Programmation de cartes à puces
Rapport sur le projet porte-monnaie électronique

Michel Nguyen
Etudiant en première année de master de mathématiques

Paris VIII - Enseignant : Loïc Demange

Avril 2021

1 Description du projet

Le XXIe siècle marque l'avènement des cartes à puces. En effet, elles sont de plus en plus présentes dans notre quotidien que ce soit dans le domaine de la santé par la carte vitale, dans le domaine du paiement par la carte bleue, mais aussi dans le domaine du transport par la carte Navigo. Autrement dit, les cartes à puces sont des outils d'identification ou monétique. La technologie des cartes à puces sont en pleine évolution où , récemment, les cartes à puces sans contact font leur apparition, ce qui changent les usages du quotidien. Les enjeux autour des cartes à puces sont multiples. Elles doivent contenir des données infalsifiables dans des composantes miniaturisées et répondre à des normes de sécurité strictes.

Dans le cadre du cours "Programmation de cartes à puces", l'objectif du projet est de comprendre le fonctionnement interne d'une carte à puce dans le cas d'un porte-monnaie électronique. Concrètement, il s'agit de comprendre la communication entre les commandes envoyées par un ordinateur et une carte à puce, et implémenter en langage C les fonctions `intro_data()`, `sortir_data()`, `debiter()`, `crediter()`.

2 Spécifications

2.1 Matériel et envoie des commandes

Dans un premier temps, nous allons détailler le matériel utilisé et toutes les étapes pour envoyer des commandes à la carte à puce. Nous avons le code source contenu dans "hello.c" possédant les fonctions qui réalisent les tâches ainsi qu'un script "hello.script" pour envoyer des commandes via le terminal. De plus, nous possédons un boîtier et un lecteur de cartes. Pour envoyer des commandes à la carte, nous devons

- mettre la carte dans le boîtier puis réaliser la commande "make" et "make progcarte" dans le répertoire correspond au fichier "hello.c" afin de pouvoir utiliser les fonctions implémentées ;
- retirer la carte du boîtier et l'insérer dans le lecteur puis exécuter la commande "scat hello.script" pour envoyer des commandes.

2.2 Donnée et status word

Dans un second temps, nous allons définir la donnée de la carte et les status word. Une donnée à envoyer à la carte se décompose en six variables globales : CLA, INS, P1, P2, P3 et DATA. La première indique la classe. Le deuxième indique la fonction à exécuter. P1, P2 et P3 représentent la taille des données à envoyer. DATA représente les octets. L'écriture est en base hexadécimal et nous considérons des octets. Par exemple, nous avons la donnée suivante

CLA	INS	P1	P2	P3	DATA
80	01	00	00	02	a b

Ici, la classe est 0x80 ; on exécute la fonction 1 située dans l'instruction ; il y a 2 octets à envoyer ; ces deux octets sont a et b.

Ensuite, il existe la notion d'erreur qui est notamment importante en cas d'erreur de taille. Nous définissons les status word SW1 et SW2. Dans le cas d'une erreur de taille, SW1 indique un code erreur spécifique à l'erreur de taille et SW2 indique la bonne taille. Par exemple, supposons que j'envoie $P3 = 02$ et $DATA = a$ b c, alors il y a un problème car il y a trois octets alors que P3 indique que deux octets. C'est pourquoi nous avons

```
1          if (p3!=2)       # si p3 ne possède pas la bonne taille
2          {
3              sw1=0x6c;    # code erreur qui indique une mauvaise taille
4              sw2=2;       # sw2 retourne la taille attendue
```

```
5                 return;
6             }
7
```

Il est à noter que chaque erreur possède son propre code erreur et qu'il existe le code $0x90$ qui indique une bonne exécution de la fonction.

2.3 Les fonctions `intro_data()`, `sortir_data()`, `debiter()`, `crediter()`

Dans un troisième temps, nous nous intéresserons aux instructions principaux du `main()` et les quatre fonctions étudiées au cours du projet : introduction et sortie des données, débiter et créditer. Ces deux dernières sont absolument essentiels pour un porte-monnaie électronique. Nous considérons le solde `ee_solde` de type EEPROM (nous définirons son rôle dans la partie 3.2) codé sur deux octets et les fonctions `sendbytet0()` et `recbytet0()` pour envoyer et recevoir des octets.

Tout d'abord, dans le `main()`, on attribue les valeurs contenues dans le script aux cinq variables globales CLA, INS, P1, P2 et P3 ainsi que SW2 telles que

```
1  int main()
2  [...]
3  # lecture de l'entête
4      cla=recbytet0();
5      ins=recbytet0();
6      p1=recbytet0();
7      p2=recbytet0();
8      p3=recbytet0();
9      sw2=0;
```

Ensuite, suivant la valeur de INS, on exécute la fonction associée. Par exemple, supposons que j'attribue la valeur 1 au fonction `intro_data()`, et si la commande dans le script contient $INS = 01$, alors j'exécute `intro_data()`. Les deux premières fonctions qu'on utilise sont les suivantes

```
1  # Je souhaite introduire des octets dans ma variable EEPROM
2
3  void intro_data()
4  {
5      int i;
6
7      # vérification de la taille
8      if (p3>MAXI)     # MAXI est la taille maximale
9      {
10         sw1=0x6c;    # code erreur correspondant à l'erreur de taille
11         sw2=MAXI;    # sw2 contient l'information de la taille correcte
12         return;
13     }
14
15     sendbytet0(ins);     # acquitement
16
17     uint16_t data[p3];  # déclaration du tableau data à p3 cases
18
19     for(i=0;i<p3;i++)
20     {
21             data[i]=recbytet0();      # le tableau data reçoit les octets
22             eeprom_write_block(data,&ee_solde,p3); # écriture des octets dans le solde via la
       fonction eeprom_write_block
23     }
24
25     taille=p3;  # la variable globale taille est de type uint16_t
```

```
26      sw1=0x90;   # code qui valide une bonne exécution de la fonction
27  }
```

```
1   # Je souhaite sortir ou lire les octets de ma variable EEPROM
2
3   void sortir_data()
4   {
5       uint16_t data[p3];  # déclaration du tableau data à p3 cases
6       eeprom_read_block(data,&ee_solde,p3);   # data reçoit les valeurs de ee_solde
7
8       int i;
9
10      # vérification de la taille
11      if (p3!=taille)
12          {
13              sw1=0x6c;   # code erreur indiquant une taille incorrecte
14              sw2=taille; # sw2 prend la taille attendue
15              return;
16          }
17
18      sendbytet0(ins);    # acquittement
19
20      # émission des données
21      for(i=0;i<p3;i++)
22              sendbytet0(data[i]);
23      sw1=0x90; # code qui valide une bonne exécution de la fonction
24  }
```

Nous remarquons que la vérification de taille est toujours vérifiée. De plus, lorsqu'on manipule une variable EEPROM, il faut toujours créer une variable intermédiaire pour manipuler les octets car il est impossible de modifier directement à partir de la variable EEPROM. Aussi, il faut manipuler des fonctions spécifiques à EEPROM.

Maintenant, nous nous intéressons aux deux dernières fonctions

```
1   # Je souhaite ajouter de l'argent dans ma variable EEPROM
2
3   void crediter(){
4
5       # vérification de la taille
6       if(p3!=taille)
7       {
8         sw1=0x6c;
9         sw2=taille;
10        return;
11      }
12
13      # récupération des deux octets
14      uint8_t x1=recbytet0();
15      uint8_t x2=recbytet0();
16
17      # déclaration du solde
18      uint16_t solde;
19
20      # solde reçoit les valeurs de ee_solde
21      eeprom_read_block(&solde,&ee_solde,2);
22
23      # addition des deux octets par méthode de décalage
24      uint16_t somme=(x1<<8)+x2;
```

```
25
26      if(solde+somme>=solde)   # test de vérification , plus de détails dans la partie 3.4
27      {
28           solde=solde+somme;  # additioner solde et somme
29           eeprom_write_block(&solde,&ee_solde,2); # mettre à jour le solde
30           sw1=0x90;    # code qui valide une bonne exécution de la fonction
31       }
32       else
33         sw1=0x6c; # code erreur
34  }
```

```
1   # Je souhaite retirer de l'argent dans ma variable EEPROM
2
3   void debiter(){
4
5       # premières instructions identiques à crediter()
6
7       if(p3!=taille)
8       {
9         sw1=0x6c;
10        sw2=taille;
11        return;
12      }
13
14      uint8_t x1=recbytet0();
15      uint8_t x2=recbytet0();
16
17      uint16_t solde;
18
19      eeprom_read_block(&solde,&ee_solde,2);
20
21      uint16_t somme=(x1<<8)+x2;
22
23      # faire la soustraction
24      if(solde-somme<=solde)  # test de vérification
25      {
26      solde = solde-somme; //soustraire somme à solde
27      eeprom_write_block(&solde,&ee_solde,2); # mettre à jour le solde
28      sw1=0x90;
29      }
30     else
31      sw1=0x6c;
32  }
```

Comme dans les deux premières fonctions, nous réalisons une vérification de la taille. Ensuite, nous récupérons les octets pour, soit les additionner, soit les soustraire au solde. Dans les parties **3.3** et **3.4**, nous détaillons les méthodes pour réaliser ces opérations.

3 Problématiques

3.1 Le problème de l'*endianness*

Il est important d'aborder le sujet de la lecture des octets par l'ordinateur. En effet, il existe une lecture de gauche à droite nommé *big endian* et une lecture de droite à gauche nommé *little endian*. Par défaut, l'ordinateur utilise le *little endian*. Ce qui est à comprendre, c'est qu'il ne faut pas appliquer les deux lectures en même temps. En effet, cela créera une confusion pour l'ordinateur et faussera les opérations. Donc, il faut avoir à l'esprit qu'il faut utiliser soit l'un, soit l'autre lors de l'implémentation des fonctions `debiter()`

et `crediter()`. Dans nos fonctions, le *big endian* est appliqué.

3.2 Utilisation du mémoire EEPROM

La carte à puce doit répondre à un besoin évident : que la mémoire ne s'efface lorsque la carte à puce n'est plus alimentée en électricité. Autrement dit, la mémoire doit être non volatile. Pour répondre à ce besoin, il existe la mémoire EEPROM qui est une mémoire non volatile. C'est pourquoi le solde est une variable de type EEPROM. La librairie EEPROM propose une bibliothèque de fonctions où nous utilisons `eeprom_write_block(adresse_données, adresse_destination_eeprom, taille_données)` pour écrire dans une variable EEPROM et `eeprom_read_block(adresse_destination, adresse_données_eeprom, taille_données)` pour lire.

3.3 Solde sur deux octets

Il existe un conflit de type lorsque qu'on manipule des octets et le solde dans le sens où le solde est codé sur 2 octets. Ceci implique qu'on doit faire attention aux opérations. Pour arranger ce problème, un octet reçoit huit décalages vers gauche puis on l'additionne / soustrait à un autre octet tel que

```
1 # déclaration des octets x1, x2
2     uint8_t x1;
3     uint8_t x2;
4
5 # déclaration du solde codé sur deux octets
6     uint16_t solde;
7
8 # addition en faisant huit décalages à gauche pour x1
9     solde = (x1<<8) + x2;
```

3.4 En cas de dépassement de débit ou de crédit

La variable solde ne doit pas fonctionner de manière circulaire dans le sens où dépasser la valeur limite ($0xff$ en écriture hexadécimal) revient à retourner à zéro et vice versa. Par exemple, si je débite un montant plus élevé que le solde, il faut que le programme identifie ce problème et m'indique une impossibilité de la transaction. Pour régler ce problème, on utilise une propriété mathématique telle qu'il existe trois entiers a, b et c, si $c = a + b$ et $c > a$ alors c est bien supérieur à $a + b$.

3.5 Notion d'anti-arrachement

Les cartes à puces doivent s'adapter en cas d'arrachement. En effet, lors de l'écriture dans la mémoire, il se peut qu'il y ait une rupture d'électricité ou bien un arrachement involontaire, abordant ainsi le problème de l'anti-arrachement. Il s'agit d'une mauvaise exécution du programme à cause d'une interruption involontaire. C'est pourquoi les cartes à puces intègrent le concept ACID (atomicité, cohérence, isolation et durabilité). L'idée est que lorsqu'on appelle le programme, soit il se réalise entièrement, soit il ne se passe rien, réglant ainsi le problème de l'anti-arrachement. Autrement dit, ceci garantit une fiabilité de l'exécution du programme.