GIF-3002 Programmation de périphériques

Ce cours discute de la programmation de périphériques en trois volets. D'abord, il traite de configuration des interruptions et du DMA. Ensuite, il donne quelques conseils généraux sur la création d'une interface logicielle à travers un exemple. Enfin, ce document donne quelques conseils pratiques sur la programmation des diverses interfaces vues en classe, d'un point de vue matériel.

1 Configuration des interruptions et du DMA

1.1 Configuration des interruptions

Dans presque tous les systèmes microprocesseurs, les adresses des routines traitant les interruptions (ISR) sont dans un tableau 1D, la table des vecteurs d'interruption. Pour initialiser les interruptions, il faut donc presque toujours écrire l'adresse des ISRs dans la table des vecteurs d'interruptions.

Il faut aussi parfois associer les périphériques à un indice de la table des vecteurs d'interruption de façon logicielle. Par exemple, il peut être possible d'associer le vecteur d'interruption #5 aux interruptions du timer 0. Dans les autres systèmes, plus communs, l'association entre les périphériques et le numéro de vecteur d'interruption est déterminée par le matériel et, de ce fait, inchangeable par logiciel.

> *Pour associer les périphériques à un vecteur d'interruption, chaque périphérique possède un numéro d'identification déterminé matériellement.*
>
> *Quels sont les avantages de relier les périphériques aux vecteurs d'interruptions avec du logiciel par rapport au matériel? Les désavantages?*

Habituellement, il existe plusieurs niveaux d'activation et de désactivation des interruptions. Par exemple, pour activer l'interruption d'un timer, il faudra: écrire un registre du contrôleur d'interruption qui permet un nouveau déclenchement de l'interruption du timer (ôter le signal d'une interruption antérieure); écrire un registre du contrôleur d'interruption qui permet les interruption du timer; écrire un registre du contrôleur d'interruption qui permet les interruptions de manière générale; écrire un registre de mode du timer qui déclenche une interruption à tous les débordements du timer; et, finalement, écrire un registre du timer pour que celui-ci commence à compter…

> *Lorsqu'on manipule les interruptions d'un nouveau microcontrôleur, rien ne vaut un bon exemple!!!*

Pour chaque périphérique, il y a généralement une ou des conditions bien précises qui génèrent l'interruption du périphérique. Par exemple, pour un UART, vous pourriez avoir : octet reçu, octet transmis, faute de parité, erreur dans les lignes de contrôle (DTR, DSR, RTS, CTS…) si elles sont utilisée, …

Lorsque plusieurs conditions peuvent déclencher l'interruption d'un périphérique, un registre de statut du périphérique est habituellement utilisé pour indiquer la(les) cause(s)

de l'interruption. Ce registre est généralement un champ de bits où chaque bit identifie une cause possible de l'interruption du périphérique.

Pour chaque cause d'interruption, il y a généralement une méthode spécifique pour désactiver le signal d'interruption, désactiver le signal d'interruption étant généralement un pré requis nécessaire à ce que l'interruption se reproduise. Parfois, les signaux d'interruptions sont gérés automatiquement par le matériel, mais il faut souvent lire ou écrire un registre de contrôle afin de réactiver l'interruption.

Certains microprocesseurs ont plusieurs signaux d'interruption pour les périphériques. On retrouve presque toujours la combinaison NMI et IRQ, mais il y en a d'autres comme les processeur ARM qui ont IRQ et FIQ. Habituellement, dans le premier cas (NMI et IRQ), seul un périphérique contrôle NMI alors que tous les autres périphériques sont associés à IRQ. Dans le second cas, des registres du contrôleur d'interruption permettent généralement d'associer chaque périphérique à l'une ou l'autre interruption.

Pour configurer les interruptions, il faut toujours déterminer leurs priorités. Cela peut se faire de plusieurs façons, en fonction du contrôleur d'interruption : par numéro de vecteur (les numéros les plus bas ont la plus haute priorité), avec des registres qui déterminent la priorité de chaque interruption, avec du logiciel si le matériel peut indiquer plusieurs sources d'interruptions dans la même interruption… Dans tous les cas, configurer les interruptions demande d'attribuer des priorités à celles-ci.

La table des vecteurs d'interruption peut habituellement être déplacée en mémoire RAM pour plusieurs raisons, mais principalement pour permettre l'exécution d'interruptions pendant l'écriture de la mémoire non-volatile du système : la plupart des mémoires non-volatiles (toutes ?!?) ne permettent pas un accès en lecture pendant l'écriture de n'importe qu'elle partie de la mémoire. Par exemple, une interruption survenant pendant l'écriture de la page 21 de la FLASH peut causer une erreur critique du système si la table des vecteurs d'interruption est à la page 0 de la FLASH…

> *Quelles sont les autres raisons qui pourraient motiver de mettre la table des vecteurs d'interruption en RAM?*

1.2 Configuration du DMA

Habituellement, configurer un transfert par DMA est une opération assez complexe : il faut configurer la source, configurer la configuration et configurer le contrôleur de DMA pour atteindre ses objectifs. Comme pour les transferts par interruption, lire la datasheet du microcontrôleur ou utiliser un exemple bien documenté est une approche nécessaire.

Le DMA fonctionne généralement par canaux : on établit un canal entre un périphérique et la mémoire ou entre la mémoire et la mémoire. Les informations de base sur un canal sont les adresses de départ du transfert (adresse source et adresse destination), l'incrément d'adresse entre chaque transfert (adresse source et adresse destination), la taille des mots à transférer et celle de ceux à recevoir (parfois différent!) et le mode d'opération du canal (voir plus loin).

Si on transfert des données d'un ADC 12-bits vers une mémoire 32-bits par exemple, l'adresse de source sera celle du registre de données de l'ADC, l'adresse de source n'incrémentera pas pendant le transfert et les mots lus seront de 2 octets. L'adresse de la destination, elle, sera celle du buffer de données en mémoire, elle incrémentera de 4 et les mots écrits seront de 4 octets (avec 20 bits inutiles par mot).

Les informations nécessaires pour décrire le canal de DMA sont soient à l'intérieur du contrôleur de DMA, soit, plus vraisemblablement, à l'intérieur de la mémoire. Il est fréquent de devoir attribuer une plage de mémoire pour décrire les canaux de DMA.

La plupart des contrôleurs de DMA supportent trois modes de transfert sur un canal: un transfert unique, un transfert ping-pong et un transfert qui regroupe des éléments de plusieurs sources vers une destination ou d'une source vers plusieurs destinations (scatter-gather) :

- Le mode transfert unique permet de transférer un bloc de données seulement. Il s'agit d'un évènement ponctuel.
- Le transfert ping-pong implique deux ensembles source-destination : le transfert par DMA se fait entre l'ensemble source-destination 1, puis il se fait entre la source-destination 2, puis il se fait de nouveau entre la source-destination 1… en ping-pong entre les deux ensembles. Le mode ping-pong est habituellement utilisé lors de traitement de signal : pendant qu'un buffer X de mémoire est rempli, on effectue des FFTs sur un buffer Y. Quand le buffer X est plein et que le traitement sur le buffer Y est fini, le rôle des buffers X et Y est interchangé…
- Le scatter-gather est un mode dans lequel le transfert par DMA est une séquence de petits transferts de différents périphériques vers la mémoire ou vice versa. Le contrôleur de DMA permet d'effectuer automatiquement une séquence de transferts.

Les transferts de DMA peuvent être déclenchés par les périphériques s'ils sont configurés adéquatement. Par exemple, le circuit de I2C pourrait déclencher automatiquement un transfert par DMA s'il a mis plusieurs octets dans une FIFO de réception… Les transferts par DMA peuvent également être déclenchés par le logiciel, par le code de l'usager. Dans ce cas, ils génèrent habituellement une interruption lorsqu'ils sont terminés.

Dans les systèmes microprocesseurs modernes, les interactions entre le DMA et le microprocesseur sont souvent minimales, gérée par le matériel, et souvent masqués d'un point de vue logiciel: à moins d'avoir une application demandant des transferts de données qui sont près des limites du système, le programmeur n'a pas à gérer l'arbitration des bus de communication ou les signaux de contrôle du DMA. Par contre, le programmeur doit s'assurer de configurer le contrôleur de DMA pour que les priorités des différentes tâches du système soient respectées.

2 Interfaces logicielles

Voici un exemple de programmation de périphérique afin d'illustrer plusieurs concepts reliés aux interfaces logicielles de périphérique : le code d'un module de communication UART sera présenté dans cette section.

2.1 Organisation du code

Habituellement, les fonctions et les variables servant au contrôle d'un périphérique sont dans un (ou plusieurs) fichier .c ou .cpp. Certaines de ces fonctions servent d'interface entre le restant du code et le périphérique. Elles sont alors définies dans le fichier .c et déclarées dans un fichier .h dont le nom est identique au fichier .c. Les fichiers où se retrouvent des applications devant accéder au périphérique incluront le fichier .h.

Main.c	Uart.c	Uart.h

#include "Uart.h"
 	
void main(void)
{
  InitUart();
 	
  while(1)
  {
  if(Event())
  SendBytesOnUART(…);
  …
  }
}
	#include "Uart.h"
	
	//Fonction locales
	void InitUartIntrerrupts();
	void InitUartBuffers();
	
	void InitUart(void)
	{
	  InitUartIntrerrupts();
	  InitUartBuffers();
	}
	…
		#ifndef _UART_H_
		#define _UART_H_
		
		void InitUart(void);
		void SendBytesOnUART(…);
		void IsByteReceivedOnUART(…);
		…
		
		#endif

2.2 Accès au module d'E/S

La plupart des accès aux modules d'E/S de font à partir des registres de ces derniers. Les registres sont à des adresses (ports) prédéterminées par le matériel.

Par ailleurs, les accès aux périphériques lents comme le UART (disons une vitesse de communication de 19200bps –bits par seconde-) se font habituellement par interruption. Lorsque des données sont reçues ou transmises le module d'E/S génère une interruption et un registre de statut indique la cause de l'interruption.

```
__irq void UART_ISR(void)
{
        unsigned char UARTStatus;

        //On assume que lire UART_STATUS_REGISTER permettra le déclenchement
        //d'un nouvel interrupt et permet de déterminer la source de l'interruption
        //duUART.
        UARTStatus = UART_STATUS_REGISTER;

        if(UARTStatus & BitIndiquantOctetRecu)
        {
                //Le registre BYTE_RECEIVED_REGISTER contient l'octet reçu
                MetDansLeBufferDeReception(BYTE_RECEIVED_REGISTER);
        }

        if(UARTStatus & BitIndiquantOctetTransmis)
        {
```

```
            if(EncoreDesBytesATransmettre())
            {
                //Le registre UART_TX_REGISTER contient l'octet à transmettre
                UART_TX_REGISTER = GetNextByteÀTransmettre();
            }
        }
}
```

Le code dans l'interruption doit être court pour plusieurs raisons. Les raisons principales sont: l'interruption retarde le traitement des autres interruptions de priorité égale ou plus petite; il est possible de manquer des interruptions de priorité égale ou plus petite si ces interruptions se produisent plus d'une fois pendant le traitement d'une autre interruption.

Dans l'exemple ci-dessus, les interruptions du UART peuvent se produire aux 520us environ (le temps pour transmettre un octet est 10 bits/octets / (19200bits/seconde). Si traiter un octet dans l'interruption prend plus de 520us ou si une autre interruption de priorité égale ou supérieure à celle de l'UART dure plus de 520 us, un octet peut se perdre…

2.3 Données partagées entre l'interruption et le main : programmation concurrente

Lors de l'interruption du périphérique, il arrive que des données doivent être écrite dans un tampon de mémoire partagé avec d'autre tâches ou que certains autres périphériques, également utilisé par d'autres tâches doivent être utilisés. Dans ces cas, il faut appliquer les principes de programmation concurrent : mettre des sémaphores pour gérer l'accès aux ressources communes, réduire les tailles des sections critiques, désactiver les interruptions…

Voici un exemple de mauvaise programmation. Dans cet exemple, un tableau linéaire est utilisé pour entreposer les octets reçus du port série. Lorsqu'un octet est reçu, la variable NombreOctetsRecusUART est incrémenté et l'octet reçu est mis à la fin du tableau. Dans le main, on vérifie si un octet est reçu du UART (si NombreOctetsRecusUART > 0). Si c'est le cas, l'octet reçu est traité. Puis, les octets dans tableau sont décalés vers l'indice 0 et NombreOctetsRecusUART est décrémenté.

```
__irq void UART_ISR(void)
{
        if( UnOctetEstRecu() )
        {
            Tableau[NombreOctetsRecusUART] = OctetRecu();
            NombreOctetsRecusUART++;
        }
        …
}

void Main(void)
{
        int  i;
        init();
        while(1)
        {
        …
```

```
        if(NombreOctetsRecusUART > 0)
        {
                for(i  = 0; i < NombreOctetsRecusUART; i++)
                {
                        OctetAtraiter = Tableau[i];
                        TraiteOctet(OctetAtraiter);
                }
                NombreOctetsRecusUART = 0;
        }
        }
}
```

Cet exemple est mauvais parce qu'un octet est perdu si un octet est reçu entre le traitement du dernier octet et la mise à 0 du nombre d'octets reçus. Les interruptions du UART se produisent de manière asynchrone par rapport au main…

Il y a plusieurs solutions pour rendre l'implémentation correcte :

1) Désactiver les interruptions dans le main, lorsqu'on manipule NombreOctetsRecusUART, qui est une variable partagée entre deux contextes.
2) Utiliser un drapeau pour indiquer que le Tableau est utilisé ou disponible (les interruptions seront aussi désactivées lors du test), mais moins longtemps.
3) Éliminer la section critique : utiliser un buffer circulaire avec une tête de lecture et une tête d'écriture pour gérer les octets reçus.

Ces trois solutions s'appliquent dans la plupart des cas de conflits de ressource ou de mémoire (variables partagées) :

1) Désactiver les interruptions est simple et facile, mais cela retarde le traitement de celles-ci avec toutes les conséquences négatives que cela peut entraîner (voir ci-dessous).
2) Utiliser un sémaphore ou des variables de contrôle afin d'entrer ou de sortir de la section critique est une bonne solution. Toutefois, cela est plus complexe que désactiver les interruptions et cela peut entraîner d'autres problèmes (voir par exemple l'inversion de priorité et les deadlocks dans le cours sur les RTOS).
3) Éliminer la section critique est toujours une solution idéale. Seulement, cette solution demande toujours une duplication des ressources utilisées et elle ne s'applique que lorsqu'il est possible d'écrire une variable dans un seul contexte seulement (dans l'exemple, il est nécessaire d'écrire le tableau dans l'interruption seulement)…

Il faut noter que les sémaphores s'utilisent habituellement avec des tâches ou processus ayant des priorités souvent égales et qui seront appelées de nouveau par le système d'exploitation si la ressource protégée par le sémaphore n'est pas disponible.

> *Par définition, un sémaphore désactive l'interruption du système d'exploitation afin d'éviter un changement de contexte pendant que la valeur du sémaphore est évaluée.*

La relation entre le main et une interruption qui peuvent partager des ressources, est totalement différente : l'interruption a, par définition, une priorité plus grande que la tâche et elle ne sera pas appelée de nouveau par le système d'exploitation si elle ne peut pas s'exécuter… Pour ces deux raisons, la solution 2 s'applique rarement dans un contexte de programmation des interruptions sans qu'elle ne redevienne la solution 1, c'est-à-dire désactiver les interruptions!

2.3.1 Écriture de variables dans plusieurs contextes

Une bonne pratique de programmation consiste à surveiller attentivement toutes les variables qui sont écrites dans des contextes différents et à éviter la situation lorsque possible.

Dans le cas de l'exemple ci-dessus, la variable NombreOctetsRecusUART change de valeur dans l'interruption et dans le main. Cette simple observation peut conduire à trouver des erreurs dans votre code…

Dans le même ordre d'idée, un drapeau utilisé pour indiquer un évènement devrait être levé à un seul endroit et rabaisser à un seul autre endroit. Écrire la valeur d'un drapeau à plusieurs endroits peut causer des erreurs…

> *Supposons, par exemple simpliste, que le drapeau « OctetReçu » soit mis à 1 dans l'interruption du UART et dans l'interruption du SPI. Supposons que ce drapeau soit remis à 0 dans le main lorsqu'il vaut 1 : si des octets sont reçus du SPI et du UART simultanément, un des deux octets sera perdu… Il aurait fallu deux drapeaux : OctetReçuDuUART et OctetReçuDuSPI…*

2.3.2 « Buffer » circulaire

Les données sont écrites dans un buffer circulaire jusqu'à ce que la fin du buffer ne soit atteinte. Quand cela survient, les données sont écrites au début du buffer, écrasant celle en place.

Un buffer circulaire a habituellement une tête de lecture et une tête d'écriture. La tête d'écriture est écrite dans le contexte qui met les données dans le buffer. La tête de lecture est écrite dans le contexte qui lit les données dans le buffer. Un « buffer » circulaire est très pratique pour éviter d'écrire une variable dans deux contextes.

```
unsigned char MonBufferCirculaire[TAILLE_DU_BUFFER];
unsigned int TeteDEcriture = 0;
unsigned int TeteDeLecture = 0;

void EcritOctetDansMonBuffer(unsigned char OctetAEcrire)
{
        MonBufferCirculaire[TeteDEcriture] = OctetAEcrire;
        TeteDEcriture = TeteDEcriture + 1;
        if(TeteDEcriture >= TAILLE_DU_BUFFER)
                TeteDEcriture = 0;
}

unsigned char LireOctetDansMonBuffer(void)
{
        unsigned char OctetLu;
        OctetLu = MonBufferCirculaire[TeteDeLecture];
        TeteDeLecture = TeteDeLecture + 1;
        if(TeteDeLecture >= TAILLE_DU_BUFFER)
                TeteDeLecture = 0;
        return OctetLu;
}
```

2.3.3 Désactiver les interruptions ou interruption haute-priorité trop longue

Désactiver les interruptions, que ce soit volontairement ou parce qu'une interruption de priorité supérieure est exécutée, peut causer plusieurs erreurs dans l'exécution d'un code :

- Manquer des interruptions de périphériques si une interruption se produit plusieurs fois pendant qu'elle est désactivée.
    - Par exemple, si une interruption haute priorité dure 2 millisecondes et que vous recevez 4 octets du UART à 115200bps pendant ce temps, le code ne rapportera que le dernier octet reçu.
- Masquer l'interruption en cours si elle se reproduit pendant son traitement.
- Causer la famine des autres interruptions ou du main
    - Par exemple, si une interruption de timer revenant à toutes les milliseconds s'exécute en 0.99 millisecondes, il est possible que les autres interruptions ou le main ne soient jamais exécutés.
- Fausser les intervalles de temps du système si les interruptions de timer sont désactivées.

- En mode auto-reload cependant, les intervals de temps moyens entre deux interruptions peuvent être préservés.

3 Conseils pratiques pour chaque périphérique

Voici quelques conseils simples ou des astuces pour la programmation des périphériques vus en classe.

3.1 Conseils généraux

Si vous ne connaissez pas les ressources utilisées par une fonction ou si vous ne connaissez pas toute les sorties possibles d'une fonction par rapport aux entrées, n'utilisez pas la fonction…

3.2 GPIO, Clavier et LEDs

Les décharges électrostatiques sont toujours à prévoir lorsqu'on utilise des GPIOs. Vos applications logicielles et matérielles devraient être robustes aux variations temporaires et imprévues causées par ces dernières.

Pour éviter d'éventuels court-circuit lorsque l'on active une ligne de clavier, il n'est pas toujours nécessaire d'ajouter du matériel. Si une seule ligne à la fois est une sortie, il ne peut y avoir de collision…

Un LCD est un périphérique lent qu'il faut initialiser avec soin. Quand un LCD ne fonctionne pas, il arrive souvent que le problème soit lié à l'intensité de l'afficheur.

3.3 Timers

L'interruption du timer n'est pas obligatoirement haute priorité. Habituellement, en mode auto-reload, il est possible de garder une base de temps précise même si d'autres interruptions retardent le traitement de l'interruption du timer.

Il faut habituellement éviter de mettre plusieurs variables pour compter le temps dans l'interruption du timer. Une pratique intéressante (permettant de minimiser le temps dans l'interruption consacré au traitement des compteurs) consiste à utiliser une seule variable pour compter le temps (exemple : unsigned int tickcount, incrémenté à toutes les interruptions de timer). Dans le main, on utilise des timestamp = tickcount,.lors d'un évènement, et des tickcount – timestamps pour déterminer le temps écoulé depuis l'évènement.

Il faut toujours gérer le débordement des variables utilisées pour compter le temps (quant tickcount passe de 2^32-1 à 0…). Faire des différences de temps permet habituellement de régler le problème. Par ailleurs, il faut idéalement utiliser des variables de même dimension/nature pour manipuler/gérer le temps : comparer des shorts avec des ints conduit souvent à des résultats catastrophiques…

3.4 ADC et DAC

Il doit toujours y avoir un filtre matériel, anti-recouvrement spectral, à l'entrée d'un ADC. Ce filtre passe-bas est déterminé en fonction de la vitesse d'échantillonnage…

Transférer des échantillons de l'ADC vers la mémoire ou des échantillons de la mémoire vers un DAC se fait habituellement, avec du DMA.

Il est commun d'additionner une tension DC (de valeur égale à la moitié de la plage de référence de l'ADC) au signal AC échantillonné pour lire un signal AC négatif. Le programmeur doit généralement mesurer ou filtrer les variations de cette tension DC. Par ailleurs, une valeur DC sera ajoutée aux échantillons de sortie d'un DAC pour produire un signal AC…

3.5 Interfaces séries

La séquence d'interruptions des interfaces séries devrait toujours être bien maîtrisée lorsqu'on construit un pilote pour une de ces interfaces.

Par exemple, le UART est souvent construit avec deux registres de transmission. Le premier registre est disponible pour le programmeur : il s'agit de l'octet transmettre (UART_TX). Le second registre, un registre à décalage, est interne : il s'agit de l'octet en cours de transmission. Une séquence d'interruptions survenant au début de la transmission pourrait être :

- Le programmeur écrit UART_TX pour commencer la transmission
- Le module d'E/S du UART transfère le contenu de UART_TX dans le registre à décalage interne.
- Après la transmission d'un seul bit, le module d'E/S génère une interruption pour dire que UART_TX est vide et peut être écrit de nouveau.
- Si on écrit UART_TX de nouveau dans l'interruption, une nouvelle interruption se produira après la transmission d'un octet…

Autre exemple, le module d'E/S du I2C pourra générer une interruption après avoir transmis un condition de START sur le bus, après le byte d'adresse, après chaque bytes de données…

Vos applications devraient toujours considérer des erreurs potentielles de communication sur les interfaces séries. Ces erreurs, causées par du bruit, de long fils ou des décharges électrostatiques devraient être gérées adéquatement par vos programmes.

3.6 Real Time Clock

La programmation d'un RTC devrait toujours rester simple. Ne vous lancez pas dans l'implémentation de « feature hot» sur le temps à moins d'y être obligé. Vous devriez également toujours programmer un ensemble de fonctions permettant de re-synchroniser aisément le RTC par une interface externe du système.

3.7 Interfaces parallèles

Souvent, le plus difficile dans l'utilisation d'une interface externe est son initialisation. Utiliser un analyseur logique est fortement recommandé…