C5-160551-INFO
Objet Connectés :
Programmation microcontrôleur - TD3

©B. Besserer

année universitaire 2021-2022

1 Entrées / Sorties (E/S ou *I/O*) et framework

1. Parmi les équipements suivants, lesquels sont généralement utilisés somme capteurs
   - [x] Potentiomètre
   - [x] Photo-résistance
   - [ ] Servo-moteur
   - [ ] Diode laser
2. Parmi les équipements suivants, lesquels délivrent généralement une grandeur sous forme d'un signal variable analogique (le plus souvent une tension variable)
   - [x] Potentiomètre
   - [ ] GPS
   - [x] Bouton poussoir
   - [ ] Microphone
3. Le framework Arduino a été développé pour faciliter la programmation pour des nocives en informatique. Au lieu d'écrire une fonction `main()` l'utilisateur doit implémenter le code qui ne doit s'exécuter qu'une seule fois dans une fonction `setup()` et le code récurrent dans une fonction `loop()`.
   Au final, c'est quand même un programme en langage C qui est traduit par le compilateur... lequel ?

A [x]
```
int main()
{
      init();
      setup();
      for (;;)
            loop();
      return 0;
}
```

B [ ]
```
 int main()
{
      init();
      for (;;)
            setup();
            loop();
      return 0;
}
```

C [ ]
```
int main()
{
      init();
      setup();
      loop();
      return 0;
}
```

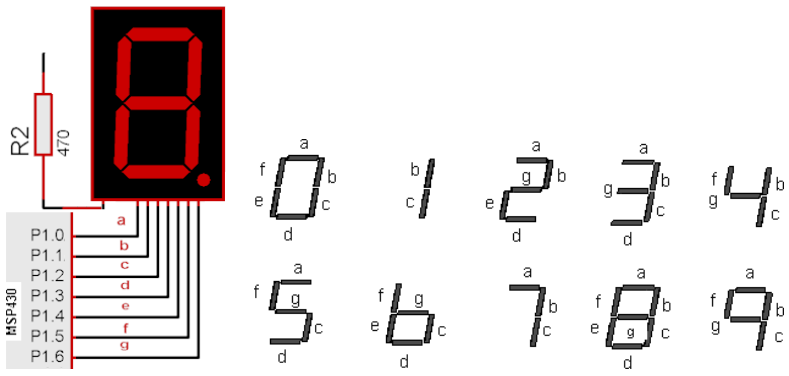
2 Table de correspondance pour affichage 7 segments

Pour les "petits" microcontrôleurs, Les broches d'E/S sont généralement regroupées par paquet de 8 constituant un **port** d'E/S (nommé PORT C, PORT D ou bien P1, P2, etc... selon les constructeurs). Un registre de 8 bits (un octet) permet de configurer individuellement chaque broche, et un autre registre de 8 bits permet d'écrire ou de lire la valeur de chacune des broches. On suppose qu'un afficheur de type 7 segments est **directement** connecté sur un port D d'un microcontrôleur de type ATMEL AVR (par ex. Arduino Uno), voir ci-dessous. Si l'on utilise les commandes du framework pour afficher un chiffre, il faudrait par exemple écrire :

```
// declaration et définitions
const byte LED_A = 1;
const byte LED_B = 2;
const byte LED_C = 3;

setup()
{
   pinMode(LED_A,OUTPUT);
   pinMode(LED_B,OUTPUT);
   pinMode(LED_C,OUTPUT ;
   // etc..
}

loop()
{
   // afficher le nombre 7
   digitalWrite(LED_A,1);
   digitalWrite(LED_B,1);
   digitalWrite(LED_C,1);
}
```



C'est un peu lourd, même si l'on écrit une fonction spécifique pour l'affichage. Si l'on peut s'arranger pour que les segments a, b, ..., g soient connecté sur des bits **consécutifs** d'un même port, on peux essayer d'écrire en une fois, dans le registre qui contrôle la valeur de sortie des broches de ce port, un mot binaire qui provoque l'allumage des segments souhaités. Dans l'exemple, le bit de poids faible (bit 0) de l'octet qui sera écrit dans le port D contrôle l'allumage de la LED (segment) a, le bit 1 l'allumage du segment b, etc...
Donc, par un grand hasard, `PORTD = 0x07;` affiche le chiffre 7... mais `PORTD = 0x06;` affiche le chiffre 1 !!!
Et comme les ports sont accessibles en lecture et en écriture, le code peut contenir des lignes de type : `PORTD = PORTD & 0x7F;`
Le principe se nomme *Direct IO Port Manipulation*, et ce type d'opérations se nomme *Read-Modify-Write Operation* ou encore *Load-Modify-Store*)

1. Donnez une table de correspondance donnant le mot (au format hexadecimal) à écrire dans le port D afin de former le chiffre souhaité sur l'afficheur 7 segments (On considère l'afficheur ayant une cathode commune, donc *active high* : l'écriture d'un 1 sur le bit correspondant allume la LED).
   La table est à tracer comme une table de vérité :
   Sous Moodle entrez la succession de valeurs

	g	f	e	d	c	b	a	hex value to Port D
0								

2. En langage 'C', on souhaite déclarer cette table de correspondance telle qu'à l'indice N de cette table, on trouve la valeur hexadécimale correspondant à l'affichage de la valeur N sur l'afficheur 7 segments. Comment déclarer et initialiser cette table ? (une ligne de code, en précisant si s'agit d'une déclaration locale à une fonction ou globale)

3. Ecrivez une fonction `void DisplayDigit(int value)` qui :
   - Effectue un test pour savoir si `value` peut être affichée (*in range*)
   - Puis, si la valeur est affichable, la ligne de code permettant d'ecrire dans le port la bonne combinaison, allumant les segments permettant d'afficher `value`. Pas de dépot sur Moodle, correction en séance.

3 Programmation : Faire deux choses en même temps

La fonction delay() est bien utile pour temporiser mais celle-ci est bloquante. Comment procéder si l'on veut faire plusieurs choses cadencées différemment. On peut structurer un cadencement avec la fonction millis() et l'utilisation d'une structure de type switch ... case dont chaque cas correspond à un état du système à piloter.
Syntax : time = millis()
Description : Returns the number of milliseconds since the Arduino board began running the current program. This number will overflow (go back to zero), after approximately 50 days.

On souhaite faire clignoter une LED selon la cadence : ON (10ms), OFF (200ms), ON (10ms) OFF (1000ms) et il faut durant cette séquence de clignotement appeler le plus souvent possible une fonction **doSomeTask()** dont la durée d'exécution n'est pas prévisible, mais se situe entre 10ms et 100ms.

1. Pourquoi 50 jours (50 days) ?
2. Examinez le code et écrivez la ligne déclarant et initialisant le tableau `d`.
3. A quel(s) endroit(s) du code peut-on appeler la fonction `doSomeTask()` (Précisez au moins deux numéros de ligne)
4. Avec ces appels de la fonction, les durées de chaque phase sont-elles toujours :
   ☐ égales aux valeurs ☐ sup. ou égales aux valeurs souhaitées ☐ inf. ou égales au valeurs

```
01    #define ON1   0
02    #define OFF1  1             // etc... pour identifier les états du cycle
03    void setup()
04    {
05      phase=ON1;
06      millisMem=millis();
07    }
08    void loop()
09    {
10      switch(phase)
11      {
12        case ON1
13                  if(millis()-millisMem < d[0])
14                  { digitalWrite(LEDPin, HIGH); }
15                  else
16                  { phase=OFF1;}
17                  break;
18        case OFF1:
19                  if(millis()-millisMem < d[0]+d[1])
20                  { digitalWrite(LEDPin, LOW); }
21                  else
22                  { phase=ON2;}
23                  break;
24        case ON2:
25                  if(millis()-millisMem < d[0]+d[1]+d[2])
26                  { digitalWrite(LEDPin, HIGH); }
27                  else
28                  { phase=OFF2;}
29                  break;
30        case OFF2:
31                  if(millis()-millisMem < d[0]+d[1]+d[2]+d[3])
32                  { digitalWrite(LEDPin, LOW); }
33                  else
34                  { phase=ON1; millisMem=millis(); }
35                  break;
36      }
37    }
```

4 Préparation au TP : Affichage 7 segments avec l'ESP32

Pas de réponses à donner sur moodle. Réaliser ces exercices vous donnera un avantage considérable lors des travaux pratiques.

L'ESP32 est basé sur une ALU 32bits dual-core. Les concepteurs de l'ESP32 ont regroupés 32 entrées/sorties dont le contrôle est regroupés dans un seul port 32 bits. En raisons de l'architecture du processeur, ces entrées/sorties ne sont pas toutes identiques, pas toutes accessibles et ont des fonctionnalités différentes (voir lien annexe B).
Pour piloter un afficheur 7 segments de la façon décrite **??** il faut au moins 7 broches de type GPIO consécutives. On propose d'utiliser les GPIO numérotés 12..19, reproduisant l'utilisation d'un port 8 bits (schéma annexe A).

1. L'ESP32 est multi-coeur. Cette architecture interdit les opérations non atomiques
   voir `https://fr.wikipedia.org/wiki/Atomicité_(informatique)`.
   Par exemple, la ligne de code : `GPIO_OUT_W1TS_REG = GPIO_OUT_W1TS_REG | 0x7F;`
   n'est plus atomique, car il s'agit d'une opération en 3 temps (*Read-Modify-Write Operation*) et après avoir lu le registre, mis le bit 12 à 1, il n'est pas exclu qu'un thread 2 qui tourne sur le second coeur modifie aussi le registre, modification qui sera écrasée lors de la dernière étape (écriture) de l'instruction du thread 1.

   Donc, pour le processeur ESP32, l'accès direct vers un registre utilisera un mode bien spécifique.
   Le framework Arduino/Wiring propose plusieurs macros définis dans le fichier "soc.h" (REG_READ, REG_WRITE, REG_SET_FIELD, etc...). REG_WRITE permet d'écrire dans un registre (_r le registre, _v la valeur) :

   ```
   #define REG_WRITE(_r, _v) ({            (*(volatile uint32_t *)(_r)) = (_v);                })
   ```

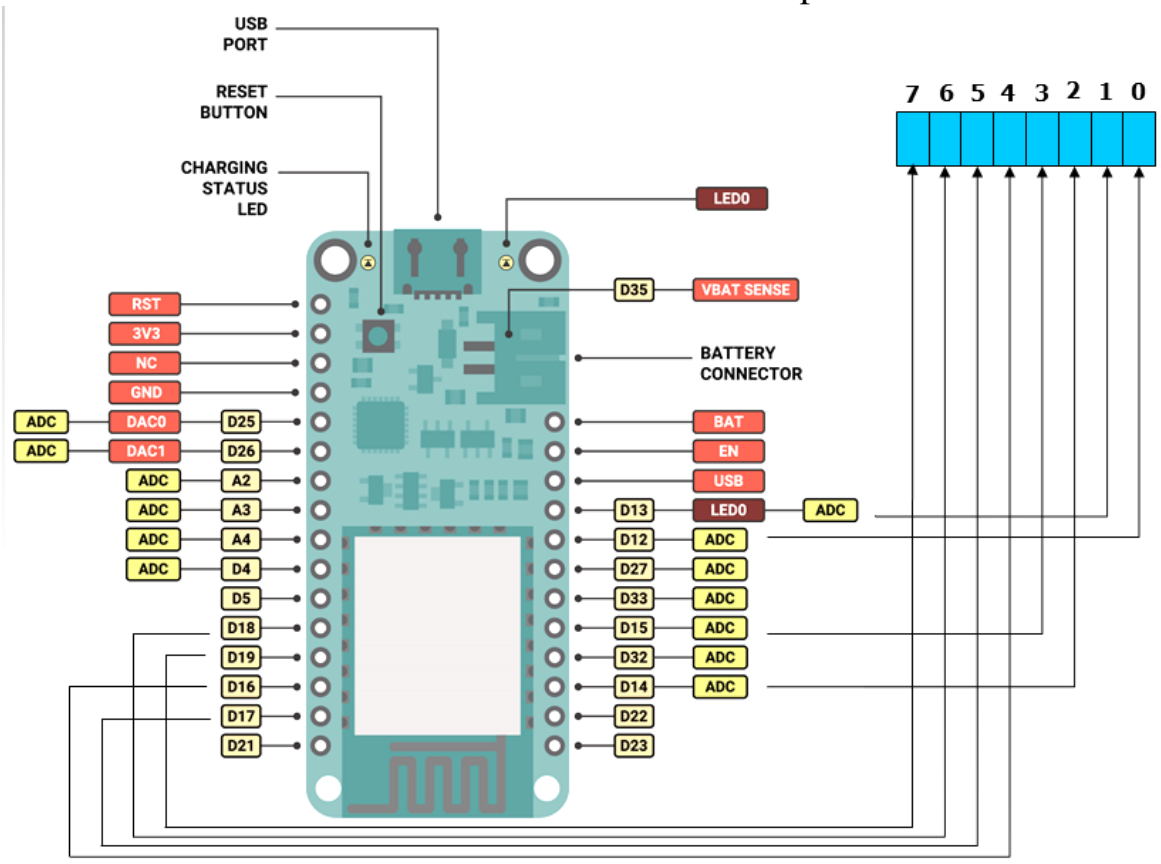
   et l'ESP32 est doté de deux registres spéciaux garantissant des opérations atomiques, l'un permettant la mise à 1 des ports, l'autre la mise à 0 des ports (une différence dans les noms : S pour Set et C pour Clear)
   `GPIO_OUT_W1TS_REG` : Si l'on écrit un 1 sur le BITx de ce registre, la sortie BITx passe à 1 (les autres sont inchangés)
   `GPIO_OUT_W1TC_REG` : Si l'on écrit un 1 sur le BITx de ce registre, la sortie BITx passe à 0 (les autres sont inchangés)

Quel sont les bits qui sont mis à 1 avec l'instruction `REG_WRITE ( GPIO_OUT_W1TS_REG , 0x03C0 );` ?

2. Peut-on réutiliser la table déclarée au 2 question 2 ? ☐ Oui ☐ Non
3. Écrivez la fonction `void DisplayDigit(int value)` pour qu'elle fonctionne avec l'ESP32
4. Que pensez vous de l'utilisation du mode *Direct Register Manipulation* plutôt que d'utiliser digitalWrite() pour chaque segment de l'afficheur 7 segments ?

A Platine ESP32 utilisée en TP - Adafruit Huzzah 32

Ici avec l'utilisation des broches 12 .. 19 comme un port 8 bits.



B Informations sur les GPIO utilisables de l'ESP32

`https://randomnerdtutorials.com/esp32-pinout-reference-gpios/`