COMPTE RENDU

SYSTÈMES DE TÉLÉCOMMUNICATIONS SANS-FILS

24/11/17

TP MODULE ITC43

AU SEIN DE

L'ECOLE

Esirem Dijon

VINCENT **CANDAPPANE** : ETUDIANT 4A INFOTRONIQUE (ILC)
KEVIN **NAUDIN** : ETUDIANT 4A INFOTRONIQUE (ILC)
M. VINCENT **THIVENT** : ENSEIGNANT & DIRIGEANT DE **ODALID**

Activités confiées : Développer l'interface graphique pour la gestion de la carte sans-contact MIFARE Classic

TABLE DES MATIÈRES

INTRODUCTION

Dans le cadre de nos travaux pratiques des systèmes de communication sans fil au sein de l'ESIREM *École Supérieure d'Ingénieurs de Recherche en Matériaux et Infotronique* de Dijon, nous allons appliquer et résumer les connaissances acquises au cours du module ITC43, enseigné par **M. Vincent THIVENT**.

En utilisant l'environnement de développement multiplate-forme Qt Creator, les différentes librairies liées à la technologoie RFID (Radio Frequency Identification) ainsi qu'un lecteur RFID développé par la société Odalid, il nous a été demandé de programmer une partie IHM et une interface permettant l'utilisation de la carte RFID : Mifare Classic.

Après une description plus approfondie des objectifs de TP et des outils mis en œuvre, nous allons étudier, par étape, chacunes des tâches demandées.

Dans une première partie, nous étudierons de manière approfondie l'architecture de la carte et du lecteur RFID. Nous mettrons ensuite en place un système basique d'identification pour identifier un utilisateur nécessitant les notions de lecture écriture. Pour finir, nous programmerons un compteur d'incrémentation et de décrémentation, modélisant le porte-monnaie de l'utilisateur.

La réalisation de ce dernier devra vous inclure les différents paradigmes (pas nécessairement tous) vus en cours : technologie RFID, carte Mifare Classic, lecteur, opérations, clé de sécurité, Qt Creator, C++, etc.

I – DÉFINITION DES ARCHITECTURES

1) FONCTIONNEMENT D'UN SYSTÈME RFID

Un système RFID (Radio Fréquence Identification) se compose de transpondeurs (aussi nommés étiquettes, marqueurs, tags, identifiants...) et d'un ou plusieurs interrogateurs (aussi nommés coupleurs, base station...). Il s'agit d'une technologie très attractive pour l'entreprise qui offre la possibilité d'une gestion automatique du nombre conséquent d'informations qu'elle doit traiter. Les équipements adaptés à ce système permettent de synchroniser les flux physiques avec les flux d'informations.

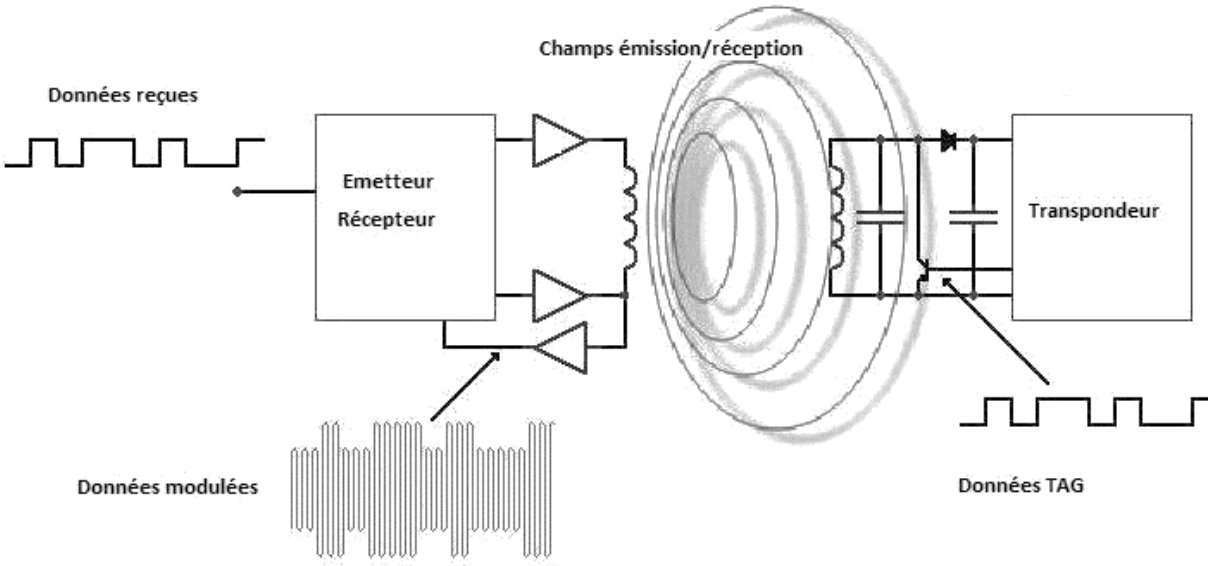
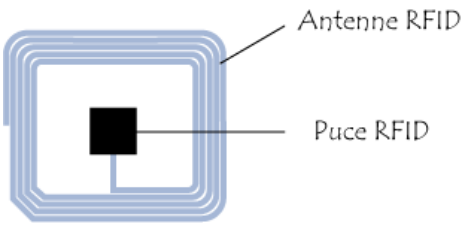


Figure 1 : Schéma de principe de la technologie RFID

Interrogateurs RFID : Ce sont des dispositifs actifs, émetteurs de radiofréquences qui vont activer les tags qui passent devant eux en leur fournissant l'énergie dont ils ont besoin pour fonctionner. Outre de l'énergie pour la carte, le lecteur envoie des commandes particulières auxquelles répond le tag. L'une des réponses les plus simples possible est le renvoi d'une identification numérique. La fréquence utilisée par les interrogateurs est variable selon le type d'application visé et les performances recherchées. Ces dernières sont détaillées dans la partie « Gammes de fréquences »

Carte RFID : C'est un dispositif récepteur, que l'on place sur les éléments à tracer (objet, animal...). Ils sont munis d'une puce contenant les informations et d'une antenne pour permettre les échanges d'informations.



Il existe trois types de cartes RFID : des cartes passives, des cartes actives et des cartes semi-actives. Dans notre étude de cas, nous allons étudier la carte sans contact de type passive : Mifare Classic.

2) ARCHITECTURE DE LA CARTE : MIFARE CLASSIC

- La carte sans contact de type Mifare Classic réagit à la norme ISO/IEC 14443 Type A. La fréquence porteuse est 13,56 MHz et le débit de communication est de 106 kbit/s.
- La carte possède une capacité mémoire de type EEPROM de 1 kilo-octet et est constituée de 16 secteurs de 4 blocs (un bloc = 16 octets).
- Les commandes que peut supporter la carte sont divisées en trois catégories distinctes :

1. Procédure d'authentification :

Authentication with Key A : Possibilité de s'authentifier avec la clé A
Authentication with Key B : Possibilité de s'authentifier avec la clé B 8.
Personalize UID Usage **:** Changer UID de la carte

2. Procédure de sélection et identification :

Select : Gérer les anti-collisions
Request : Réaliser une requête
Wake-up : Réveiller la carte
Anticollision : Fonction d'anticollision intelligente permettant l'utilisation plusieurs carte dans le champ simultanément. L'exécution d'une transaction avec une carte sélectionnée est effectuée correctement sans Interférence d'une autre carte sur le terrain.

3. Opérations mémoires

- **Read** **:** Lecture d'un bloc mémoire
- **Write** **:** Écriture d'un bloc mémoire
- **Increment** **:** Incrémente le contenu d'un bloc et stocke le résultat dans le registre interne
- **Decrement :** Décrémente le contenu d'un bloc et stocke le résultat dans le registre interne
- **Transfer** **:** Écrit le contenu du registre de données internes dans le bloc
- **Restore** **:** Lit le contenu d'un bloc dans le registre de données internes
- **Halt** **:** Désactive la carte

3) SÉCURITÉ DE LA CARTE

- La sécurité de l'accès à la carte se déroule en plusieurs étapes :
  - Le lecteur spécifie le secteur à accéder et choisit la clé A ou B.
  - La carte lit le code secret et les conditions d'accès, puis envoie un nombre sur le lecteur.
  - Le lecteur calcule la réponse en utilisant la clé secrète et renvoie sa réponse à la carte.
  - La carte vérifie la réponse du lecteur en la comparant à son propre calcul.
  - Le lecteur vérifie la réponse de la carte en la comparant à son propre calcul.
  - Établissement de la connexion cryptée.

- On peut stocker les informations dans l'EEPROM. Un backup doit être prévu pour le débit de crédit d'unité. Il pourra se faire grâce à la commande 'Restore'.

4) ARCHITECTURE DU LECTEUR

- La prise de contact avec le lecteur peut se faire de 2 manières différentes :
  - *ISO14443_3_A_Pollcard*
  - *ISO14443_3_A_PollcardWU*
- L'authentification se fait à l'aide de la fonction *"Mf_Classic_Authenticate"*.
- Les options sur la mémoire se font grâce à une multitude de fonctions :
  - *Mf_Classic_Read_Block*
  - *Mf_Classic_Write_Block*
  - *Mf_Classic_Read_Sector*
  - *Mf_Classic_Write_Sector*
  - *Mf_Classic_Read_Value*
  - *Mf_Classic_Write_Value*
  - *Mf_Classic_Increment_Value*
  - *Mf_Classic_Decrement_Value*
  - *Mf_Classic_Restore_Value*

Figure 2 : Lecteur de carte MIFARE Classique

5) OUTILS REQUIS

- Qt Creator, un IDE centré sur l'utilisation du **langage C++** et mettant à disposition une bibliothèque graphique permettant de réaliser des applications de bureau
- Qt, **une bibliothèque dérivée de C++** apportant son lot de modifications au langage de même que des types et permettant de faire le lien entre le C++ pur et l'Interface Homme-Machine (IHM) de Qt Creator
- Le Software Development Kit (SDK) de la société **Odalid**, mettant à disposition de notre code de nombreuses opérations permettant d'interagir avec notre lecteur RFID.

II – INTERFACE GRAPHIQUE

Avec Qt Creator, la création d'une interface graphique simple et efficace a été intuitive et facile à mettre en place. De plus, l'interface se devait d'être facile d'utilisation, basique et rapidement compréhensible aux yeux des utilisateurs. L'interface graphique permet l'affichage et la modification des éléments présents dans la carte.

Pour sa réalisation nous avions listé toutes les informations nécessaires lors du développement. Après une lecture approfondie du sujet, l'interface finale doit être capable d'afficher :

1. Version et type du lecteur
2. Statuts du lecteur
3. Nom
4. Prenom
5. Relevé des dernières opérations effectuées (Incrément - Décrément)
6. Crédits

Pour cela nous avions pris l'initiative de créer différents boutons interactifs rendant possible via l'utilisateur de réaliser les services ci-dessus, suivant un ordre prédéfini :

Bouton 1 : Connexion du lecteur RFID
Bouton 2 : Deconnexion du lecteur RFID
Bouton 3 : Enregistrement des données → Écriture (send)
Bouton 4 : Lecture des données rentrées précédemment → Lecture (get)
Bouton 6 : Formattage manuelle des données de la carte (réaliser directement dans IHM.cpp)
Bouton 7 : Incrémentation
Bouton 8 : Décrémentation

Afin de déterminer si la bonne carte est bien posée sur le lecteur, nous avons ajouté un voyant changeant de couleur via la méthode **LEDBUZZER(ReaderName *Name, uint8_t LEDBuzzer)** :

- En rouge, si une mauvaise carte est placée (comme la carte Mobigo de DIVIA)
- En vert, s'il s'agit de la carte sans contact Mifare CLASSIC

Figure 3 : Lecteur RFID inactive (rouge) et active (vert)

Pour l'amélioration de notre interface et le contrôle des boutons, nous nous sommes mis dans l'optique de limiter au maximum les erreurs utilisateurs. C'est pour cela que nous avons choisi de bloquer le bouton Connexion (modélisé par **le logo 'clé' bleu**) rendant possible la connexion du lecteur à l'interface. Ceci évite une double connexion qui provoque une erreur par la suite. Cependant, les boutons liés à la modification de la carte n'ont pas été désactivés, en contrepartie un message d'erreur s'affiche indiquant qu'aucun lecteur n'a été trouvé et donc qu'aucune information ne peut être lue. Voici notre interface de départ.

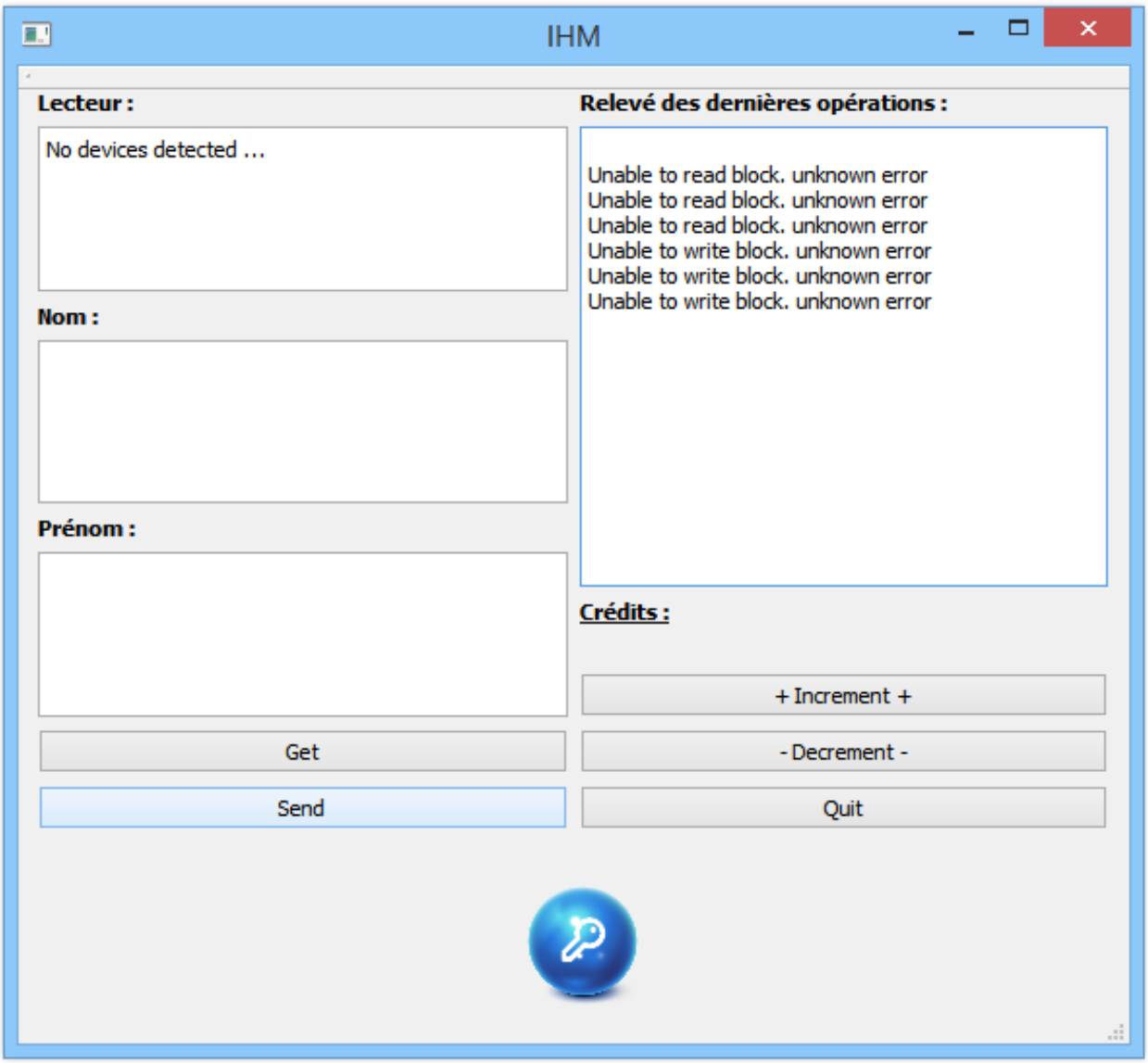


Figure 4 : Interface non activée sans carte présente

Nous sommes ici dans le cas où aucune carte n'est présente sur le lecteur. Aucune information n'est lisible. De ce fait, le bouton Connexion est verrouillé et des messages d'erreurs volontaires sont apparents en cas de mauvaise manipulation. Nous allons maintenant voir l'interface avec une carte présente sur le lecteur.

Notre application permet dès son lancement d'activer le champ RFID du lecteur, puis par la suite de se connecter à sa carte (si l'utilisateur utilise une carte déjà configurée)

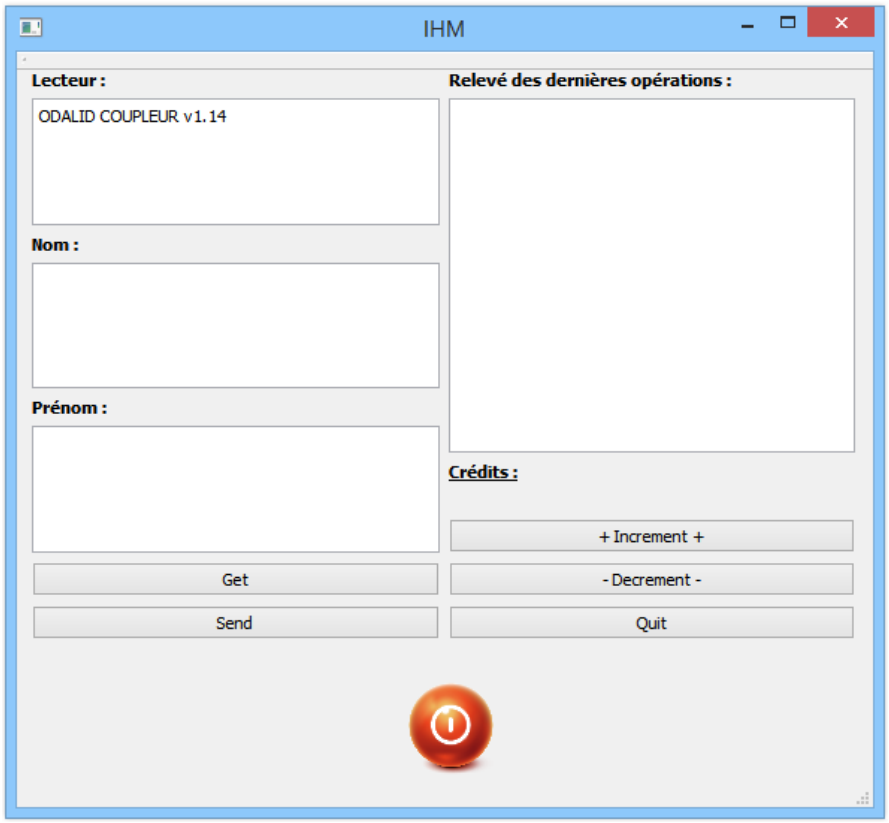


Figure 5 : Interface activée avec carte, mais non utilisé pour le moment

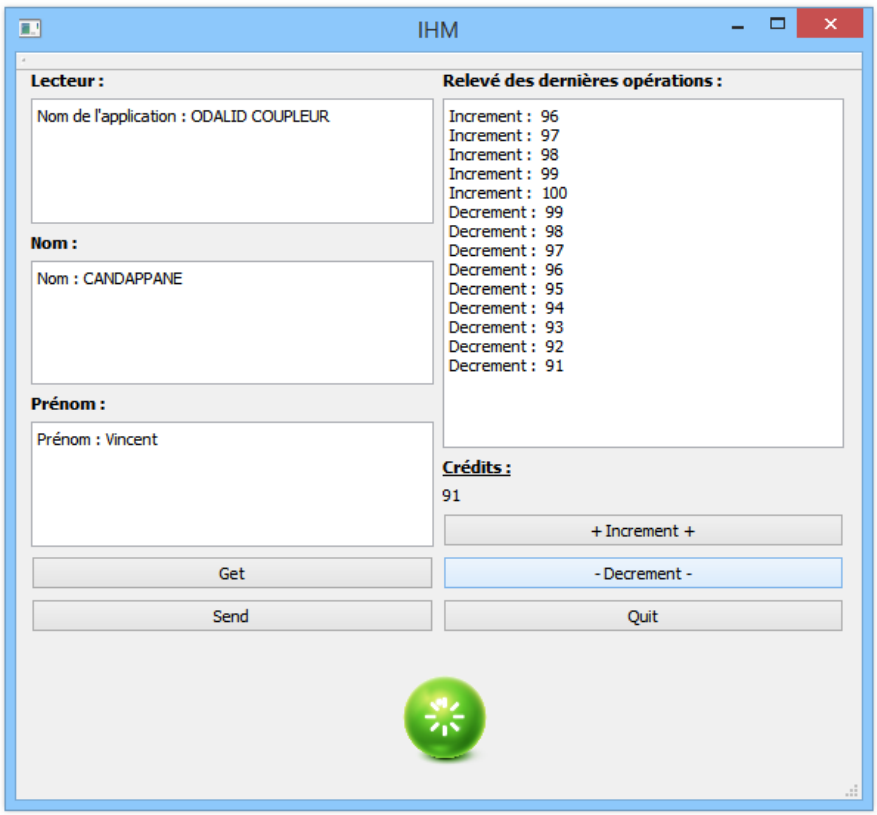
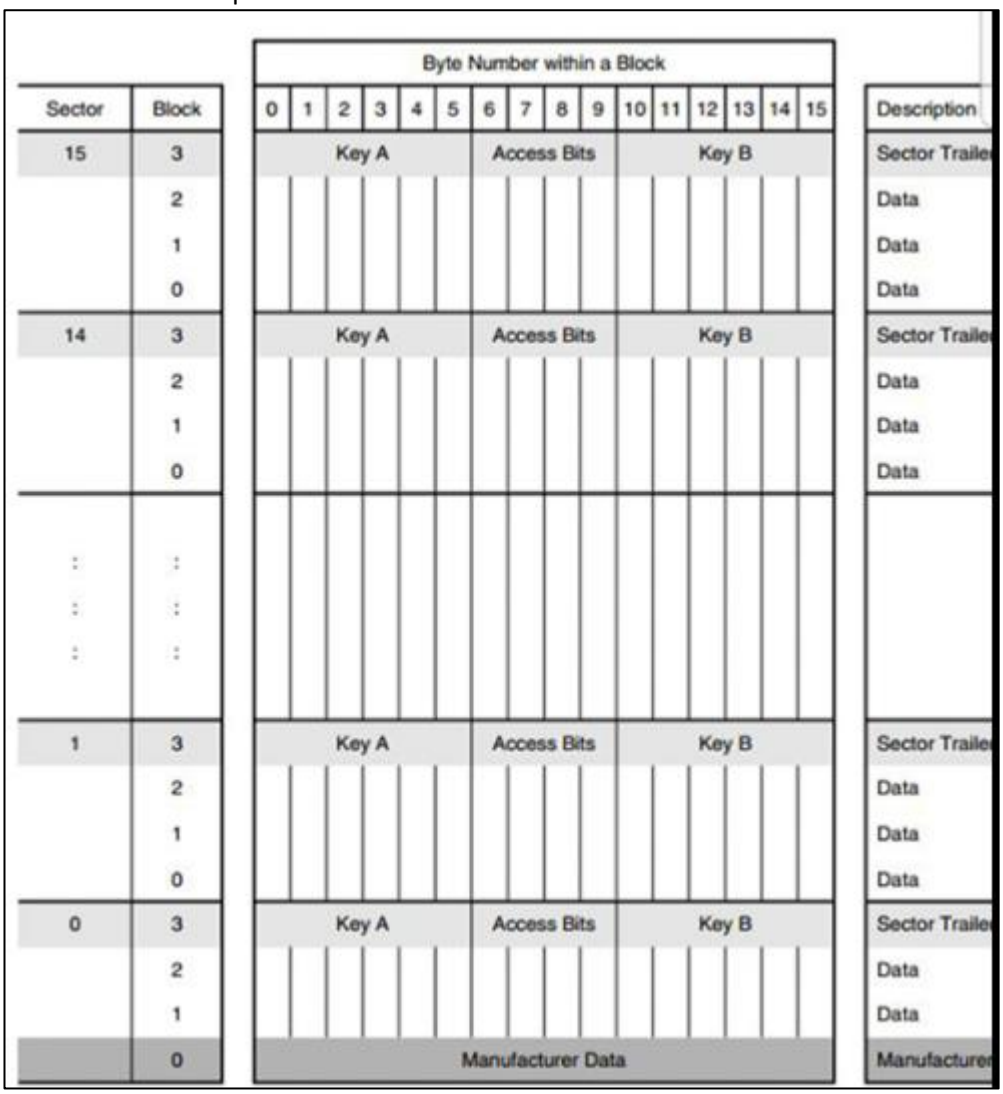


Figure 6 : Interface avec carte présente et en pleine activité

Nous voyons ici que les informations dans la carte sont correctement affichées dans les cases correspondantes, le bouton connexion change de statut. L'utilisateur trouvera plusieurs options :
- Personnaliser son identité
- Créditer ou débiter ses unités

III – LECTURE / ÉCRITURE

Pour mieux comprendre le déroulement de ce TP, il est important de voir en détail l'architecture d'une carte Mifare classique.



Sector	Block	Byte Number within a Block 0 1 2 3 4 5	6 7 8 9	10 11 12 13 14 15	Description
15	3	Key A	Access Bits	Key B	Sector Trailer
	2				Data
	1				Data
	0				Data
14	3	Key A	Access Bits	Key B	Sector Trailer
	2				Data
	1				Data
	0				Data
:	:				
:	:				
:	:				
1	3	Key A	Access Bits	Key B	Sector Trailer
	2				Data
	1				Data
	0				Data
0	3	Key A	Access Bits	Key B	Sector Trailer
	2				Data
	1				Data
	0	Manufacturer Data			Manufacturer

Figure 7 : Schéma de l'architecture mémoire d'une carte Mifare Classique

La carte Mifare Classic possède une mémoire reprogrammable constituée de 16 secteurs contenant chacun 4 blocs de 16 octets. Cela signifie que l'on peut stocker 1ko de données sur ce type de carte. Lorsque l'on souhaite stocker une information dans la carte, il faut savoir que la donnée prendra forcément un bloc de mémoire. Ainsi, une donnée stockée ne pourra dépasser 16 octets en taille. Il y a également un aspect de sécurité à la carte. Pour pouvoir lire ou écrire certains blocs et secteurs, nous avons besoin des clefs d'autorisation nécessaires. Sans ces clefs, l'accès nous sera refusé. Nous allons désormais étudier comment connecter la carte à l'ordinateur via le lecteur RFID puis comment se passent la lecture et l'écriture au sein du programme. Il sera précisé lorsque l'opération provient de la bibliothèque Odalid.

1) DÉFINITION DES CONDITIONS D'ACCÈS

- Définition des conditions d'accès pour le secteur 2 :

Nous utiliserons le secteur 2 pour stocker l'identité du propriétaire de la carte.

Sector	Block	Data
2	11	Clé A + Access Bit + Clé B
	10	Nom
	9	Prenom
	8	"Identité"

- **Clé A = A0 A1 A2 A3 A4 A5**
- **Clé B = B0 B1 B2 B3 B4 B5**

Les nombres de clés sont importants dans la mesure d'avoir un accès à différentes opérations de mémoire ou d'authentification de la carte. Une clé peut avoir tout les droits d'accès ou plusieurs clés peuvent être utilisées pour un droit d'accès en particulier.

Voici les accès pour 4 applications :

Lecture *Decrement*	***Moins sensible*** *KEY A*
Écriture *Incrément*	***Plus sensible*** *KEY B*

2) CONNEXION À LA CARTE

Avant de procéder à une quelconque opération, nous devons nous assurer que la carte est connectée au PC via le lecteur RFID. Pour ce faire, nous créons une référence de type ReaderName dans notre code vers le lecteur de carte grâce à la bibliothèque Odalid. Nous créons une variable pour retenir le statut renvoyé par la carte, contenant 0 si les opérations se déroulent comme prévu ou un autre chiffre correspondant à un code d'erreur dans le cas contraire. Pour éviter les conflits à la connexion, nous désactivons le port USB de connexion du Reader puis le réactivons. Ainsi si la carte était déconnectée cela nous éviter de tenter une seconde connexion inutile. Nous demandons ensuite au lecteur de s'alimenter grâce à l'opération RF_Power_Control, puis demandons au lecteur d'initialiser la connexion entre lui et la carte grâce à l'opération ISO14443_3_A_PollCard. Cette opération permet également de récupérer l'identifiant du *manufacturer*, le type du tag ainsi que le numéro d'identification de celui-ci.

```
status = CloseCOM1(&cardReader);
cardReader.Type = ReaderCDC;
cardReader.device = 0;
status = OpenCOM1(&cardReader);

RF_Power_Control(&cardReader, TRUE, 0);

status = ISO14443_3_A_PollCard(&cardReader, atq, sak, uid, &uid_len);
```

Figure 8 : Reconnexion USB, alimentation du lecteur et connexion à la carte

Lorsque le lecteur est connecté et que la carte est reconnue (cela suppose que la carte a été posée sur le lecteur avant d'appuyer sur le bouton de connexion) le programme chargera les clefs de sécurité (dont nous parlions précédemment) dans le lecteur RFID. Ces clefs seront ensuite associées aux secteurs ou elles seront utilisées.

```
status = Mf_Classic_LoadKey(&cardReader, Auth_KeyA, key1, 2);
status = Mf_Classic_LoadKey(&cardReader, Auth_KeyB, key4, 3);
```

Figure 9 : Chargement des clefs de sécurité dans le lecteur

Dans cet exemple, cardReader est notre lecteur de carte. key1 sera chargée dans le lecteur et sera utilisée dans la slot KeyA pour le secteur 2. key4 sera chargée dans le lecteur et sera utilisée dans le slot KeyB pour le secteur 3. Si quoi ce soit de mal intervient lors de la connexion, l'application affichera un message d'erreur dans la fenêtre de droite (voir la partie INTERFACE pour de plus amples informations).

3) LECTURE / ÉCRITURE

Passons à la partie essentielle de l'application : la lecture et l'écriture de données au sein de la carte. Comme nous pouvons le voir sur notre interface, nous disposons de plusieurs zones de texte permettant d'afficher ou de rentrer du texte. Nous mettons également en place deux boutons, « send » et « get », l'un permettant de transmettre les informations des zones de texte à la carte, l'autre permettant de récupérer les données de la carte et afficher celles-ci. La lecture et l'écriture ne sont utilisées que dans le cas de la première application de la carte appelée « Identité » ; c'est-à-dire que toutes les informations de cette application seront stockées dans le secteur 2 de la carte. On rappelle ici qu'un bloc d'un secteur ne peut contenir plus de 16 octets. Comme 1 caractère = 1 octet on ne pourra donc stocker que 16 caractères par bloc. Nous allons donc créer une fonction permettant l'écriture dans un bloc donné à partir d'une zone de texte, ainsi que la lecture d'un bloc donné pour l'afficher dans une zone de texte. On pourrait se demander pourquoi nous n'informons pas le code du secteur correspondant. C'est tout simplement, car les blocs d'un secteur ont en fait pour index.

Index_bloc = (Index_secteur*4)+numero_bloc

De plus, le secteur dans lequel nous sommes est enregistré au sein du code. Il reste à 2 pour la lecture et l'écriture, à 3 pour le reste. Ainsi, si nous souhaitons écrire ou lire les données du bloc 2 du secteur 2, nous devrons donner comme index de bloc la valeur 10. Pour la lecture et l'écriture, les fonctions à appeler possèdent les mêmes paramètres. Nous les décrivons ici :

Mf_Classic_Write_Block(&cardReader, TRUE, block_index, data, key, key_index);

- cardReader correspond au lecteur de carte.
- TRUE signifie que nous souhaitons effectuer nos opérations sans avoir besoin de s'authentifier. Si ce booléen était placé à FALSE, toute opération échouerait, car nous n'avons à aucun moment procédé à l'authentification (non demandée par le TP)
- Data correspond à un tableau de caractères renfermant les données à envoyer ou alors sert d'interface permettant de récupérer les données renfermées dans le bloc
- Key correspond à la clef que nous souhaitons utiliser. Dans notre cas on utilisera la clef A pour la lecture et la clef B pour l'écriture.
- Key_index correspond à l'index dans lequel nous avons chargé la clef de sécurité permettant d'accéder au secteur correspondant.

Nous passons désormais à la fonction permettant l'écriture dans un bloc :

sendData(int sector, int block, int display)

Cette fonction sert à préciser quelle zone de texte doit être copiée dans quel bloc. Les valeurs de display sont 0 pour le nom de l'application, 1 pour le nom et 2 pour le prénom. Ainsi, pour envoyer nos données, nous faisons simplement

```
std::string toWrite;

switch(display)
{
    case 0: toWrite = this->ui->statusDisplay->toPlainText().toStdString();
    break;
    case 1: toWrite = this->ui->nameBox->toPlainText().toStdString();
    break;
    case 2: toWrite = this->ui->surnameBox->toPlainText().toStdString();
    break;
}

strcpy( (char *) dataToWrite, toWrite.c_str() );

status = Mf_Classic_Write_Block(&cardReader, TRUE, block, dataToWrite, Auth_KeyB, sector);
```

Figure 10 : Écriture d'un bloc

Ce code présent la recopie des valeurs des zones de texte dans un tableau de caractères qui est ensuite envoyé dans le bloc choisi.

Nous nous rendons toutefois compte que cela présente des problèmes. En effet lorsque nous envoyons des données dans un bloc, le bloc recopie caractère par caractère l'ensemble des données jusqu'à ne plus rien pouvoir copier. A aucun moment la valeur du bloc n'est réinitialisée pour faire de la place. Imaginons un bloc d'un secteur.

Celui-ci contient bien 16 octets. Nous allons envoyer les données suivantes :

J	e		m	‘	a	p	p	e	l	l	e		L	e	o

Ces données sont recopiées entièrement dans le bloc qui ressemble désormais à cela (exacte recopie du tableau de données envoyées) :

J	e		m	‘	a	p	p	e	l	l	e		L	e	o

Nous souhaitons désormais envoyer les données suivantes :

B	o	n	j	o	u	r	!								

Ces données vont être recopiées caractère par caractère jusqu'au « ! », mais sans réinitialisation du bloc ! Cela signifie que le bloc ressemble à ceci :

B	o	n	j	o	u	r	!	e	l	l	e		L	e	o

Il ne s'agit donc pas des données que nous avons envoyées. Un moyen de corriger ce problème est de faire en sorte qu'avant chaque insertion de données, nous réinitialisons à la main le bloc en y insérant 16 fois le caractère « espace ».

```
//Clears block
for(int i = 0; i<16; i++) dataToWrite[i] = ' ';
status = Mf_Classic_Write_Block(&cardReader, TRUE, block, dataToWrite, Auth_KeyB, 2);
```

Figure 11 : Réinitialisation du bloc

Cela ne pose aucun problème à notre application, car ces blocs ne sont faits que pour stocker des chaines de caractères. Nous pouvons maintenant faire en sorte qu'à chaque appui sur le bouton « send », l'utilisateur enverra les informations de toutes les zones de texte à la carte.

```
void IHM::on_writeBlockButton_clicked()
{
    sendData(sect, 8, 0);
    sendData(sect, 9, 1);
    sendData(sect, 10, 2);
}
```

Pour la lecture il s'agit de la même chose, mais en inversé. Nous utiliserons cette fois ci la clef A.

```
void IHM::getData(int sector, int block, int display)
{
    status = Mf_Classic_Read_Block(&cardReader, TRUE, block, dataToRead, Auth_KeyA, 2);
    if(status != MI_OK) {
        this->ui->textSend->setText(this->ui->textSend->toPlainText()+"\nUnable to read block. "+GetErrorMessage(status));
        this->ui->textSend->update();
    }
    else
    {
        QString result = "";
        // Un bloc ne contient que 16 octets de données. Inutile d'aller plus loin.
        for(int i = 0; i < 16; i++) result += (QString) dataToRead[i];

        switch(display)
        {
            case 0: this->ui->statusDisplay->setText("Nom de l'application : "+result);
                    this->ui->statusDisplay->update();
                break;
            case 1: this->ui->nameBox->setText("Nom : "+result);
                    this->ui->nameBox->update();
                break;
            case 2: this->ui->surnameBox->setText("Prénom : "+result);
                    this->ui->surnameBox->update();
                break;
        }
    }
}
```

Figure 12 : Lecture d'un bloc

On récupère les données du bloc choisi puis, selon l'endroit d'affichage choisi, on recopie les 16 premiers caractères du tableau (celui-ci en fait 240 à la base, mais techniquement un bloc ne fait que 16 octets en données. Inutile d'aller lire le reste de notre tableau donc), les convertissons en string et les affichons au bon endroit. L'utilisateur pourra donc désormais cliquer sur le bouton « get » afin d'obtenir ceci :

```
void IHM::on_readBlockButton_clicked()
{
    getData(sect, 8, 0);
    getData(sect, 9, 1);
    getData(sect, 10,2);
}
```

IV – INCRÉMENTATION / DÉCRÉMATATION

1) DÉFINITION DES CONDITIONS D'ACCÈS

- Définition des conditions d'accès pour le secteur 3 :

Nous utiliserons le secteur 3 pour stocker le crédit d'unité en entier (U_int32) du propriétaire.

Sector	Block	Data
3	15	Clé C + Access Bit + Clé D
	14	Compteur
	13	Backup Compteur
	12	"Porte-monnaie"

- **Clé C** = **C0 C1 C2 C3 C4 C5**
- **Clé D** = **D0 D1 D2 D3 D4 D5**

2) INCRÉMENTATION

```
1.  void IHM::on_Increment_Button_clicked()
2.  {
3.  readIncrementDecrementStatus();
4.  uint32_t value =0;
5.  status = Mf_Classic_Increment_Value(&cardReader, TRUE, 14, 1, 13, Auth_KeyB, 3);
6.  status = Mf_Classic_Restore_Value(&cardReader, TRUE, 13, 14, Auth_KeyB, 3);
7.  status = Mf_Classic_Read_Value(&cardReader, TRUE, 13, &value, Auth_KeyA, 3);
8.  std::cout << status << std::endl;
9.  status = Mf_Classic_Read_Value(&cardReader, TRUE, 14, &value, Auth_KeyA, 3);
10.     std::cout << status << std::endl;
11.
12.     QString displayValue = QString::number(value);
13.
14.     this->ui->textSend->setText("Increment :  "+ displayValue+ "\n");
15.     this->ui->textSend->update();
16.     this->ui->creditLabel->setText(displayValue);
17.     this->ui->creditLabel->update();
18.     }
```

Il est très important de partionner le plus possible les codes partiels dans d'autres méthodes afin de ne pas surplomber le code et afin de pouvoir coder proprement et débugger étape par étape comme l'on peut le voir ici avec la méthode `readIncrementDecrementStatus()`. La fonction précédente concatène 3 fonctionnalités : l'écriture, la lecture et l'affichage du bloque 12 en secteur 3, soit "Porte monnaie".

Voici ce qui se passe en détail grâce au 2 méthodes principaux fournis :

- Increment :
Mf_Classic_Increment_Value(&cardReader, TRUE, 14, 1, 13, Auth_KeyB, 3);

Secteur 3	Block 13 (Backup) = 15
	Block 14 (value) = 14

- Restore :
Mf_Classic_Restore_Value(&cardReader, TRUE, 13, 14, Auth_KeyB, 3);

Secteur 3	Block 13 (Backup) = 15
	Block 14 (value) = 15

Remarque : L'incrémentation n'est pas infinie, elle est limitée à la valeur : 4294967295

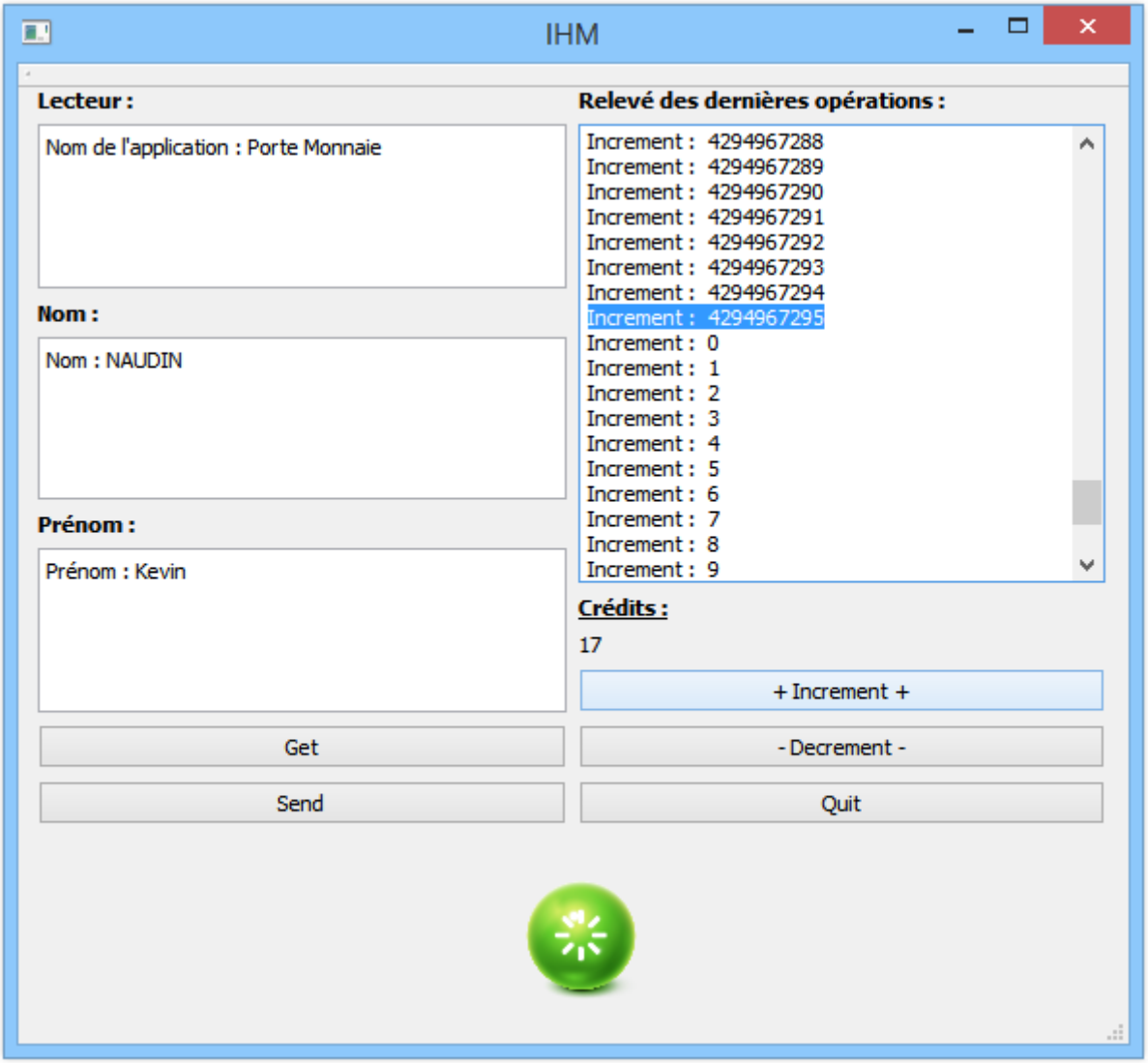


Figure 13 : Limite de l'incrémentation

3) DÉCRÉMENTATION

```
1. void IHM::on_Decrement_Button_clicked()
2. {
3. readIncrementDecrementStatus();
4. uint32_t value =0;
5. status = Mf_Classic_Decrement_Value(&cardReader,TRUE, 14, 1, 13, Auth_KeyA, 3);
6. status = Mf_Classic_Restore_Value(&cardReader, TRUE, 13, 14, Auth_KeyA, 3);
7. status = Mf_Classic_Read_Value(&cardReader, TRUE, 13, &value, Auth_KeyA, 3);
8. std::cout << status << std::endl;
9. status = Mf_Classic_Read_Value(&cardReader, TRUE, 14, &value, Auth_KeyA, 3);
10.     std::cout << status << std::endl;
11.
12.     QString displayValue = QString::number(value);
13.
14.     this->ui->textSend->setText("Decrement :   "+ displayValue+ "\n");
15.     this->ui->textSend->update();
16.     this->ui->creditLabel->setText(displayValue);
17.     this->ui->creditLabel->update();
18.     }
```

Pour la décrémentation, il en va de même que l'incrémentation, voici un schéma de fonctionnement de la méthode Mf_Classic_Decrement_Value(…) :

- Decrement :
Mf_Classic_Decrement_Value(&cardReader, TRUE, 14, 1, 13, Auth_KeyA, 3);

Secteur 3	Block 13 (Backup) = 13
	Block 14 (value) = 14

- Restore :
Mf_Classic_Restore_Value(&cardReader, TRUE, 13, 14, Auth_KeyA, 3);

Secteur 3	Block 13 (Backup) = 13
	Block 14 (value) = 13

Afin de créditer ou débiter une valeur, l'utilisateur peut se servir des deux boutons vus en détail précédemment ayant pour rôle d'augmenter ou de diminuer par pas de 1 la valeur de 'value'. L'appui sur une des deux touches validera l'action. Le fait d'appuyer sur la touche opposée permettra de revenir à la valeur précédente en cas d'erreur de transaction.

V – CONCLUSION

A travers ce TP, nous avons approfondis nos connaissances sur le sujet des lecteurs RFID. De la manipulation de l'IDE Qt Creator en C++ à la manipulation du materiel provenant de l'entreprise Odalid, nous avions pu assembler petit à petit les pièces de manière à pouvoir réaliser un produit fini et fonctionnel.
La technologie RFID est une technologie très attractive pour les entreprises qui offre la possibilité d'une gestion automatique du nombre conséquent d'informations qu'elle doit traiter.
Cependant, le champ d'application privilégié des RFID dans le futur pourrait bien être l'IoT : l'internet des objets. Il s'agit du croisement entre le monde réel et le monde virtuel. Autrement dit, on applique un identifiant unique à chaque objet afin de pouvoir l'identifier rapidement. Mais les puces RFID permettront bien d'autres perspectives. Elles pourront stocker des données et communiquer en temps réel avec les autres objets, avec son propriétaire, avec les utilisateurs etc. La technologie RFID, le début d'une nouvelle ère technologique.

CANDAPPANE Vincent
NAUDIN Kevin
4A – Infotronique