LANGAGES INTERPRÉTÉS SPÉCIALISÉS

Pierre-Yves Rochat, EPFL

rév 2016/07/09

Motivation

Animer une enseigne à LED consiste en une suite d'opérations sur les groupes de LED. Animer un afficheur matriciel consiste aussi à envoyer des séquences graphiques. Dans les deux cas, une jolie animation ne se limitera pas à quelques étapes, mais pourra vite devenir longue. Les programmes correspondants vont donc avoir tendance à s'allonger, ce qui va rendre leur lecture fastidieuse et qui risque aussi de remplir rapidement la mémoire du microcontrôleur.

Une technique souvent utilisée consiste à **inventer un langage** pour décrire ce qui se passe sur l'enseigne ou l'afficheur et programmer les animations dans ce langage.

Langage Arduino

Prenons l'exemple très simple. Pour décrire une animation sur une enseigne, deux ordres suffisent pour décrire les actions :

- allumer un groupe de LED avec une certaine intensité,
- attendre un certain temps.

Dans le cas simple de sorties tout-ou-rien, voici les procédures Arduino qui vont suffire :

- *`digitalWrite()`* pour donner un état à une sortie,
- *`delay()`* pour une attente.

En observant la taille d'un petit programme sur Energia et en ajoutant des appels à ces procédures, on constate que :

- *`digitalWrite()`* prend 8 octets en mémoire,
- *`delay()`* prend 10 octets en mémoire.

En prenant par exemple un microcontrôleur MSP430G2213, qui dispose d'une mémoire *flash* de 2 ko (2048 octets), on sera limité à moins de 80 pas de programme, constitués de paires `digitalWrite() - delay()`. En constatant qu'un simple chenillard dans les deux sens sur 8 bits en utilise déjà 16, c'est réellement limitatif !

```
loop() {
  digitalWrite (P2_0, 1); delay (100);
  digitalWrite (P2_1, 1); delay (100);
  digitalWrite (P2_2, 1); delay (100);
  digitalWrite (P2_3, 1); delay (100);
  digitalWrite (P2_4, 1); delay (100);
  digitalWrite (P2_5, 1); delay (100);
  digitalWrite (P2_6, 1); delay (100);
  digitalWrite (P2_7, 1); delay (200);
  digitalWrite (P2_7, 0); delay (100);
  digitalWrite (P2_6, 0); delay (100);
  digitalWrite (P2_5, 0); delay (100);
  digitalWrite (P2_4, 0); delay (100);
  digitalWrite (P2_3, 0); delay (100);
  digitalWrite (P2_2, 0); delay (100);
  digitalWrite (P2_1, 0); delay (100);
  digitalWrite (P2_0, 0); delay (300);
}
```

Bien entendu, les instructions permettant l'accès direct aux registres du microcontrôleur permettent d'économiser la place en mémoire. L'instruction `P1OUT |= (1<<0);` utilise 4 octets. C'est déja mieux ! Mais cherchons une autre solution.

Inventer un langage

Une solution élégante est d'inventer un langage. Notre premier langage aura les deux instructions :

- **Mettre une intensité sur une sortie**. Paramètres : numéro de la sorte et intensité (0 ou 1)
- **Attendre**. Paramètre : durée de l'attente.

Le programme peut alors se présenter sous forme d'un tableau. Nous avons utilisé ici un tableau d'octets. Le programme pour notre chenillard aura alors la forme suivante :

```
uint8_t Animation[] = { // définition d'un tableau d'octets
  Sortie0+On, Attente+10,
  Sortie1+On, Attente+10,
  Sortie2+On, Attente+10,
  Sortie3+On, Attente+10,
```

```
  Sortie4+On, Attente+10,
  Sortie5+On, Attente+10,
  Sortie6+On, Attente+10,
  Sortie7+On, Attente+20,
  Sortie7+Off, Attente+10,
  Sortie6+Off, Attente+10,
  Sortie5+Off, Attente+10,
  Sortie4+Off, Attente+10,
  Sortie3+Off, Attente+10,
  Sortie2+Off, Attente+10,
  Sortie1+Off, Attente+10,
  Sortie0+Off, Attente+30,
  Fin
}
```

Sa taille n'est que de 33 octets. Voici les définitions nécessaires pour que ce tableau se compile correctement :

```
#define On 0b01000000
#define Sortie0 0
#define Sortie1 1
#define Sortie2 2
#define Sortie3 3
#define Sortie4 4
#define Sortie5 5
#define Sortie6 6
#define Sortie7 7

#define Attente 0b10000000
#define Fin 0b1111111
```

LANGAGE BINAIRE

Voici la description binaire de notre langage :

```
// Description des instructions :
// b7 b6 b5 b4 b3 b2 b1 b0  : instructions sur 8 bits
// -----------------------
//  0 i0 s5-s4-s3-s2-s1-s0  : met une intensité sur une sortie
//  1 d6-d5-d4-d3-d2-d1-d0  : attente
// -----------------------
//
// Sorties sur 6 bits (maximum 64 sorties)
// Intensité sur 1 bit (On ou OFF)
// Durée sur 7 bits, exprimée en dixième de seconde (0 à 12.6 secondes)
```

Ceux qui ont déjà programmé en assembleur trouveront une grande similitude avec la description des instructions en binaire !

On voit que des choix ont été faits pour utiliser au mieux les instructions, qui sont des champs de 8 bits. Le bit de poids fort *b7* détermine s'il s'agit d'une instruction pour définir l'intensité ou pour l'attente. Ensuite, les 7 bits restants se répartissent selon l'instruction : une intensité et un numéro de sortie pour l'action sur une sortie, une valeur en dixième de seconde pour l'attente. L'usage de la milliseconde comme unité aurait été trop limitatif, étant donné que seuls 7 bits sont à disposition.

Interpréteur

Il reste à écrire une procédure qui va interpréter notre langage et le traduire en instructions pour un microcontrôleur :

```
void Exec () {
  uint8_t instr = Programme[pc++]; // lit l'instruction
  if (instr==Fin) { // gère la fin du programme
    pc = 0;
  } else {
    if (instr & 0x8000) {  // attente
      AtenteMs(10 * (instr & 0x7F));
    } else {               // set intensité
      if (instr & 0x40) {
        Allume(instr & 0x3F);
      } else {
        Eteint(instr & 0x3F);
      }
    }
  }
}
```

Langages plus complexes

Plusieurs compléments permettent de créer un environnement réellement intéressant pour programmer des enseignes complexes :

- L'ajout de la gestion de l'intensité des LED par BCM (*Binary Coded Modulation*).
- La possibilité d'agir sur des groupes de LED, permettant de simplifier l'écriture des programmes.
- La gestion de plusieurs programmes en parallèle, pour gérer plus facilement différentes parties de l'enseigne.

Exemple de langage pour un afficheur matriciel

Pour piloter des animations sur une petite enseigne de pharmacie constituée d'une matrice monochrome de 16×16 LED, un langage plus complet a été imaginé. Il est basé sur le dessin de droites horizontales et verticales, à partir d'un curseur courant. Pour pouvoir réutiliser des parties de programme, un mécanisme d'appel de sous-routines a été mis en place, avec une instruction *Label* pour indiquer le début de la routine et une instruction *Return* pour en indiquer la fin. Une pile a été implémentée, rendant possibles des appels imbriqués.

Les animations étant très souvent de répétitions de motifs élémentaires, on a ajouté une instruction de répétition, qui préfixe n'importe quelle autre instruction, pour la répéter un certain nombre de fois. Elle est particulièrement utile pour préfixer un appel de routine.

Voici en détail la définition des instructions :

```
#define DrH 0x30 // + dx (sur 4 bits) : droite horizontale, depuis le curseur
#define DrV 0x40 // + dy (sur 4 bits) : droite verticale, depuis le curseur
#define PlusX 0x50 // + dx (sur 4 bits) : avance le curseur en X
#define PlusY 0x60 // + dy (sur 4 bits) : avance le curseur en Y
#define MoinsX 0x70 // + dx (sur 4 bits) : recule le curseur en X
#define MoinsY 0x80 // + dy (sur 4 bits) : recule le curseur en Y
#define Repete 0x90 // + 4 bits : préfixe de répétition pour l'instr. suivante
#define Delai 0xA0 // + 4 bits : Attente, valeur exposant de 2
#define SetAccu 0xB0 // + 4 bits : Charge l'accumulateur (utilisé pour Intens)
#define Label 0xC0 // + 5 bits (32 routines max)
#define Call 0xE0 // + 5 bits
#define Fin 0 // fin du programme
#define Vide 1 // efface l'écran
#define Ret 2 // retour de sous-routine (saut à l'adresse sur la pile)
#define Origine 3 // place le curseur à 0,0
#define ZeroX 4 // met X à zéro
#define Intens 5 // détermine l'intensité, selon la valeur de l'accumulateur
#define Masque 0x9
#define InvMasque 0xA // inverse le masque courant
#define SetDelai 0xB // définit délai utilisé entre l'affichage de chaque point
#define SetDelaiDef 0xC // définit la valeur du délai 0
#define Effet 0xD
#define Libre2 0xE // instructions non utilisées
#define Libre1 0xF
```

A noter qu'on aurait pu utiliser une notation plus sûre : *#define DrH(x) (0x30+((x)&0x0f))*.

Voici un exemple d'animation. Attention, c'est comme l'assembleur : il faut un peu de pratique pour s'y retrouver !

```
Label+Croix7x7, // Affiche une croix 7x7 (1/4 de la surface), curseur courant
PlusX+2,
DrH+2, PlusY+1, MoinsX+2,
DrH+2, PlusY+1, MoinsX+4,
DrH+6, PlusY+1, MoinsX+6,
DrH+6, PlusY+1, MoinsX+6,
DrH+6, PlusY+1, MoinsX+4,
DrH+2, PlusY+1, MoinsX+2,
DrH+2, MoinsY+6, MoinsX+4,
Ret,

Label+Croix7x7plusDel, // Affiche croix 7x7, déplace le curseur en X et attend
Vide, Call+Croix7x7, PlusX+1, Delai+4, Ret,

Label+Croix7x7passeX, // Fait passer une croix X
ZeroX, MoinsX+8, Repete+12, Call+Croix7x7plusDel,
Repete+12, Call+Croix7x7plusDel, Vide, Delai+6, PlusY+2, Ret,
```

La première version de ce programme a permis de placer un bon nombre d'animations graphiques dans un microcontrôleur MSP430G2202, qui ne dispose que de 2 ko de mémoire *flash*.