Université Paris 8

Faculté des sciences et technologies

Master Arithméiques, Codage et Cryptologie

Programmation Carte à puce
projet porte-monnaie

Auteur :
NKANJE TIOMOU Kevin
LINON Romuald
AMAOUCHE Kaci

Sous la direction de
Loïc DEMANGE

2021/2022

Contents

1 Introduction

Une carte à puce est une carte en plastique dotée d'au moins un circuit intégré ou « puce » contenant de l'information. Ce circuit intégré peut se limiter à des circuits de mémoire non volatile ou bien comporter un microprocesseur capable de traiter ces informations. La lecture de la carte nécessite des équipements spécialisés et peut s'effectuer avec ou sans contact avec la puce. Cette technologie est principalement utilisée comme moyen d'identification personnelle (dans le cas des badges d'accès, d'une carte SIM ou même de la carte vitale) ou comme moyen de paiement (notamment dans le cas de la carte bancaire et du porte-monnaie électronique).

2 Architecture

2.1 Architecture physique

Dans notre projet, on dispose d'une carte, ses principales caractéristiques sont:

- FLASH: mémoire programme de 32 ko.
- RAM: données (volatile) 2 ko.
  Une mémoire volatile nécessitant une alimentation électrique continue pour pouvoir conserver l'information qui y est enregistrée, en cas de coupure, la mémoire est perdue.
- EEPROM : données (non volatile) 1ko.
  Une mémoire non volatile, elle ne nécessite pas une alimentation électrique continue pour pouvoir conserver l'information. Par exemple, dans le cas des cartes bancaires, pour pouvoir effectuer des opérations de retrait il faut que le solde du compte soit enregistré pour savoir si le propriétaire a assez d'argent sur son compte.

2.2 Architecture logicielle

2.2.1 La mémoire RAM

Les variables sont déclarées dans la mémoire RAM par défaut. Et on dispose de deux fonctions d'entrée/sortie dont on a besoin pour entrer et sortir des informations de la carte :

- recbytet0(): lecture d'un octet sur l'interface de la carte
- sendbytet0() : sortir d'un octet de la carte.

2.2.2 La mémoire EEPROM

Pour déclarer des variables dans l'EEPROM, il va falloir indiquer au compilateur qu'elles ne sont pas dans la RAM. Pour faire ça, on utilise l'attribut EEMEM, « type nom EEMEM ». Mais tout d'abord il faut inclure le fichier avr/eeprom.h à l'entête du programme, en utilisant ce fichier: include <avr/eeprom.h>. Les données de la mémoire EEPROM ne peuvent pas être traitées directement donc il faut d'abord les transférer dans la RAM pour pouvoir ensuite faire des manipulations.
Pour cela on aura des fonctions de lecture et d'écriture pour la mémoire EEPROM :

- uint8_t eeprom_read_byte(uint8_t * src) : lecture d'un octet, elle prend en paramètre l'adresse de la variable.
- uint8_t eeprom_write_byte(uint8_t * dest, uint8_t val): écriture d'un octet, elle prend en paramètre l'adresse de la variable où on veut écrire et sa valeur.

On a aussi :

- eeprom_read_word (uint16_t * src) ; Permet de lire un mot (entier de 16 octets) dans l'EEPROM

- eeprom_read_block (void * dest, void * src, uint8_t n); Permet de lire n octets de donn´ees dans l'EEPROM

- uint8_t eeprom_read_byte(uint8_t * src); Permet de lire un entier de 8 octets dans l'EEPROM

- eeprom_write_word (uint16_t * dest, uint16_t val); Permet d'ecrire un mot (entier de 16 octets) dans l'EEPROM

- eeprom_write_block (void * src, void * dest, uint8_t n); Permet d'ecrire n octets de donn´ees dans l'EEPROM

- void eeprom_write_byte(uint8_t * dest, uint8_t val); Permet d'ecrire un entier de 8 octets dans l'EEPROM

2.2.3 La mémoire FLASH:

La mémoire FLASH est de l'EEPROM rapide, la seule différence c'est qu'on ne peut pas effacer octet par octet, et on doit l'effacer avant de réécrire dessus.

Avantage: Rapidité.

Inconvénient: Pour effacer un octet, on est obligé d'effacer tout un secteur de la mémoire flash, elle est organisée en plusieurs secteurs: par exemple pour une mémoire de 8 Mo, on peut avoir 64 secteurs de 128 ko. Et pour réécrire un octet de la mémoire flash, on est obligé d'effacer tout le secteur de 128 ko avant de réécrire dessus.

2.3 Outils pour programmer

- AVRDUD: Est un programme opensource pour la programmation des processeurs AVR. Il est indépendant de l'outil qui a été utilisé pour produire le code exécutable, il n'est pas important que le programme soit écrit en C, C++ ou assembleur. AVRDUD se charge simplement de le transférer dans le processeur cible. On dispose d'une carte à puce insérable dans un programmateur et dans un lecteur de carte. Une fois alimentée par le lecteur, elle lui envoie un ATR (Answer To Reset) qui a une réponse de RESET, et qui contient une séquence d'octets émise par la puce qui est censée donner au lecteur des informations sur le fonctionnement de la puce.

- SCAT: Est un programme qui permet d'envoyer des commandes via un lecteur. Il peut s'utiliser en mode interactif ou par l'envoie d'un lot de commande depuis un fichier. Une fois lancé, le programme vérifie si un ou plusieurs lecteurs sont branchés et affiche la liste. S'il n'existe aucun lecteur il s'arrête. Et si au moins un lecteur est présent il affiche une invite de commande « * » .

2.4 APDU (Application Protocole Data Unit)

Format de commande APDU à envoyer à la carte : CLA INS P1 P2 P3 (ISO 7816-3 et 4)

- CLA : correspond à la classe d'instruction.
- INS : désigne le numéro d'instruction.
- P1 : désigne le paramètre 1.
- P2 : désigne le paramètre 2.
- P3 : désigne la taille des données transmises.

Une fois que la carte exécute la commande, elle se met en attente d'une autre commande grâce à la boucle principale «for», qui se trouve dans la fonction principale «main», et qui reçoit la commande et attribue les cinq premiers octets aux variables globales en static en RAM (CLA, INS, P1, P2, P3). Puis elle traite le fonctionnement de la commande par une boucle «switch» qui vérifie s'il n'y a pas d'erreur au niveau de la structure de la commande et l'exécute. Sinon elle envoie un status word d'erreur qui dépend de la faute commise.

Les différentes réponses de la carte sont:

- 90 00: Fin normale
- 6E 00: CLA inconnue
- 6D 00: CLA connue, mais INS inconnu
- 6B 00: CLA, INS connues, mais P1 et P2 incorrects
- 6C xx: CLA,INS,P1,P2 corrects mais P3 incorrect et xx désigne le P3 attendu
- 6F 00: Commande inconnue

3 Description du projet

Dans cette partie nous allons s'intéresser à résoudre les problématiques qui concernent le projet.
Le but du projet est de réaliser un porte monnaie électronique, avec pour fonctionnalités:

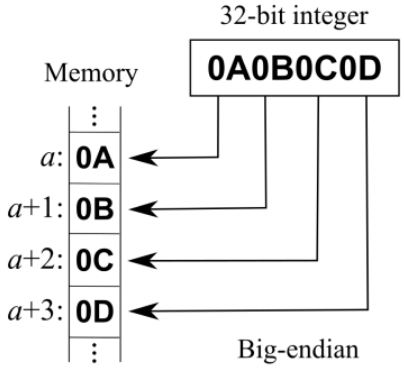
- Introduction des données personnelles
- Lecture des données personnelles
- Lecture du solde
- Retrait d'argent
- Dépôt d'argent

Le solde sera sur 16 bits, et stock´e dans un entier non signé uint16_t.
Les données personnelles et le solde seront dans une mémoire non volatile.
La classe de la carte sera la classe libre 0x81.

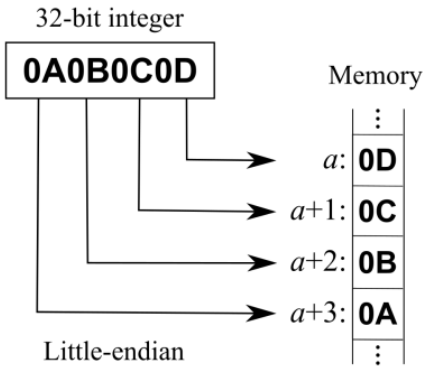
3.1 Problèmes à prendre en compte

3.1.1 L'Endianness

Le solde est stocké sur deux octets. Or, on ne peut envoyer et recevoir qu'octet par octet. On va donc devoir découper un entier de 16 bits en deux entiers de 8 bits, et inversement reformer un entier de 16 bits avec deux entiers de 8 bits. Ce qui vous nous amener à l'utilisation de l'Endianness: C'est la manière d'organiser l'octet en mémoire.
On peut stocker le bit de poids fort en premier.



Ou on peut stocker le bit de poids faible en premier.



3.1.2 Retrait ou dépôt trop important

Le solde est stock´e comme un entier non signé, il faut donc faire attention aux dépôts et aux retraits invalides.

- Si on essaye de retirer une somme trop importante, notre solde va boucler ($0 \;\; 1 = 65535$).
- Si on essaye de déposer une somme trop importante, notre solde va boucler ($65535 + 1 = 0$).

3.2 Écriture en EEPROM

On initialise d'abord les données:

```
172  typedef enum{vide = 0, plein = 0x1c} etat_struct;
173
174  etat_struct ee_etat EEMEM = vide;
175
176  struct donnees{
177      uint8_t nb_ope;
178      uint8_t tailleTotale;
179      uint8_t n[NBROPE];
180      uint8_t * d[NBROPE];
181      uint8_t buffer[TAIILEMAX];
182  } ee_aEcrire EEMEM;
183
```

Il y a trois fonctions:

3.2.1 La fonction "solde()"

Transmet d'abord l'octet de poids fort, puis celui de poids faible, "big endian", on initialise le solde.

```
184 void solde(){
185     int i;
186     // vérification de la taille
187     if (p3 != 2)
188     {
189         sw1=0x6c;   // P3 incorrect
190         sw2=2;  // sw2 contient l'information de la taille correcte
191         return;
192     }
193     sendbytet0(ins);    // acquitement
194     uint16_t x = eeprom_read_word(&ee_solde);
195     d1 = x>>8;
196     d2 = x&0xFF;
197     sendbytet0(d1);
198     sendbytet0(d2);
199     sw1=0x90;
200 }
201
```

3.2.2 La fonction "retrait()"

Le contrôle du montant: si le montant du retrait est plus élevé que le solde, il affiche le status word d'erreur 91 00. Sinon, il effectue l'opération demandé et affiche le status word 90 00

3.2.3 La fonction "dépôt()"

Permet d'ajouter un montant au solde, et d'effectuer le contrôle de débordement: car avec les deux octets que nous avons choisis comme taille du solde au départ, son maximum est de 65535 centimes. Et si on ajoute 1 centime de plus à ce montant, le solde sera de 0 centime comme expliquer de le point 3.1.2 .

```
202 void depot(){
203     int i;
204     // vérification de la taille
205     if (p3!=2)
206     {
207         sw1=0x6c;   // P3 incorrect
208         sw2=2;  // sw2 contient l'information de la taille correcte
209         return;
210     }
211     sendbytet0(ins);    // acquitement
212     uint16_t x = eeprom_read_word(&ee_solde);
213     uint16_t y = eeprom_read_word(&ee_depot);
214     d1 = x>>8;
215     d2 = x&0xFF;
216     sendbytet0(d1);
217     sendbytet0(d2);
218
219 }
```

3.3 Transactions ACID

Au moment de la transmission des données, si l'utilisateur arrache sa carte, cela pourrait produire des problèmes de cohérences au niveau des données. Soit une opération se déroule entièrement, soit pas du tout. Si l'engagement échoue, on perd l'opération. Si la validation échoue, on recommence jusqu'à succès. Alors pour minimiser le risque de ce problème, nous avons utilisé deux procédures.

3.3.1 La fonction "engager_donnees()"

Permet d'écrire la taille, les données à écrire ainsi que leurs destinations dans une structure (elle-même dans l'EEPROM) en suivant cet algorithme:

- état ← vide;
- Placer les données en mémoire auxiliaire;
- état ← plein;

L'engagement ne passe 'a l'état plein que lorsqu'il a entièrement fini de remplir la structure.

3.3.2 La fonction "valider()"

Permet de copier les données de la mémoire auxiliaire dans la destination, en suivant cet algorithme:

- Si l'état est plein; alors déplacer l'auxiliaire dans la destination;
- Si non aucune opération n'est effectuée.

La validation ne passe 'a l'état vide que lorsqu'elle a entièrement fini d'écrire les données.

4 Chiffrement

Le but est d'avoir une certaine sécurité du porte monnaie. On doit vérifier l'intégrité et l'authenticité de chaque action, les commandes de débit et de crédit ne sont faites que pour être réalisées une seule fois. Il faut donc faire en sorte qu'une commande ne puisse être éditée que par la banque, qu'elle ne puisse pas être réalisée plusieurs fois, que la somme ne puisse pas être modifiée et qu'elle soit exécutée seulement si elle est authentique, intègre et unique.

Pour cela on va rajouter une entête et un compteur qui seront chiffrés. On aura donc:

- Une entête fixe de 4 octets
- Un compteur de 2 octets qui s'incrémente à chaque opération de la banque
- Un solde de 2 octets

La somme ne pourra pas être falsifiée car elle sera chiffrée, le souci d'intégrité sera repéré grâce à une entête erronée, et la vérification du compteur permettra de vérifier si la commande a déjà été exécutée.
Pour le chiffrement nous allons donc nous servir de l'algorithme symétrique TEA , qui prend des clés de taille 128 bits.
void tea_chiffre(uint32_t * clair,uint32_t * crypto, uint32_t * k);
void tea_dechiffre(uint32_t * crypto,uint32_t * clair, uint32_t * k);
clair et crypto prennent des tailles de 64 bits, k est la clé de 128 bits

4.1 TEA _chiffre

```
225  void tea_chiffre(uint32_t * clair,uint32_t * crypto, uint32_t * k)
226  {
227      uint32_t    y=clair[0],
228                  z=clair[1],
229                  sum=0;
230      int i;
231      for (i=0;i<32;i++)
232      {
233          sum += 0x9E3779B9L;
234          y += ((z << 4)+k[0]) ^ (z+sum) ^ ((z >> 5)+k[1]);
235          z += ((y << 4)+k[2]) ^ (y+sum) ^ ((y >> 5)+k[3]);
236      }
237      crypto[0]=y; crypto[1]=z;
238  }
239
```

4.2 TEA _dechiffre

```
241  void tea_dechiffre(uint32_t* crypto ,uint32_t* clair, uint32_t*k)
242  {
243      uint32_t    y=crypto[0],
244                  z=crypto[1],
245                  sum=0xC6EF3720L;
246      int i;
247      for (i=0;i<32;i++)
248      {
249          z -= ((y << 4)+k[2]) ^ (y+sum) ^ ((y >> 5)+k[3]);
250          y -= ((z << 4)+k[0]) ^ (z+sum) ^ ((z >> 5)+k[1]);
251          sum -= 0x9E3779B9L;
252      }
253      clair[0]=y; clair[1]=z;
254  }
```