La gestion du temps par programmation

La gestion des sorties

Pierre-Yves Rochat

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Les Microcontrôleurs disposent d'un outil matériel très puissant permettant la gestion du temps : les Timers, associés généralement aux interruptions. Ces sujets feront partie de la suite du cours. Mais il est possible dès maintenant de résoudre de nombreux problèmes demandant une gestion du temps, avec les connaissance de programmation que nous avons déjà vues.

La durée d'une instruction

Le processeur qui se trouve dans un microcontrôleur exécute en permanence des instructions. La durée de chaque instruction est très courte : même dans un *petit* microcontrôleur, une instruction ne dure souvent qu'une fraction de **micro-seconde** (µs = 10^{-6} secondes). Mais ce temps n'est pas infiniment court et il peut être utilisé pour gérer le temps qui s'écoule.

Sur beaucoup de microcontrôleurs, une instruction dure un certain nombre de cycles élémentaires de l'horloge-mère du processeur, ce nombre pouvant varier selon les instructions. Par exemple les AVR possèdent une horloge interne, souvent à une fréquence de 8 MHz, calibrée à la production. En ajoutant un Quartz externe, cette fréquence peut monter à 20 MHz, en offrant en plus une précision beaucoup plus grande. Les processeurs AVR ont la particularité que presque toutes les instructions ne durent qu'un cycle d'horloge. Une instruction dure donc généralement 125 ns. Sur ces processeurs, il est possible de demander, par un fanion non-volatile de passer à une fréquence 8 fois plus basse que l'horloge de base. Son intérêt est de limiter la consommation de courant, vu que la technologie C-MOS utilisée implique une consommation qui comporte un facteur proportionnel à la fréquence.

Les MSP430 ont aussi une horloge interne. Elle peut fonctionner jusqu'à 16 MHz. Il est possible à tout moment de modifier la fréquence du processeur, en cours d'exécution. Deux fréquences calibrées à la production (1MHz et 16MHz) peuvent être facilement choisies.

Attente active

Il n'est pas toujours facile de prévoir avec exactitude la durée d'une instruction. Les processeurs complexes ont par exemple des mécanismes d'accélération du type *pipe-line* qui optimisent le temps d'exécution en fonction de l'environnement. Mais la durée d'une instruction est **répétitive**. Il est donc simple d'écrire des boucles d'**attentes active,** dont la durée sera répétitive.

Prenons par exemple la procédure suivante :

```
#define BaseTempsMs 460
void AtteneMs (int duree) {
    volatile int j; // variable de comptage pour l'unité de temps
    int i; // variable de comptage du nombre d'unités de temps
    for (i=0; i<duree; i++) {
```

```
        for (j=0; j<BaseTempsMs; j++){
        }
    }
}
```

Le `volatile` est indispensable, sinon le compilateur va optimiser le code et supprimer la seconde boucle.

Cette procédure permet donc de laisser passer un certain temps. Si on souhaite par exemple que ce temps soit exprimé en millisecondes (ms), il faudra calibrer la procédure. On exécutera le programme suivant :

```
while (1) { // boucle de test, censée durer 10 secondes
    AtteenteMs (10000);
    PORTB ^= (1<<0); // fait changer d'état une LED sur PB0
}
```

Au moyen d'un chronomètre, on mesurera le temps de clignotement, qui devrait durer 10 secondes. On ajustera ensuite la constante `BaseMs`. Je l'ai souvent mise à 460 avec un AVR à 8 MHz.

Arduino : delay

Encore une fois, l'environnement Arduino (ou Energia pour le MSP430) offre une procédure similaire : `delay (int duree);`

Elle avait déjà été utilisée lors du premier exemple du cours, qui faisait clignoter une LED :

```
void loop () {
    digitalWrite (LED_ROUGE, HIGH);
    delay (500);
    digitalWrite (LED_ROUGE, LOW);
    delay (500);
}
```

Rappel : cette procédure est bloquante.

Une faible précision

On comprend que cette manière de gérer le temps n'est pas très précise. Elle ne convient visiblement pas à gérer l'heure et la date, pour lesquels même une précision de 0.001 % génère des erreurs de plus d'une seconde par jour. Dans ce cas, on utilise des circuits spécialisés (RTC = Real Time Clock) associés à un quartz (généralement de 32'768 Hz, donc 2^{15} hertz) et une petite pile pour être indépendant de l'alimentation. Dans la suite du cours, nous verrons comment ces circuits communiquent avec un microcontrôleur (par ligne série, I2C ou SPI).

Mais de nombreux problèmes pratiques ne nécessitent pas d'une mesure précise du temps.

Réalisation d'un séquenceur

Nous avons dans les mains tout ce qu'il faut pour programmer des séquenceurs, en

faisant succéder des assignations de sorties et des délais.

On peut ainsi réaliser un générateur de mélodie (par actions successives sur la membrane d'un haut-parleur), la commande d'une enseigne lumineuse (ce sujet sera détaillé dans la suite du cours) et bien d'autres applications. La commander par Modulation de Largeur d'Impulsion (PWM) peut aussi être mise en œuvre de cette manière.

Exemple des feux tricolores

Voici une application, qui réalise une commande de feux tricolores pour un carrefour, en laissant passer successivement les voitures sur un axe, puis sur l'autre.

```
enum {Rouge, Orange, Vert}; // Définitions des couleurs

// Allume une couleur sur le feu de l'axe 1 :
void FeuAxe1 (int couleur) {
    digitalWrite(bitRougeAxe1, LOW); // éteint les 3 couleurs...
    digitalWrite(bitOrangeAxe1, LOW);
    digitalWrite(bitVertAxe1, LOW);
    switch (couleur){ // ...allume la bonne :
      case Rouge  : digitalWrite(bitRougeAxe1, HIGH);
      case Orange : digitalWrite(bitOrangeAxe1, HIGH);
      case Vert   : digitalWrite(bitVertAxe1, HIGH);
    }
}
// Allume une couleur sur le feu de l'axe 2 :
void FeuAxe2 (int couleur) {
...
while (1) {  // boucle principale
  FeuAxe2(Rouge); FeuAxe1(Vert);  // passage sur l'axe routier 1
  AttenteSec(20);
  FeuAxe1(Orange); AttenteSec(3); // fin du passage
  FeuAxe1(Rouge); AttenteSec(1); // bloque l'axe 1
  FeuAxe2(Vert); AttenteSec(20); // passage sur l'axe routier 2
  FeuAxe2(Orange); AttenteSec(3); // fin du passage
  FeuAxe2(Rouge); AttenteSec(1); // bloque l'axe 2
}
```

En observant ce programme, on remarque que la boucle principale dure 48 secondes.

On écrira de manière très similaire un feu pour passage piétons. Il sera alors souvent déclenché par le bouton par lequel les piétons peuvent demander le passage.

Gérer plusieurs tâches

Il faut toujours garder à l'esprit que le processeur ne peut rien faire d'autre durant l'exécution d'une boucle d'attente.

Il est simple de faire clignoter une LED à 2 Hz. Mais faire clignoter en même temps une seconde LED à 3 Hz n'est déjà pas si simple !

A vrai dire, la programmation de processus multiples est tout un chapitre de l'informatique technique, qui dépasse le cadre de ce cours. Mais quelques solutions simples peuvent être appliquées avec les microcontrôleurs.

Dans un programme comme celui des feux tricolores, la boucle principale s'exécute une fois par cycle. Une solution intéressante est de travailler avec une boucle principale qui a une durée courte, par exemple une ms.

On utilisera alors des variables et des tests pour actionner les sorties. Voici par exemple comment réaliser le double clignotant à fréquences inégales :

```
#define ToogleRedLed {P1OUT ^=(1<<0);} // inverse la LED rouge
#define ToogleRGreenLed {P1OUT ^=(1<<6);} // inverse la LED verte
int compteur1=0; int compteur2=0;

while (1) { // boucle principale
    if (compteur1==0) {ToggleRedLed;}
    compteur1++;
    if (compteur1==250) compteur1=0; // 2Hz, demi période : 250ms
    if (compteur2==0) {ToggleGreenLed;}
    compteur2++;
    if (compteur2==166) {compteur2=0;}// 3Hz, demi période : 166ms
    AttenteMs(1);
}
```

Dans ce cas, on va considérer que le temps pour exécuter les premières instructions contenue dans la boucle `while (1)` est négligeable par rapport au temps d'exécution de la routine `AttenteMs(1)`. Et il est possible d'ajouter ainsi d'autres instructions, pour d'autres processus.

Temps « absolu »

Dans les exemples que nous avons vu, le temps a toujours été traité de manière relative. Il est aussi possible d'utiliser le temps de manière « absolue ». Il faut une horloge précise pour avoir la date et l'heure, mais il est facile de connaître le temps écoulé depuis le début de l'exécution du programme, donc généralement depuis que le dispositif comportant le microcontrôleur est allumé :

```
long int TempsMs=0;
while (1) { // boucle principale
    AttenteMs (1);
    TempsMs++;
}
```

Pour ne pas avoir un fonctionnement périodique de 65'536 ms, soit seulement à peine plus d'une minute, on peut utiliser ici une variable `long int`, utilisant 64 bits. Pour être certain que le temps soit toujours un nombre positif, on utilise plutôt `unsigned long int`, qu'on peut écrire simplement `unsigned long`. La variable `TempsMs` ne reviendra alors à zéro qu'après plus d'un mois !

A tout moment, il est possible de consulter la variable TempsMs et effectuer des calculs sur ces valeurs.

L'exemple précédent pourrait s'écrire de cette manière :

```
long int TempsMs=0;
while (1) { // boucle principale
    if ((temps%250) ==0) {ToggleRedLed;}
    if ((temps%166) ==0) {ToggleGreenLed;}
    AttenteMs(1); TempsMs++;
}
```

Rappelons que le signe % correspond au reste de la division entière.

Cette solution peut paraître élégante, vu que le programme est plus court, en ce qui concerne les lignes en langage C. Mais le résultat de la compilation pourrait être décevant : la division, même entière, nécessite beaucoup d'instructions en assembleur, donc de la place en mémoire de programme (flash) et du temps d'exécution !

Arduino : millis et micros

Une fois de plus, l'environnement Arduino (ou Energia pour le MSP430) offre une fonctionnalité similaire . La procédure :

`unsigned long millis()`

rend le nombre de milli-secondes depuis le début du programme.

Ce qui est plus étonnant, c'est qu'il existe aussi une procédure similaire pour les µs :

`unsigned long micros()`

Elle ne pourrait pas être réalisée avec la technique que nous venons de voir. La durée d'une ou de quelques instructions ne peut plus alors être considérée comme négligeable par rapport à une µs. C'est donc en utilisant les Timers et les interruptions, que nous étudierons prochainement dans le cours, que la procédure `micros()` a été réalisée, tout comme aussi la fonction `millis()`.

Rappel : ces deux procédures ne sont pas bloquantes.

Les sorties et les entrées !

Dans la pratique, les applications utilisant seulement les sorties d'un microcontrôleur et le temps qui s'écoule sont rares, bien qu'elles ils existent : feu tricolore à cycle, boîte à musique, afficheur matriciels à LED, enseignes lumineuses animées, etc.

Il est évident que la grande majorité des programmes utilisent aussi les entrées ! Nous verrons dans le chapitre suivant comment gérer les entrées en fonction du temps qui s'écoule.