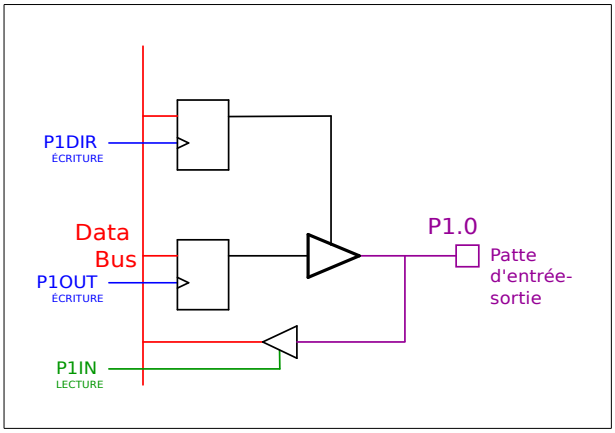
La gestion du temps par programmation

La gestion des entrées

Pierre-Yves Rochat

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE



Les systèmes informatique ont une particularité. Ils offrent un contrôle complet de leurs sorties, alors qu'un incertitude plane toujours sur les entrées. Je m'explique. Dès le moment où un programme a imposé une valeur sur une sortie, cette valeur est maintenue jusqu'à la prochaine écriture sur cette sortie. C'est la bascule associée à chaque sortie qui offre cette certitude.

Par contre, une entrée peut changer à tout moment. Le passeur (driver trois état) associé à l'entrée permet la saisie de la valeur d'entrée à un instant donné. Lors d'une lecture ultérieure de l'entrée, une nouvelle valeur sera lue. Mais rien ne permet de savoir ce qui s'est passé entre ces deux lectures !

Le mécanisme des interruptions, qu'il est possible d'associer à un changement sur une entrée, permet dans une certaine mesure de résoudre ce problème (P*in Change Interrupts*). Il fera l'objet d'une étude approfondie dans la suite du cours.

La scrutation et ses contraintes

En attendant, une entrée doit être lue suffisamment souvent au cours du temps pour s'assurer de prendre connaissance de son état d'une manière qui soit compatible avec la nature de l'entrée. On parle de « lecture des entrées par scrutation » (*polling* en anglais). Chaque application a des contraintes différentes.

Prenons deux exemples :

- Un tourniquet est placé à l'entrée d'une gare de métro. Chaque fois qu'un passager passe, un contact est actionné. La nature mécanique du tourniquet et de son utilisateur (le passager, même s'il est pressé...) implique un certain temps entre chaque changement d'état du contact. Une lecture de l'entrée chaque dixième de seconde (100ms) suffira probablement à compter correctement le nombre de passagers.

- Un moteur moto, pouvant tourner à un maximum de 8'000 tours par minute est associé à un capteur optique composé de 12 bandes noires (signal à 0) séparées par 12 bandes blanches (signal à 1). Dans ce cas, une scrutation de l'entrée chaque ms serait très insuffisante : lors d'une rotation à 8'000 tours/minute, le signal change de valeurs trois fois par ms (précisément 312.5 µs) !

Lecture d'une entrée

Rappelons que la lecture d'une entrée permet de savoir si elle est active ou non au moment de la lecture. Le programme suivant n'est vraiment pas intéressant :

```
while (1) { // boucle principale
    if (PoussoirOn) {
        AllumeLed;
    }
}
```

Il suffit en effet de presser une fois sur le bouton-poussoir pour que la LED reste allumée en permanence.

Ce programme est déjà plus utile :

```
while (1) { // boucle principale
    if (PoussoirOn) {
        AllumeLed;
    } else {
        EteintLed ;
    }
}
```

Il permet d'allumer le LED lorsque le bouton-poussoir est pressé, mais aussi de l'éteindre lorsque qu'il est relâché.

Détection d'un flanc

Très souvent, c'est au moment de l'activation d'une entrée qu'une action doit être réalisée. Il faut donc être capable de détecter un flan. Le programme suivant détecte un flanc montant :

```
while (1) { // boucle principale
    while (!EntreeOn) {
        // on attends le flanc montant
    }
    ... // action
    while (EntreeOn) {
        // on attends le flanc descendant
    }
}
```

Ce programme comporte trois boucle `while` : la boucle principale (infinie) et les deux boucles qui attendent les flancs du signal.

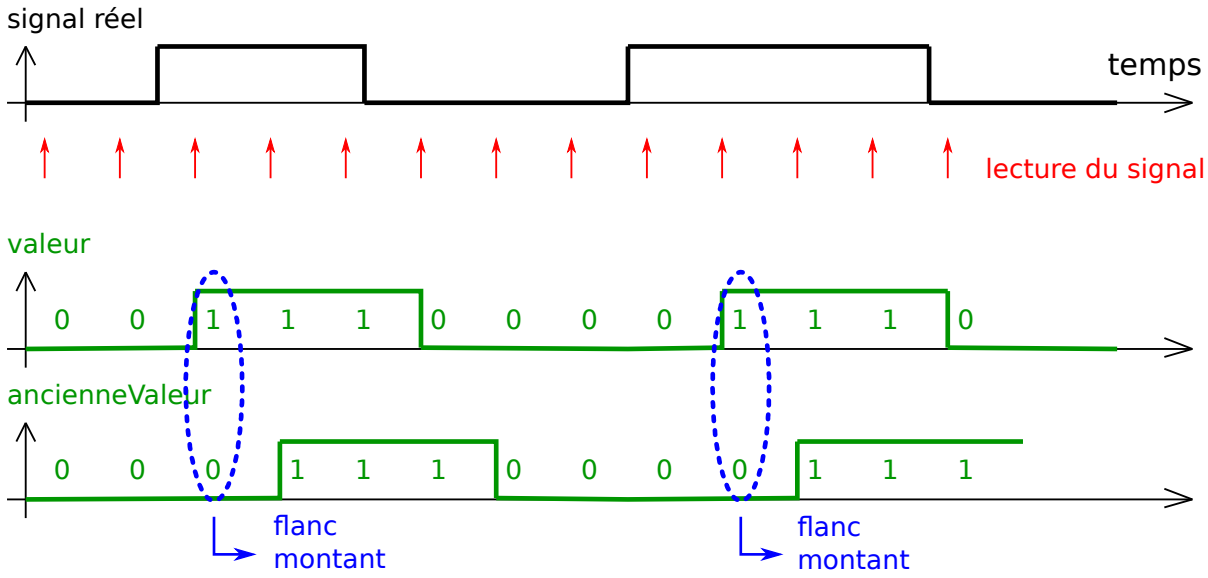
Remarquons que pour la boucle qui attend le flanc montant du signal, il faut mettre dans la condition du `while` le fait que l'entrée soit à zéro : c'est la condition bloquante.

Prenons le réflexe de regarder combien de temps dure la boucle principale. Ici, elle va durer le temps d'un cycle bas-haut de l'entrée.

Cette manière de programmer limite à la scrutation d'une seule entrée, par la présence des boucles `while`, pendant lesquelles rien ne se passe.

Mémorisation de l'ancienne valeur

Le diagramme des temps suivant montre comment il est possible de détecter un flanc sans utiliser un boucle bloquante :



Voici le programme correspondant :

```
#define EntreeOn (digitalRead(entree))

int ancienneValeur=EntreeOn;
int valeur;

while (1) { // boucle principale
    valeur = EntreeOn;
    if (!ancienneValeur && valeur) {
      // action à effectuer sur le flanc montant
    }
    ancienneValeur=valeur;
}
```

On remarque l'utilisation de deux variables : `valeur` et `ancienneValeur`. La variable `valeur` est assignée avec la valeur de l'entrée au début de la boucle principale. Cette variable est copiée dans `ancienneValeur` à la fin de la boucle principale. On remarque que l'entrée n'est lue qu'une fois dans la boucle principale, mais qu'elle est lue au début du programme pour que la variable `ancienneValeur` puisse jouer son rôle déjà durant la première exécution de la boucle principale.

Durant l'exécution de la boucle principale, on dispose donc dans des variables de la valeur de l'entrée qui vient d'être lue, mais aussi de la valeur de l'entrée lors de l'exécution précédente de la boucle. C'est en comparant ces deux deux valeurs qu'on peut facilement détecter des flancs :

- flanc montant si `valeur` vaut 1 et `ancienneValeur` vaut 0
- flanc montant si `valeur` vaut 0 et `ancienneValeur` vaut 1
- flanc quelconque si `valeur` et `ancienneValeur` sont différentes.

Gardons le réflexe de regarder combien de temps dure la boucle principale. Ici, elle va durer un temps très court.

Multi-tâches

Lorsqu'aucune boucle ne se trouve dans la boucle principale, il est facile de gérer plusieurs tâches en même temps. Par exemple, la scrutation d'une seconde entrée peut être ajoutée au programme précédent :

```
int ancienneValeur1=Entree1On;
int valeur1;

int ancienneValeur2=Entree2On;
int valeur2;

while (1) { // boucle principale
    valeur1 = Entree1On; // Lecture les entrées
    valeur2 = Entree2On; // en début de boucle

    if (!ancienneValeur1 && valeur1) {
      // action à effectuer sur le flanc montant de l'entrée 1
    }
    if (ancienneValeur2 && !valeur2) {
      // action à effectuer sur le flanc descendant de l'entrée 2
    }
    ancienneValeur1=valeur1; // mémorisation des valeurs
    ancienneValeur2=valeur2; // en fin de boucle
}
```

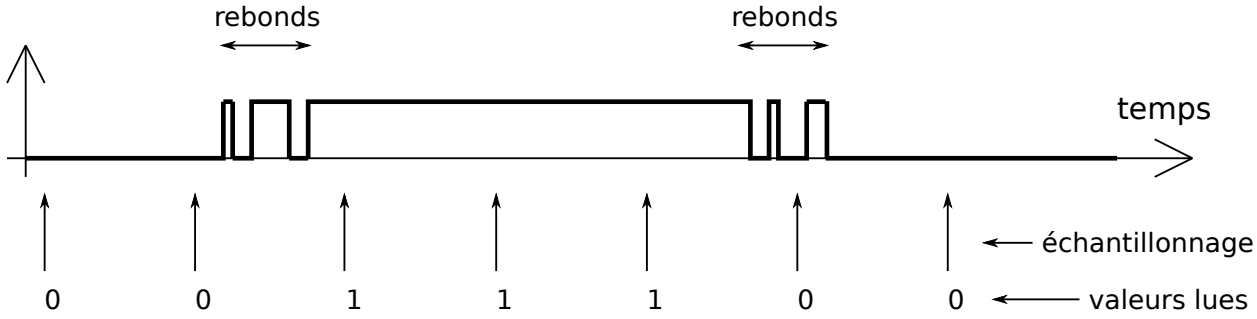
La partie qui gère le premier bouton ne gène pas celle qui gère le second. La boucle principale va s'exécuter à une fréquence très élevée, vu que les quelques lignes de programme en C ne vont générer que quelques dizaines d'instructions assembleur, qui ne prennent chacune qu'une fraction de µs.

La programmation multi-tâche est en fait un chapitre important et complexe de l'informatique industrielle, qui sortirait du cadre de ce cours. Mais pour des applications des microcontrôleurs, on se satisfait souvent de technique simple comme celle qui vient d'être présentée.

Rebonds de contact

Est-ce que les programmes que nous venons d'étudier fonctionnent avec un bouton-poussoir ? On observe en fait un fonctionnement curieux...

Le problème qui se pose est la présence de rebonds de contact. Le bouton-poussoir est un dispositif mécanique et l'élasticité des pièces métalliques qui le composent entraînent des rebonds. Plusieurs solutions sont possible pour résoudre ce problème. Parmi elles, la diminution de la fréquence d'échantillonnage, en ajoutant par exemple dans notre boucle principale un délai. Une valeur d'environ 20 ms conviendra bien. On voit sur le diagramme des temps ci dessous que, même si la lecture se produit durant le rebond, on ne risque pas d'enregistrer le rebond si la période l'échantillonnage est plus longue que la durée maximale des rebonds.




```
while (1) { // boucle principale
    valeur = digitalRead(poussoir);
    if (!ancienneValeur && valeur) {
      // action à effectuer sur le flanc montant
    }
    ancienneValeur=valeur;
    delay(20);
}
```

On remarque ici que la boucle principale est de durée constante, soit ici environ 20ms.

Une autre solution est d'ajouter un délai dès qu'un changement d'état est détecté, pendant lequel aucune lecture ne sera faite. Mais il est alors nécessaire de détecter les deux flancs :

```
while (1) { // boucle principale
    valeur = digitalRead(poussoir);
    if (!ancienneValeur && valeur) {
      delay(20); // masque les rebonds
      // action à effectuer sur le flanc montant
    }
     if (!ancienneValeur && valeur) {
      delay(20); // masque les rebonds
    }
    ancienneValeur=valeur;
}
```

Exemple de la machine à café

Sur une machine à café se trouve deux boutons, permettant de tirer un café « Normal » ou un « Expresso », qui se différencient par la quantité d'eau qui traverse la capsule de café. La pompe assurant un débit presque constant, c'est le temps qui différenciera les deux options.

Ajoutons au cahier des charges que, durant le temps de préparation du café, une pression sur l'un des boutons arrête la pompe : c'est un arrêt d'urgence.

Voici une première solution à ce problème :

```
while (1) { // boucle principale
    PompeOff;
```

```
    do { // attente du départ
        if (BoutonNormalOn) {
            TempsRestant=30000; // 30 secondes
        }
        if (BoutonExpressoOn) {
            TempsRestant=8000; // 18 secondes
        }
    } while (TempsRestant == 0);
    PompeOn;
    do { // Pompage
        AttenteMs(1); TempsRestant--; // décomptage des ms
        if (BoutonNormalOn || BoutonExpressoOn) {
            TempsRestant=0; // arrêt d'urgence
        }
    } while (TempsRestant != 0);
}
```

Est-ce que que ce programme peut fonctionner selon le cahier des charges ?

Non ! Lorsqu'on va presser sur un des boutons, la pompe va bien se mettre en marche, mais elle va s'arrêter une ms plus tard, vu que l'utilisateur n'aura certainement pas encore relâcher le bouton !

Voici une correction possible :

```
    PompeOn; Attente (500); TempsRestant = TempsRestant-500;
```

Durant une demi-seconde, le système ne scrute plus les entrées. Ce temps va permettre à l'utilisateur de relâcher le bouton.

On remarque donc qu'il faut être très attentif au déroulement des opérations au cours du temps en écrivant un programme pour microcontrôleur.

Dans cet exemple, la boucle principale a une durée qui correspond au cycle Attente-Pompage. Il est alors difficile d'envisager de gérer d'autres processus, comme par exemple la régulation de la chaleur de l'eau dans le corps de chauffe.

Essayons de récrire ce programme avec une boucle principale à durée constante :

```
while (1) { // boucle principale
    AttenteMs (1);
    if (TempsRestant == 0) {
        PompeOff;
        if (BoutonNormalOn) {
            TempsRestant=30000; // 30 secondes
            TempsMasqueBoutons=500; // invalide le bouton une 1/2s
        }
        if (BoutonExpressoOn) {
            TempsRestant=8000; // 18 secondes
            TempsMasqueBoutons=500; // invalide le bouton une 1/2s
        }
    }
    else { // Temps restant non nul
```

```
        PompeOn;
        TempsRestant--;
        if ( (BoutonNormalOn || BoutonExpressoOn) &&
             (TempsMasqueBoutons == 0) {
            TempsRestant=0; // arrêt d'urgence
        }
        if (TempsMasqueBoutons > 0) {
            TempsMasqueBoutons--;
        }
    }
    // il est possible de gérer ici d'autre processus
}
```

Fréquence d'échantillonnage

Lorsqu'une entrée est lue de manière régulière au cours du temps, la scrutation de l'entrée correspond à une fréquence, appelée fréquence d'échantillonnage. La théorie du traitement de signal donne des indications sur la relation qu'il existe entre la fréquence maximale du signal d'entrée (***FreqInMax***) et la fréquence d'échantillonnage minimale (***FreqEchMin***) nécessaire pour restituer une information correcte sur le signal d'entrée :

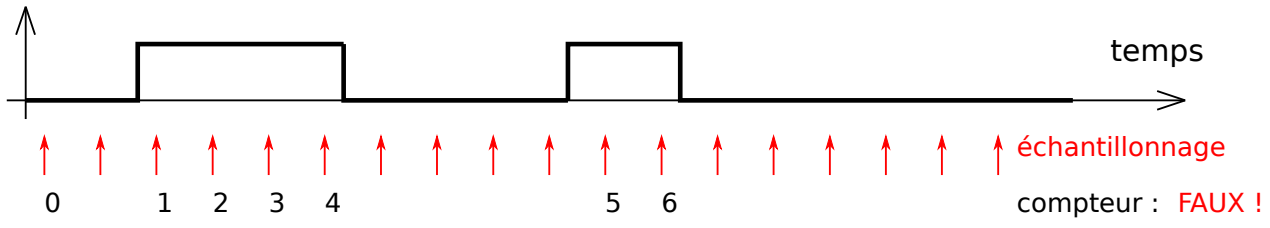
***Freq*$_{EchMin}$ >= 2 . *Freq*$_{InMax}$**

Comptage d'événements

Examinons le programme suivant :

```
while (1) { // boucle principale
    if (PoussoirOn) {
        compteur++;
    }
}
```

Est-ce que ce programme permet de compter le nombre de pression sur le bouton-poussoir ? Un digramme des temps permet de répondre facilement à cette question :



Il ne faut pas compter le nombre de fois que l'entrée prend un état donné, mais chercher à compter les flancs (par exemple les flancs montants).

Le programme suivant permet un tel comptage :

```
#define EntreeOn (digitalRead(Entree))
```

```
int compteur=0;
while (1) { // boucle principale
    while (!EntreeOn) {
      // attente du passage de l'entrée à 1
    }
    compteur++;
    while (EntreeOn) {
      // attente du passage de l'entrée à 0
    }
}
```

Le programme suivant permet aussi un comptage, avec la technique de la boucle principale sans attente, compatible avec le multi-tâche :

```
int ancienneValeur=EntreeOn;
int valeur;
int compteur=0;

while (1) { // boucle principale
    valeur = EntreeOn;
    if (!ancienneValeur && valeur) {
      compteur++;
    }
    ancienneValeur=valeur;
}
```

C'est alors très simple de compter les flancs sur deux entrées indépendantes :

```
int ancienneValeur1=Entree1On;
int valeur1; int compteur=0;
int ancienneValeur2=Entree2On;
int valeur2; int compteur2=0;

while (1) { // boucle principale
    valeur1 = Entree1On;
    if (!ancienneValeur1 && valeur1) {
      compteur1++;
    }
    ancienneValeur1=valeur1;

    valeur2 = EntreeOn;
    if (!ancienneValeur2 && valeur2) {
      compteur2++;
    }
    ancienneValeur2=valeur2;
}
```

Mesure du temps

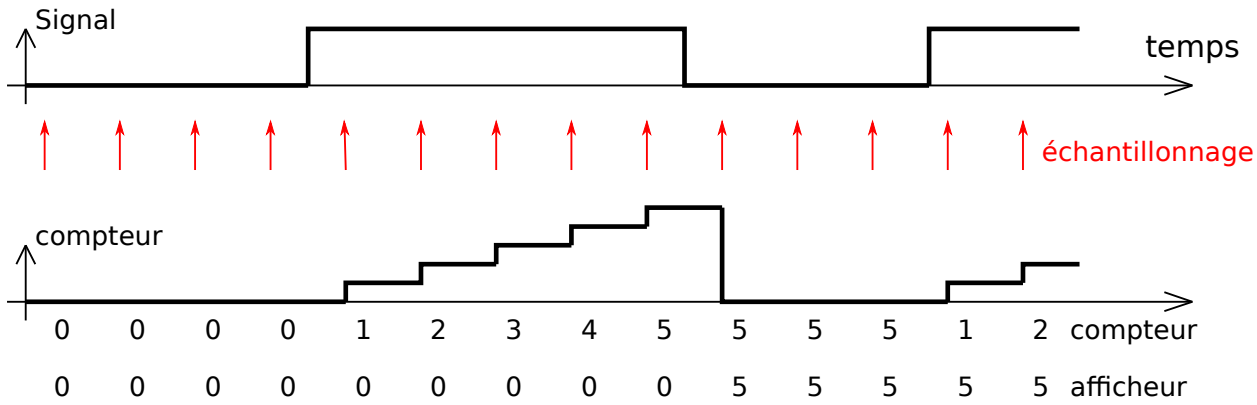
Avec ce que nous avons vu jusqu'ici, il n'est pas très compliqué de mesurer le temps entre deux événements.

Prenons un exemple. Des impulsions arrivent sur une entrée, on souhaite afficher leur

durée. Examinons le programme suivant :

```
int compteur=0;
while (1) { // boucle principale
    Affiche(compteur); // affiche la valeur précédemment mesurée
    while (!EntreeOn) {
      // attente du passage de l'entrée à 1
    }
    compteur=0;
    while (EntreeOn) { // attente du passage à 0
       delay(1); compteur++; // compte le temps
    }
}
```

Voici un diagramme des temps, avec les valeurs successives de la variable `compteur` :



La seconde boucle permet de compter les ms qui s'écoulent pendant que le signal est à 1. La boucle principale dure à nouveau le temps d'un cycle du signal d'entrée.

Voici une technique basée sur une boucle principale de durée constante :

```
int ancienneValeur=EntreeOn;
int valeur;
int compteur=0;

while (1) { // boucle principale
    valeur = EntreeOn;
    if (valeur) { // entrée active ?
        compteur++; // totalise la durée de l'impulsion
    }
    if (ancienneValeur && !valeur) { // flanc descendant ?
      Affiche(compteur); // affiche la valeur
      compteur=0;
    }
    ancienneValeur=valeur;
    delay(1);
}
```

Dans ce programme, le premier des `if` est vrai à chaque échantillonnage d'une valeur active de l'entrée. On reconnaît que le second `if` ne se produit qu'une fois par cycle. Il

détecte le flanc descendant et affiche la durée, qui sera visible jusqu'au flanc descendant suivant.

Machines d'état

Dans la pratique, les applications sont souvent plus complexes que les exemples que nous avons présentés. Ces problèmes peuvent souvent être décrits par des graphes d'état, étudiés dans la première partie de ce cours. Dans un prochains chapitre, nous verront comment écrire des programmes qui réalisent une machine d'état.

2013/09/15 Pierre-Yves ROCHAT pyr@pyr.ch Version du 2013/11/16