LISV Laboratoire d'ingénierie des systèmes de Versailles
UVSQ université PARIS-SACLAY

Cartes à puce
TP 2 et TP 3 - Porte-monnaie électronique

Halim Djerroud

révision 0.1

Sources : https://github.com/hdd-robot/tps_carte_a_puce.git

Préambule

Avant tous les TP, il ne faut pas oublier d'inclure les bonnes bibliothèques : celle relative aux types de données que nous utilisons, celle permettant la réception ou l'envoie de données ou celles concernant la gestion des différents espaces mémoires de la carte :

```
#include "/usr/lib/avr/include/stdint.h"
#include "/usr/lib/gcc/avr/4.7.2/include/stddef.h"
#include "/usr/lib/avr/include/inttypes.h"
// Pour utiliser les types uint8_t, uint16_t, etc.

#include <avr/io.h>
// io.s contient les fonctions sendbytet0() et recbytet0() codées en assembleur
#include <avr/eeprom.h> // Pour lire/écrire dans l'EEPROM
#include <avr/pgmspace.h> // Pour lire/écrire dans la ROM
```

Pour manipuler une variable dans la RAM, rien de particulier à faire, il suffit de déclarer et affecter la variable normalement. En revanche, dans l'EEPROM ou dans la Mémoire Programme, les choses ne sont pas aussi simple. Pour déclarer une variable dans l'EEPROM, il faut ajouter EEMEM à la déclaration et il faut obligatoirement initialiser la variable. Idem pour déclarer une variable dans la Mémoire Programme en remplaçant EEMEM par PROGMEM. Dans ce dernier cas, la variable ne pourra être modifiée : ce sera une constante. Pour accéder à une variable enregistrée dans l'EEPROM, il faut utiliser des fonctions particulières contenues dans la bibliothèque "avr/eeprom". De la même façon, la bibliothèque "avr/pgmspace" contient des fonctions pour accéder à une variable enregistrée dans la Mémoire Programme sauf qu'il n'y aura pas de fonctions pour écrire cette fois-ci.

```
// Déclarations
uint8_t a; // Dans la RAM
uint8_t b EEMEM = 0; // Dans l'EEPROM, variable initialisée à 0
uint8_t c PROGMEM = 5; // Dans la Mémoire Programme : constante égale à 5

// Transférer les données de l'EEPROM vers la RAM
a = eeprom_read_byte(&b); // Prototype : uint8_t eeprom_read_byte(uint8_t *ptr)

// Transférer les données de la RAM vers l'EEPROM
eeprom_write_byte(&b, a); // Prototype : void eeprom_read_byte(uint8_t *ptr, uint8_t val)

// Transférer les données de la Mémoire Programme vers la RAM
a = pgm_read_byte(&c); // Prototype : uint8_t pgm_read_byte(uint8_t *ptr)
```

Remarques :

— le type `char` est différent de `uint8_t`. Mais il y a quand même une correspondance avec `unsigned char`.
— Faire un `#define c 5` n'équivaut pas à utiliser `PROGMEM` car cela ne définit pas une vraie constante mais seulement un raccourci.

— Lors de la programmation de la carte, il faut penser à inclure le fichier `.eep.hex` ou `.eep` dans le champ `EEPROM` du programmateur `AVRDUDE` pour bien initialiser l'`EEPROM`.
— Dans les différentes fonctions, on peut remplacer *"byte"* par *"word"*, *"dword"* ou *"block"*. Il faut alors changer les types en conséquence : "word" définit un `uint16_t` (2 octets), *"dword"* définit un `uint32_t` (4 octets) et *"block"* définit une chaine de `uint8_t`. Les fonctions avec *"block"* sont légèrement différentes, voici leurs prototypes :

```
void eeprom_write_block (const void * src, void * dst, size_t n);
void eeprom_read_block (void * dst, const void * src, size_t n);
void pgm_read_block (void * dst, const void * src, size_t n);
```

1 Présentation du TP et indications

Le but de la séance est de se familiariser avec l'environnement de développement et d'apprendre à accéder aux mémoires non volatiles : on va vouloir écrire dans la mémoire programme, et lire et écrire dans la mémoire EEPROM via une application de porte-monnaie électronique.

Rappelons qu'une carte à puce possède 3 types de mémoires :
— la **mémoire RAM** qui est volatile. Elle correspond à la mémoire de travail. C'est dans cette mémoire que les variables déclarées sont allouées.
— la **mémoire programme** qui est non volatile. Lorsque l'on compile un programme, l'exécutable se trouve dans cette mémoire qui est accessible par le programmateur.
— la **mémoire EEPROM** qui est non volatile.

1.1 Les fonctions d'entrée/sortie

Le fichier `io.s` est écrit en langage assembleur. Il contient la définition des deux fonctions d'entrée/sortie qu'on aura besoin pour pouvoir entrer et sortir des informations dans/de la carte :
— `recbytet0()` : lecture d'un octet sur l'interface de la carte
— `sendbytet0()` : « sort » un octet de la carte

1.2 Consignes :

À partir du fichier `hello.c` écrire les commandes suivantes :
— Classe `0x80`
  — Instruction `0x00` :
    — Lire version : 82 00 00 00 04 → "1.00" 90 00
    — Fonction : `version();`
    — Description : Donne la version de la carte, cette dernière est directement inscrite dans le code source lors du développement de la carte :

```
#define size_ver 4
const char ver_str[size_ver] PROGMEM = "1.00";
```

    — Solution :

```
/**
  * La commande sortante associée est 82 00 00 00 04 .
  * Elle doit renvoyer 1.00 90 00.
  */

#define size_ver 4
const char ver_str[size_ver] PROGMEM = "1.00";
// émission de la version
// t est la taille de la chaîne sv
void version(){
 int i;
 // vérification de la taille
 if (p3!=size_ver){
     sw1=0x6c; // taille incorrecte
```

1.2 Consignes :

```
        sw2=size_ver; // taille attendue
        return;
    }
    sendbytet0(ins); // acquittement
    // émission des données
    for(i=0;i<p3;i++){
        sendbytet0(pgm_read_byte(ver_str+i));
       }
       sw1=0x90;
   }
```

— Instruction `0x01` :
  — Introduire personnalisation : 82 01 00 00 05 "Jean" → 90 00
  — Fonction : `intro_perso();`
  — Description : Fonction de personnalisation, les données doivent être écrites dans l'eeprome. Attention `P3` ne devra pas excéder une certaine valeur.

```
#define MAX_PERSO 32
uint8_t ee_taille_perso EEMEM=0;
unsigned char ee_perso[MAX_PERSO] EEMEM;
```

  — Solution :

```
void intro_perso(){
    int i;
 unsigned char data[MAX_PERSO];
    // vérification de la taille
    if (p3>MAX_PERSO){
     sw1=0x6c; // P3 incorrect
        sw2=MAX_PERSO; // sw2 contient l'information de la taille correcte
 return;
    }
 sendbytet0(ins); // acquitement
 for(i=0;i<p3;i++){  // boucle d'envoi du message
     data[i]=recbytet0();
 }
 eeprom_write_block(data,ee_perso,p3);
 eeprom_write_byte(&ee_taille_perso,p3);
 sw1=0x90;
}
```

— Instruction `0x02` :
  — Lire personnalisation : 82 02 00 00 05 → "Jean" 90 00
  — Fonction : `lire_perso();`
  — Description : Si P3 est inconnu, alors mettre `P3` à `00`. On recevra alors une erreur avec la taille attendue.
  — Solution :

```
void lire_perso(){
 int i;
 uint8_t taille;
 taille=eeprom_read_byte(&ee_taille_perso);
 if (p3!=taille){
 sw1=0x6c;
 sw2=taille;
 return;
 }
 sendbytet0(ins);
 for (i=0;i<p3;i++){
 sendbytet0(eeprom_read_byte(data+i));
 }
```

1.2 Consignes :

```
    sw1=0x90;
}
```

— Instruction `0x03` :
  — Lire solde : 82 03 00 00 02 → xx xx 90 00
  — Fonction : `lire_solde();`
  — Description : `0€` au départ. Le solde est un `uint16_t` sauvegardé dans l'EEPROM. Il est donc compris entre 0 et 65535 centimes d'euro
  — Solution : Le solde est un entier 16 bits (= 2 octets). Par convention, un entier 16 bits se traduit par 4 chiffres hexadécimaux qui vont de 0000 à FFFF. Il existe deux ordres de transmission des octets :
    — big endian : on transmet d'abord l'octet de poids fort, puis celui de poids faible
    — little endian : on transmet d'abord l'octet de poids faible, puis celui de poids fort

    Les machines peuvent avoir les deux conventions mais cela dépend du modèle du processeur. Dans ce TP, on utilise la convention big endian. Si le solde est à 1€ (100 en hexadécimal donne 0x0064), on devra recevoir 00 64 90 00.

```
uint16_t solde EEMEM = 0;
void LectureSolde(){
    if(p3 != 2){
        sw1 = 0x6c ;
        sw2 = 2;
        return ;
    }
    sendbytet0(ins) ;
    uint16_t mot = eeprom_read_word(&solde) ;
    sendbytet0(mot >> 8) ; //on envoie d'abord le bit de poids fort
    sendbytet0(mot) ; //on envoie le bit de poids faible
    sw1 = 0x90 ;
}
```

— Instruction `0x04` :
  — Créditer : 82 04 00 00 02 xx xx → 90 00 ou 61 00 (capacité maximale de rechargement dépassée)
  — Fonction : `credit();`
  — Description : instruction permettant de créditer la carte lors d'un rechargement. `61 00` si le crédit dépasse les capacités
  — Solution :

```
void credit(){
    if(p3 != 2){
        sw1 = 0x6c ;
        sw2 = 2;
        return ;
    }
    sendbytet0(ins) ;
    uint16_t ajout = ((uint16_t)recbytet0() << 8) + (uint16_t)recbytet0();
    uint16_t solde_mot = eeprom_read_word(&solde) ;
    uint16_t montant = ajout + solde_mot ;
    if(montant < ajout){ //il y a eu un debordement
        sw1 = 0x61 ;
        return ;
    }
    eeprom_write_word(&solde, montant) ;
    sw1 = 0x90 ;
}
```

— Instruction `0x05` :
  — Débiter : 82 05 00 00 02 xx xx → 90 00 ou 61 00 (solde est insuffisant)
  — Fonction : `debit();`
  — Description : instruction permettant de débiter la carte lors d'un paiement. 61 00 si le solde est

1.2 Consignes :

insuffisant

— Solution :

```
void Depenser(){
   if(p3 != 2){
       sw1 = 0x6c ;
       sw2 = 2;
       return ;
   }
   sendbytet0(ins) ;
   uint16_t retrait = ((uint16_t)recbytet0()<<8) + (uint16_t)recbytet0() ;
   uint16_t solde_mot = eeprom_read_word(&solde) ;
   if(solde_mot < retrait){
       sw1 = 0x61 ; // solde insuffisant
       return ;
   }
   uint16_t montant = solde_mot - retrait ;
   eeprom_write_word(&solde, montant) ;
   sw1 = 0x90 ;
}
```

Remarques importante

— Pas d'opérations trop lentes durant la réception des octets. Par exemple, il faut dissocier la réception des données et l'écriture de celles-ci dans l'EEPROM.

— Le solde est enregistré en uint16_t. Il faut donc convertir `2` × `uint8_t` en `uint16_t`. Pour cela, il faut faire savoir si on utilise la convention *Little endian* ou *Big endian* :

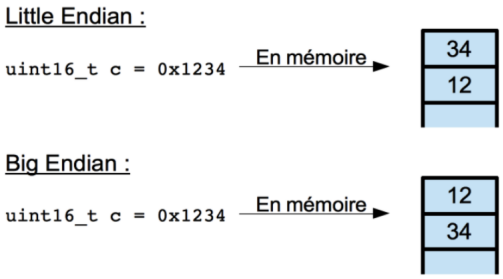

FIGURE 1 – Little Endian et Big Endian

2 l'anti-arrachement

Le but du TP est de gérer l'anti-arrachement de la carte pendant une opération, c'est-à-dire qu'on aimerait maintenir la carte dans un état cohérent même si on l'arrache pendant une opération.

On peut avoir des incohérences si lors d'une exécution de la carte, celle-ci est arrachée. En effet, imaginons que nous arrachions la carte du **TP Porte_monnaie** pendant que celle-ci effectue un "crédit" ou un "débit". Si cette arrachage se fait pendant l'écriture dans l'EEPROM, alors, le solde peut être erroné... De plus, lors de la personnalisation de la carte, on introduit un nom et la taille de ce nom dans l'EEPROM. Si la carte est arrachée pendant cette opération, peut être que l'on perdra la cohérence entre la vraie taille du nom et la taille sauvegardée. On peut alors avoir le problème du buffer overflow, c'est-à-dire que l'on dépasse la capacité de la mémoire donnant ainsi accès à des zones pouvant contenir des données privées.

Il faut donc rendre certaines opérations atomiques, indivisible : soit on fait TOUT ou alors RIEN.

Nous allons utiliser une mémoire tampon. Avant d'écrire définitivement les données, on passera par cette mémoire. L'opération se déroulera en 2 temps avec une fonction pour chacun de ces temps. On va travailler avec un automate à 2 états.

- État VIDE → Pas de données dans la mémoire tampon
- État DATA → Il y a des données dans la mémoire tampon

- 1er temps : Engagement → `engage(taille, addr_dest, données)`
  - (1) Met l'état à valeur `VIDE`
  - (2) Place les données à écrire dans la mémoire tampon
  - (3) Met l'état à valeur `DATA`
- 2eme temps : Validation → `valide()()`
  - (4) Si l'état est `DATA`, alors placer les données dans la mémoire tampon vers l'`EEPROM`
  - (5) Mettre l'état à `VIDE`

Au lancement (au Reset), appeler la fonction `valide()`. Ainsi, si jamais la carte à précédemment été arrachée entre un appel à `engage()` et un appel à `valide()`, l'opération se terminera quand même.

En cas d'arrachement à l'étape :

- (1) On a une chance sur 256 d'écrire DATA au lieu de VIDE : problème de mémoire résiduelle
- (2) Pas d'engagement dans ce cas. On écrira donc rien
- (3) L'état est à VIDE au lieu de DATA. Il n'y a donc pas d'engagement
- (4) La validation n'est pas terminée mais l'état est à DATA donc la validation sera faite au prochain Reset
- (5) Un problème survient si on écrit DATA au lieu de VIDE. On refera alors la validation

À propos de la mémoire tampon : La mémoire tampon doit évidement se situer dans l'EEPROM. En effet, il faut pouvoir mémoriser des choses malgré un arrachement de carte donc il faut mémoriser ces choses dans une mémoire non volatile.

La mémoire tampon doit contenir :

- L'état (1 octet)
- La taille des données (1 octet)
- L'adresse de destination dans l'EEPROM (2 octets)
- Les données (taille maximale compatible avec la taille des données usuelles)

Remarque : Nous travaillerons avec mémoire tampon de 128 octets, soit 1/4 de la mémoire EEPROM disponible. Il ne faut pas choisir cette taille trop grande car cela prendrait trop de place dans l'EEPROM mais il ne faut pas non plus la choisir trop petite car on perdrait la tonicité du programme.

2.1 Consignes : Porte monnaie avec transaction

Modifier le TP précédent en prenant en compte d'éventuels arrachements de la carte. Il faudra lier la "taille" et le "nom" en mettant ces deux variables dans une même structure.