Module Master M1

Systèmes temps réel et Informatique Industrielle
Chapitre VI : Interruptions, signaux et événements

Projet n°03 d'exposé des étudiants de Master M1 Informatique

Présenté par : Prof. Kholladi Mohamed-Khireddine
Département d'Informatique
Facultés des Sciences Exactes
Université Echahid Hamma Lakhdar d'El Oued
Tél. 0770314924
Email. kholladi@univ-eloued.dz et kholladi@yahoo.fr
Site Web. www.univ-eloued.dz
http://kholladi.doomby.com/ et http://kholladi.e-monsite.com/

VI – Interruptions, signaux et événements

VI.1 - Exceptions et interruptions

a - Exceptions

Dans l'exception, n'importe quel événement qui interrompt l'exécution normale du processeur et l'oblige à exécuter un ensemble d'instructions spéciales dans un état privilégié. Les exceptions peuvent être synchrones ou asynchrones. Les exceptions générées par des événements internes sont des exceptions synchrones, par exemple la division par zéro, non alignement des données. Les exceptions générées par des événements externes sont des exceptions asynchrones, par exemple la pression sur le bouton de reset, arrivée de données sur le module de communication du processeur.

b - Interruptions

Une Interruption est une classe particulière d'exceptions externes (asynchrones) provoquées par un événement déclenché par du matériel extérieur au processeur. Les interruptions se distinguent des exceptions synchrones par la source (le processeur lui-même pour les exceptions synchrones, un matériel externe pour les interruptions). Les exceptions et les interruptions sont utilisées pour communiquer entre le processeur et le matériel extérieur pour les erreurs internes, la gestion de conditions spéciales, l'utilisation concurrente du matériel et la gestion des demandes de service.

VI.2 - Contrôleur programmable d'interruptions

La plupart des applications ont de multiples sources d'interruptions. Le contrôleur programmable d'interruptions (PIC) a deux fonctions essentielles :

- Établir des priorités entre les différentes sources,
- Décharger la CPU de la détermination de la source d'une interruption, quand celle-ci se présente.

Il utilise une ou des lignes d'interruptions (IRQ : Interrupt ReQuest), chacune étant associée avec un niveau de priorité. Il connaît l'adresse des instructions à charger (ISR : Interrupt Service Routine) quand l'interruption est détectée (vecteur d'interruption) comme sur la figure VI.1 et le tableau suivant.

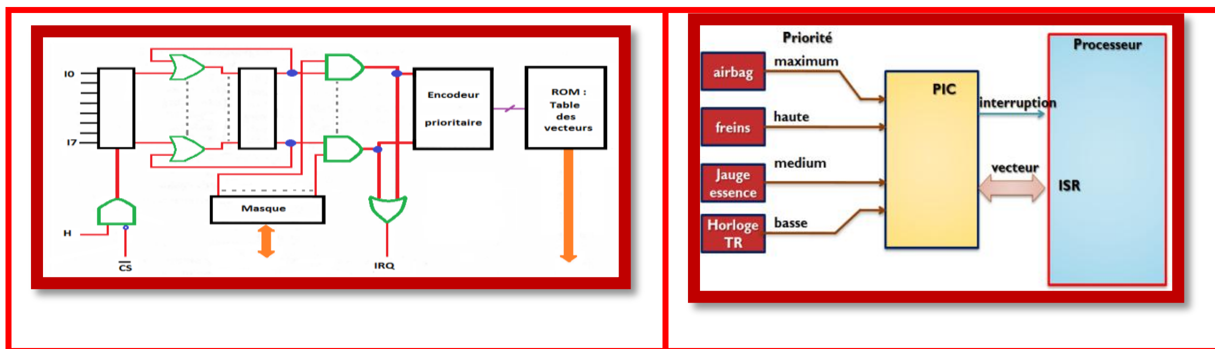

Figure VI.1 – Les schémas d'un contrôleur programmable d'interruptions

Source	Priorité	Adresse IRQ	Vecteur	Fréq. max	Description
Capteur airbag	Maxi	14h	8	N/A	Déclenche l'airbag
Capteur freins	Haute	18h	7	N/A	Déclenche le freinage
Capteur jauge	médium	1Bh	6	20Hz	Mesure le niveau
Horloge TR	Basse	1Dh	5	100Hz	Horloge précise à 10ms.

VI.3 - Quelques considérations sur le temps

La figure VI.2 illustre les quelques considérations sur le temps. On sait que le temps de latence dépend du processeur, des interruptions de plus haute priorité en cours et du masquage. Le temps de traitement ne dépend que du programmeur.

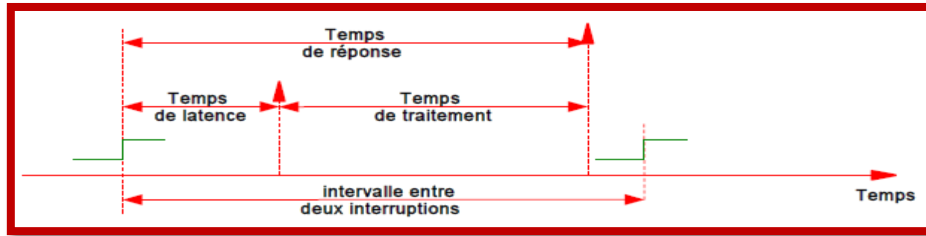

Figure VI.2 – Quelques considérations sur le temps

La figure VI.3 illustre un peu plus en détail pris en considération sur le temps.

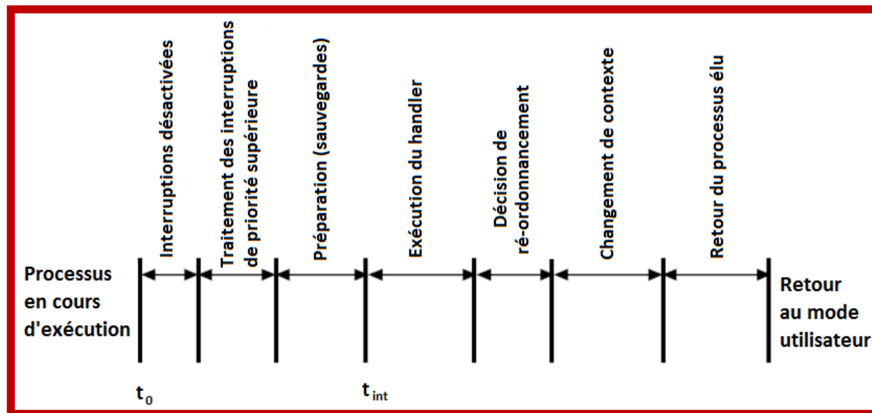

Figure VI.3 – Un peu plus en détail

Le traitement souvent séparé en un traitement minimal dans l'**ISR** et un traitement reporté effectué par une autre tâche (éventuellement un démon) comme sur la figure VI.4.

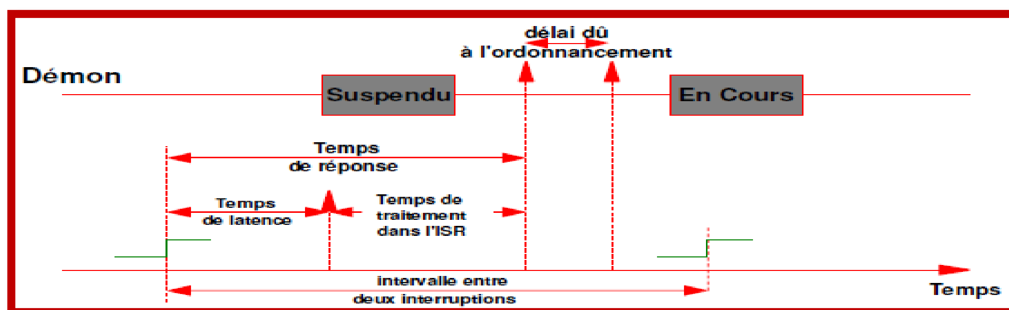

Figure VI.4 – Le traitement séparé d'une ISR

La gestion optimale des interruptions de basse priorité diminue les risques de manquer des interruptions au détriment du temps de réponse.

VI.4 - Gestion des interruptions

On ne peut pas normalement gérer les interruptions matérielles à partir de l'espace utilisateur et cela nécessite de passer par un driver (dans l'espace noyau).

VI.5 - Interruptions et Xenomai

La peau POSIX de Xenomai propose plusieurs fonctions (extensions non-portables du standard POSIX.4) pour manipuler les interruptions. Un objet interruption associe un handler (ISR) à un numéro d'IRQ. Suivant que l'on se trouve dans l'espace noyau (module) ou utilisateur (thread Xenomai), la manipulation d'un objet interruption est différente, on ne considérera que l'espace utilisateur. Les développeurs de Xenomai conseillent d'utiliser la peau RTDM (Real Time Driver Model) pour gérer les interruptions.

L'installation d'un handler pour une interruption par :

```
int pthread_intr_attach_np(pthread_intr_t *intrp,
        unsigned irq,
        int mode);
```

- Crée un objet "interruption"
- Un thread qui sera bloqué en attente de l'interruption **irq** ***sera*** automatiquement débloqué
- Mode : OU de PTHREAD_IPROPAGATE et PTHREAD_INOAUTOENA

L'attente de l'interruption est faite par :

```
int pthread_intr_wait_np(pthread_intr_t intr,
        const struct timespec *to);
```

- Bloque le thread jusqu'à ce que l'interruption irq soit activée,
- Retourne de suite si l'interruption a déjà été activée,
- Un timeout peut être spécifié dans la structure **to.**

L'annulation de la prise en compte de l'interruption est faite par :

```
int pthread_intr_detach_np(pthread_intr_t *intrp)
```

- Détruit l'objet intrp,
- Si un thread était en attente de l'interruption, l'appel retourne avec un statut d'erreur EIDRM.

Le contrôle de la prise en compte de l'interruption est faite par :

```
int pthread_intr_control_np(pthread_intr_t *intrp,
    int cmd)
```

- cmd peut prendre les valeurs PTHREAD_IENABLE ou PTHREAD_IDISABLE.

VI.6 – Signaux

L'équivalent software d'une interruption ou d'une exception, les signaux peuvent être générés par :

- Une exception (pagination, division par zéro, etc.)
- Une interruption (CTRL-C, CTRL-Z, terminaison d'une opération d'entrée / sortie, expiration d'un timer, etc.),

Une signalisation explicite (kill).

Les signaux autorisent une gestion asynchrone par la notification immédiate et la gestion de la concurrence (signalisation de la fin de l'exécution de l'autre tâche). Mais, cela peut avoir des conséquences :

- Complexes à gérer,
- Non déterministes,
- Moins efficaces que les communications synchrones en termes de performances.

VI.7 - Signaux POSIX.1

La figure VI.5 illustre les différents signaux de la norme PSIX.1.

Signal	*Num.*	*Effet*	*Origine*	*Signal*	*Num.*	*Effet*	*Origine*
SIGUO	1	Exit	Hangup	SIGUSR1	16	Exit	User signal
SIGINT	2	Exit	Interrupt	SIGUSR2	17	Exit	User signal 2
SIGQUIT	3	Core	Quit	SIGCHLD	18	Ignore	Child status changed
SIGILL	4	Core	Illegal instruction	SIGPWR	19	Ignore	Power fail/restart
SIGTRAP	5	Core	Trace/breakpoint trap	SIGWINCH	20	Ignore	Window size changed
SIGABRT	6	Core	Abort	SIGURG	21	Ignore	Urgent socket condition
SIGEMT	7	Core	Emulation trap	SIGPOLL	22	Exit	Pollable event
SIGSFPE	8	Core	Arithmetic exception	SIGSTOP	23	Stop	Stopped (signal)
SIGKILL	9	Exit	Killed	SIGSTP	24	Stop	Stopped (user)
SIGBUS	10	Core	Bus error	SIGCONT	25	Ignore	Continued
SIGSEGV	11	Core	Segmentation fault	SIGTTIN	26	Stop	Stopped (by input)
SIGSYS	12	Core	Bad system call	SIGTTOU	27	Stop	Stopped (by output)
SIGPIPE	13	Exit	Broken pipe	SIGSVTALRM	28	Exit	Virtual timer expired
SIGALRM	14	Exit	Alarm clock	SIGPROF	29	Exit	Profiling timer expired
SIGTERM	15	Exit	Terminated	SIGXCPU	30	Core	CPU time limit exceeded
				SIGXFSZ	31	Core	File size limit exceeded

Figure VI.5 – Les signaux POSIX.1

L'envoi est effectué par : kill(pid_t pid, int SIGNAL). Les comportements à la réception d'un signal donné :

- Blocage, en attendant de le traiter ultérieurement,
- Ignorance : aucune action spécifique n'est prise,
- Gestion par l'établissement d'une fonction (handler) qui va être appelée à chaque fois que le signal va être reçu.

VI.8 - Gestion du comportement à la réception

La gestion du comportement à la réception est effectuée par la primitive sigaction**.** Elle précise les détails du comportement du processus à la réception du signal et utilise une structure de type :

```
struct sigaction { void (*sa_handler)();
           sigset_t sa_mask; int sa_flags;
 void (*sa_sigaction)(int siginfo_t *,
           void *);
                     }
int sigaction(SIGNAL, struct sigaction *p_action,
           struct sigaction *p_action_anc);
```

Son handler "sa_handler" est un pointeur sur une fonction qui sera appelée à chaque fois que le processus recevra un signal : void handler_pour_SIGNAL(int signum).Il peut être remplacé par SIG_IGN ou SIG_DFL, mais attention à la définition d'autres handlers (cachés) pour le même signal.

Son masque "sa_mask" permet d'effectuer la liste des signaux bloqués pendant l'exécution du handler. Elle est constituée des signaux normalement bloqués, du signal reçu et de la liste donnée. SIGKILL et SIGSTOP ne peuvent être bloqués.

Le flag "sa_flags" est pour indiquer la façon de gérer les signaux :

- SA_RESTART : les appels systèmes lents interrompus par le signal sont redémarrés automatiquement.
- SA_NODEFER : le signal ne sera pas bloqué pendant l'exécution du handler.
- SA_RESETHAND : après exécution du gestionnaire, le comportement par défaut est rétabli.

Plus quelques autres flags peu utilisés (voir les pages man).

La gestion simplifiée est effectué par : signal(int signum, (void *) handler). Elle associe le gestionnaire handler au signal signum. Elle retourne : un pointeur sur l'ancienne routine de

gestion du signal ou SIG_ERR en cas d'erreur (signal inexistant ou non géré, comme SIGKILL). Le handler a la même forme que pour sigaction (y compris l'usage de SIG_IGN et SIG_DFL) : void handler(int signum). Il y a beaucoup de limitations dans signal qu'il vaut mieux donc éviter d'utiliser.

VI.9 - Blocage des signaux

Pendant l'exécution d'un handler et pendant toute une partie du programme, l'ensemble des signaux bloqués défini par un masque de type sigset_t sont effectués par :

```
sigprocmask(int opération,
    struct sigset_t *p_masque,
    struct sigset_t *p_masque_anc);
```

Les opérations sont :

- SIG_BLOCK: ajout d'un signal,
- SIG_UNBLOCK : retrait d'un signal,
- SIG_SETMASK : établissement du masque.

VI.10 - Gestion du masque des signaux

La gestion du masque des signaux est effectué par les commandes suivantes :

```
sigemptyset(sigset_t *p_m);
sigfillset(sigset_t *p_m);
sigaddset(sigset_t *p_m, int sn);
sigdelset(sigset_t *p_m, int sn);
sigismember(sigset_t *p_m, int sn);
```

VI.11 - Synchronisation sur les signaux

La commande pause() met le processus en hibernation jusqu'à l'arrivée d'un signal non bloqué (masque général des signaux bloqués pour le processus) et exécute le handler ou effectue le comportement par défaut, puis retourne.

La commande sigsuspend(sigset_t *p_m) prend le masque de signaux qui lui est passé et l'installe temporairement comme masque des signaux bloqués du processus, met le processus en hibernation jusqu'à l'arrivée d'un signal non bloqué, exécute le handler du signal reçu et retourne à l'instruction suivante quand l'exécution du handler est terminée. Toutes les opérations internes sont atomiques (pas de risques d'interférences).

VI.12 - Contenu du gestionnaire de signaux

En théorie, on ne devrait pas faire autre chose que modifier des variables globales de type sig_atomic_t (déclarées en mode volatile). En pratique, on peut accéder à 'autres types de données globales si le masque des signaux bloqués est géré correctement (problème d'accès simultané par des gestionnaires). On ne doit appeler que des fonctions ou appels systèmes réentrants : Proscrire malloc ou free. Il existe une liste minimale des fonctions non interruptibles ou réentrantes (async-system-safe), Proscrire l'utilisation de fonctions graphiques.

VI.13 – Exemple

a - Processus attendant une interaction du monde extérieur

```
void interaction_terminee (int sig_num) {
        …
        }
main (int argc, char **argv) {
        struct sigaction sa;
        sigset_t signaux_en_attente, signaux_bloques;
        sigemptyset(&signaux_en_attente);
        sigaddset(&signaux_en_attente, SIGUSRI);
        sigprocmask(SIG_BLOCK, &signaux_en_attente, NULL);
        sa.sa_handler = interaction_terminee;
        sigemptyset(&sa.sa_mask);
        sa.sa_flags =0;
        if (sigaction(SIGUSRI, &sa, NULL)) {
                perror(sigaction);
                exit(1);
        }
```

b - Processus attendant une interaction du monde extérieur (suite)

```
        sigemptyset(&signaux_bloques);
        while (1) { /* sequence d’instruction */
                sigsuspend(&signaux_bloques);
                …
```

```
    }
    …
}
```

VI.14 - Limitations des signaux POSIX.1

Elles ne sont pas suffisamment nombreuses. Elles ne peuvent pas être mises en queue. Elles ne respectent pas l’ordre d’arrivée dans le traitement. Elles ne transportent pas d’information autre que leur identité. Elles sont lentes. **POSIX.4** fournit une première extension de la gestion des signaux qui solutionne la plupart de ces limitations. Les signaux **POSIX.4** sont identiques aux signaux **POSIX.1** (seule la gestion diffère). Les signaux temps réel ajoutent de la fonctionnalité :

- Plus nombreux,
- Notion de priorité entre les signaux,
- Possibilité de les mettre en queue.

VI.15 - Utilisation des signaux POSIX.4

On utilise encore la structure sigaction et la primitive sigaction. flag SA_SIGINFO défini par POSIX.4 pour les signaux temps réel, mais s’applique aussi aux autres signaux et permet de passer d’autres informations que le numéro du signal au handler. Le handler est alors pointé par sa_sigaction (au lieu de sa_handler). Les informations sont stockées dans une structure de type siginfo_t. Le handler a la forme :

```
void my_handler(int signum,
    siginfo_t *info,
    void *foo);
```

VI.16 - Passage de l'information

Le passage de l’information est effectué par :

```
typedef struct {
    int si_signo /* signal number */
    int si_errno /* error number */
    int si_code /* signal code */
    union sigval si_value /* signal value */
} siginfo_t;
union sigval {
```

```
        …
        int sival_int;
        void *sival_ptr;
        …
};
si_code
        SI_QUEUE
        SI_TIMER
        SI_ASYNCIO
        SI_MESGQ
        SI_USER
```

VI.17 - Gestion des signaux POSIX.4

L'envoi d'un signal POSIX.4 est effectué par :

```
int sigqueue(pid_t pid,int signo,
        const union sigval value);
```

Sa mise en queue du signal (remplace kill) et sa consommation de ressources système.

VI.18 - Signaux temps réel

Les extensions des signaux SIGUSR1 et SIGUSR2 exigent un ombre plus importants de signaux utilisateurs, la mise en queue des occurrences, la délivrance prioritaire et l'informations supplémentaires pour le handler. Il n'y a pas de noms spécifiques définis par des numéros compris entre SIGRTMIN et SIGRTMAX, d'où la possibilité d'empiler jusqu'à mille vingt quatre occurrences. Quand plusieurs signaux temps réel sont présents, ils sont délivrés par ordre de numéro croissant. La commande "SIGRTMIN" est donc le signal le plus prioritaire. Mais, attention aux effets pervers (l'ordre de délivrance ne sera pas l'ordre d 'émission), etc.

VI.19 - Autres mécanismes de génération

Des signaux peuvent être générés par d'autres mécanismes que "sigqueue" : notification d'écriture dans une file vide, timers POSIX.4 et Entrées/sorties asynchrones. Ils utilisent la structure "sigevent" suivante :

```
struct sigevent{
```

```
...
int sigev_notify; /* type de notification */
...
int sigev_signo; /* numéro du signal */
...
union sigval sigev_value; /* valeur */
...
}
```

On reviendra sur cette structure à propos des timers.

VI.20 - Synchronisation sur les signaux POSIX.4

En complément de "sigsuspend", on a :

```
int sigwaitinfo (const sigset_t *set,
        siginfo_t *info);
int sigtimedwait(const sigset_t *set,
        siginfo_t *info,
        const struct timespec *timeout);
```

Set est le masque des signaux attendus. Le handler n'est pas exécuté à la réception d'un signal attendu. À la réception d'un signal attendu, celui-ci est retiré de la liste des signaux en attente et le numéro du signal reçu est retourné comme statut de sigwaitinfo.

VI.21 - Signaux et threads

Le tableau suivant illustre la sélection des signaux et des threads selon les origines et les cibles effectives.

Origine	**Cible effective**	**Sélection du thread**
Asynchrone	Un thread	Thread d'origine
pthread_kill	Un thread	Le thread visé
kill	Le processus	Sur la base du masque des signaux des threads

L'envoi d'un signal est faite par :

pthread_kill(pthread_t tid, int signo);

Et cela uniquement à l'intérieur d'un processus donné.

La gestion du masque est faite par :

```
pthread_sigmask(int op, const sigset_t *set,
    sigset_t *oldset);
```

op : SIG_SETMASK, SIG_BLOCK, SIG_UNBLOCK

L’attente est faite par :

```
sigwait(const sigset_t *set, int *sig);
```

L’attente d'un des signaux contenus dans set, stockage du numéro du signal reçu dans signo et retour.