme suivant est un exemple complet de programmation des interruptions avec *CodeWarrior* pour *HC12*.
Dans la fonction main (ligne 12) le bit *IE (interrupt enable)* correspondant à l'autorisation des interruption associées au port H *(registre PIEH)* est activé pour les 4 bits de poids faibles. Il s'agit d'une autorisation individuelle d'émettre des requêtes d'interruption. L'autorisation globales des interruptions se fait (à la ligne 18) par la macro *EnableInterrupts* qui correspond à l'instruction assembleur *CLI*. Cette instruction met à zéro le bit *I* du registre d'état *CCR*. Le *MCU* devient alors sensible aux interruptions. En l'absence d'interruption, ce programme attend sans rien faire dans la boucle infinie de la ligne 20. Lorsque l'un des 4 boutons poussoir de la carte *Star12* est appuyé, l'interruption est déclenchée. La fonction *KWH_ISR* (ligne 23 à 28) s'exécute, ce qui a pour effet d'inverser l'état de l'une des *LED* associé au port M. Notez qu'à la ligne 26 **on écrit des UN** dans les indicateurs d'interruption (*IF*) du registre *PIFH* **ce qui a pour effet de les remettre à ZERO**. Cela correspond à un ***acquittement de l'interruption***. S'il n'est pas fait, l'*ISR* se relance indéfiniment dès qu'elle se termine ce qui bloque le *CPU* et plante le programme.

Comme la table des vecteurs se trouve dans la mémoire flash du *MCU* 9S12DP256, il n'est pas possible de charger le code uniquement dans la RAM, il faut programmer la mémoire flash. A défaut, la table des vecteurs n'est pas modifiée, le vecteur n°25 pointe donc n'importe où (mais sûrement pas sur son *ISR*) et le programme plante dès le déclenchement de la première interruption.

Listing 5: Programme complet d'interruption sur CW / HC12

```
1     /////////////////////////////
2     // DEMO INTERRUPT HC12     //
3     // TARGET = 9S12DP526      //
4     // Compiler = CodeWarrior  //
5     /////////////////////////////
6
7     #include <hidef.h>        //  for "EnableInterrupts" macro
8     #include "interrupt.h"     //  for "PORTH_ISR" macro
9     #include "6812dp256.h"     // for 9S12dp256's registers
10
11    void main(void){
12         PIEH = 0x0F;        // Enable interrupt on port H3-0
13         PERH = 0xFF;        // Enable Pull-Up on Port H
```

Listing 5: Programme complet d'interruption sur CW / HC12

```
14
15          DDRA = 0xFF;          // Port A OUT (Bargraph)
16          DDRM |= 0xC0;         // Port M 7-6 OUT (Bargraph)
17
18          EnableInterrupts      // Idem  {__asm CLI;}
19
20                 while(TRUE);         // Infinite loop.
21      }
22
23      interrupt PORTH_ISR       // Is vector num 25
24      void KWH_isr(void){
25          PTM ^= 0x80;          // Toggle Port M bit 7
26          PIFH = 0xFF;          // Write 1s in port H IT falgs
27                                // to clear them.
28      }
29      ///////////////////END OF DEMO////////////////
```

Le mot clé ***interrupt*** (ligne 23), qui **n'est pas standard**, indique au compilateur que cette fonction est associée au vecteur d'interruption n°25, c'est-à-dire ici les interruptions du port H. Le compilateur va générer une *ISR* assembleur, qui se termine par l'instruction *RTI*, et place lui-même le vecteur dans la table des vecteurs d'interruption.

7.2 Avec un autre compilateur

Avec d'autres outils, la déclaration d'une fonction d'interruption peut-être différente et l'écriture dans la table des vecteurs doit, le plus souvent, se faire explicitement. Voici par exemple comment peut une *ISR* avec des outils de développement différents. La ligne 7 correspond à la définition du pointeur sur la fonction *TOFhandler* dans la table des vecteurs *TOF_vector[]*. L'adresse du vecteur est défini par la directive *#pragma abs_address* à la ligne 6.

Listing 6: Routine de service en C avec un autre compilateur

```
1       #pragma interrupt_handler TOFhandler
2       void TOFhandler(void){
3         TFLG2 = 0x80;             // TOF interrupt acknowledge
4         PORTT ^= 0x40;            // toggle bit 6
5       }
6       #pragma abs_address:0xffde
7       void (*TOF_vector[])() = { TOFhandler };
8       #pragma end_abs_address
```

8. Trouvez vos BUGS

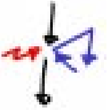
8.1 Si le programme plante avant de déclencher une interruption :

- Vérifiez avec le debugger dans la mémoire que les vecteurs associés aux interruptions autorisées pointent effectivement sur vos *ISR*.

- Vérifier que votre projet est bien conçu pour télécharger le programme en *FLASH*. Si seule la *RAM* est programmée, les vecteurs d'interruption ne seront pas modifiées et pointeront n'importe où.

8.2 Si une *ISR* se déclenche une seule fois puis reste bloquée

- Les fonctions d'interruption ne doivent jamais être appelées à partir d'une autre fonction comme des fonctions usuelles. Autrement l'instruction de fin d'interruption (*RTI*) qui ne trouve pas dans la pile ce qu'elle attend fait planter le système.
- Avant la fin d'une routine d'interruption, la cause de l'interruption doit être supprimée (généralement en écrivant un UN dans un registre *IF*) sinon l'interruption se redéclenche indéfiniment et le système est bloqué.



8.3 Si l'interruption ne se déclenche pas du tout

- Vérifiez avec le debugger dans les registres du *MCU* que la requête d'interruption se fait bien. Le bit *IF* correspondant doit être à 1.
- Vérifiez avec le debugger que l'interruption est pas autorisée. Le bit *IE* correspondant doit être à 1.
- Vérifiez avec le debugger que les interruptions ne sont pas globalement inhibées. Le bit I du registre d'état doit être à 0.
- Vérifiez avec le debbuger dans la mémoire que le vecteur associé à cette interruption pointe effectivement sur votre *ISR*.
- Vérifiez avec le debbuger que l'interruption ne se déclenche pas en y plaçant un point d'arrêt.

9. Pour aller plus loin

Certains points liés à la programmation avec des interruptions sont assez délicats. Ils ne seront qu'évoqués ici.

9.1 Interruptions et réentrance.

Lorsqu'une *ISR* appelle une fonction qui est appelée également par d'autre *ISR* ou par le programme d'arrière plan, la fonction doit présenter certaines caractéristiques pour fonctionner normalement. Le fait qu'ils puissent exister différentes occurrences simultanées d'une fonction implique qu'elle soit *réentrante*. En particulier, elle **ne doit pas utiliser de données statiques comme variables temporaires**.

9.2 Section critiques

Parfois il est ennuyeux qu'une interruption intervienne au cours d'une opération manipulant une donnée. Cela peut conduire à un état des variables incohérent. Il est alors nécessaire de définir des sections de code qui doivent s'exécuter de façon atomique, c'est à dire sans pouvoir être interrompu au milieu, comme s'il s'agissait d'une seule instruction assembleur. On

appelle section critique de telles section de code où les interruptions sont inhibées. Bien sûr, les sections critiques nuisent à la réactivité du système qui risque de mettre plus longtemps pour répondre aux interruptions. Elles doivent donc être aussi courtes que possible.

9.3 Quand le Retour d'interruption se fait *rouler*

Dans un *RTOS* multitâche préemptif, chaque tâche dispose de sa propre pile. Le lancement d'une telle tâche est un joli tour de passe-passe. En effet, pour démarrer une tâche, la fonction de création de tâche créé *artificiellement* dans la mémoire une pile qui simule celle qu'aurait eut la tâche si elle avait été interrompu. Puis le pointeur de pile est modifié pour pointer sur cette nouvelle pile. La tâche est démarrée par l'instruction de retour d'interruption *RTI*. En effet, cette instruction va chercher dans la pile les registres qu'elle pense avoir sauvegardés. Trompée par cette nouvelle pile, elle initialise en fait ses registres avec un contexte créé de toute pièce pour lui faire exécuter la nouvelle tâche.
De même, pour basculer d'une tache à l'autre, une interruption logicielle lance une routine de changement de contexte. Cette routine modifie le pointeur de pile pour qu'il pointe sur la pile de la tache à relancer. Ainsi, le retour d'interruption ne retournera pas à la tache interrompue mais à une autre tâche.
Il est clair que la programmation d'une routine de création de tâche ou de changement de contexte nécessite une parfaite connaissance de l'état de la pile au moment des interruptions. C'est probablement la partie la plus délicate dans un *RTOS*.