RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE

MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITÉ IBN-KHALDOUN DE TIARET

FACULTÉ DES SCIENCES APPLIQUEES

DÉPARTEMENT DE GENIE ELECTRIQUE

MEMOIRE DE FIN D'ETUDES

Pour l'obtention du diplôme de Master

Domaine : Sciences et Technologie

Filière : Electronique

Spécialité : Electronique des Systèmes Embarqués

THÈME

Réalisation d'un programmateur de PIC Partie Firmware & Software

Préparé par : M. BELMESSAOUD MOHAMED NADIR

Devant le Jury :

Nom et prénoms	Grade	Université	Qualité
SEBAA Morsli	MCA	U-Tiaret	Président
BELARBI Mustapha	MCA	U-Tiaret	Encadreur
BENABID Houari	MAA	U-Tiaret	Examinateur

PROMOTION 2018 /2019

REMERCIEMENT

D'abord et avant tout, Grâces vont à Allah pour m'avoir béni avec la patience, la connaissance et les moyens nécessaires pour accomplir ce travail.

Mes plus sincères remerciements à mon encadreur. Dr. BELARBI Mustapha, dont le soutien et les conseils ont rendu ce mémoire possible. J'aimerais également exprimer mes remerciements aux membres du jury, qui ont pris le temps d'évaluer ce travail.

Je voudrais exprimer ma gratitude envers ma famille, en particulier ma chère maman, ma précieuse grand-mère et ma tante Hanane qui a toujours été d'un grand soutien.

Mes remerciements et mon appréciation vont également à mes camarades de classe pour avoir fait de l'université une deuxième maison, ainsi qu'à mes amis dont je peux toujours compter sur l'aide.

Ce mémoire est dédié à moi-même
à ma famille
à mes amis
et à tous ceux qui ont contribué à sa rédaction.

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

LISTINGS

ACRONYMES

ADC	Analog to Digital Converter
API	Application Programming Interface
CAN	Controller Area Network
CCP	Capture/Compare/PWM
CPU	Central Processing Unit
DC	Direct Current
DMA	Direct Memory Access
DSC	Digital Signal Controller
DSP	Digital Signal Processing
EEPROM	Electrically Erasable Programmable Read-Only Memory
HID	Human Interface Device
HVP	High Voltage Programming
I^2C	Inter-Integrated Circuit
ICSP	In-Circuit Serial Programming
ISP	In System Programming
JAR	Java ARchive
JDK	Java Development Kit
JNA	Java Native Access
JTAG	Joint Test Action Group
JVM	Java Virtual Machine
LVP	Low Voltage Programming
$\overline{\text{MCLR}}$	Master Clear
MCU	MicroController Unit
MIPS	Million Instructions Per Second
PC	Personal Computer
PID	Product ID
PLL	Phase Locked Loop
PWM	Pulse Width Modulation
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
SFR	Special Function Registers
SGBD	Système de Gestion de Base de Données

SPI	Serial Peripheral Interface
SQL	Structured Query Language
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
VID	Vendor ID

INTRODUCTION

INTRODUCTION

La prévalence des composants numériques programmables dans l'électronique d'aujourd'hui est irréfutable. leur utilisation facilite la mise en œuvre de différents schémas électroniques et rend possible la reprogrammation d'un même équipement pour atteindre divers objectifs et par conséquent réduire le coût et le temps de fabrication d'un produit. Il en résulte des systèmes plus complexes et performants pour un espace réduit.

En raison de leur abondance, de leur prix relativement réduit et de leur simplicité, aucun autre composant n'a été plus dominant dans ce domaine que les microcontrôleurs, ces derniers sont des microprocesseurs intégrés avec d'autres périphériques différents dans une seule puce.

Les périphériques que l'on trouve couramment dans les microcontrôleurs comprennent notamment les suivants :

- Les mémoires (Flash, RAM, EEPROM, . . .)
- Timers (8 bit, 16 bit)
- PWM
- Mode de capture
- Convertisseur A/N
- Interface E/S paralléle
- Interface de communication série asynchrone (UART), synchrones (SPI, I^2C, . . .)

Les microcontrôleurs interagissent et supervisent leurs environnements selon un programme dont les instructions, écrites en langage machine, sont obtenues à l'aide d'un compiler ou d'un assembler. Ces derniers génèrent un fichier HEX contenant le code machine à partir d'un langage de niveau haut (e.g., Assembly, C, C++, Ada, . . .).

Un dispositif appelé programmateur est utilisé pour charger le contenu du fichier HEX dans la mémoire ROM incorporée dans le microcontrôleur, d'écrire les données permanentes dans l'EEPROM, d'effacer et de vérifier leur contenu. Une variété de ces programmateurs est disponible, certains sont dotés de fonctions de débogage, d'autres communiquent avec le logiciel hôte sur l'ordinateur à travers le port USB ou bien le port série, alors qu'on trouve d'autres qui utilisent le port parallèle. Cette communication bidirectionnelle permet de faire l'interface entre la personne et le microcontrôleur.

Cependant, et pour un appareil de cette importance, il semble évident qu'il n'y a pas suffisamment de documentation adéquate qui détaille son comportement, ce manque de clarté peut rendre difficile la réalisation de certains circuits tels que les cartes de développement et de prototypage basées sur le microcontrôleur PIC, ainsi que la conception de versions simplifiées des programmateurs disponibles sur le marché, qui pourraient être beaucoup plus économiques et accessibles pour les étudiants.

Le but de ce mémoire est de lever l'ambiguïté qui entoure ces dispositifs, en se préoccupant des programmateurs des microcontrôleurs PIC de Microchip, et plus spécifiquement les programmateurs USB. Ces derniers sont eux-mêmes construits autour d'un microcontrôleur,

ce qui implique le développement d'un firmware approprié qui, d'une part, assure la communication avec le logiciel hôte et, d'autre part, génère les différents signaux nécessaires pour programmer le microcontrôleur cible.

Le premier chapitre sera consacré à la présentation des bases concernant les microcontrôleurs PIC et leurs méthodes de programmation, le deuxième chapitre décrira le processus de programmation du point de vue du firmware, le troisième chapitre sera consacré au développement du logiciel en utilisant le langage de programmation Java et le framework JavaFX et le dernier chapitre se concentrera sur l'implémentation d'un bootloader, qui est une approche qui permet à un microcontrôleur d'écrire ses propres régions de mémoire sans avoir besoin de programmateurs externes. .

CHAPITRE 1

1

APERÇU GÉNÉRAL DES PIC ET DE LEUR PROGRAMMATION

1.1 INTRODUCTION

Les PIC sont des microcontrôleurs développés par Microchip. Ils utilisent une architecture Harvard ou Harvard modifée et un jeu d'instructions RISC, le premier de cette ligne était le PIC16C84, il a été introduit en 1993 et il intégre un algorithme de programmation en série et une mémoire EEPROM. Ils sont largement utilisés dans différentes industries en raison de leurs performances élevées et de leur faible consommation d'énergie. Ils sont également très populaires parmi les amateurs en raison de leur coût modéré et de la disponibilité de leurs outils logiciels comme les compilateurs, simulateurs et débogueurs.

Il convient de mentionner que Microchip n'a jamais utilisé PIC comme acronyme, et récemment ils ont commencé à appeler leurs microcontrôleurs PICmicro MCU.

1.2 GÉNÉRALITÉS SUR LES MICROCONTRÔLEURS PIC

Historiquement, La division Microelectronics de General Instruments a développé un MCU appelé PIC1650 en 1975, décrit comme « Programmable Intelligent Computer » ou Ordinateur Intelligent Programmable. Ce microcontrôleur est le prédécesseur de tous les PIC. il a été conçue comme un périphérique pour leur microprocesseur CP1600.

En 1985, General Instruments a converti sa division Microelectronics en Microchip Technology. Cette dernière a lancé le PIC16C84 en 1993, et sa version améliorée, le PIC16F84 en 1998. Depuis lors, Microchips a produit différents contrôleurs avec différentes architectures pour une variété d'applications, les plus populaires sont les suivants :

- PIC12F683 : un petit dispositif à 8 broches. C'est un bon microcontrôleur pour les petites applications en raison de sa petite taille, de sa puissance relativement élevée et de ses diverses caractéristiques, comme 4 canaux ADC et un oscillateur interne 4MHZ.
- PIC16F1936 : équipé d'un ADC 11 canaux, 10 bits, XLP (eXtreme Low Power) pour une faible consommation d'énergie, considéré comme un bon et peu coûteux remplacement pour le PIC16F877.
- PIC16F628A : un substitut au PIC16F84A discontinué, il est compatible avec ce dernier, mais dispose de plus de périphériques tels qu'un UART et une mémoire Flash plus grande de 3,5 KB pour un prix moins cher en comparaison.
- PIC16F88 : une version améliorée du PIC16F268A, il possède toutes les fonctionnalités du 16F628A, avec une mémoire de programme plus grande qui permet l'autoprogrammation (self-programming).

- PIC16F877A : c'est probablement le PIC le plus populaire utilisé par la communauté des hobbyistes. Dispose de 14 KB de mémoire de programme, 368 bytes de RAM, un boîtier de 40 broches, 2 modules CCP, 8 canaux ADC 10-bits.
- PIC18F4550 : l'un des dispositifs les plus utilisés dans la famille PIC18, dispose d'une interface USB full speed, 13 canaux ADC, vitesse jusqu'à 12 MIPS et une mémoire flash de 32 KB.

1.2.1 *Les différentes architectures*

Actuellement, une grande variété de microcontrôleurs est disponible, selon la taille de leurs registres et de leur bus de données, on peut distinguer les architectures suivantes :

1.2.1.1 *Architecture 8-bit*

Ces microcontrôleurs peuvent être classés dans l'une des 4 catégories ci-après :

- Baseline : mot instruction de 12 bits, comprend PIC10 et certains microcontrôleurs des famillies PIC12 et PIC16, ils contiennent quelques périphériques comme un comparateur, ADC 8-bit, mémoire donnée, oscillateur interne.
- Mid-Range : mot instruction de 14 bits, cette catégorie inclue les PIC12 et PIC16, en plus des caractéristiques mentionnées dans la catégorie baseline, celle-ci comprend les interfaces SPI, I^2C, UART, modules PWM et ADC 10- bits.
- Enhanced Mid-Range : mot instruction de 14 bits, cette catégorie inclue les familles 12F1xxx et 16F1xxx, ces microcontrôleurs sont compatibles avec ceux de la catégorie mid-range, ils comprennent certaines améliorations telles qu'une consommation d'énergie réduite, une mémoire de programme plus large et des fréquences de fonctionnement plus élevées
- High-End : mot instruction de 16 bits, comprend la famille PIC18, ces microcontrôleurs sont équipés de périphériques plus avancés tels que USB, CAN, Ethernet et ADC 12-bits.

Le Tableau 1.1 résume les principales différences entre les catégories.

1.2.1.2 *Architecture 16-bit*

Deux familles utilisent cette architecture , PIC24 et dsPIC, les microcontrôleurs de cette architecture ont un mot instruction de 24 bits et un compteur ordinal de taille 23 bits capable d'adresser jusqu'à 4 MB de la ROM. Cependant, la mémoire implémentée varie d'un dispositif à l'autre. L'accès à la mémoire de données s'effectue à l'aide d'une unité de génération d'adresse qui accède la RAM entière comme un seul espace linéaire.

Le PIC24 dispose de 76 instructions, avec une vitesse d'exécution allant jusqu'à 40 MIPS, un DMA à 8 canaux permettant le transfert de données entre la RAM et les SFR des périphériques avec une intervention minimale du CPU, jusqu'à neuf timers 16 bits et jusqu'à deux de chacun des modules de communication (SPI, I^2C, UART, CAN).

	BASELINE	MID-RANGE	ENHANCED MID-RANGE	HIGH-END
Nombre de broches	6-40	8-64	8-64	18-100
Interruptions	Non	Capacité d'interruption unique	Capacité d'interruption unique avec sauvegarde du contexte en matériel	Capacité d'interruptions multiple avec sauvegarde du contexte en matériel
Performance	5 MIPS	5 MIPS	8 MIPS	Jusqu'à 16 MIPS
Instructions	33	35	49	83
Mémoire programme	Jusqu'à 3 KB	Jusqu'à 14 KB	Jusqu'à 28 KB	Jusqu'à 128 KB
Mémoire données	Jusqu'à 138 Bytes	Jusqu'à 368 Bytes	Jusqu'à 1.5 KB	Jusqu'à 4 KB
La pile	2 niveaux	8 niveaux	16 niveaux	32 niveaux

Tableau 1.1 – Comparaison des différentes catégories des PIC 8-bit.[1]

Le dsPIC DSC est basé sur le PIC24 , mais inclut également d'autres fonctionnalités pour supporter les opérations du DSP, 19 instructions supplémentaires sont ajoutées et la RAM est accessible via deux bus de 16 bits qui permettent de récupérer simultanément les opérandes dans les opérations de multiplication.

1.2.1.3 *Architecture 32-bit*

Les membres de la famille PIC32 sont destinés à des applications de calcul intensif qui nécessitent une vitesse d'éxecution élevée, ils sont équipés d'une interface JTAG pour la programmation et le débogage. Microchip a introduit 4 sous-familles du PIC32.

- PIC32MM : il s'agit du plus petit membre de la famille PIC32, il utilise des instructions 16 bits et 32 bits, il se compose de la série GPL qui offre la plus faible consommation d'énergie et de la série GPM qui dispose d'interface USB, une mémoire plus grande et plus de broches.
- PIC32MX : ces microcontrôleurs disposent d'un ensemble de périphériques beaucoup plus riche et d'une vitesse de fonctionnement supérieure à celle du PIC32MM.
- PIC32MZ : ces dispositifs sont parmi les contrôleurs les plus performants de la famille PIC32, ils sont équipés d'un ensemble de périphériques avancés qui comprend un FPU (Floating Point Unit) double précision, un GPU (Graphics Processing Unit) et une vitesse de 252 MHz.
- PIC32MK : ils sont destinés au domaine industriel tel que le contrôle des moteurs et les applications industrielles d'Internet des Objets.

1.2.2 *Outils matériels et logiciels*

Les microcontrôleurs sont utilisés conjointement avec une variété d'outils différents, tant logiciels que matériels, allant des éditeurs de texte utilisés pour écrire le code source, aux compilateurs utilisés pour générer le code machine, aux programmateurs qui servent à le charger en mémoire et les débogueurs qui permettent de vérifier le fonctionnement adéquat du programme. Les microcontrôleurs PIC ne sont pas différents, une myriade d'outils sont disponibles auprès de Microchip et d'autres développeurs tiers, qui peuvent être classés dans l'une des catégories suivantes :

1.2.2.1 *Les compilateurs*

Généralement, les fichiers HEX des systèmes embarqués sont obtenus à l'aide d'un compilateur appelé cross-compiler, contrairement aux compilateurs natifs, un cross-compiler génère du code machine pour une plate-forme autre que celle sur laquelle le compilateur est déployé. Cross-assemblers, peuvent être considérés comme des compilateurs, mais ceux qui font une tâche plus simple qui se réduit au mappage direct d'une instruction d'assembleur aux opcodes correspondants. [2]

La gamme de compilateurs XC est disponible chez Microchip, elle comprend le compilateur XC8 qui supporte toutes les familles 8 bits, le XC16 pour le PIC24 et le dsPIC, et le XC32 pour les contrôleurs 32 bits. Les compilateurs tiers comprennent le compilateur IAR C/C++ de IAR Systems, Hi-Tech PIC C, MicroC et CCS PIC C.

Tous les compilateurs mentionnés précédemment sont des compilateurs du langage C, qui est de loin le langage le plus populaire dans le domaine des systèmes embarqués, l'une des principales raisons pour cela est que C permet le contrôle direct et la gestion de la mémoire sans connaissance approfondie du matériel requis par un langage tel que l'assembleur. Mais il y a des efforts pour apporter des langages de plus haut niveau à la programmation des PIC tels que Python avec Pyastra.

L'assembleur est utilisé en complément du langage de haut niveau, généralement uniquement pour les petits morceaux de code qui doivent être extrêmement efficaces ou compactes, i.e., inline-assembly. Cependant, pour les fichiers sources écrits entièrement en assembleur, un assembler tel que MPASM ou GPASM peut être utilisé pour générer le code exécutable.

1.2.2.2 *Les Simulateurs*

Un simulateur permet d'exécuter un programme sans avoir le matériel réel. Le contenu des registres, la mémoire interne et le code source sont affichés dans des fenêtres séparées. L'utilisateur peut définir des points d'arrêt pour le programme et examiner les résultats de l'exécution. Le logiciel de développement MPLAB intègre un simulateur pour différents PIC. Cependant, des outils plus puissants et polyvalents peuvent être utilisés, tels que Proteus de Labcenter.

1.2.2.3 *Les débogueurs en-circuit*

Ces outils sont utilisés pour le débogage au niveau matériel, ils permettent de démarrer l'exécution, d'arrêter, d'exécuter le programme une étape à la fois et d'insérer des points d'arrêt pendant que le microcontrôleur est intégré dans le circuit réel. L'utilisateur peut avoir accès en lecture et en écriture à différentes zones de mémoire. Certains des débogueurs utilisés pour les microcontrôleurs PIC sont l'ICD2, l'ICD3 et PICKit 3.

1.2.2.4 *Les cartes de développement*

Les cartes de démonstration ou de développement sont utiles pour apprendre le microcontrôleur et tester le programme avant que le matériel final ne soit terminé. En tant qu'outil d'apprentissage, une carte de développement bien conçue devrait permettre à l'utilisateur d'expérimenter toutes les fonctions de chaque périphérique.[3]

1.2.2.5 *Les programmateurs*

Un programmateur est un dispositif matériel accompagné d'un logiciel qui est utilisé pour transférer le code du langage machine au microcontrôleur à partir du PC.

Certains programmateurs exigent que le microcontrôleur cible soit retiré de son circuit et placé dans celui-ci, généralement dans un socket ZIF (Zero Insertion Force), ce qui est problématique pour les microcontrôleurs SMD (Surface Mount Device) qui n'ont pas de socket standard et qui peut aussi rendre difficile la mise à jour du firmware dans le cas des dispositifs THT (Through Hole Technology) . Cependant, les microcontrôleurs plus modernes offrent la possibilité d'être programmer dans leur circuit final, cette caractéristique est appelée In System Programming (ISP).

Il existe principalement quatre types de programmateurs, USB, série, parallèle et bootloaders. les trois premiers fonctionnent de la même manière, ils génèrent un flux de données série en utilisant deux lignes de signaux horloge et données. Une autre broche délivre la tension de programmation et deux autres fournissent l'alimentation et la masse. Dans le dernier cas, le microcontrôleur est préchargé avec un programme appelé bootloader, l'application hôte envoie le contenu du fichier HEX au bootloader qui écrit le contenu des mémoires internes en conséquence.

1.3 LA PROGRAMMATION DES PIC

La programmation dans ce contexte fait référence au processus de chargement du code exécutable dans le microcontrôleur et non pas au développement du code source. En général, six connexions sont nécessaires pour programmer un PIC.

- V_{DD} & V_{SS} : l'alimentation et la masse respectivement, le microcontrôleur peut être alimenté soit par le circuit dans lequel il est placé, soit par le programmateur.
- V_{PP} : voltage de programmation, cette tension est d'environ 13V, mais elle varie d'une famille de dispositifs à l'autre.

- PGD & PGC : broches de données et d'horloge, les bits de données sont transmis en série et sont lus avec le front. descendant du signal de l'horloge.
- PGM : permet de sélectionner le mode de programmation.

Les algorithmes utilisés pour écrire et lire les données du PIC sont détaillés dans différents documents appelés programming specifications. Ces documents seront référencés tout au long du développement du logiciel et du firmware, mais l'une des principales complications est qu'il n'existe pas d'ensemble standard d'algorithmes utilisés pour tous les PIC, chaque document est plutôt destiné à un petit sous-ensemble de microcontrôleurs, ce qui peut rendre la conception d'un programmateur universel difficile.

1.3.1 *In-Circuit Serial Programming (ICSP)*

La programmation série en circuit est une technique ISP améliorée mise en œuvre dans les microcontrôleurs PICmicro de Microchip. L'utilisation de seulement deux broches d'E/S pour les données d'entrée et de sortie en série rend le ICSP facile à utiliser et moins intrusif pour le fonctionnement normal du MCU.[4]

la plupart des PICmicros peuvent être programmés avec ICSP dans l'un des deux modes suivants :

1.3.1.1 *High Voltage Programming (HVP)*

Dans ce mode, la programmation s'effectue en maintenant les broches PGD et PGC à un niveau bas, puis en augmentant la tension de la broche $\overline{\text{MCLR}}$ à V_{IHH} (la tension de programmation en mode HVP) , cette dernière peut varier selon les familles, le Tableau 1.2 résume les tensions de programmation pour quelques familles de microcontrôleurs.

Il est clair d'après le Tableau 1.2 que la plupart des microcontrôleurs nécessitent une tension de programmation plus élevée que le 5V fourni par l'USB. Par conséquent, les programmateurs USB doivent appliquer un circuit supplémentaire, en particulier un convertisseur DC-DC qui peut générer la tension de programmation à partir des 5V disponibles, ce convertisseur peut être soit un boost ou une pompe de charge.

les programmateurs qui utilisent le port série peuvent résoudre ce problème dans la plupart des cas, puisque les tensions de communication série peuvent être comprises entre $\pm 3V$ et $\pm 25V$, un niveau logique haut est représenté par une tension négative et une tension positive représente un niveau logique bas, sur la plupart des ordinateurs, ces tensions sont approximativement -12V et +12V respectivement. Par conséquent, maintenir la ligne Tx à un niveau logique bas peut être suffisant pour générer la tension V_{IHH} pour de nombreux microcontrôleurs. Ceci peut être fait en envoyant un signal 'BREAK' sur la ligne Tx qui le maintiendra à un '0' logique.

Cependant, et puisque l'objectif est de développer un programmateur USB, l'implémentation d'un circuit de boost est nécessaire, ceci va générer une plage de tensions qui va varier selon la fréquence et le rapport cyclique du signal PWM commandant le commutateur du boost, la Figure 1.1 montre l'interface du circuit de boost avec le microcontrôleur embarqué dans le programmateur.

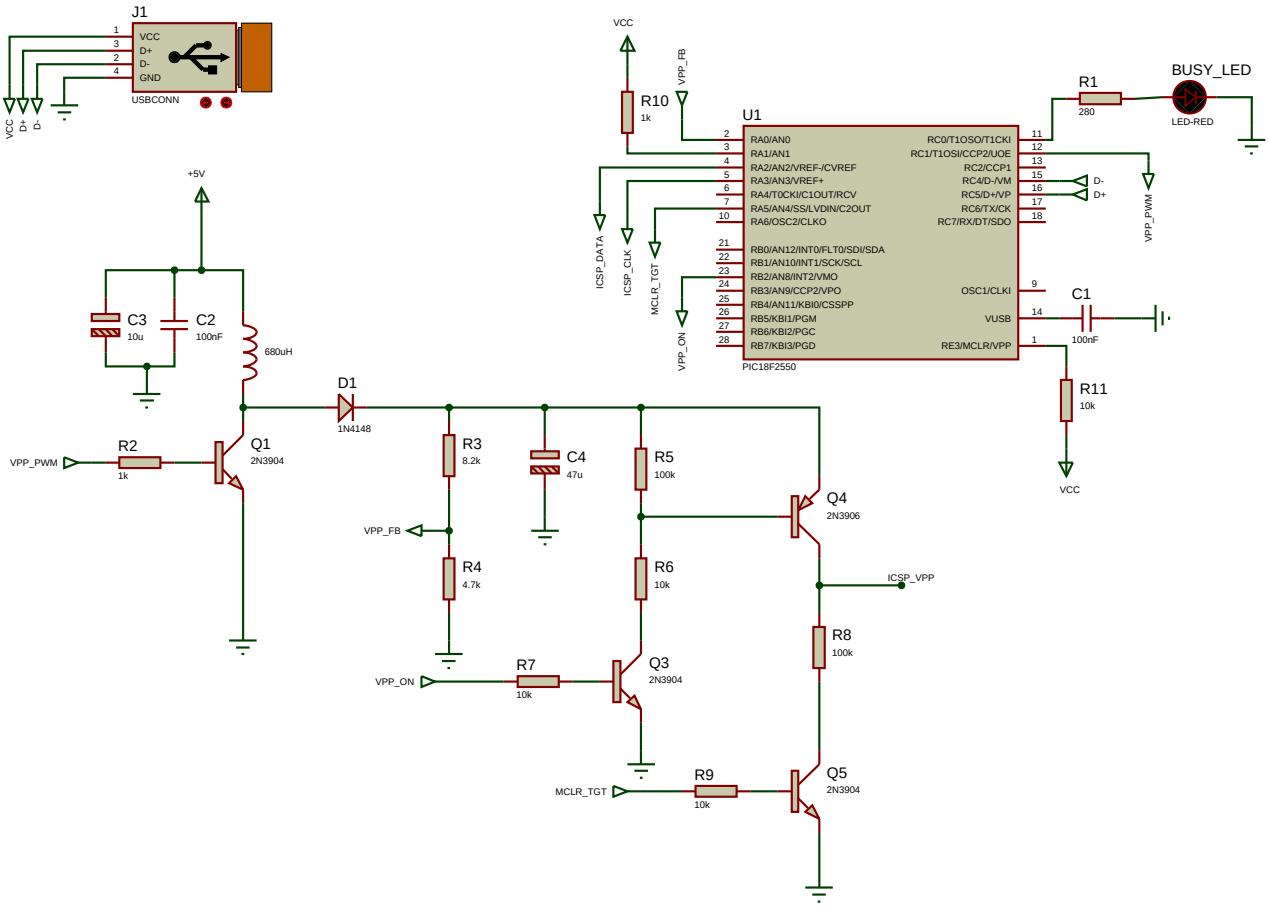

FIGURE 1.1 – L'interfaçage du boost avec le MCU du programmateur

1.3.1.2 *Low Voltage Programming (LVP)*

Aussi appelé programmation à alimentation unique, ce mode permet de programmer le MCU à l'aide de la tension V_{DD} du microcontrôleur, ce qui permet d'éliminer les circuits supplémentaires et les risques liés à l'utilisation de tensions relativement élevées exigées par le mode HVP.

La programmation en mode LVP est contrôlée par un bit dans les registres de configuration du MCU, Lorsque le bit LVP est mis à '1', la programmation ICSP basse tension est activée. Pour désactiver le mode ICSP basse tension, le bit LVP doit être programmé à '0'. Voici quelques considérations à prendre en compte lors de la programmation en mode LVP :

- Lorsque le bit LVP est mis à '1', la broche PGM est réservée aux opérations de programmation et le microcontrôleur entre en mode programmation lorsque cette broche est à un niveau logique élevé, lorsque le bit LVP est à '0', la broche PGM fonctionne comme une entrées/sorties à usage général.
- Les microcontrôleurs sont fabriqués avec le bit LVP activé.
- Quelque soit l'état du bit LVP, le MCU peut être programmé en mode HVP.

MCU	MIN	MAX	MCU	MIN	MAX
PIC10F2XX	12.5 V	13.5 V	PIC16F88X	10 V	12 V
PIC10(L)F3	8 V	9 V	PIC16HV	10 V	13 V
PIC12F1XXX	8 V	9 V	PIC16LF	8 V	9 V
PIC12LF	8 V	9 V	PIC18F1XXX	9 V	13.25 V
PIC12F5XX PIC12F61X PIC12F635 PIC12F683	10 V	12 V	PIC18F1XK2	5.75 V	9 V
PIC12F629 PIC12F675	12.75 V	13.25 V	PIC18F1XK5	3.3 V	9 V
PIC12HV	10 V	13 V	PIC18F22XX PIC18F23XX	9 V	13.25 V
PIC16F1XXX	8 V	9 V	PIC18F24XX PIC18F25XX PIC18F26XX	9.5 V	12.5 V
PIC16F5XX PIC16F61X PIC16F63X	10 V	12 V	PIC18F44XX PIC18F45XX PIC18F46XX	9.5 V	12.5 V
PIC16F62X PIC16F648A PIC16F676 PIC16F72	12.75 V	13.25 V	PIC18F63XX	10 V	12 V
PIC16F677 PIC16F68X PIC16F690 PIC16F716	10 V	12 V	PIC18FXXJXX	2.125 V	3.5 V
PIC16F72X	8 V	9 V	PIC24EP	3 V	3.5 V
PIC16F73X PIC16F74X PIC16F76X PIC16F77X	12.5 V	13.5 V	dsPIC30F	9 V	13.25 V
PIC16F8XX	12.5 V	13.5 V	dsPIC33EP	3 V	3.5 V

Tableau 1.2 – Les voltages de programmation pour quelques familles de microcontrôleurs PIC

- L'état du bit LVP ne peut pas être modifié lorsque le MCU est en train d'être programmé en mode LVP. Par exemple, si le bit LVP est mis à '0', un programmateur haute tension est nécessaire pour le remettre à '1' et ré-activer cette fonctionnalité.

Bien que le mode LVP semble convaincant, il souffre toujours de deux problèmes principaux :

1. La séquence nécessaire pour faire entrer un microcontrôleur en mode de programmation est assez facile à se produire pendant le fonctionnement normal du MCU, ce qui consiste principalement à maintenir PGC et PGD à un niveau bas, placer un niveau logique élevé sur PGM et ensuite élever $\overline{\text{MCLR}}$ à V_{DD}, comme le montre la Figure 1.2.

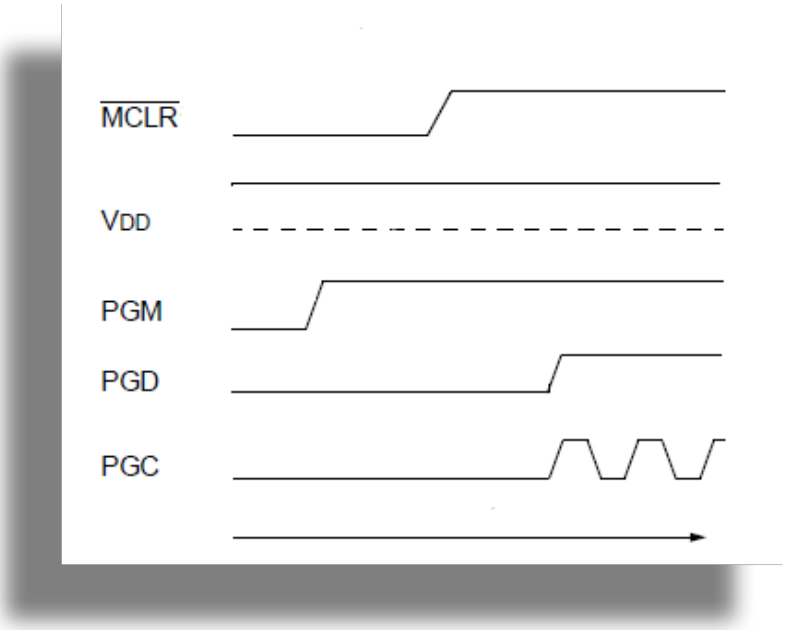

FIGURE 1.2 – Entrée en mode programmation basse tension

   Pour y remédier, PGM est souvent reliée à la masse pendant le temps de fonctionnement, ce qui rendrait toutefois cette broche inutilisable.
2. Un programmateur basse tension est obsolète si le microcontrôleur est préprogrammé avec le bit LVP désactivé.

1.3.2 *Les contraintes de conception*

Lors de la construction d'un circuit qui permet les opérations du ICSP, le concepteur doit tenir compte de plusieurs facteurs et doit compenser les problèmes suivants :

1. Le circuit final doit être conçu pour permettre la connexion directe de tous les signaux de programmation au PIC, ce qui implique l'inclusion d'un connecteur ICSP.
2. Isolation de $\overline{\text{MCLR}}$ du reste du circuit, cette broche est souvent liée à V_{DD} par une résistance. Lors de la programmation du PIC, elle est amenée à environ 13V et également à la masse. Par conséquent, le circuit d'application doit être isolé de cette tension fournie par le programmateur par une diode de type Schottky.
3. La charge sur PGD et PGC, ces broches doivent être isolées du reste du circuit d'application afin de ne pas affecter les signaux pendant la programmation, la connexion de tous composants à ces broches qui nécessitent une quantité importante de courant peut entraîner une distorsion des signaux ICSP de données et de l'horloge.

Il est recommandé par Microchip que PGC et PGD soient dédiées uniquement aux opérations du ICSP [5], et de ne pas être utilisés à d'autres fins dans le circuit d'application. Dans ce cas, ces broches doivent être configurées comme des sorties en fonctionnement normal. Ils ne doivent pas être laissés en tant qu'entrées flottantes. Sinon ils peuvent être isolés des autres parties du circuit par une résistance appropriée.

4. Interférence entre PGD et PGC, sur les microcontrôleurs PIC, ces deux broches sont adjacentes l'une à l'autre, ce qui peut entraîner des interférences entre les deux dans certains cas, en raison de la nature bidirectionnelle du PGD, cela nécessite que le problème soit traité par le circuit cible, dans le cas où le PGD est piloté par le PIC, il s'agit d'une sortie numérique normale. Ceux-ci sont conçus pour passer d'un état à l'autre le plus rapidement possible. Un tel front produit sur le PGD par le PIC cible peut être couplé sur la ligne PGC. Le PIC cible voit alors une impulsion PGC que le programmateur n'a pas émise et la communication série se désynchronise. Cette question est particulièrement importante pour les dsPIC car ils sont plus rapides et ont donc des pilotes de sortie numérique plus puissants et des transitions plus promptes qui se couplent mieux entre signaux. Bien qu'un peu moins grave, ce problème a également été observé chez les PIC18F.[6]

   Ce problème peut être résolu en mettant un condensateur de 22 à 47 pF sur les lignes PGD et PGC à la masse à proximité du microcontrôleur cible. De plus, la mise en série d'une résistance de 100 Ω avec la ligne PGD entre le microcontrôleur cible et le condensateur permet de filtrer le signal PGD lorsqu'il est piloté par le PIC cible. Cela réduit les hautes fréquences qui peuvent se coupler sur la ligne PGC. Un condenseur sur la ligne PGC la rend moins sensible au bruit induit.

En tenant compte des préoccupations mentionnées ci-dessus, un circuit d'application typique qui met en œuvre l'ICSP est illustré dans la Figure 1.3.

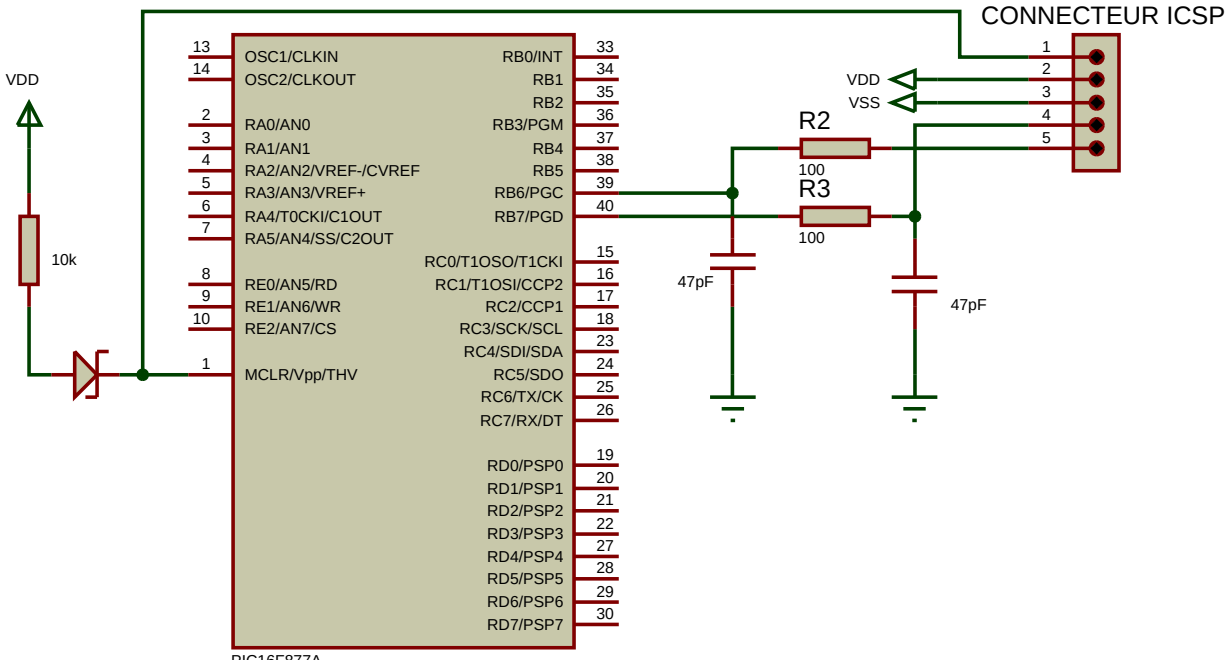

FIGURE 1.3 – Exemple de mise en œuvre du ICSP

1.4 LES PROGRAMMATEURS USB

Les programmateurs USB se distinguent des autres types qui utilisent des ports différents par leur versatilité, leur fiabilité et leur facilité de mise à jour. et dans le fait que les ports USB sont abondants dans tous les ordinateurs de bureau et les ordinateurs portables, ce qui est à la différence des autres ports .

La versatilité des programmateurs USB fait référence à la capacité de construire un programmateur universel avec une aisance relative et même d'ajouter des fonctionnalités de débogage, la fiabilité dans la transmission des signaux de programmation comme prévu au microcontrôleur cible et la facilité de mise à jour se réfère à la simplicité de l'ajout du support des nouveaux microcontrôleurs, qui dans la plupart des cas se fait simplement en ajoutant quelques entrées à la base de données et en mettant le firmware du MCU du programmateur à jour et ne nécessite aucune modification au circuit.

L'intérêt pour les programmateurs USB a été suscité par Microchip qui a lancé le programmateur PICKit 2 en open source. Cette étape avait pour but d'attirer la communauté des amateurs, surtout après la popularité du Arduino, qui était basé sur les microcontrôleurs AVR d'Atmel, cela a en effet donné naissance à de nombreux dispositifs tiers, la plupart commercialisés et peu sont eux-mêmes open source. Certains qui méritent d'être mentionnés sont : LProg, USBProg2, usbpicprog et l'utilisation du PICKit 2 pour programmer les microcontrôleurs AVR avec avrdude.

Tout programmateur USB doit inclure tous les éléments suivants :

- Un API qui gère la communication USB aux deux extrémités (PC et PIC).
- Une base de données ou une autre forme de stockage de données qui préserve les informations relatives aux différents microcontrôleurs.
- Un microcontrôleur qui sera incorporé dans le programmateur.

Une vue d'ensemble des composants qui constituent le programmateur qui sera développé dans ce mémoire est présentée dans la Figure 1.4.

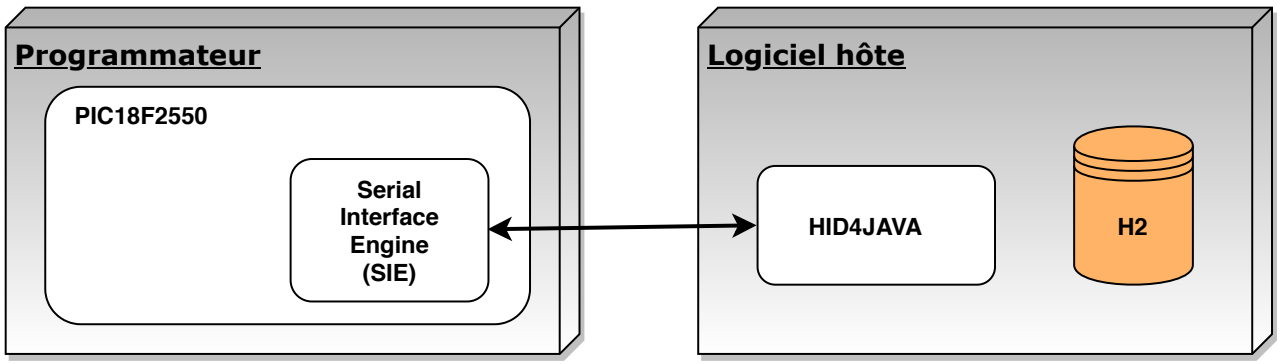

FIGURE 1.4 – Les constituants du programmateur

1.4.1 *hid4java*

Bien qu'il fasse partie intégrante de nombreux ordinateurs, Java ne supporte pas officiellement l'USB. Pour qu'un programme Java puisse interagir avec des périphériques USB, il faut donc un API Java/USB tiers. L'API utilisée par le logiciel du programmateur pour communiquer via USB est hid4java [7], ce dernier est une bibliothèque Java multiplateforme qui est basée sur hidapi qui permet à une application d'interfacer avec USB et Bluetooth HID-Class sous Windows, Linux, FreeBSD, et Mac OS X. Et utilise Java Native Access (JNA) pour accéder aux bibliothèques du système d'exploitation natif.

hid4java fournit les fonctions HID les plus fréquemment utilisées, y compris l'envoi de rapports et le blocage des lectures avec des timeouts, ainsi que les événements qui permettent à une application de répondre au matériel qui a été placé ou retiré. Ces fonctions sont utilisées pour communiquer avec les périphériques de la classe Human Interface Device (HID) du standard USB.

La classe HID comprend les claviers, les souris et les contrôleurs de jeu. Avec ces appareils, l'hôte lit et agit sur les entrées humaines telles que les pressions de touches et les mouvements de souris. Les hôtes doivent répondre assez rapidement pour que les utilisateurs ne remarquent pas un délai entre une action et la réponse attendue. Outre les claviers, les souris et les manettes, la classe HID comprend des panneaux avec boutons, interrupteurs, curseurs, les télécommandes et les claviers de téléphone.

Les dispositifs HID sont définis au niveau de l'interface. Dans le descripteur d'interface, bInterfaceClass = 03h pour indiquer la classe HID et ils communiquent en échangeant des rapports en utilisant des transferts de contrôle et d'interruption.[8]

1.4.1.1 *Inclusion de hid4java dans un projet Java*

hid4java est un projet Maven, les étapes nécessaires pour l'intégrer dans un project qui n'utilise pas ce dernier peuvent paraître moins évidentes. Ce qui suit décrit les étapes qui ont été suivies pour inclure l'API dans l'application hôte :

1. Téléchargez hid4java depuis Github sous la forme d'un fichier .ZIP, et enregistrez-le quelque part sur l'ordinateur, sur le bureau par exemple, puis extrayez les fichiers.
2. Téléchargez et installez Maven, après cela vérifiez s'il est installé correctement en tapant la commande montrée dans le Listing 1.1, le résultat devrait être similaire à celui montré ci-dessous.

```
Microsoft Windows [Version 6.3.9600]
Copyright (c) 2013 Microsoft Corporation. All rights reserved.

C:\User\Foo\Desktop> mvn --version
Apache Maven 3.5.4 (1edded0938998edf8bf061f1ceb3cfdeccf443fe; 2018-06-17T19:33:1
Maven home: C:\apache-maven-3.5.4\bin\..
Java version: 1.8.0_171, vendor: Oracle Corporation, runtime: C:\Program Files\J
Default locale: en_US, platform encoding: Cp1252
OS name: "windows 8.1", version: "6.3", arch: "amd64", family: "windows"
```

Listing 1.1 – Vérification de l'installation de Maven

3. Ouvrez le CMD sous Windows ou le Terminal sous Linux, et naviguez vers l'emplacement du dossier hid4java puis saisissez-y la commande "mvn clean install", comme indiqué dans Listing 1.2.

```
C:\User\Foo> cd Desktop
C:\User\Foo\Desktop> cd hid4java
C:\User\Foo\Desktop\hid4java> mvn clean install
```

Listing 1.2 – Building hid4java

4. Si tout se passe bien, un "BUILD SUCCESS" sera affiché, et un nouveau dossier appelé "target" apparaîtra à l'intérieur duquel se trouvent les deux fichiers .JAR que nous devons y incorporer dans notre projet.
5. Ensuite, téléchargez JNA comme un fichier JAR, la version 4.5.2 semble bien fonctionner dans ce projet contrairement aux versions précédentes.
6. Enfin, ajoutez les trois fichiers au projet, sur NetBeans par un clic droit sur le nom du projet, puis Properties > Libraries > Add JAR/Folder comme le montre la Figure 1.5, puis appuyez sur OK.

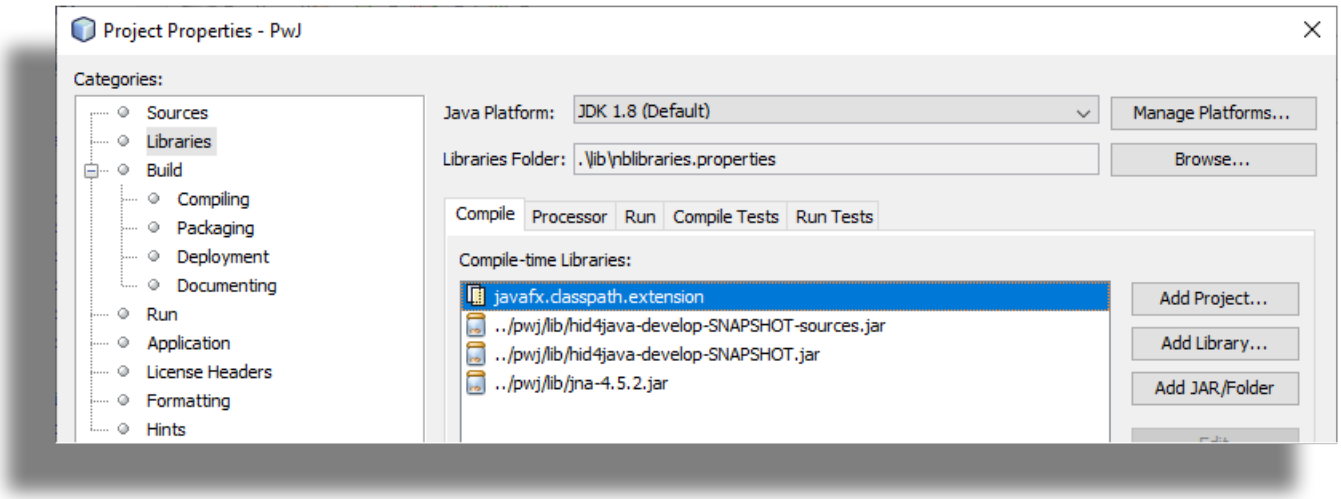

FIGURE 1.5 – Ajout de fichiers JAR dans NetBeans

Après avoir terminé les étapes mentionnées ci-dessus, hid4java devrait être prêt à être utilisé.

1.4.1.2 *Les alternatives*

Tout au long du développement du logiciel, la communication USB a été tentée en utilisant de nombreuses bibliothèques, hid4java étant celle qui fonctionnait de manière cohérente. Cependant, dans le cas où l'utilisateur rencontrerait des difficultés à l'intégrer dans son projet, certaines des alternatives qui peuvent être utilisées sont :

- javahidapi
- PureJavaHidApi
- usb4java

1.4.2 *Base de données H2*

Comme la plupart des autres applications, la base de données fait partie intégrante du logiciel hôte. Par conséquent, la mise en œuvre d'un Système de Gestion de Base de Données (SGBD) adéquat est cruciale pour assurer l'exécution correcte du logiciel.

Une base de données est un ensemble de données apparentées organisées de manière à ce que les données puissent être facilement accessibles, gérées et modifiées. Un système de gestion de base de données est un logiciel qui permet la création, la définition et la manipulation d'une base de données, permettant aux utilisateurs de stocker, traiter et analyser les données facilement. Le SGBD nous fournit une interface pour effectuer diverses opérations comme le stockage et la mise à jour des données ainsi que la création de tables dans la base de données. Il assure également la protection et la sécurité des bases de données. Il maintient également la cohérence des données en cas d'utilisateurs multiples.

Le SGBD qui sera utilisé dans l'application est H2 [9] qui est une base de données SQL open source entièrement écrite en Java.

1.4.2.1 *Les différentes technologies de bases de données*

Il existe trois types de technologies de bases de données :

1. Client/serveur : un système de base de données client/serveur est un système pour lequel un processus dédié, le serveur de base de données, traite les demandes d'opérations de base de données provenant de plusieurs applications client. Ces applications peuvent toutes résider sur le même système physique que le serveur de base de données ou sur des systèmes différents. Des exemples de tels systèmes sont : Derby, MySQL et PostgreSQL.
2. En mémoire : un système de base de données en mémoire est un SGBD qui stocke les données entièrement dans la mémoire principale (RAM). Ceci contraste avec les systèmes de base de données traditionnels (sur disque), qui sont conçus pour le stockage de données sur des supports persistants. Parce que travailler avec des données en mémoire est beaucoup plus rapide que d'écrire et de lire depuis un système de fichiers. Certaines des bases de données en mémoire sont SQLite, Hazelcast et Apache Ignite.
3. Embarquées : Une base de données embarquée est une technologie de base de données dans laquelle les solutions de gestion de base de données sont intégrées dans une application, plutôt que fournies en tant que systèmes de base de données séparés. Les fichiers JAR contiennent tous les composants nécessaires aux différentes opérations. Quelques-uns des systèmes populaires sont HyperSQL, Berkley DB et Java DB.

H2 peut être utilisé dans tous les types mentionnés ci-dessus, mais dans ce projet, il sera utilisé en mode embarqué, car le mode client/serveur est destiné à de plus grandes applications avec accès simultané et le mode en mémoire ne persiste pas les données.

1.4.2.2 *L'utilisation de la base de données H2*

Les étapes pour incorporer la base de données H2 dans un programme java sont les suivantes :

1. Télécharger le fichier JAR et l'inclure dans le projet dans NetBeans comme démontré précédemment.
2. Charger le pilote et créer une connexion à partir du code Java, comme indiqué à la ligne 1 du Listing 1.3.
3. la ligne 2 dans Listing 1.3 va tenter de se connecter à la base de données "test", si la base de données n'existe pas, elle sera créée.

```
1 Class.forName("org.h2.Driver").newInstance();
2 Connection conn = DriverManager.getConnection("jdbc:h2:./test", "nomUtilisateur",
      "motDePasse");
```

Listing 1.3 – Création de la connexion à la base de données

4. H2 inclut une console basée sur Mozilla Firefox, il fournit les outils graphiques pour interagir avec la base de données et un espace pour exécuter des requêtes SQL. Il peut être utilisé en téléchargeant et en installant H2 setup et il est accessible en double-cliquant sur le fichier JAR de H2, la base de données est accessible en précisant son emplacement, le nom d'utilisateur et le mot de passe.

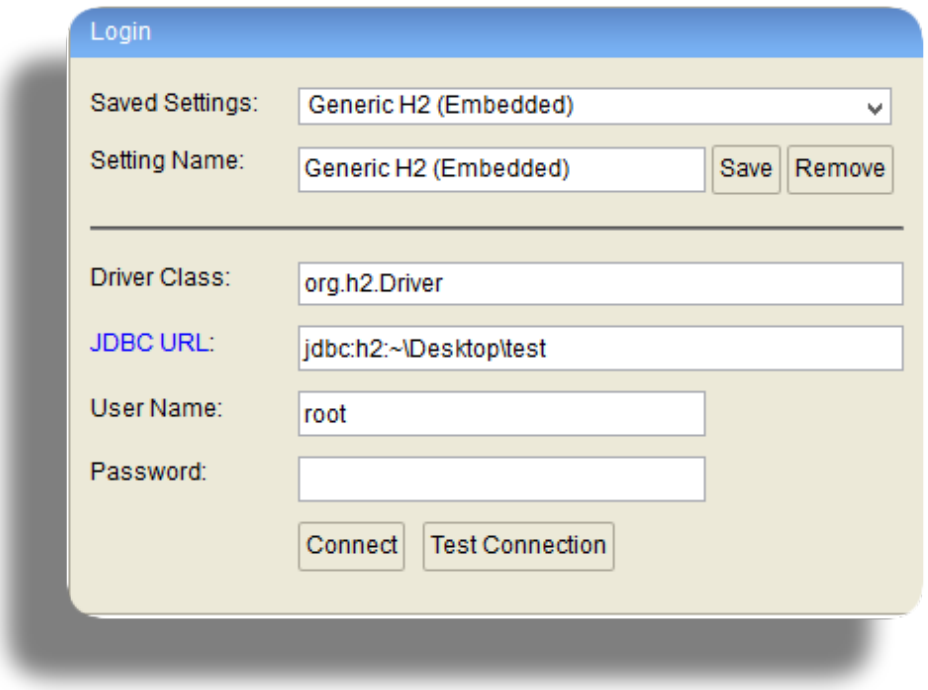

FIGURE 1.6 – La console H2

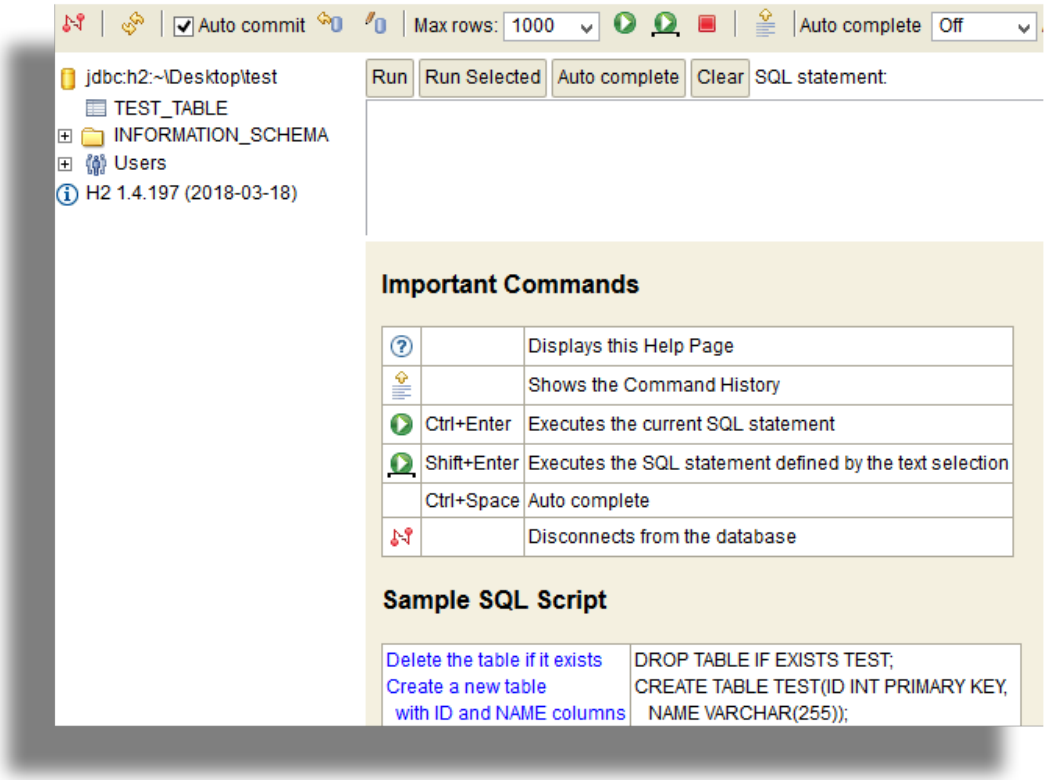

FIGURE 1.7 – L'espace de travail de H2

1.4.3 *Le PIC18F2550*

Comme indiqué précédemment, les programmateurs USB sont eux-mêmes construits autour d'un microcontrôleur, ce MCU gère la communication avec l'hôte, génère le PWM qui va piloter le boost et les différents signaux ICSP.

Le microcontrôleur que nous allons utiliser est le PIC18F2550. Puisque ce MCU répond aux exigences requises pour le développement de ce projet, principalement, une interface USB, un module CCP, module ADC, une mémoire de programme suffisamment grande pour le firmware.

1.4.3.1 *Caractéristiques*

Le PIC18F2550 est un MCU 8 bits de la famille des microcontrôleurs High-end, il intègre un large éventail de fonctionnalités, celles qui nous intéressent le plus sont :

- USB Low-Speed et Full-Speed.
- 24 broches E/S programmables.
- multiplicateur 8x8.
- ADC avec une résolution de 10 bits.
- 2 sorties PWM.

- 32 KB de ROM, 2 KB de RAM et 256 bytes d'EEPROM.
- code compatible avec le populaire PIC18F4550

1.4.3.2 *Développement pour le PIC18F2550*

L'obtention du code exécutable pour le microcontrôleur comme pour tout système programmable moderne nécessite un environnement de développement, nous utiliserons le compilateur PIC C de Custom Computer Services (CCS) qui fournit plusieurs des fonctionnalités sur lesquelles repose le développement du firmware. dont certains sont :

- L'insertion de données constantes dans l'espace ROM à une adresse spécifiée, cela peut être fait dans PIC C en utilisant la directive "#rom". L'exemple Listing 1.4 place les chiffres de 0 à 9 dans la ROM à partir de l'adresse 0x1400.

```
#rom int8 0x1400 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Listing 1.4 – Utilisation de la directive #rom dans PIC C

- L'adressage des emplacements exacts en RAM avec "#byte" et "#bit", En utilisant le code du Listing 1.5, la variable "`my_var`" sera créée et placée à l'adresse 0x100 en RAM, le premier bit de cette variable est accessible par "`my_bit`".

```
1 #byte my_var = 0x100
2 #bit my_bit = my_var.0
```

Listing 1.5 – Directives #byte et #bit

- L'utilisation des variables C dans l'assembleur, comme indiqué dans Listing 1.6.

```
1 int8 foo = 10;
2 #asm
3     LOOP:
4     decfsz foo
5     bra LOOP
6 #endasm
```

Listing 1.6 – Accès aux variables C dans l'assembleur

- Réservation de l'espace ROM.

```
#org 0, 0x1000 {}
```

Listing 1.7 – Utilisation de la directive #org

Cette ligne indique au compilateur de ne pas insérer de code dans l'espace allant de l'adresse 0 à l'adresse 0x1000.

- Remappage des vecteurs de reset et d'interruption.

```
#build (reset=0x200, interrupt=0x208)
```

Listing 1.8 – La réallocation des vecteurs de reset et d'interruption

la directive "#build" changera l'emplacement du vecteur reset à l'adresse 0x200 au lieu de l'adresse 0, et le vecteur interruption à l'adresse 0x208 au lieu de 8.

Les fonctionnalités susmentionnées du PIC C seront particulièrement utiles lors du développement du firmware, les deux dernières devenant utiles lors de la réalisation du bootloader par la suite.

1.5 CONCLUSION

Ce chapitre a établi les concepts fondamentaux concernant les microcontrôleurs PIC, leurs architectures et une variété d'outils de développement qui sont fréquemment employés par l'utilisateur final, tout en donnant une introduction générale concernant leur programmation. Il a également servi à justifier le choix du type de programmateur à développer dans ce projet, un qui utilise USB au lieu de ports série ou parallèles, ICSP au lieu de programmateurs qui emploient un socket, et un programmateur qui opère en mode HVP plutôt que LVP.

Les chapitres suivants vont étudier les différents aspects de ce programmateur, en commençant par le micrologiciel s'exécutant sur le PIC18F2550 puis l'application PC.

CHAPITRE 2

2

LE FIRMWARE

2.1 INTRODUCTION

Le firmware ou micrologiciel est le programme qui gère le fonctionnement du matériel sous-jacent, il diffère d'un logiciel traditionnel par le fait qu'il est intégré dans un système embarqué. Par conséquent, il est plus limité en ressources, les contraintes de temps sont plus strictes, moins tolérant aux erreurs et généralement spécifique à un seul but. Dans le cas du programmateur, ce micrologiciel est chargé dans la mémoire Flash du PIC18F2550 et sert à gérer la communication USB et à opérer les différents périphériques du MCU.

2.2 LA COMMUNICATION USB

Le programmateur se présentera au système d'exploitation hôte comme un dispositif HID. Ceci est dû à un avantage majeur par rapport à d'autres classes de périphériques USB, qui est que tous les principaux systèmes d'exploitation ont des pilotes HID intégrés, ce qui signifie qu'aucune installation de pilote n'est nécessaire.

Un HID n'a pas besoin d'avoir une interface humaine. L'appareil doit juste pouvoir fonctionner dans les limites de la spécification de la classe HID. Certaines de ces limites sont les suivantes :

- Une interface HID peut avoir au maximum un endpoint d'interruption 'IN' et un endpoint d'interruption 'OUT'.

  Pour plus de clarté, il convient de mentionner que tout le trafic dans le bus USB se déplace vers ou à partir d'un endpoint de périphérique. Le endpoint est un buffer en mémoire qui stocke plusieurs bytes. Les données stockées peuvent être des données reçues (endpoint 'OUT') ou des données en attente de transmission (endpoint 'IN').
- Les dispositifs HID sont limités au transfert d'un seul paquet de 64 octets et à un débit de 64 kB/s.

Ces limites ne seront pas problématiques dans l'application visée. Un seul endpoint suffit pour la communication bidirectionnelle avec l'hôte, et la longueur limitée des paquets sera prise en compte lors de l'élaboration du protocole de communication dans la section 2.3.

2.2.1 *Implémentation de l'USB avec le compilateur PIC C*

Le développement du driver pour le module USB du PIC18F2550 à partir de zéro serait une tâche qui nécessiterait énormement de temps. Mais cela est rendu beaucoup plus facile par l'utilisation des bibliothèques du compilateur PIC C.

La configuration d'un PIC18F pour USB peut s'effectuer à l'aide d'un fichier header ressemblant à celui du Listing 2.1.

```
#include <18F2550.h>

#fuse HSPLL
#fuse PLL5
#fuse CPUDIV2
#fuse USBDIV
#fuse VREGEN

#use delay(clock=48000000)

#define USB_CONFIG_HID_TX_SIZE 64 // Le maximum est de 64 octets
#define USB_CONFIG_HID_RX_SIZE 64 // Le maximum est de 64 octets
#define USB_CONFIG_PID 1234 // Valeur 16 bits non signée
#define USB_CONFIG_VID 5678  // Valeur 16 bits non signée

#include <pic18_usb.h>
#include <usb_desc_hid.h>
#include <usb.c>
```

Listing 2.1 – Configuration du PIC18F2550 pour USB

La directive "#fuse" définit quels bits de configuration doivent être activés dans le MCU lorsqu'il est programmé. Ci-après la description de chacune des configurations utilisées dans le Listing 2.1 :

1. HSPLL : Mode haute vitesse avec PLL activé. Ce mode est choisi car nous utilisons un quartz de 20 MHz comme oscillateur externe et une PLL pour fournir le signal d'horloge nécessaire pour le module USB.
2. PLL5 : Le module USB nécessite une source d'horloge de 48 MHz lorsqu'il fonctionne en mode Full-Speed. Dans le PIC18F2550, ce signal peut être généré avec une PLL ayant une entrée de référence constante de 4 MHz et une sortie constante de 96 MHz. L'entrée 4 MHz peut être obtenue en utilisant un prescaler qui divise la fréquence du quartz par 5. La sortie 96 MHz est ensuite divisée par 2 pour être utilisée par le module. Les autres valeurs du prescaler sont 1, 2, 3, 4, 6, 10 et 12.
3. CPUDIV2 : La source d'horloge pour le noyau, elle est obtenue en divisant les 96 MHz par 2.
4. USBDIV : Source d'horloge USB. L'horloge vient de la PLL divisée par 2.
5. VREGEN : Active le régulateur de tension interne 3,3 V de l'USB, Cela se fait au lieu d'utiliser une source de tension externe connectée à la broche V_{USB}.
6. use delay(clock=48000000) : indique au compilateur la vitesse du processeur et permet l'utilisation des fonctions de temporisation telles que "`delay_ms`" et "`delay_us`".
7. define `USB_CONFIG_HID_TX_SIZE` : définit la taille des paquets qui sont envoyés du PIC vers le PC.

8. define `USB_CONFIG_HID_RX_SIZE` : définit la taille des paquets reçus par le PIC depuis le PC.
9. define `USB_CONFIG_PID` : configurer le PID du PIC
10. define `USB_CONFIG_VID` : configurer le VID.

Les fonctions les plus couramment utilisées sont :

- `usb_init_cs` : initialise le module USB mais sans bloquer le noyau du MCU jusqu'à ce que le périphérique ait été énuméré. Lorsque le module USB a été initialisé à l'aide de cette fonction, "`usb_task`" doit être appelé avant toute opération pour vérifier si le périphérique est connecté au bus et a été énuméré.
- `usb_enumerated` : vérifie si le dispositif a été énuméré, aucune opération ne doit être effectuée tant qu'il n'est pas.
- `usb_put_packet(num_endpoint, pointeur_données, longueur, USB_DTS_TOGGLE)` : envoie un paquet à l'hôte.
- `usb_kbhit(endpoint)` : vérifie si le endpoint spécifié a reçu un paquet.
- `usb_get_packet(numéro_du_endpoint, pointeur_données, longueur)` : lit un paquet qui a été reçu de l'hôte et le copie à l'emplacement spécifié par "`pointeur_données`".

2.3 LES COMMANDES DU FIRMWARE ET DES SCRIPTS

Comme pour chaque paire d'appareils connectés, le firmware et le logiciel doivent établir un ensemble de règles pour que le firmware puisse interpréter correctement les commandes envoyées par le logiciel.

2.3.1 *Les commandes du firmware*

Le programmateur étant configuré en tant que dispositif HID, le type de transfert utilisé est "interruption", la communication est donc limitée à 64 octets par paquet USB, et un débit maximal de 64 kB/s. Ainsi, pour surmonter cette limitation et accélérer les différentes opérations, les commandes conçues pour ce programmateur ont été élaborées de manière à minimiser le nombre de paquets transférés sur l'USB. Le Tableau 2.1 répertorie ces commandes.

COMMANDE	VALEUR	PARAMÈTRES	DESCRIPTION
GET_VERSION	0x00	Pas de paramètres	Lire la version du firmware du programmateur
SET_VPP	0x01	1. VPP du PIC	Établit la tension de programmation pour le PIC cible
READ_VOLTAGES	0x02	Pas de paramètres	Lire les tensions V_{DD} et V_{PP} du PIC cible
RUN_ROM_SCRIPT	0x03	1. Longueur du script 2. Octet de poids faible de l'adresse du script. 3. Octet de poids fort de l'adresse du script	Exécuter un script situé dans la ROM
RUN_ROM_SCRIPT_ITR	0x04	1. Longueur du script. 2. Octet de poids faible de l'adresse du script 3. Octet de poids fort de l'adresse du script 4. Nombre d'itérations	Exécuter un script situé dans la ROM pour plusieurs itérations
RUN_USB_SCRIPT	0x05	1. Longueur du script 2. Le script	Exécuter le script inclus dans le paquet USB courant
CLEAR_DOWN_BUFF	0x06	Pas de paramètres	Vide le buffer utilisé pour les transferts PC → PIC
WRITE_DOWN_BUFF	0x07	1. Nombre d'octets à stocker 2. Les données	Ecrire les données dans le buffer des transferts PC → PIC
CLEAR_UP_BUFF	0x08	Pas de paramètres	Vide le buffer utilisé pour les transferts PIC → PC
UPLOAD	0x09	Pas de paramètres	Envoyer le contenu du buffer au logiciel, le premier byte est le nombre d'octets envoyés
UPLOAD_WITHOUT_LEN	0x0A	Pas de paramètres	Envoyer le contenu du buffer au logiciel, sans inclure le nombre d'octets

Tableau 2.1 – Les commandes du logiciel hôte

Les scripts utilisés pour identifier, lire, programmer ou effacer le PIC cible sont l'un des facteurs qui affectent le débit, le logiciel de PICKit 2 ainsi que les divers autres programmateurs open source stockent ces scripts sur le côté PC et sont envoyés un par un quand ils sont requis. L'approche que nous avons suivie est un peu différente, les scripts sont déjà stockés dans le programmateur. Par conséquent, un script peut être exécuté en n'utilisant que quatre octets du paquet USB comme le montre le Listing 2.2, au lieu de transférer la totalité du script, qui peut dépasser 50 octets de longueur.

```
1 byte[] runScriptCmd = new byte[4];
2 runScriptCmd[0] = RUN_ROM_SCRIPT;
3 runScriptCmd[1] = 56; // Longueur du script, 56 bytes
4 // Le script se trouve à l'adresse 0x647D dans la ROM du PIC18F2550.
5 runScriptCmd[2] = (byte) 0x7D;
6 runScriptCmd[3] = (byte) 0x64;
7 USBFunctions.hidWrite(runScriptCmd);
```

Listing 2.2 – Exécution d'un script situé dans la ROM

Avec cette méthode, les 64 octets disponibles dans un paquet USB sont amplement suffisants pour l'exécution séquentielle de plusieurs scripts.

Un autre point qui devrait être adressé est que les programmateurs qui stockent les scripts sur le côté hôte essaient de surmonter le problème d'envoyer les scripts un par un au firmware utilisent un buffer qui retient plusieurs scripts dans la RAM. Lorsque le PIC cible est identifié, certains des scripts utilisés pour le programmer sont stockés dans ce tampon. Dans le cas du PICKit 2, la taille de du buffer est de 768 bytes. En stockant les scripts dans le firmware, ce buffer n'est pas utilisé et cet espace est libre pour être utilisé à d'autres fins.

2.3.2 *Les scripts*

La programmation d'un PIC consiste à envoyer différentes commandes suivies de données. Ces données peuvent être écrites dans le MCU, ou lues à partir de celui-ci. Ces commandes varient d'une famille d'appareils à l'autre. A titre d'exemple, celles utilisées pour les microcontrôleurs PIC18F2XXX/4XXX sont présentées dans le Tableau 2.2.

L'exécution d'une séquence de ces commandes permet d'effectuer différentes opérations sur le PIC cible. Par exemple, la séquence nécessaire pour écrire un bloc de mémoire programme dans un PIC18F2XXX est celle indiquée dans le Tableau 2.3.

Pour faciliter la tâche du logiciel, les différentes séquences utilisées à diverses fins sont stockées dans le firmware sous forme de scripts. Comme exemple, les scripts qui représentent celles du Tableau 2.3 Sont stockés comme indiqué sur Listing 2.3 et Listing 2.4. Le premier script accomplit la première et la deuxième étape qui sont mentionnées dans le Tableau 2.3 et le deuxième script écrit les 32 octets de données puis quitte le mode de programmation.

COMMANDE	CODE	DONNÉE	DESCRIPTION
Instruction de base	0000	16 bits	Utilisée pour transmettre une instruction 16 bits au CPU
Lire le registre TABLAT	0010	8 bits	Lire le contenu du registre “TABLAT”
Table Read	1000	8 bits	Lire l’emplacement vers lequel “Table Pointer” pointe
Table Read, Post-incrémentation	1001	8 bits	Lire l’emplacement puis incrémenter l’adresse
Table Read, Post-Décrémentation	1010	8 bits	Lire l’emplacement puis décrémenter l’adresse
Table Read, Pré-incrémentation	1011	8 bits	Incrémenter l’adresse puis lire l’emplacement
Table Write	1100	16 bits	Ecrire à l‘emplacement indiqué par “Table Pointer”
Table Write, Post-incrémentation par 2	1101	16 bits	Ecrire à l‘emplacement puis incrémenter l’adresse par 2
Table Write, Commencer la programmation, Post-incrémentation par 2	1110	16 bits	Ecriture et incrémentation en mode programmation
Table Write, Commencer la programmation	1111	16 bits	Commencer la programmation après l’écriture

Tableau 2.2 – Les commandes des microcontrôleurs PIC18F [10]

```
1 // Type = PROG_MEM_WR_PREP
2 // Address = 0x6740 – Length = 33
3 #rom int8 0x6740 = {SHIFT_BITS_OUT_CMD, 4, 0, SHIFT_BYTE_OUT_BUFFER,
      SHIFT_BYTE_OUT, 0x0E, COREINST18, 0xF6, 0x6E, SHIFT_BITS_OUT_CMD, 4, 0,
      SHIFT_BYTE_OUT_BUFFER, SHIFT_BYTE_OUT, 0x0E, COREINST18, 0xF7, 0x6E,
      SHIFT_BITS_OUT_CMD, 4, 00, SHIFT_BYTE_OUT_BUFFER, SHIFT_BYTE_OUT,
      0x0E, COREINST18, 0xF8, 0x6E, COREINST18, 0xA6, 0x8E, COREINST18, 0xA6, 0x96}
```

Listing 2.3 – Le script qui prépare l’écriture d’un bloc mémoire Flash

```
1 // Type = PROG_MEM_WR
2 // Address = 0x6760 – Length = 35
3 #rom int8 0x6760 = {SHIFT_BITS_OUT_CMD, 4, 0x0D, SHIFT_BYTE_OUT_BUFFER,
      SHIFT_BYTE_OUT_BUFFER, LOOP, 5, 0x0E, SHIFT_BITS_OUT_CMD, 4, 0x0F,
      SHIFT_BYTE_OUT_BUFFER, SHIFT_BYTE_OUT_BUFFER, SHIFT_BITS_OUT_CMD,
      3, 0, SET_ICSP_PINS_CMD, 4, SHORT_DELAY, 47, SET_ICSP_PINS_CMD, 0,
      SHORT_DELAY, 5, SHIFT_BYTE_OUT, 0, SHIFT_BYTE_OUT, 0,
```

SHIFT_BITS_OUT_CMD, 4, 0x0D, SHIFT_BYTE_OUT, 0xFF, SHIFT_BYTE_OUT, 0xFF}

Listing 2.4 – Le script qui écrit un bloc de mémoire Flash

COMMANDE	DONNÉES	INSTRUCTION ÉQUIVALENTE
Étape 1. Accès direct à la mémoire de programme et activation des écritures		
0000	0x8E 0xA6	BSF EECON1, EEPGD
0000	0x9C 0xA6	BCF EECON1, CFGS
Étape 2. Charger l'adresse du bloc dans "Table Pointer"		
0000	0x0E <Addr[21 :16]>	MOVLW <Addr[21 :16]>
0000	0x6E 0xF8	MOVWF TBLPTRU
0000	0x0E <Addr[15 :8]>	MOVLW <Addr[15 :8]>
0000	0x6E 0xF7	MOVWF TBLPTRH
0000	0x0E <Addr[7 :0]>	MOVLW <Addr[7 :0]>
0000	0x6E 0xF6	MOVWF TBLPTRL
Étape 3. Répétez l'instruction suivante pour écrire 30 octets		
1101	<MSB ><LSB >	Table Write, Post-incrémentation par 2
Étape 4. Ecrire les deux derniers octets du bloc, et commencer la programmation		
1111	<MSB ><LSB >	Table Write, Commencer la programmation
0000	00 00	NOP - Maintenir le PGC à un niveau haut

Tableau 2.3 – Programmation d'un bloc de mémoire de code [10]

Les scripts sont définis par l'adresse à laquelle ils se trouvent dans la ROM, leur longueur et leur type. L'adresse sert d'identifiant à chaque script puisqu'elle est unique à chacun d'entre eux, la longueur est utile lors de son traitement et la définition d'un type est utile puisque la base de données est recherchée pour un script en fonction de la fonctionnalité de ce script et à quel périphérique ou famille de périphériques il est associé. Les différents types sont listés ci-dessous :

- `ROW_ERASE`
- `PROG_ENTRY`
- `PROG_EXIT`
- `READ_DEV_ID`
- `CHIP_ERASE`
- `PROG_MEM_ADDR_SET`
- `PROG_MEM_READ`
- `EE_RD_PREP`
- `EE_RD`
- `USER_ID_RD_PREP`
- `USER_ID_RD`
- `CONFIG_RD_PREP`
- `CONFIG_RD`
- `PROG_MEM_WR_PREP`
- `PROG_MEM_WR`
- `EE_MEM_WR_PREP`
- `EE_MEM_WR`
- `USER_ID_WR_PREP`
- `USER_ID_WR`
- `CONFIG_WR_PREP`
- `CONFIG_WR`

- OSCCAL_RD
- OSCCAL_WR
- CHIP_ERASE_PREP
- PROG_MEM_ERASE
- EE_MEM_ERASE
- CONFIG_MEM_ERASE
- EE_ROW_ERASE

2.3.2.1 *Les commandes du script*

Comme le montrent Listing 2.3 et Listing 2.4, chaque script est composé d'un ensemble de commandes, qui ont pour but de simplifier le processus de développement des scripts. Le Tableau 2.4 énumère les diverses commandes.

COMMANDE	VALEUR	PARAMÈTRES	DESCRIPTION
COREINST18	0xC0	1. Premier octet qui sera envoyé 2. Deuxième octet	Envoie quatre zéros puis les deux octets de paramètres
POP_DOWNLOAD_BUFFER	0xC1	Pas de paramètres	Renvoie le premier octet dans le buffer PC → PIC
LOOP_BUFFER	0xC2	1. Point de début de la boucle	Exécute les commandes qui précèdent la commande courante plusieurs fois, le nombre de commandes est le paramètre, et le nombre d'itérations est dans le buffer PC → PIC
EXIT_SCRIPT	0xC3	Pas de paramètres	Arrêter l'exécution du script courant
GOTO_IDX	0xCs4	1. L'indice de destination	Aller à un index spécifique dans le script
SHORT_DELAY	0xC5	1. La durée du délai, en multiples de 21.3 us	Suspend l'exécution d'une durée déterminée
LONG_DELAY	0xC6	1. La durée du délai, en multiples de 5.4 ms	Suspend l'exécution d'une durée déterminée
LOOP	0xC7	1. Point de début de la boucle 2. Nombre d'itérations	Exécute les commandes qui précèdent la commande courante plusieurs fois

SHIFT_BITS_OUT_CMD	0xC8	1. Nombre de bits 2. La valeur des bits	Envoie un maximum de huit bits au PIC cible
SHIFT_BYTE_IN_BUFFER	0xC9	Pas de paramètres	Lire un octet du PIC cible et le stocker dans le tampon PIC → PC
SHIFT_BYTE_OUT_BUFFER	0xCA	Pas de paramètres	Envoie un byte du tampon vers le PIC cible
SHIFT_BYTE_OUT	0xCB	1. L'octet à envoyer	Envoie un byte vers le PIC cible
SET_ICSP_PINS_CMD	0xCC	1. Un octet, les premier et deuxième bits définissent la direction des signaux d'horloge et de données, '1' = entrée et '0' = sortie. Les troisième et quatrième bits définissent l'état de ces signaux	Définit la direction et l'état des broches PGD et PGC
MCLR_TGT_GND_OFF	0xCD	Pas de paramètres	Bloque le transistor utilisé pour réinitialiser le PIC cible
MCLR_TGT_GND_ON	0xCE	Pas de paramètres	Sature le transistor utilisé pour réinitialiser le PIC cible
VPP_PWM_OFF	0xCF	Pas de paramètres	Arrête le signal PWM qui commande le boost
VPP_PWM_ON	0xD0	Pas de paramètres	Démarre le signal PWM qui commande le boost
VPP_ON_CMD	0xD1	Pas de paramètres	Délivre la tension V_{PP} au PIC cible
VPP_OFF	0xD2	Pas de paramètres	Connecte V_{PP} à la masse

Tableau 2.4 – Les commandes des scripts

2.4 LES PROCÉDURES DE BASE DU FIRMWARE

Les programmes qui se déroulent sur les microcontrôleurs consistent généralement à exécuter une boucle de programme principale, c'est-à-dire un ensemble déterminé de tâches qui sont exécutées indéfiniment. Le système quittait cette boucle seulement pour réagir à des interruptions internes ou externes.

Dans le cas du programmateur, le firmware initialisera les différents périphériques et l'interface USB du PIC18F2550, puis entrera dans la boucle principale qui consiste à attendre les paquets USB et les traiter quand ils arrivent. La procédure principale est illustrée à le Listing 2.5.

```
void main() {
    // Initialiser les périphériques du MCU
    pgm_init ();
    // Initialiser le module USB
    usb_init_cs ();
    while (true) {
        usb_task();
        // Vérifier si le PIC a été énuméré par le PC
        if (usb_enumerated()) {
            // Vérifier si le endpoint1 a reçu des données
            if (usb_kbhit(1)) {
                // Traiter le paquet reçu
                process_usb_packet ();
            }
        }
    }
}
```

Listing 2.5 – La fonction main du firmware

2.4.1 *Traitement du paquet USB*

Chaque paquet USB reçu par le MCU du programmateur contient les commandes listées dans le Tableau 2.1. Ces commandes dictent la manière dont le dispositif doit se comporter et sont interprétées dans la fonction "`process_usb_packet`". Listing 2.6 illustre sa mise en œuvre.

```
// Allumer la LED pour indiquer que le programmateur est occupé
BUSY_LED = 1;
// L'indice utilisé pour adresser tous les emplacements du paquet
i = 1;
// Copier le endpoint dans le tampon "data_in"
usb_get_packet(1, data_in, 64);
// Le premier octet est le nombre de données envoyées par l'hôte
unsigned int8 packet_length = data_in[0];
// Chaque commande sera copiée ici
unsigned int8 offset;
// un pointeur vers le buffer en RAM dans laquelle un script sera copié
unsigned int8 *script_buffer;
// L'adresse du script dans la ROM
```

```
14 unsigned int16 address;
15
16 while (i <= packet_length) {
17     offset = data_in[i];
18     offset *= 2;
19     offset += 8;
20
21     #asm
22         MOVF PCL, W
23         ADDWF offset, W
24         BTFSC C
25         INCF PCLATH
26         MOVWF PCL
27         BRA GET_VERSION_LBL
28         BRA SET_VPP_LBL
29         BRA READ_VOLTAGES_LBL
30         BRA RUN_ROM_SCRIPT_LBL
31         BRA RUN_ROM_SCRIIPT_ITR_LBL
32         BRA RUN_USB_SCRIPT_LBL
33         BRA CLEAR_DOWN_BUFF_LBL
34         BRA WRITE_DOWN_BUFF_LBL
35         BRA CLEAR_UP_BUFF_LBL
36         BRA UPLOAD_LBL
37         BRA UPLOAD_WITHOUT_LENGTH_LBL
38     #endasm
```

Listing 2.6 – La fonction main du firmware

La détermination de la commande à exécuter se fait en modifiant le registre PCL (Program Counter Low). Chaque code de commande représente un décalage par rapport à l'instruction "`BRA GET_VERSION_LBL` ". Donc l'envoi de la commande "`GET_VERSION`" qui a le code 0x00 entraînera l'exécution de l'instruction "`BRA GET_VERSION_LBL`", de la même manière, l'envoi de la commande "`SET_VPP`" qui a le code 0x01 provoquera l'exécution de l'instruction "`BRA SET_VPP_LB`" et ainsi de suite.

2.4.2 *Exécution d'un script*

La réception de la commande "`RUN_ROM_SCRIPT`" entraînera la récupération et l'exécution d'un script depuis la ROM du PIC18F2550. Avant que le script ne puisse être exécuté, il doit être chargé dans la RAM, ceci est fait en allouant dynamiquement un espace dans la mémoire en accord avec la taille du script. Comme le montre le Listing 2.7.

```
1 RUN_ROM_SCRIPT_LBL:
2     /*
3       * data_in[i+1] = Longueur du script
4       * data_in[i+2] = Octet le moins significatif de l'adresse du script
5       * data_in[i+3] = Octet le plus significatif de l'adresse du script
```

```
6       */
7       // Allouer un espace en RAM et passer l'adresse au pointeur "script_buffer"
8       script_buffer = malloc (data_in[i+1]);
9       // Evaluer l'adresse du script
10      address = ((data_in[i+3] * 0x100) + data_in[i+2]);
11      // Lire le script de la mémoire du programme dans le buffer de la RAM
12      read_program_memory(address, script_buffer, data_in[i+1]);
13      // Exécuter le script
14      execute_script(data_in[i+1], script_buffer);
15      // Libérer l'espace alloué en RAM
16      free (script_buffer);
17      // Incrémenter l'indice pour qu'il s'adresse à la commande suivante
18      i += 4;
19      // Passer à l'itération suivante de la boucle "while"
20      continue;
```

Listing 2.7 – Récupération d'un script depuis la ROM

Après le chargement du script dans la RAM, "`execute_script`" est invoqué pour le traiter, les commandes incluses dans les différents scripts sont listées dans le Tableau 2.4 et sont interprétées comme indiqué dans le Listing 2.8.

```
1   // Initialiser l'indice du script à zéro
2   unsigned int8 si = 0;
3   unsigned int8 offset;
4
5   while (si < scrpt_len) {
6       offset = *(script_location + si);
7       offset -= 0xC0;
8       offset *= 2;
9       offset += 8;
10      #asm
11          MOVF PCL, W
12          ADDWF offset, W
13          BTFSC C
14          INCF PCLATH
15          MOVWF PCL
16          BRA COREINST18_LBL
17          BRA POP_DOWNLOAD_BUFFER_LBL
18          BRA LOOP_BUFFER_LBL
19          BRA EXIT_SCRIPT_LBL
20          BRA GOTO_IDX_LBL
21          BRA SHORT_DELAY_LBL
22          BRA LONG_DELAY_LBL
23          BRA LOOP_LBL
24          BRA SHIFT_BITS_OUT_LBL
25          BRA SHIFT_BYTE_IN_BUFFER_LBL
26          BRA SHIFT_BYTE_OUT_BUFFER_LBL
27          BRA SHIFT_BYTE_OUT_LBL
```

```
28          BRA SET_ICSP_PINS_LBL
29          BRA MCLR_TGT_GND_OFF_LBL
30          BRA MCLR_TGT_GND_ON_LBL
31          BRA VPP_PWM_OFF_LBL
32          BRA VPP_PWM_ON_LBL
33          BRA VPP_ON_LBL
34          BRA VPP_OFF_LBL
35      #endasm
```

Listing 2.8 – Le traitement d'un script

L'exécution d'un script est similaire au traitement du paquet USB indiqué à le Listing 2.6. Ccependant, puisque la première commande ("`COREINST18`") a un code égal à 0xC0, cette valeur est soustraite.

2.5 CONCLUSION

Ce chapitre a couvert l'implémentation de la communication du point de vue du firmware, en commençant par la mise en œuvre du protocole USB jusqu'au traitement des paquets entrants et à l'exécution des scripts de programmation. Le logiciel hôte est responsable de l'envoi de ces commandes et du contrôle du comportement du firmware. Ce dernier sera abordé dans le chapitre suivant.

CHAPITRE 3

3

LE LOGICIEL HÔTE

3.1 INTRODUCTION

Il a été mentionné précédemment que les programmateurs USB nécessitent des applications fonctionnant sur le PC et le microcontrôleur incorporé dans le programmateur. Ce chapitre sera consacré à la description détaillée de l'application hôte.

Le logiciel sert à mettre à la disposition de l'utilisateur une interface graphique qui peut être utilisé pour lancer les opérations de base comme la lecture, l'écriture, l'effacement et la vérification du contenu de la mémoire du PIC cible. le logiciel qui a été développé est montrée dans la Figure 3.1.

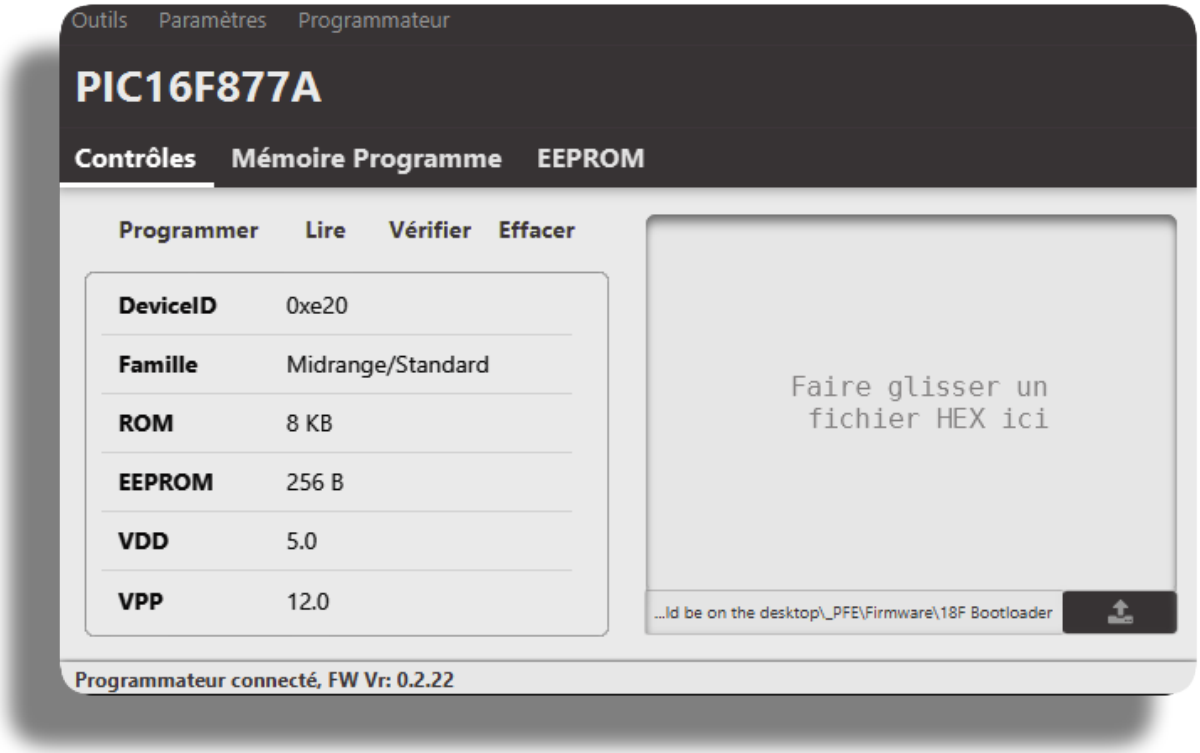

FIGURE 3.1 – L'interface graphique du logiciel

Le logiciel a été écrit en Java, et construit en utilisant le framework JavaFX, avant de décrire le fonctionnement interne de l'application, nous allons décrire le processus de développement en utilisant ces deux outils.

3.2 PROCESSUS DE DÉVELOPPEMENT

Le logiciel pour le PICKit 2 qui a suscité l'intérêt dans les programmateurs PIC de Microchip a été développé en C# et était natif de Windows et n'offrait le support à aucun autre système d'exploitation. Par la suite, un outil CLI (Command Line Interface) a été créé pour Linux. C'est la raison pour laquelle Java, et plus précisément Java 8, a été utilisé pour développer un outil graphique multi-plateforme dans ce projet.

Historiquement, la motivation initiale du développement de Java était le besoin d'un langage indépendant de la plate-forme (c'est-à-dire, indépendant de l'architecture) qui pourrait être utilisé pour créer des logiciels devant être intégrés dans divers appareils électroniques pour le grand public. avec l'émergence d'Internet, le problème de la portabilité du code est revenu. Étant donné qu'Internet se compose d'un ensemble diversifié et distribué d'ordinateurs et de systèmes d'exploitation. L'objectif était de faire en sorte que tous ces systèmes puissent exécuter le même programme.[11]

Le problème avec C et C++ (et la plupart des autres langages) est qu'ils sont conçus pour être compilés pour une cible spécifique. Bien qu'il soit possible de compiler un programme C++ pour à peu près n'importe quel type de CPU, il faut pour cela un compilateur C++ complet pour ce CPU. Le problème est que les compilateurs sont coûteux et prennent beaucoup de temps à créer.

La clé qui permet à Java de résoudre le problème de portabilité est que la sortie d'un compilateur Java n'est pas un code exécutable. Il s'agit plutôt d'un bytecode. Le bytecode est un ensemble d'instructions conçues pour être exécutées par la machine virtuelle du Java (JVM). Essentiellement, la JVM a été conçue comme un interpréteur de bytecode.

3.2.1 *JavaFX*

JavaFX est un ensemble de packages de graphiques et de médias qui permet aux développeurs de concevoir, créer, tester, déboguer et déployer des applications client riches qui fonctionnent de manière cohérente sur diverses plateformes.[12] elle a été conçu pour remplacer Swing en tant que framework de développement d'interface graphique standard, elle fournit un outil puissant et flexible qui simplifie la création des interfaces modernes.

L'un des avantages de l'utilisation de JavaFX vient de la séparation de vue et logique, puisque la vue peut être modifiée en utilisant des fichiers FXML (FX Markup Language) séparés et la logique peut être implémentée dans les classes contrôleur. Un autre avantage est que les applications JavaFX peuvent être personnalisées à l'aide de CSS (cascading style sheets), qui peut être utilisé pour accéder et modifier l'apparence de chaque noeud de l'application.

Les API JavaFX sont intégrées dans le kit de développement java depuis JDK 7, mais d'autres outils peuvent aider au processus de développement, comme Scene Builder, qui est un logiciel utilisé pour concevoir de manière interactive l'interface utilisateur graphique.

3.2.2 *Multithreading dans JavaFX*

l'interface graphique d'une application JavaFX n'est accessible et modifiable qu'à partir du thread principal également appelé « JavaFX Application thread ». L'implémentation de tâches de longue durée sur ce thread rend inévitablement une application non réactif. Une meilleure approche est d'effectuer ces tâches sur un ou plusieurs threads secondaires et de laisser le thread principal traiter les événements d'utilisateur.[13]

Les fonctions de lecture, d'écriture, d'effacement et autres que le programmateur effectue prennent un temps relativement long à accomplir, afin de ne pas risquer que le logiciel apparaisse non réactif à l'utilisateur et au système d'exploitation, toutes ces fonctionnalités doivent être implémentées dans un thread séparé, ainsi que des mécanismes qui permettent de renvoyer les résultats vers le thread principal.

La gestion du multithreading en JavaFX est différente de celle des autres environnements Java, elle est effectué avec la classe Task, tel que démontré dans Listing 3.1 ci-dessous.

```
1  Task<Void> task = new Task<Void>()
2  {
3          @Override
4      protected Void call() throws Exception {
5          // Le code à exécuter par la tâche
6      }
7  };
8
9  task.setOnSucceeded((WorkerStateEvent event) -> {
10     System.out.println ("La tâche a réussi");
11 });
12
13 task.setOnFailed((WorkerStateEvent event) -> {
14     System.out.println ("La tâche a échoué");
15 });
16
17 Thread thread = new Thread(task);
18 thread.start();
```

Listing 3.1 – Utilisation de la classe Task

Lorsque l'utilisateur lance une procédure telle que l'écriture sur le MCU cible, une tâche sera créée qui effectuera l'exécution de ce processus, ainsi que les gestionnaires d'événements comme indiqué dans les lignes 9 et 13 du Listing 3.1, ceux-ci seront exécutés lorsque la tâche est terminée. Pendant ce temps, le thread principal affichera un message tel que celui illustré sur la Figure 3.2, et les autres contrôles seront désactivés afin que l'utilisateur ne puisse pas lancer deux opérations simultanément, telles que lire et programmer.

Comme mentionné précédemment, toutes les mises à jour de l'interface graphique doivent être effectuées sur le thread principal, cependant, il y a des cas où une tâche secondaire doit affecter ses propres modifications, comme l'affichage d'un message à l'utilisateur, ceci peut être fait en utilisant le code indiqué dans Listing 3.2.

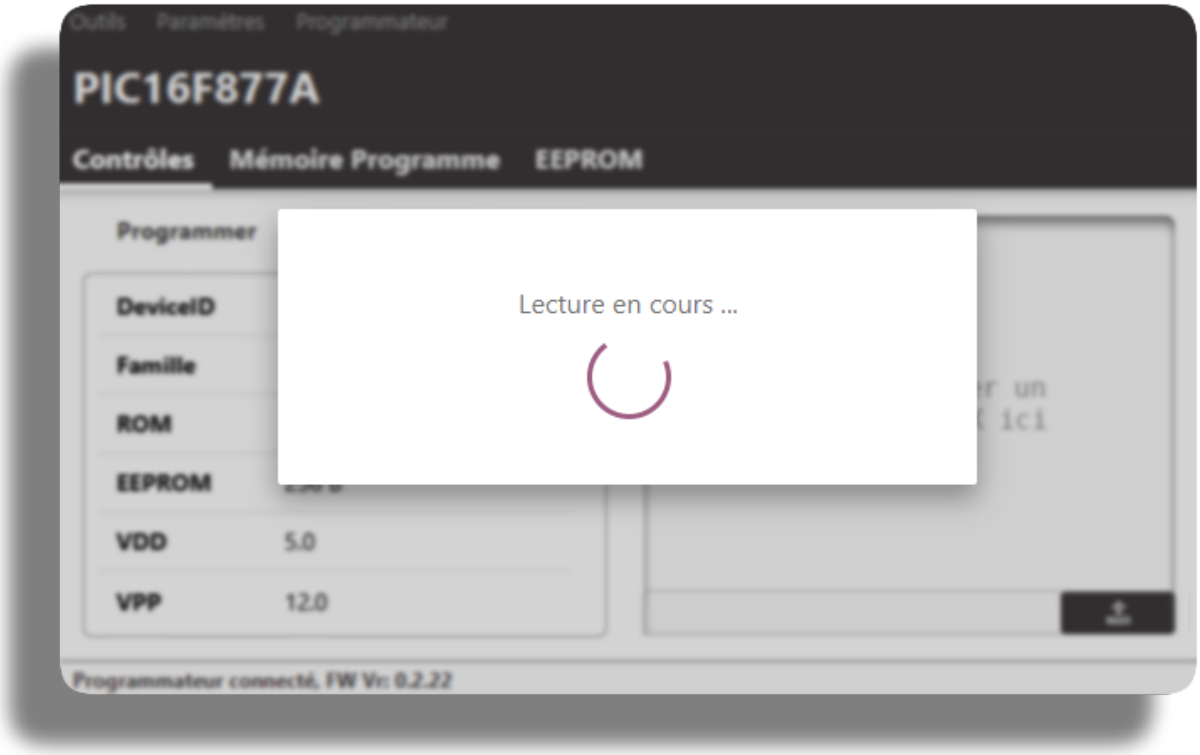

FIGURE 3.2 – Une tâche est en cours d'exécution

```
1 Platform.runLater(() -> {
2     // Les tâches qui vont être exécutées sur JavaFX Application thread
3 });
```

Listing 3.2 – Modifier l'interface graphique à partir d'une tâche d'arrière-plan

3.2.3 *Le schéma de la base de données*

Les tables de la base de données contiennent toutes les informations nécessaires aux différentes opérations, telles que la taille et l'adresse des différentes régions de mémoire, les noms et identificateurs des périphériques, les adresses et la longueur des différents scripts, les voltages de programmation ..., et puisque H2 est une base de données relationnelle, il y a quelques considérations qui doivent être prises en compte lors de la conception des schémas de ces tableaux.

3.2.3.1 *Modèle Entité-Association*

Avant de définir les schémas relationnels, les données doivent être conceptualisées en diagrammes, plus précisément en diagrammes enitité association, qui servent à donner une

idée générale des liens existant entre les différentes entités de la base de données. Il y a trois entités principales dans notre base de données :

1. Family : cette entité représente chaque famille de microcontrôleurs PIC, ses attributs sont les caractéristiques communes à tous les dispositifs qui font partie d'une même famille.
2. Device : elle modélise les microcontrôleurs individuels, les attributs sont spécifiques à un seul MCU.
3. Script : c'est l'entité qui représente les algorithmes utilisés pour la programmation ou la lecture du PIC cible, ceux-ci seront discutés dans un chapitre ultérieur.

Une notion qu'il convient de décrire avant de démontrer les liens entre ces entités est celle des cardinalités. Les cardinalités spécifient combien d'instances d'une entité se rapportent à une instance d'une autre entité. Avec cela dit, l'association entre les trois entités peut être décrite comme suit :

- une instance de la classe Device peut faire partie d'exactement une instance de la classe Family, mais une Family peut inclure plusieurs instances de Device, ce qui signifie que c'est une cardinalité 1 :n.
- une instance de l'entité Device peut utiliser plusieurs instances de Script et vice-versa, donc la cardinalité est n :n.
- comme la précédente, une instance de l'entité Famille peut utiliser plusieurs instances de l'entité Script et vice-versa, alors la cardinalité est n :n.

Les associations sont schématisées dans la Figure 3.3.

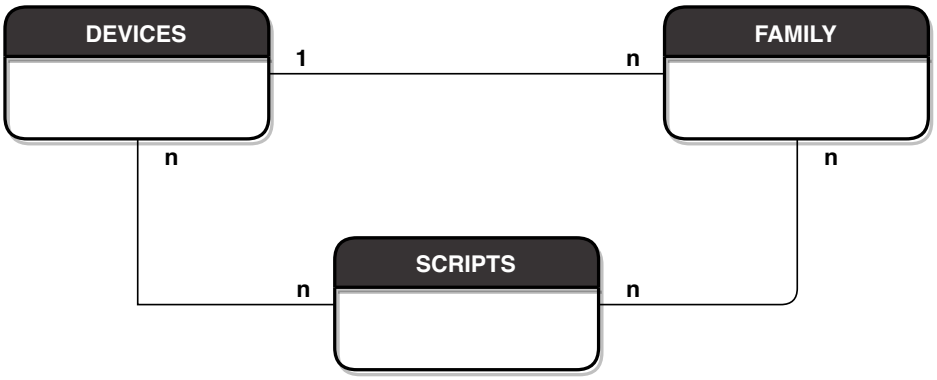

FIGURE 3.3 – Modèle conceptuel de la base de données

3.2.3.2 *Normalisation de la base de données*

Une étape fondamentale dans le processus de conception du logiciel est la normalisation de la base de données. cette dernière est une approche systématique de décomposition des différentes relations pour éliminer la redondance des données et les caractéristiques indésirables comme les anomalies d'insertion, de modification et de suppression. Il s'agit d'un processus en plusieurs étapes qui met les données sous forme de tableaux, en supprimant les données dupliquées.

Il existe plusieurs formes de normalisation des bases de données, mais il est généralement admis qu'une base de données relationnelle sous la troisième forme normale est suffisamment normalisée. Ce qui suit donne une brève description des trois premières formes normales :

1. Une relation dans une base de données est en première forme normale si tous les attributs sont atomiques, i.e., les valeurs multiples dans une colonne ne sont pas autorisées.
2. Une relation est en seconde forme normale si elle est en première forme normale et que tous les attributs dépendent de la totalité de la clé. Ceci s'applique aux relations qui ont des clés composites, c'est-à-dire des clés composées de plusieurs colonnes.
3. Une relation est dans la troisième forme normale si elle est dans la deuxième forme normale et qu'il n'y a pas de dépendance transitive. Ce qui signifie que chaque attribut ne doit dépendre que de la clé (primaire ou candidate).

Le schéma final des différentes relations de la base de données est représenté dans la Figure 3.4.

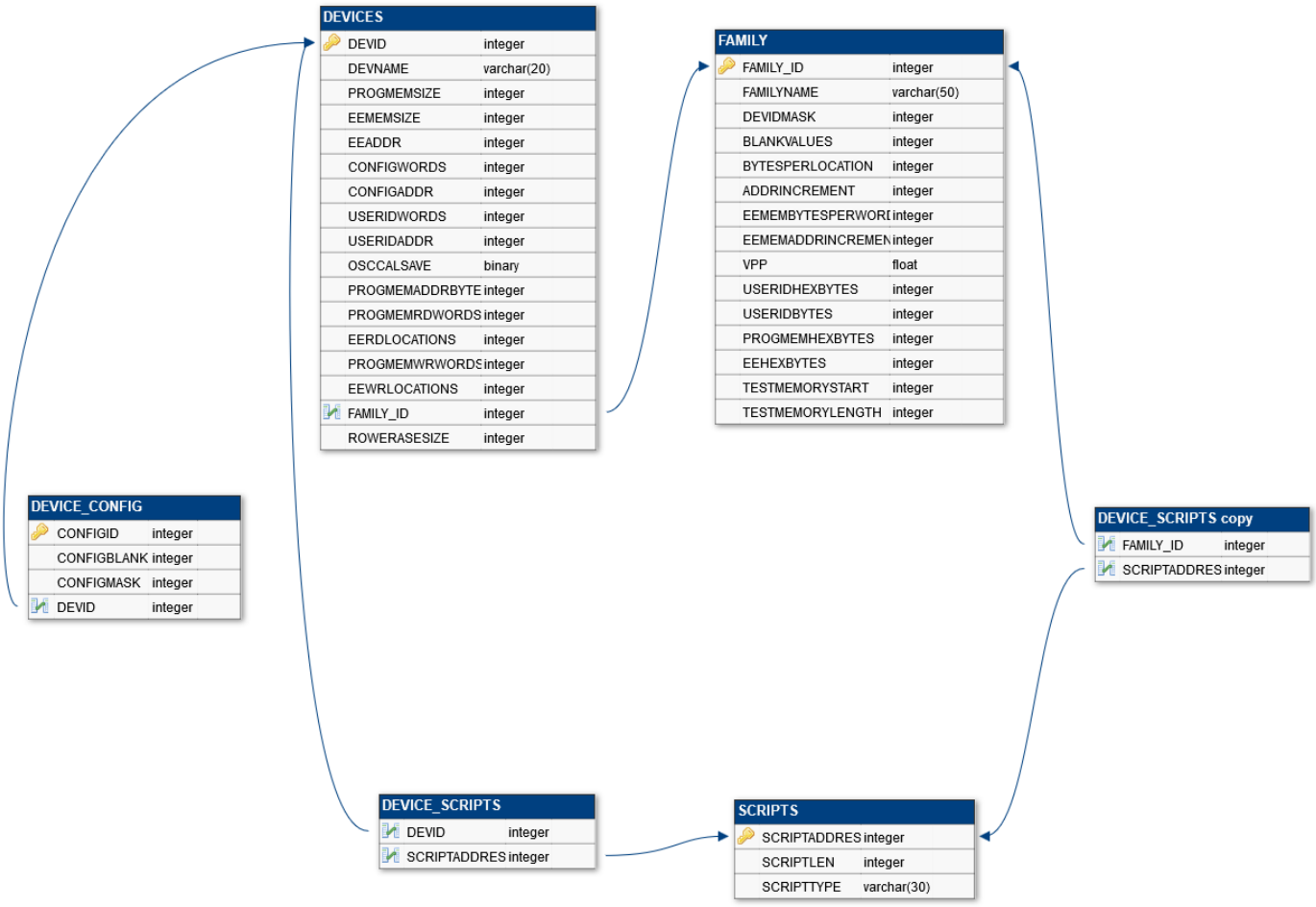

FIGURE 3.4 – Schema de la base de données

3.3 DESCRIPTION DES DIFFÉRENTES OPÉRATIONS

L'interface graphique du logiciel permet à l'utilisateur d'accomplir une multitude de tâches, ce qui suit couvre les différentes opérations que le programmateur peut effectuer sur le PIC cible.

3.3.1 *Connexion au programmateur*

La connexion au programmateur est le premier processus que le logiciel tente de faire après avoir été lancé, Ceci est rendu plus simple en utilisant les fonctions fournies par l'API hid4java. Ce qui suit décrit comment cela a été accompli :

1. La première étape consiste à importer les classes de hid4java. Tel qu'indiqué dans le Listing 3.3.

```
1 import org.hid4java.HidDevice;
2 import org.hid4java.HidManager;
3 import org.hid4java.HidServices;
4 import org.hid4java.HidServicesSpecification;
5 import org.hid4java.ScanMode;
```

Listing 3.3 – Les classes utilisées pour la communication USB

2. Après cela, il faudra initialiser les services HID.

```
1  private static HidServices hidServices;
2  public static HidDevice programmer = null;
3  public static void usbInit () {
4          HidServicesSpecification hidServicesSpecification = new
               HidServicesSpecification();
5          hidServicesSpecification.setAutoShutdown (true);
6          hidServicesSpecification.setScanInterval (500);
7          hidServicesSpecification.setPauseInterval (5000);
8          hidServicesSpecification.setScanMode
               (ScanMode.SCAN_AT_FIXED_INTERVAL_WITH_PAUSE_AFTER_WRITE);
9          hidServices = HidManager.getHidServices (hidServicesSpecification);
10         hidServices.start();
11 }
```

Listing 3.4 – Initialisation de la communication USB

3. Effectuer une recherche des périphériques HID connectés.

```
1 @SuppressWarnings("empty-statement")
2 public static byte[] checkForProgrammer () {
3     // Boucle à travers les appareils HID connectés
4     for (HidDevice hidDevice : hidServices.getAttachedHidDevices()) {
```

```
5            // Comparer le VID et le PID de l'appareil détecté à ceux du programmateur.
6            if (hidDevice.getProductId() == 0x0001 && hidDevice.getVendorId() == 0x0025)
7            {
8                programmer = hidDevice;
9                programmer.open();
10               byte[] cmd = new byte[1];
11               cmd[0] = GET_VERSION;
12               USBFunctions.hidWrite(cmd);
13               byte[] response = new byte[4];
14               while (programmer.read(response, 500) < 0);
15               PwJ.setUsbFound(true);
16               return response;
17           }
18       }
19       return null;
20   }
```

Listing 3.5 – Identification du programmateur

4. Chaque périphérique HID est identifié par la combinaison VID/PID, le firmware a un PID de 1 et un VID de 25, ces valeurs sont aléatoirement choisis. Si un périphérique avec ces valeurs est trouvé, le canal de communication s'ouvre et une demande de numéro de version est envoyée, ceci est fait pour s'assurer que la communication USB fonctionne correctement. et si aucun périphérique n'a été trouvé, la méthode "checkForProgramme" retourne "null".

3.3.2 *L'identification*

Le premier processus qui doit être effectué par le programmateur est de reconnaître l'identité du PIC cible. Cela se fait en lisant un identifiant stocké dans la mémoire de du MCU. L'emplacement exact de cet identificateur varie d'une famille de microcontrôleurs à l'autre. Le processus s'effectue comme suit :

1. Tout d'abord, la fonction "searchForDevice" itère à travers toutes les familles, chaque famille possède la clé primaire `FAMILY_ID` comme indiqué précédemment dans la Figure 3.4. Actuellement la dernière famille un a `FAMILY_ID` = 11.

```
1  public static void identifyDevice() {
2      for (byte i = 0; i < 12; i++) {
3          /* Si le MCU a été identifiée, "searchForDevice" retournera son ID, sinon elle
               retournera 0 */
4          if (searchForDevice(i) != 0) {
5              PGMMainController.setActiveFamily(i);
6              break; // Nous avons identifié le PIC. Pas besoin de chercher d'autres familles
7          }
8      }
9  }
```

Listing 3.6 – Itération à travers les familles PIC

2. Pour chaque valeur de `FAMILY_ID`, récupérer toutes les informations nécessaires pour lire l'ID du PIC cible. Listing 3.7 montre une version simplifiée de la façon dont ce processus est mis en œuvre.

```
1  public static int searchForDevice (byte id) {
2      int progEntryScript = 0; // Utilisé pour stocker l'adresse du script "ProgEntry"
3      int progExitScript = 0;
4      int readDevIdScript = 0;
5      int devIdMask = 0;
6      byte progEntryLen = 0;
7      byte progExitLen = 0;
8      byte readDevIdLen = 0;
9      float familyVpp = 0;
10     // Joindre les relarions Scripts et Family
11     String query = "SELECT *"
12                    + " FROM FAMILY_SCRIPTS"
13                    + " INNER JOIN FAMILY ON
                          FAMILY.FAMILYID=FAMILY_SCRIPTS.FAMILYID"
14                    + " INNER JOIN SCRIPTS ON FAMILY_SCRIPTS.SCRIPTADDRESS
                          = SCRIPTS.SCRIPTADDRESS"
15                    + " WHERE FAMILY.FAMILYID ="+id;
16     ResultSet rs = DbUtil.execQuery(query);
17     try {
18         // La requête va retourner VPP et tous les scripts de cette famille
19         while (rs.next()) {
20             familyVpp = rs.getFloat("VPP");
21             devIdMask = rs.getInt("DEVIDMASK");
22             if (rs.getString("SCRIPTTYPE").equals("PROG_ENTRY")) {
23                 progEntryScript = rs.getInt("SCRIPTADDRESS");
24                 progEntryLen = rs.getByte("SCRIPTLEN");
25             }
26             else if (rs.getString("SCRIPTTYPE").equals("PROG_EXIT")) {
27                 progExitScript = rs.getInt("SCRIPTADDRESS");
28                 progExitLen = rs.getByte("SCRIPTLEN");
29             }
30             else if (rs.getString("SCRIPTTYPE").equals("READ_DEV_ID")) {
31                 readDevIdScript = rs.getInt("SCRIPTADDRESS");
32                 readDevIdLen = rs.getByte("SCRIPTLEN");
33             }
34         }
35     } catch (SQLException ex) {
36         Logger.getLogger(PwJFunctions.class.getName()).log(Level.SEVERE, null, ex);
37     }
38     // Quitter si l'une des adresses n'a pas été trouvée
39     if (progEntryScript == 0 || progExitScript == 0 || readDevIdScript == 0)
40         return 0;
41     byte[] cmd = new byte [15];
42     cmd [0] = SET_VPP;
43     cmd [1] = (byte) familyVpp;
44     cmd [2] = RUN_ROM_SCRIPT;
45     cmd [3] = progEntryLen;
```

```
46      cmd [4] = (byte) progEntryScript;
47      cmd [5] = (byte)(progEntryScript >> 8);
48      cmd [6] = RUN_ROM_SCRIPT;
49      cmd [7] = readDevIdLen;
50      cmd [8] = (byte) readDevIdScript;
51      cmd [9] = (byte)(readDevIdScript >> 8);
52      cmd [10] = RUN_ROM_SCRIPT;
53      cmd [11] = progExitLen;
54      cmd [12] = (byte) progExitScript;
55      cmd [13] = (byte)(progExitScript >> 8);
56      cmd [14] = UPLOAD;
57      // Envoyer le paquet USB
58      USBFunctions.hidWrite(cmd);
59      byte[] response = new byte[3];
60      while (programmer.read(response, 500) < 0);
61      // response[1] contient le byte le plus significatif
62      int devID = response[1] * 0x100 + response[0];
63      // Masquer les bits en excès.
64      devID &= devIdMask;
65
66      String deviceQuery = "SELECT * FROM DEVICES WHERE DEVID="+devID;
67      rs = DbUtil.execQuery(deviceQuery);
68      // Si la requête ne renvoie aucune ligne, un PIC avec cet identifiant n'existe pas.
69      if (!rs.first()) return 0;
70      return devID;
71  }
```

Listing 3.7 – Identification d'un PIC

3. Après avoir identifié le PIC, un objet du type DeviceInfo est créé pour contenir toutes les informations relatives à ce MCU.

3.3.3 *Importation du fichier HEX*

Avant de programmer le dispositif, un fichier contenant le nouveau programme doit être importé dans le logiciel. Les fichiers HEX utilisés pour les microcontrôleurs PIC suivent le format "Intel HEX". Dans ce format, le fichier est écrit sous forme d'un ensemble de lignes de texte ASCII (American Standard Code for Information Interchange).

Chaque ligne du fichier se compose de six champs :

```
:llaaaatt[dd...]cc
```

Listing 3.8 – Format de ligne en Intel HEX

Ci-après la description de chaque champ de la ligne :

1. Code de début, le caractère qui précède chaque ligne est le deux-points ' :', à l'exception des commentaires insérés par le compilateur qui commencent par un ' ;'.

2. Nombre d'octets (ll), représente le nombre d'octets de données (dd) dans la ligne (en hexadécimal).
3. L'adresse (aaaa), c'est l'adresse absolue (ou relative) du début de la ligne.
4. Type de ligne (tt), indique le type de la ligne. Dans le cas des microcontrôleurs PIC, il peut prendre l'une des valeurs suivantes :
   4.1. 00, indique que la ligne contient des données.
   4.2. 01, EOF (End Of File) il apparaît une fois dans la dernière ligne du fichier.
   4.3. 02, Champ d'adresse étendue. Utilisé quand une adresse 16 bits n'est pas suffisante. L'adresse spécifiée par le champ 02 est décalée de 4 bits vers la gauche et additionnée aux adresses contenues dans les champs de type 00.
   4.4. 04, Adresse linéaire étendu. Permet un mode d'adressage 32 bits. Cette ligne contient deux octets de données, ces octets représentent les 16 bits de poids fort des adresses 32 bits, quand ils sont combinés avec les adresses dans les lignes de type 00.
5. Données (dd...), représente le champ des données.
6. Checksum (cc), compliment à deux de la somme de tous les octets de la ligne.

L'importation du fichier HEX dans le logiciel consiste à parser chaque ligne et à calculer l'emplacement des octets de données, s'ils sont dans la mémoire programme, EEPROM, registres de configuration ...

La lecture du fichier peut se faire en Java en utilisant "BufferedReader".

```
try (BufferedReader hexReader = new BufferedReader(new FileReader(hexFile))) {
    String hexLine;
    while ((hexLine = hexReader.readLine()) != null) {
            // Traitement de la ligne
    }
    hexReader.close();
} catch (IOException e) {}
```

Listing 3.9 – Lecture du fichier HEX en Java

La version simplifiée de la méthodologie utilisée pour interpréter chaque ligne du fichier est illustrée dans le Listing 3.10 suivant.

```
private boolean importHex() {
    // Les adresses de ligne sont relatives à cette adresse
    int addressbase = 0;
    /*
    ObyteIdxenir la taille de la mémoire programme en octets, taille en mots * # d'octets dans un
        mot
    "device" est l'objet de la classe DeviceInfo, qui a été créé après l'identification du PIC cible.
    */
    int progMemSizeBytes = device.getProgMemSize() * device.getProgMemHexBytes();
```

```
9   // Déclaration et initialisation des tampons
10  int[] progMemBuffer = device.getProgMem();
11  int[] eeMemBuffer = device.getEepromMem();
12  int[] configs = device.getConfig();
13  int[] userIdBuffer = device.getUserID();
14  // Adresse de la mémoire EEPROM
15  int eeMemAddress = device.getEeMemAddr();
16  // Nombre d'octets dans un mot EEPROM
17  byte eeMemBytes = device.getEeMemHexBytes();
18  if (hexLine.charAt(0) == ':' && hexLine.length() > 10) {
19      int byteCount = Integer.parseInt(hexLine.substring(1, 3), 16); // LL
20      int fileAddress = addressbase + Integer.parseInt(hexLine.substring(3, 7), 16); // AAAA
21      int lineType = Integer.parseInt(hexLine.substring(7, 9), 16); // TT
22      if (lineType == 2 || lineType == 4) {
23          addressbase = Integer.parseInt(hexLine.substring(9,13), 16);
24          // Adresse étendue
25          if (lineType == 2) {
26              addressbase <<= 4;
27          } else {
28              // Adresse linéaire étendu
29              addressbase <<= 16;
30          }
31      } else if (lineType == 1) { // EOF
32          break; // Sortir de la boucle while
33      } else if (lineType == 0) { // Ligne de données
34          for (int lineByte = 0; lineByte < byteCount; lineByte++) {
35              int byteAddress = fileAddress + lineByte;
36              int arrayAddress = byteAddress / device.getProgMemHexBytes();
37              // Position de l'octet dans le mot
38              int bytePosition = byteAddress % device.getProgMemHexBytes();
39              int wordByte = 0x7FFFFF00 | Integer.parseInt(hexLine.substring(9+(2 *
                    lineByte), 11 + (2 * lineByte)), 16);
40              // Mettre l'octet en position correcte
41              for (byte shift = 0; shift < bytePosition; shift++) {
42                  wordByte <<= 8;
43                  wordByte |= 0xFF; // Mettre les bits à un
44              }
45              /* PROGRAM MEMORY */
46              // Ajouter l'octet au progMemBuffer s'il se trouve dans l'espace de la mémoire
                    programme.
47              if (byteAddress >= 0 && byteAddress < progMemSizeBytes) {
48                  progMemBuffer[arrayAddress] &= wordByte;
49              }
50              /* EEPROM */
51              if ((eeMemAddress > 0) && (byteAddress >= eeMemAddress) &&
                    (device.getEeMemSize() > 0)) {
52                  // Emplacement de l'octet dans l'EEPROM
53                  int eeAddress = (byteAddress - eeMemAddress) / eeMemBytes;
54                  if (eeMemBytes == device.getProgMemHexBytes()) {
55                      eeMemBuffer[eeAddress] &= wordByte;
56                  } else {
57                      int eeShift = (bytePosition / eeMemBytes) * eeMemBytes;
```

```
58                          for (byte shift = 0; shift < eeShift; shift++) {
59                              wordByte >>= 8;
60                          }
61                          eeMemBuffer[eeAddress] &= wordByte;
62                      }
63                  }
64              }
65          }
66      }
67  }
```

Listing 3.10 – Parsing de chaque ligne HEX

Le Listing 3.10 montre l'importation des octets qui se trouvent dans la mémoire programme ainsi que ceux qui se trouvent dans l'EEPROM, idem pour les mots de configuration et les User ID.

Une fois que tous les octets ont été importés dans les tampons de mémoire programme, EEPROM, configuration et User ID, ils sont passés dans l'objet "device" et peuvent être utilisés pour des opérations telles que la programmation et la vérification. Si l'octet ne correspond à aucune des régions de mémoire, alors c'est un fichier HEX invalide, et le message indiqué dans la Figure 3.5 sera affiché.

```
1  device.setProgMem(progMemBuffer);
2  device.setEepromMem(eeMemBuffer);
3  device.setConfig(configs);
4  device.setUserID(userIdBuffer);
```

Listing 3.11 – Passage des buffers à l'objet "device"

3.3.4 *La programmation du PIC cible*

Le processus fondamental que le programmateur doit effectuer est l'écriture des zones de mémoire interne du MCU cible. Ce qui suit est la façon dont cette opération est réalisée.

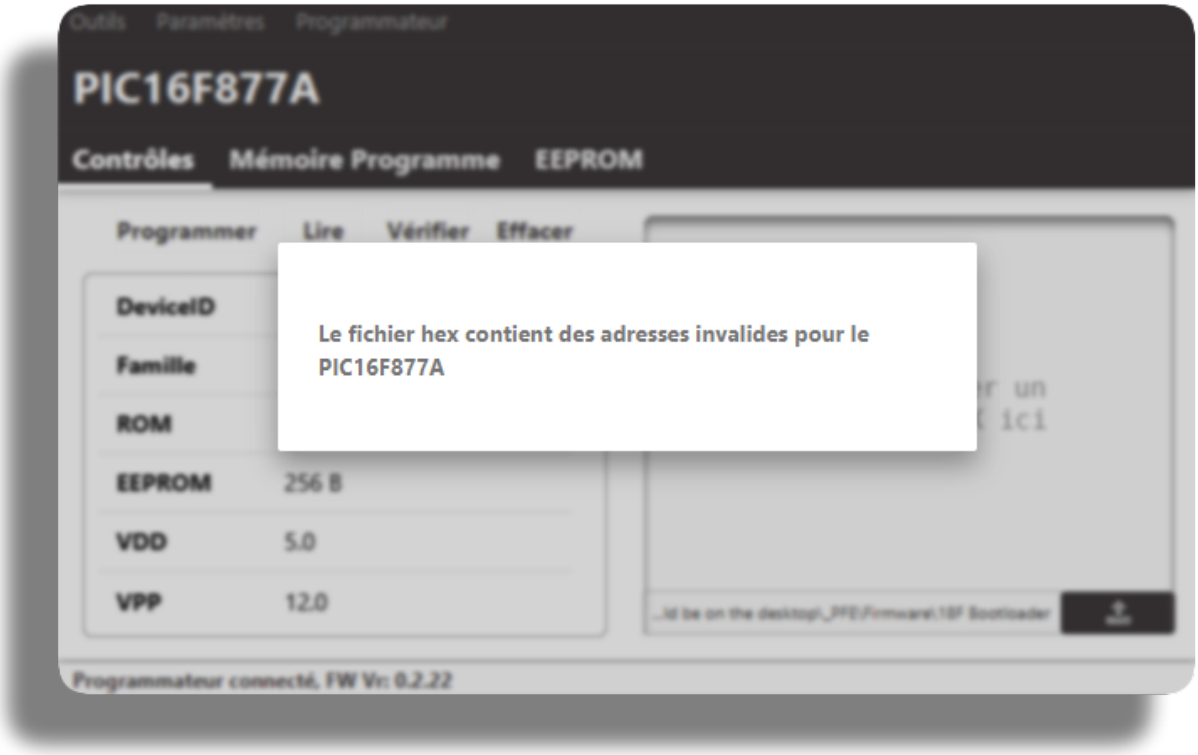

FIGURE 3.5 – Message d'erreur de fichier HEX invalide

```
// Copier le contenu des mémoires dans des tampons
int[] progMemBuffer = device.getProgMem();
int[] eeMemBuffer = device.getEepromMem();
int[] configBuffer = device.getConfig();
iint[] userIdBuffer = device.getUserID();

// Faire entrer le PIC en mode programmation
PwJFunctions.runScript(device.getProgEntryScriptLen(), device.getProgEntryScript());
/*
Un bloc entier de Flash est écrit en une seule fois, le nombre de mots dans un bloc
change d'un MCU à un autre.
*/
int wordsPerWrite = device.getProgMemWrWords();
// Combien d'octets dans un mot instruction
int bytesPerWord = device.getBytesPerLocation();
/*
Le MCU du programmateur dispose d'un buffer de 256 B pour stocker les octets envoyés par le
    PC.
Les scripts qui écrivent dans la mémoire du cible, écrivent plusieurs mots simultanément,
ce nombre est indiqué par "wordsPerWrite".
Combien de fois les scripts doivent s'exécuter avant qu'ils écrivent tous les octets dans le buffer.
*/
```

```
22 int scriptRunsToUseDownload = 256 / (bytesPerWord * wordsPerWrite);
23
24 // Le nombre de mots qui ont été écrits
25 int wordsWritten = 0;
26
27 // Indice de la dernière valeur non vide dans la mémoire programme
28 endOfBuffer = PwJFunctions.findLastUsedInBuffer (progMemBuffer, device.getBlankValue(),
       endOfBuffer);
29
30 /*
31 Nombre de fois que la boucle 'for' ci-dessous va itérer pour envoyer la totalité du buffer de
32 la mémoire programme
33 */
34 int writes = (endOfBuffer + 1) / wordsPerLoop;
35 // Nombre de mots à écrire
36 endOfBuffer = writes * wordsPerLoop;
37 // Utilisé pour simuler le buffer qui se trouve dans le firmware
38 byte[] downloadBuffer = new byte[256]
39 do {
40       int downloadIndex = 0;
41      for (int word = 0; word < wordsPerLoop; word++) {
42           if (wordsWritten == endOfBuffer) {
43                                break;
44           }
45           int memWord = progMemBuffer[wordsWritten++];
46           /*
47                Certaines familles exigent que le mot soit décalé d'une position.
48                De ces familles, actuellement seule la famille avec le
49                FAMILY_ID de 11 est implémentée.
50           */
51           if (activeFamily == 11)
52                                memWord = memWord << 1;
53
54           downloadBuffer[downloadIndex++] = (byte) memWord;
55           for (int byteIdx = 1; byteIdx < bytesPerWord; byteIdx++) {
56                memWord >>= 8;
57                downloadBuffer[downloadIndex++] = (byte) memWord;
58           }
59      }
60      /*
61           Envoi des octets au firmware, puisque la longueur maximale des paquets
62           est de 64 octets, le processus doit être répété plusieurs fois.
63      */
64      int dataIndex = PwJFunctions.clearAndDownload(downloadBuffer, 0);
65      while (dataIndex < downloadIndex)
66      {
67           dataIndex = PwJFunctions.downloadData(downloadBuffer, dataIndex, downloadIndex);
68      }
69      /*
70           Exécutez le script qui écrira la mémoire du programme jusqu'à  ce que
71           tous les octets du buffer soient écrits.
```

```
72      */
73      PwJFunctions.runScriptItr (device.getProgMemWrScriptLen(),
            device.getProgMemWrScript(), (byte) scriptRunsToUseDownload);
74  } while (wordsWritten < endOfBuffer);
75  // Quitter le mode de programmation
76  PwJFunctions.runScript(device.getProgExitScriptLen(), device.getProgExitScript());
```

Listing 3.12 – Implémentation de l'écriture du PIC cible

Listing 3.12 ci-dessus décrit le processus utilisé pour écrire la mémoire programme. Une approche similaire peut être utilisée pour programmer les autres régions.

3.3.5 *La lecture du PIC*

La lecture du MCU cible consiste à afficher le contenu de la mémoire dans l'interface graphique, ce qui se fait généralement à des fins de débogage.

La Figure 3.6 montre comment le contenu de la mémoire est affiché dans le logiciel.

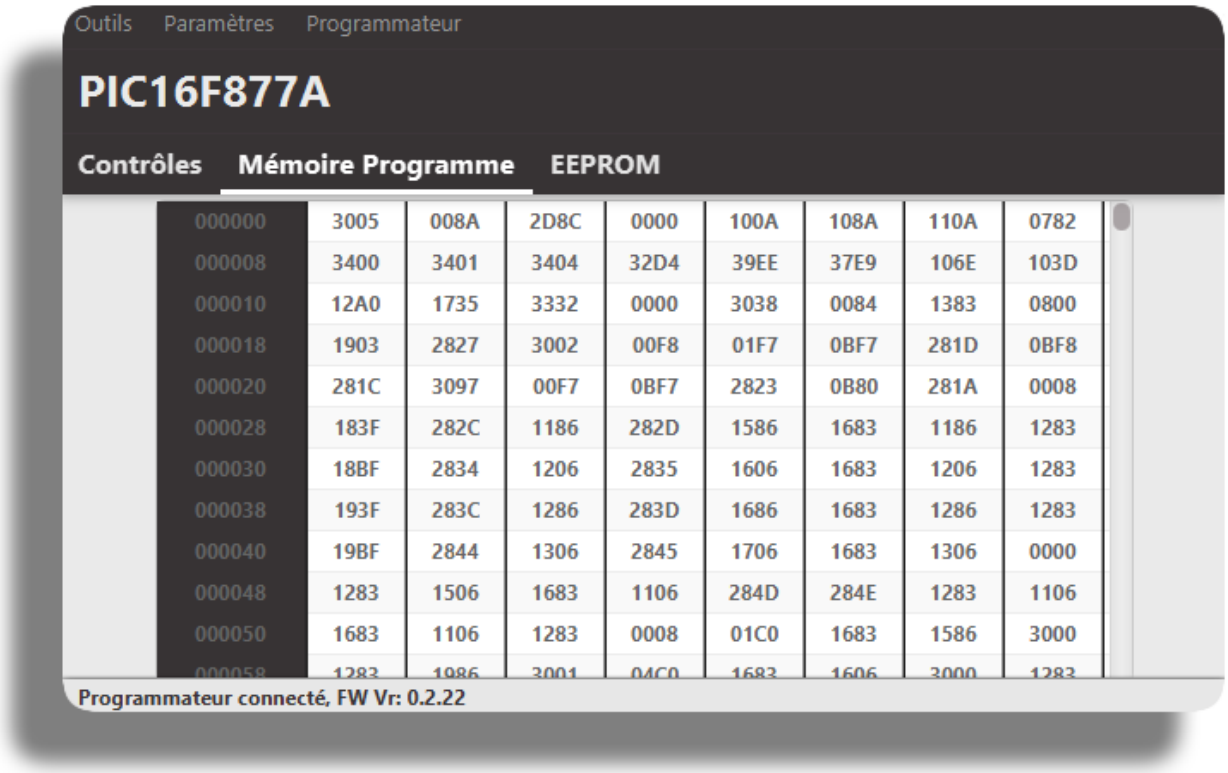

FIGURE 3.6 – Visualisation du contenu de la mémoire programme

Le processus est mis en œuvre du côté de l'hôte comme suit :

```
1  // Copier le contenu des mémoires dans des tampons
```

```
int[] progMembuffer = device.getProgMem();
int[] eeMemBufer = device.getEepromMem();
int[] userIdBuffer = device.getUserID();
int[] configBuffer = device.getConfig();
// Utilisé pour simuler le buffer utilisé pour les transferts PIC vers PC
byte[] uploadBuffer = new byte[128];
// Faire entrer le PIC en mode programmation
PwJFunctions.runScript(device.getProgEntryScriptLen(), device.getProgEntryScript());
// Nombre d'octets dans un seul mot d'instruction
int bytesPerWord = device.getBytesPerLocation();
/* Le nombre de fois que le script qui lit la mémoire doit être exécuté
    pour remplir le tampon.
*/
int scriptRunsToFillUpload =128 / (device.getProgMemRdWords() * bytesPerWord);

// Le nombre de mots qui seront lus jusqu'à ce que le tampon PIC vers PC se remplisse.
int wordsPerLoop = scriptRunsToFillUpload * device.getProgMemRdWords();
// Le nombre de mots qui ont été lus
int wordsRead = 0;
do {
    PwJFunctions.runScriptItr(device.getProgMemRdScriptLen(),
        device.getProgMemRdScript(), (byte) scriptRunsToFillUpload);
    // Utilisé pour contenir les données du paquet USB
    byte[] uploadedData;
    // Envoyer les données au PC sans inclure la longueur et sans effacer le tampon.
    uploadedData = PwJFunctions.uploadData(false, false);
    if (uploadedData == null) {
        // L'opération a échoué, notifier l'utilisateur
        Platform.runLater(() -> {
        // Cela produira un message similaire à celui de la Figure 2.5 avec le texte "Lecture
            échouée"
            Prompt.alert("Lecture échouée", rootPane, rootAnchorPane);
        });
        return false;
    }
    // Copiez les 64 octets du paquet USB dans le tampon uploadBuffer à partir de la position 0 .
    System.arraycopy(uploadedData, 0, uploadBuffer, 0, 64);
    uploadedData = PwJFunctions.uploadData(false, false);
    // Copiez les 64 octets du paquet USB dans le tampon uploadBuffer à partir de la position 64.
    System.arraycopy(uploadedData, 0, uploadBuffer, 64, 64);
    int uploadIndex = 0;
    for (int word = 0; word < wordsPerLoop; word++) {
        // Les octets d'un seul mot qui ont été lus
        int byteIdx = 0;
        int memWord = uploadBuffer[uploadIndex + byteIdx++];
        // Ajuster les valeurs négatives
        if (memWord < 0) memWord += 256;
        // Lire le deuxième octet du mot
        if (byteIdx < bytesPerWord)
            memWord |= (uploadBuffer[uploadIndex + byteIdx++] << 8);

        uploadIndex += byteIdx;
```

```
52          /*
53              Certaines familles exigent que le mot soit décalé d'une position.
54              De ces familles, actuellement seule la famille avec le
55              FAMILY_ID de 11 est implémentée.
56          */
57          if (activeFamily == 11)
58              memWord = (memWord >> 1) & device.getBlankValue();
59
60          progMembuffer[wordsRead++] = memWord;
61          // Si tous les mots ont été lus, quitter la boucle for
62          if (wordsRead == device.getProgMemSize())
63              break;
64      }
65
66  } while (wordsRead < device.getProgMemSize());
67
68  PwJFunctions.runScript(device.getProgExitScriptLen(), device.getProgExitScript());
```

Listing 3.13 – Lecture du PIC

3.3.6 *L'éffacement du PIC cible*

Les microcontrôleurs PIC peuvent être effacés par l'une des deux méthodes suivantes : effacement "global" (bulk erase) ou effacement par ligne (row erase). Dans le premier cas, la mémoire programme et la mémoire EEPROM sont effacées au complet simultanément et, dans le second, des blocs individuels de données sont effacés. Néanmoins, la seconde méthode ne peut être utilisée par tous les microcontrôleurs.

L'effacement en "bulk" nécessite un intervalle V_{DD} plus étroit pour être implémenté avec succès, qui pour la plupart des appareils se situe entre 3,5V et 5,5V. L'effacement par ligne nécessite cependant une tension minimale d'environ 2V. De ce fait, ce dernier sera utilisé si le PIC cible le supporte.

```
1  // device.getRowEraseSize() retourne la taille de la ligne qui sera effacée
2  // Si le PIC ne supporte pas l'effacement de ligne, elle sera égale à zéro.
3  if (device.getRowEraseSize() > 0) {
4      PwJFunctions.rowErase(device);
5  } else {
6      if (device.getEeMemSize() > 0) {
7          PwJFunctions.bulkErase(device); // Utiliser l'effacement global
8      } else if (device.getProgMemEraseScript() > 0) {
9      // Si le PIC n'a pas d'EEPROM, effacer uniquement la mémoire programme
10         PwJFunctions.progMemErase(device);
11     }
12 }
```

Listing 3.14 – L'ffacement du PIC cible

Le Listing 3.14 montre comment le processus d'effacement a été implémenté, si le PIC supporte l'effacement par ligne, la méthode "rowErase" sera appelée. S'il ne le supporte pas, "bulkErase" sera exécuté si le PIC cible possède une EEPROM. Sinon, seule la mémoire programme sera effacée avec "progMemErase". Listing 3.15, Listing 3.16 et Listing 3.17 montrent comment ces trois fonctions ont été mises en œuvre.

```
// Le nombre de lignes qui seront effacées
int memoryRows = device.getProgMemSize() / device.getRowEraseSize();
// Faire entrer le PIC en mode programmation
runScript(device.getProgEntryScriptLen(), device.getProgEntryScript());
byte[] rowEraseCmd = new byte[5];
rowEraseCmd[0] = RUN_ROM_SCRIIPT_ITR;
rowEraseCmd[1] = device.getRowEraseScriptLen();
rowEraseCmd[2] = (byte)device.getRowEraseScript();
rowEraseCmd[3] = (byte)(device.getRowEraseScript() >> 8);
do {
    if (memoryRows >= 256) {
    // Le nombre maximum d'itérations est de 256, ce qui est représenté par l'envoi d'un zéro.
        rowEraseCmd[4] = 0;
    // 256 lignes ont été effacées, on les soustrait du nombre total de lignes
        memoryRows -= 256;
    }
    else {
        rowEraseCmd[4] = (byte) (memoryRows & 0xFF);
        memoryRows = 0;
    }
    USBFunctions.hidWrite(rowEraseCmd);
} while (memoryRows > 0);

runScript(device.getProgExitScriptLen(), device.getProgExitScript());
// Effacer la mémoire de configuration
if (device.getConfigMemEraseScript() > 0) {
    runScript(device.getProgEntryScriptLen(), device.getProgEntryScript());
    runScript(device.getConfigMemEraseScriptLen(), device.getConfigMemEraseScript());
    runScript(device.getProgExitScriptLen(), device.getProgExitScript());
}
```

Listing 3.15 – La fonction "rowErase"

```
// Effacer la mémoire de configuration
if (device.getConfigMemEraseScript() > 0) {
    runScript(device.getProgEntryScriptLen(), device.getProgEntryScript());
    runScript(device.getConfigWrScriptLen(), device.getConfigWrScript());
    runScript(device.getProgExitScriptLen(), device.getProgExitScript());
}
runScript(device.getProgEntryScriptLen(), device.getProgEntryScript());
// Le script qui va effectuer l'effacement en bulk
```

```
runScript(device.getChipEraseScriptLen(), device.getChipEraseScript());
runScript(device.getProgExitScriptLen(), device.getProgExitScript());
```

Listing 3.16 – La fonction “bulkErase”

```
runScript(device.getProgEntryScriptLen(), device.getProgEntryScript());
// Le script qui va effectuer l'effacement de la mémoire programme
runScript(device.getProgMemEraseScriptLen(), device.getProgMemEraseScript());
runScript(device.getProgExitScriptLen(), device.getProgExitScript());
```

Listing 3.17 – La fonction “progMemErase”

3.3.7 *Vérification du PIC cible*

La vérification d'un microcontrôleur consiste à comparer le contenu de ses mémoires avec un fichier HEX importé. Le processus est similaire à celui de la lecture. Listing 3.18 montre comment il est réalisé.

```
int[] progMemBuffer = device.getProgMem();
int[] eeMemBuffer = device.getEepromMem();
int[] userIdBuffer = device.getUserID();
int[] configBuffer = device.getConfig();
byte[] uploadBuffer = new byte[128];

// Faire entrer le PIC en mode programmation
PwJFunctions.runScript(device.getProgEntryScriptLen(), device.getProgEntryScript());
// Nombre d'octets dans un seul mot d'instruction
int bytesPerWord = device.getBytesPerLocation();
/*
    Le nombre de fois que le script qui lit la mémoire doit être exécuté
    pour remplir le tampon.
*/
int scriptRunsToFillUpload =128 / (device.getProgMemRdWords() * bytesPerWord);

// Le nombre de mots qui seront lus jusqu'à ce que le tampon PIC vers PC se remplisse.
int wordsPerLoop = scriptRunsToFillUpload * device.getProgMemRdWords();
// Le nombre de mots qui ont été lus
int wordsRead = 0;

do {
    PwJFunctions.runScriptItr(device.getProgMemRdScriptLen(),
        device.getProgMemRdScript(), (byte) scriptRunsToFillUpload);
    // Pour contenir les données du paquet USB
    byte[] uploadedData;
    // Envoyer les données au PC sans inclure la longueur et sans effacer le tampon.
```

```
27      uploadedData = PwJFunctions.uploadData(false, false);
28      System.arraycopy(uploadedData, 0, uploadBuffer, 0, 64);
29      uploadedData = PwJFunctions.uploadData(false, false);
30      System.arraycopy(uploadedData, 0, uploadBuffer, 64, 64);
31              int uploadIdx = 0;
32              for (int word = 0; word < wordsPerLoop; word++) {
33          int byteIdx = 0;
34          int memWord = uploadBuffer[uploadIdx + byteIdx++];
35          if (memWord < 0) memWord += 256;
36          if (byteIdx < bytesPerWord)
37              memWord |= (uploadBuffer[uploadIdx + byteIdx++] << 8);
38          uploadIdx += byteIdx;
39          if (activeFamily == 11)
40              memWord = (memWord >> 1) & device.getBlankValue();
41          /*
42              Si le mot lu à partir du microcontrôleur n'est pas le même que le mot correspondant
43              dans le tampon, la vérification a échoué.
44          */
45          if (memWord != progMemBuffer[wordsRead++]) {
46              // Quitter le mode de programmation
47              PwJFunctions.runScript(device.getProgExitScriptLen(), device.getProgExitScript());
48              // Afficher le message approprié
49              Platform.runLater (() -> {
50                  Prompt.alert("La vérification a échoué", rootPane, rootAnchorPane);
51              });
52              return false;
53          }
54          if (wordsRead > endOfBuffer) break;
55      }
56  } while (wordsRead < endOfBuffer);
57  PwJFunctions.runScript(device.getProgExitScriptLen(), device.getProgExitScript());
```

Listing 3.18 – La vérification du PIC cible

3.4 CONCLUSION

Ce chapitre couvrait le processus de développement d'une application JavaFX, la conception du schéma de la base de données, les protocoles établis entre le logiciel et le firmware ainsi qu'une description de la façon dont les différents processus du programmateur sont implémentés du côté logiciel.

Il convient de noter que les codes listés dans ce chapitre ne concernent que les opérations qui manipulent la mémoire programme et ne servent qu'à donner une idée générale de la façon dont les processus sont exécutés, pour une implémentation complète, il est nécessaire de se référer au code source.

CHAPITRE 4

4

LE BOOTLOADER

4.1 INTRODUCTION

Initialement, ce qui a suscité l'intérêt pour le développement du programmateur pour les microcontrôleurs PIC était le besoin d'outils tels que les cartes de développement et de prototypage, qui sont généralement des circuits simples mais le mécanisme utilisé pour injecter le code de l'application dans le microcontrôleur est moins clair.

Pour éviter d'utiliser un programmateur externe, ces cartes sont généralement préprogrammées avec un bootloader. Dans le contexte des microcontrôleurs Flash, un bootloader est un programme qui peut écrire un autre programme (celui de l'application principale) dans la ROM du microcontrôleur. Il ne peut évidemment écrire que dans la partie de la mémoire qui n'est pas occupée par le bootloader lui-même. Ce qui est une incitation à le rendre aussi petit que possible.

4.2 LES PROCESSUS HÔTES

Tout comme le firmware du programmateur, le bootloader nécessite la coordination d'un logiciel s'exécutant sur un PC et le code dans le microcontrôleur. Actuellement les fonctionnalités implémentées dans le logiciel sont la lecture du fichier HEX et l'écriture dans le bootloader. Pour communiquer avec le bootloader au lieu du firmware, il suffit de sélectionner Programmateur 〉 18F Bootloader.

4.2.1 *Connexion au bootloader*

Le bootloader est également configuré comme un périphérique HID, donc la connexion se fait de la même manière que la connexion au firmware. Listing 4.1 montre l'exemple minimal fonctionnel de la façon dont elle a été implémentée.

```
1  // Itérer sur tous les dispositifs HID connectés
2  for (HidDevice hidDevice : hidServices.getAttachedHidDevices()) {
3      if (hidDevice.getProductId() == 0x0001 && hidDevice.getVendorId() == 0x0025) {
4          programmer = hidDevice;
5          // Ouvrir le canal de communication
6          programmer.open();
7          byte[] cmd = new byte[1];
8          // Vérifier la communication en lisant la version du bootloader
9          cmd[0] = GET_BOOTLOADER_VERSION;
10         USBFunctions.hidWrite(cmd);
11         byte[] response = new byte[1];
```

```
12          while (programmer.read(response, 500) < 0);
13          /*
14              Après avoir reçu la commande "GET_BOOTLOADER_VERSION", le bootloader
                    doit retourner 17
15          */
16          if (response[0] == 17)
17              return true;
18      }
19  }
```

Listing 4.1 – Connexion au bootloader

4.2.2 *Importation du fichier HEX pour le bootloader*

L'importation du fichier HEX pour le firmware du programmateur consistait à lire chaque octet et à le placer dans des tampons représentant différentes régions de mémoire pour le PIC cible (mémoire programme, EEPROM, mots de configuration . . .). Dans le cas du bootloader, le processus est différent, seuls les octets qui se trouvent dans la mémoire programme et l'EEPROM sont importés. Le fichier HEX est envoyé ligne par ligne, chaque ligne commence par le nombre d'octets dans cette ligne, l'adresse, et les octets de données. Un exemple minimal fonctionnel est présenté à le Listing 4.2.

```
1   byteCount = (byte) Integer.parseInt(hexLine.substring(1, 3), 16); // LL
2   lineAddress = addressBase + Integer.parseInt(hexLine.substring(3, 7), 16); // AAAA
3   lineType = (byte) Integer.parseInt(hexLine.substring(7, 9), 16); // TT
4   if (lineType == 2 || lineType == 4) {
5       addressBase = Integer.parseInt(hexLine.substring(9,13), 16);
6       if (lineType == 2) addressBase <<= 4;
7       else addressBase <<= 16;
8   } else if (lineType == 0) {
9       // Ajoutez l'adresse si elle se trouve dans la mémoire EEPROM ou de programme
10      if (lineAddress < 0x7FFF || (lineAddress >= 0x00F00000 && lineAddress <
                (0x00F00000+256))) {
11          hexFileBytes.add (byteCount);
12          hexFileBytes.add ((byte) lineAddress);
13          hexFileBytes.add ((byte) (lineAddress >> 8));
14          hexFileBytes.add ((byte) (lineAddress >> 16));
15          hexFileBytes.add ((byte) (lineAddress >> 24));
16      }
17      for (int lineByte = 0; lineByte < byteCount; lineByte++) {
18          int byteAddress = lineAddress + lineByte;
19          int eeAddress = (byteAddress - eeMemAddress) / eeMemBytes;
20          int wordByte = 0x7FFFFF00 | Integer.parseInt(hexLine.substring(9+(2 * lineByte), 11 +
                    (2 * lineByte)), 16);
21          // Si l'octet est en mémoire programme ou en EEPROM
22          if (byteAddress < progMemSizeBytes) || (byteAddress >= eeMemAddress &&
                    eeAddress < 256)) {
```

```
                hexFileBytes.add ((byte) (wordByte & 0xFF));
            }
        }
    }
```

Listing 4.2 – Parsing du fichier HEX pour le bootloader

"hexFileBytes" est un "ArrayList" d'octets puisque le nombre de lignes à envoyer peut varier d'un programme à l'autre, seules les adresses et les octets qui se situent en mémoire programme ou espaces EEPROM y sont ajoutés.

4.2.3 *Envoi des données au bootloader*

Après avoir chargé les octets à envoyer à "hexFileBytes", le contenu de ce dernier est envoyé au bootloader une ligne à la fois. Comme le montre l'exemple minimal fonctionne montré dans le Listing 4.3.

```
byte[] cmd = new byte[1];
// Le contenu d'une seule ligne de fichier HEX est stocké ici
byte[] lineBytes;
// Le nombre d'octets de données dans la ligne
byte lineSize;
// Le nombre d'octets lus à partir de "hexFileBytes"
int byteIdx = 0;
// Le nombre d'octets restant dans "hexFileBytes"
int remainingBytes = hexFileBytes.size();
// Itérer jusqu'à  ce qu'il ne reste plus d'octets dans "hexFileBytes"
while (remainingBytes > 0) {
    // taille = nbr d'octets + 4 octets d'adresse + 1 octet pour le comptage des octets
    lineBytes = new byte[hexFileBytes.get(byteIdx) + 5];
    lineSize = (byte) lineBytes.length;
    // Copier la ligne depuis "hexFileBytes" vers "lineBytes"
    for (byte i = 0; i < lineSize; i++) {
        lineBytes[i] = hexFileBytes.get(byteIdx++);
    }
    // Soustraire le nombre d'octets copiés des octets restants
    remainingBytes -= lineSize;
    // Envoyer la ligne
    USBFunctions.hidWrite(lineBytes);
    // Attendez que le bootloader envoie la commande RESUME_COMM
    while (programmer.read(response, 500) < 0 && response[0] != RESUME_COMM) ;
}
// Tous les octets ont été envoyés, quittez le bootloader
cmd[0] = QUIT_BOOT;
USBFunctions.hidWrite(cmd);
```

Listing 4.3 – Envoi des données au bootloader

4.3 LE CODE MICROCONTRÔLEUR

Lorsque le microcontrôleur est mis sous tension ou lorsqu'une réinitialisation se produit, le bootloader lit une broche d'entrée pour vérifier si un bouton-poussoir est appuyé, si oui, le MCU reste en mode bootloader et attend le nouveau code firmware transmis via USB, si le bouton est relâché alors le contrôle sera transféré au firmware principal du microcontrôleur. Dans notre cas, la broche choisie est RB5 pour le bouton-poussoir, mais n'importe quelle autre entrée numérique peut être utilisée. la Figure 4.1 illustre le circuit requis pour le bootloader et Listing 4.4 montre comment cela est mis en œuvre en code.

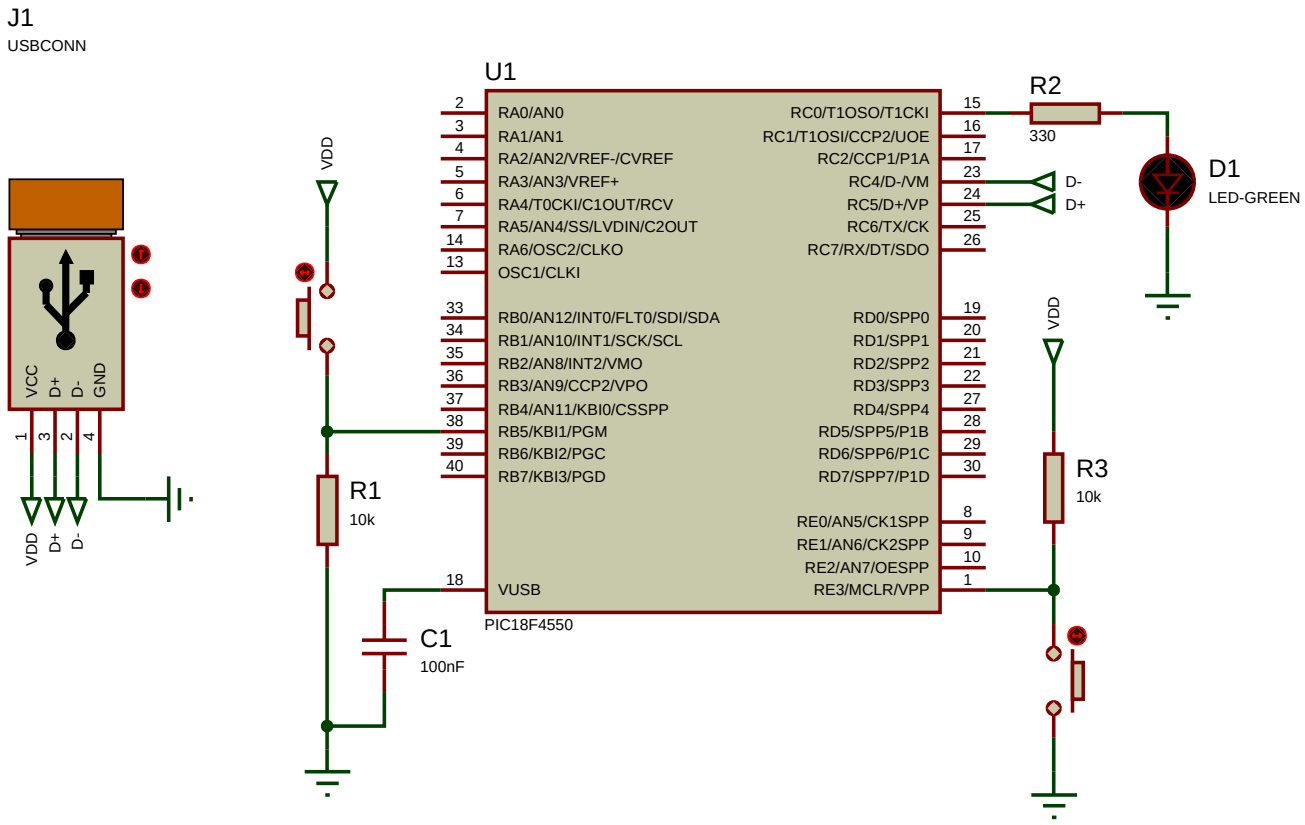

FIGURE 4.1 – Circuit minimal pour le bootloader

```
1  void main()
2  {
3      TRISD = 0x7F; // Broche LED configurée comme sortie.
4      TRISB = 0xFF; // Broche de BP configurée comme entrée.
5      LED = 0;  // Initialement la LED est éteinte
6      if (PB) {
7      // Le BP est pressé, entrer en mode bootloader
8          boot_routine ();
9          // Une fois que le MCU sort du bootloader, il sera réinitialisé.
10         reset_cpu ();
```

```
    } else {

        // Le BP est relâché, allez a l'endroit où l'application principale commence.
        // Avec ce bootloader, l'application principale commence à l'adresse 0x1400
         #asm
            goto APPLICATION_SPACE
         #endasm
    }
}
// Fonction vide, sera remplacée par la nouvelle application.
#org APPLICATION_SPACE, APPLICATION_SPACE+0x22
void empty_app () {
    while (TRUE);
}
```

Listing 4.4 – La fonction “main” du bootloader

Il est à noter que le PIC C ne reconnaît pas les noms des SFR et leurs différents bits. Ainsi, pour utiliser un registre ou un bit, les directives “#byte” et “#bit” doivent être utilisées, comme démontré dans la sous-sous-section 1.4.3.2.

Le bootloader ne génère pas d’interruptions, donc si une interruption se produit, la source doit être l’application principale, et le bootloader doit rediriger le MCU vers l’emplacement où se trouvent les vecteurs d’interruption de l’application. Ceci est fait en utilisant la fonction indiquée dans Listing 4.5 avec la fonction “main”.

```
#int_default
void isr (void) {
    jump_to_isr (APPLICATION_SPACE+8);
}
```

Listing 4.5 – Redirection des interruptions vers l’application

Le véritable point d’entrée du bootloader est la fonction “`boot_routine`”, son implémentation est la suivante :

```
void boot_routine () {
    LED = 1; // Allumer la LED pour indiquer le mode bootloader
    // Exit_boot sera mis à true lorsque l'hôte envoie la commande "QUIT_BOOT"
    int1 exit_boot = FALSE;
    // Initialiser le module USB
    usb_init_cs();
    // Attendre que le périphérique soit énuméré par l'hôte
    while (!usb_enumerated()) usb_task();
    while (!exit_boot) {
        usb_task();
        // vérifier si le endpoint1 a reçu des données
        if (usb_kbhit(1)) {
```

```
13                  exit_boot = process_hex_line ();
14              }
15          }
16          LED = 0;  // Eteindre la LED
17      }
```

Listing 4.6 – L'implémentation de la fonction `boot_routine`

Chaque paquet USB contient une ligne du fichier HEX, quand il est reçu, il est traité dans la fonction "`process_hex_line`".

```
1   int1 process_hex_line () {
2       // Copier le paquet depuis le endpoint vers un buffer de 64 octets (data_in)
3       usb_get_packet(1, data_in, 64);
4
5       // L'hôte demande le numéro de version
6       if (data_in[1] == GET_BOOTLOADER_VERSION) {
7           data_out[0] = 3; // Longueur
8           data_out[1] = 0;
9           data_out[2] = 12;
10          data_out[3] = 13;
11          // Envoyer les données
12          usb_put_packet(1, data_out, 64, USB_DTS_TOGGLE);
13          // Retourne FALSE pour que le MCU reste en mode bootloader
14          return FALSE;
15      }
16
17      // Si l'hôte envoie "QUIT_BOOT", quittez le bootloader
18      else if (data_in[1] == QUIT_BOOT) return TRUE;
19
20      unsigned int8 nbr_bytes;
21      unsigned int32 address;
22      // Le deuxième octet dans le paquet est le nombre de bytes de données dans la ligne
23      nbr_bytes = data_in[1];
24
25      // Les 4 octets suivants représentent l'adresse de départ dans laquelle les octets seront écrits
26      address = make32 (data_in[5], data_in[4], data_in[3], data_in[2]);
27
28      set_flash_buffer (address, &data_in[6], nbr_bytes);
29
30      // Informer l'hôte qu'il peut envoyer la ligne suivante
31      data_out[0] = RESUME_COMM;
32      usb_put_packet(1, data_out, 64, USB_DTS_TOGGLE);
33      // Reste en mode bootloader
34      return FALSE;
35  }
```

Listing 4.7 – La fonction "`process_hex_line`"

Les mémoires flash sont effacées et écrites en blocs, dans le PIC18F2550, la mémoire est effacée en blocs de 64 octets et écrite en blocs de 32 octets. Chaque bloc doit être effacé avant d'être écrit. Pour cela, le processus consiste à copier un bloc de 64 octets dans la RAM, à le modifier en ajoutant les données envoyées par l'hôte, à effacer le bloc original dans la ROM, puis à réécrire celui qui a été modifié en effectuant deux opérations d'écriture.

Le processus est expliqué dans la fiche technique du PIC18F2550 et est résumé dans l'organigramme présenté à la Figure 4.2. La façon dont cela a été implémenté dans le code est la suivante :

```
1  void set_flash_buffer (int32 addr, unsigned int8 *bytes_location, unsigned int8 bytes_count) {
2
3      int32 block_addr;
4      unsigned int8 addr_offset;
5      // Trouver où commence le bloc de 64 octets qui sera écrit.
6      block_addr = addr & ~((int32) 63);
7
8      // Copier le bloc en RAM
9      read_flash_to_buffer (block_addr);
10
11     // Trouver où doit commencer I' écriture des données reçues
12     addr_offset = addr - block_addr;
13
14     // Copier les données du buffer USB vers le buffer RAM
15     while (bytes_count != 0) {
16         prog_mem_buffer[addr_offset++] = *bytes_location;
17         bytes_location++; // Pointer sur l'octet suivant
18         bytes_count--; // Un octet a été copié
19     }
20     // Effacer le bloc ROM
21     erase_row (block_addr);
22
23     // Ecriture du buffer RAM modifié dans la mémoire Flash
24     write_flash_block (block_addr);
25 }
```

Listing 4.8 – Le processus d'écriture d'une ligne HEX sur la ROM

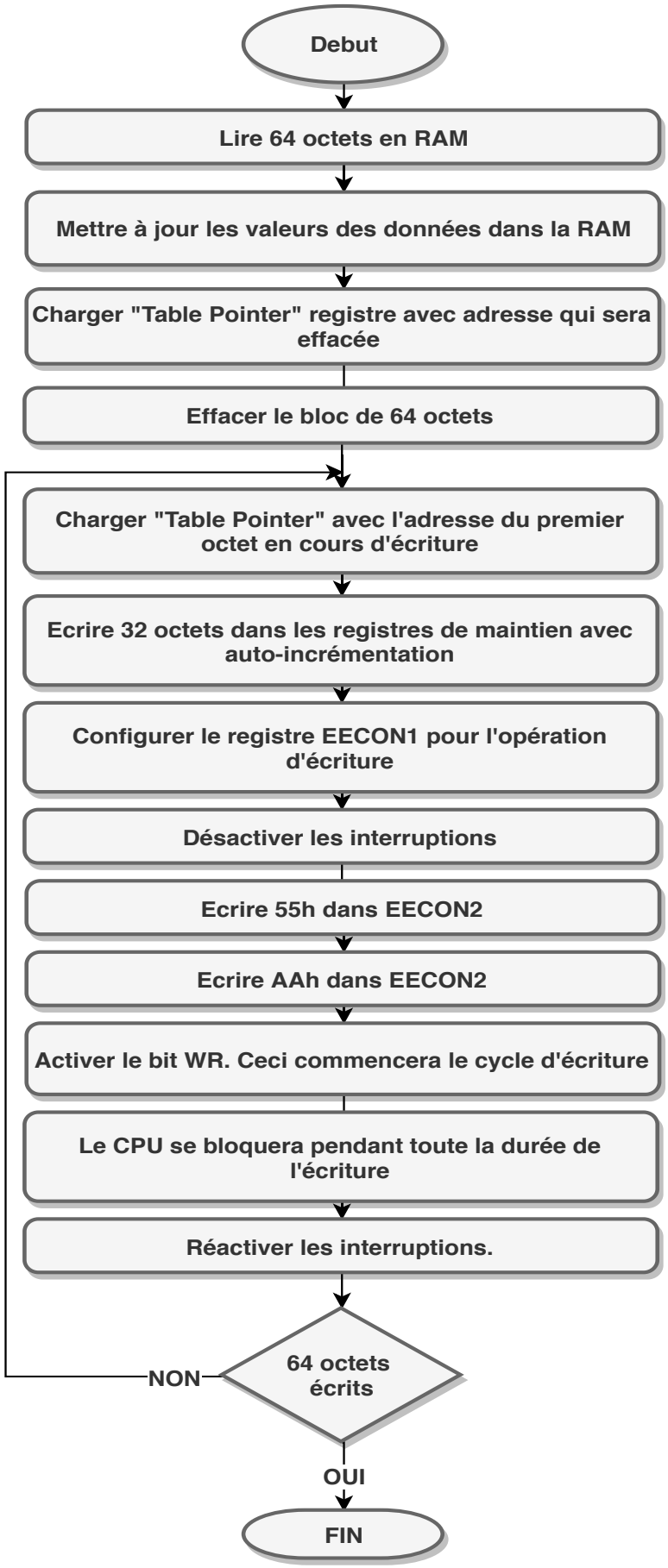

FIGURE 4.2 – Processus d'écriture d'un bloc Flash

La copie d'un bloc Flash dans la RAM s'effectue par la fonction "`read_flash_to_buffer`". Le Listing 4.9 montre sa réalisation.

```
void read_flash_to_buffer (int32 block_add) {
    // Charger l'adresse du buffer RAM dans les registres FSR0
    FSR0L = (unsigned int8) (&prog_mem_buffer & 0xFF);
    FSR0H = (unsigned int8) ((&prog_mem_buffer >> 8) & 0xFF);

    // Charge l'adresse de début du bloc Flash qui sera lu dans "Table Pointer"
    TBLPTRL = (unsigned int8) (block_add & 0xFF);
    TBLPTRH = (unsigned int8) ((block_add >> 8) & 0xFF);
    TBLPTRU = (unsigned int8) ((block_add >> 16) & 0xFF);

    // Lire 64 octets
    unsigned int8 counter = 64;
     #asm
        READ:
            // Lire la valeur pointée par "Table Pointer", puis incrémenter ce dernier
            TBLRD*+
            // Placer la valeur lue dans le registre W
            MOVF TABLAT, W
            /*
                Placer la valeur du registre W à l'adresse indiquée par les registres FSR0 puis
                    incrémenter cette adresse
            */
            MOVWF POSTINC0
            // Sortie après 64 itérations
            DECFSZ counter
            BRA READ
     #endasm
}
```

Listing 4.9 – Lecture d'un bloc de mémoire Flash

Après avoir lu le bloc Flash original et l'avoir modifié dans "`set_flash_buffer`", cet emplacement dans la ROM doit être effacé avant d'être réécrit. Ceci est réalisé par la fonction "`erase_row`".

```
void erase_row (int32 addr) {
    // Charge l'adresse de début du bloc Flash qui sera lu dans "Table Pointer"
    TBLPTRL = (unsigned int8) (addr & 0xFF);
    TBLPTRH = (unsigned int8) ((addr >> 8) & 0xFF);
    TBLPTRU = (unsigned int8) ((addr >> 16) & 0xFF);

     #asm
        BSF EECON1,7 // Bit EEPGD
        BCF EECON1,6 // Bit CFGS
        BSF EECON1,2 // Bit WREN
```

```
        BSF EECON1,4 // Bit FREE
        BCF GIE // Désactiver les interruptions
        MOVLW 0x55
        MOVWF EECON2
        MOVLW 0xAA
        MOVWF EECON2
        BSF EECON1,1 // Activer le bit WR
        BSF GIE  // Activer les interruptions
    #endasm
}
```

Listing 4.10 – Effacer un bloc de mémoire Flash

Enfin, le tampon RAM modifié peut être écrit dans le bloc de Flash qui a été effacé à l'aide de la fonction "`write_flash_block`".

```
void write_flash_block (int32 code_addr) {
    TBLPTRL = (unsigned int8) (code_addr & 0xFF);
    TBLPTRH = (unsigned int8) ((code_addr >> 8) & 0xFF);
    TBLPTRU = (unsigned int8) ((code_addr >> 16) & 0xFF);

    #asm
        // Nécessaire pour faire pointer "Table Pointer" vers l'adresse précédente
        TBLRD*-
    #endasm
    // Charger l'adresse du buffer RAM dans les registres FSR0
    FSR0L = (unsigned int8) (&prog_mem_buffer & 0xFF);
    FSR0H = (unsigned int8) ((&prog_mem_buffer >> 8) & 0xFF);

    // Compteur pour les 32 octets dans un seul bloc d'écriture
    unsigned int8 counter;

    /*
        Compteur pour les 2 blocs qui seront écrits, pour remplacer les 64 octets qui ont été
            effacés.
    */
    unsigned int8 bloc_ctr = 2;

    #asm
    WR64:
        MOVLW 32
        MOVWF counter
    WR32:
        MOVF POSTINC0, W
        MOVWF TABLAT
        /*
            Ecrire la valeur qui se trouve dans "TABLAT" à l'adresse ROM spécifiée par "Table
                Pointer". Puis incrémenter cette adresse
        */
```

```
32          TBLWT+*
33          DECFSZ counter
34          BRA WR32
35
36          BCF EECON1, 4 // Bit FREE
37          BSF EECON1, 7 // Bit EEPGD
38          BCF EECON1, 6 // Bit CFGS
39          BSF EECON1, 2 // Bit WREN
40          BCF GIE  // Désactiver les interruptions
41
42          MOVLW 0x55
43          MOVWF EECON2
44          MOVLW 0xAA
45          MOVWF EECON2
46
47          BSF EECON1,1 // Activer le bit WR
48
49          DECFSZ bloc_ctr // Compteur de blocs. Ecrire 2 blocs de 32 octets
50          BRA WR64
51
52          BSF GIE  // Activer les interruptions
53          BCF EECON1, 2 // WREN
54      #endasm
55  }
```

Listing 4.11 – Ecriture d'un bloc mémoire Flash

4.4 LA COMPATIBILITÉ D'UNE APPLICATION

Il y a quelques considérations à prendre en compte lors du développement d'une application compatible avec le bootloader.

1. Le bootloader ne modifie pas les mots de configuration, l'application doit donc être configurée de la même manière que le bootloader.

```
1  #device ADC=16
2
3  #fuses NOWDT WDT128 NOBROWNOUT NOLVP NOXINST
4  #fuses HSPLL PLL5 CPUDIV2 USBDIV VREGEN
5
6  #use delay(clock=48000000)
```

Listing 4.12 – Configuration du bootloader

2. Le compilateur devrait être informé de ne pas insérer de code dans l'espace allant de l'adresse 0 à l'adresse 0x13FF, puisque cette région est occupée par le bootloader.

```
#define APPLICATION_ADDRESS 0x1400
#org 0, (APPLICATION_ADDRESS − 1) {}
```

Listing 4.13 – Insertion du code d'application à une adresse spécifique

3. Réallocation des vecteurs de reset et d'interruption.

```
#build(reset=APPLICATION_ADDRESS, interrupt=(APPLICATION_ADDRESS+8))
```

Listing 4.14 – Remappage des vecteurs de reset et d'interruption

4.5 CONCLUSION

Ce chapitre a abordé une autre approche pour programmer les microcontrôleurs PIC, qui consiste à utiliser un bootloader. Cette méthode est optimale lorsqu'elle est implémentée dans des cartes autonomes où le logiciel PC communique directement avec le MCU cible incorporé dans ces cartes, sans avoir besoin d'un programmateur intermédiaire.

Des exemples de code qui donnent une vue d'ensemble des procédures du logiciel hôte et du bootloader sont inclus ainsi que le circuit minimal requis par ce dernier pour fonctionner.

CONCLUSION GÉNÉRALE

CONCLUSION GÉNÉRALE

La programmation des microcontrôleurs est une tâche ardue qui nécessite la coordination de différents éléments, depuis le matériel et le micrologiciel qui le contrôle jusqu'au logiciel s'exécutant sur le PC. L'objectif de ce mémoire était de développer le logiciel hôte ainsi que le firmware qui sont chargés de la conduite des différentes opérations qui sont effectuées sur le microcontrôleur PIC cible.

Le premier chapitre donnait un aperçu général des microcontrôleurs PIC, de leurs architectures et des outils logiciels qui les accompagnent, de leurs méthodes de programmation ainsi que des outils tiers qui ont été utilisés pour développer le logiciel et le firmware.

Le deuxième chapitre décrit la mise en œuvre du processus de programmation du point de vue du firmware, il détaille les étapes nécessaires pour inclure la communication USB dans les projets PIC ainsi que le protocole établi entre le logiciel et le firmware.

Le troisième chapitre était consacré au détail du fonctionnement du logiciel, au développement utilisant le langage Java et le framework JavaFX.

Le dernier chapitre a été consacré au fonctionnement du bootloader lorsqu'il est utilisé pour charger une application dans le microcontrôleur. Actuellement, il ne supporte que les opérations d'écriture, mais il peut être encore amélioré pour supporter d'autres opérations telles que la lecture et l'effacement des zones de mémoire. Il est à noter cependant que certains problèmes sont survenus lors du chargement du bootloader et du firmware du programmateur dans la mémoire Flash d'un MCU, le comportement de ce dernier devient imprévisible et le logiciel ne parvient pas à se connecter de temps en temps.

Actuellement, seul un sous-ensemble de microcontrôleurs est supporté, l'application ainsi que le firmware peuvent être améliorés en ajoutant le support de nouveaux microcontrôleurs. La compatibilité du bootloader avec le firmware du programmateur n'est pas encore résolue non plus.

Espérons que ce travail a établi la base pour de futurs projets plus avancés qui nécessitent un haut niveau de personnalisation en travaillant avec le microcontrôleur PIC.

BIBLIOGRAPHIE

BIBLIOGRAPHIE

[1] I. Microchip Technology, "8-bit microcontroller summary." http://microchipdeveloper.com/8bit:summary. Accédé le 12-04-2019.

[2] M. Barr, *Programming Embedded Systems in C and C++*. Sebastopol, CA, USA : O'Reilly & Associates, Inc, 1999.

[3] H.-W. Huang, *PIC Microcontroller : An Introduction to Software and Hardware Interfacing*. 5 Maxwell Drive, PO Box 8007, Clifton Park, NY 12065-8007, USA : Delmar Learning, 2005.

[4] R. R. . S. D. Thomas Schmidt, "In-circuit serial programming™ (icsp™) guide," tech. rep., Microchip Technology, Inc., 2355 West Chandler Blvd.Chandler, AZ 85224-6199r, 3 2003.

[5] M. T. Inc, "Section 28. in-circuit serial programming™ (icsp™)," tech. rep., 2355 West Chandler Blvd.Chandler, AZ 85224-6199r, 1997.

[6] O. Lathrop, "In-circuit serial programming (icsp)." http://www.embedinc.com/picprg/icsp.htm. Accédé le 20-05-2019.

[7] G. Rowe, "hid4java." https://github.com/gary-rowe/hid4java. Accédé le 21-05-2019.

[8] J. Axelson, *Everything You Need to Develop Custom USB Peripherals*. Madison, WI 53704 : Lakeview Research LLC, 2005.

[9] "H2 database engine." https://www.h2database.com/html/main.html. Accédé le 22-05-2019.

[10] "Pic18f2xxx/4xxx family flash microcontroller programming specification," tech. rep., Microchip Technology, Inc., 2355 West Chandler Blvd.Chandler, AZ 85224-6199r, 7 2015.

[11] H. Schildt, *Java The Complete Reference*. McGraw-Hill Educationg, 9 ed., 2014.

[12] "Javafx overview (release 8)." https://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm#JFXST784. Accédé le 23-05-2019.

[13] I. Fedortsova, "Concurrency in javafx." https://docs.oracle.com/javafx/2/threads/jfxpub-threads.htm. Accédé le 23-05-2019.

Résumé

Un programmateur est un instrument qui permet de charger le contenu d'un fichier HEX dans la mémoire du microcontrôleur cible, il est également utilisé pour lire le contenu de la mémoire et l'effacer. Il utilise le protocole USB pour communiquer avec le logiciel qui fournit une interface utilisateur graphique à l'utilisateur.

Ce projet vise à fournir les concepts de base pour la programmation des microcontrôleurs PIC. Au cours du processus de développement, un micrologiciel a été créé ainsi que le logiciel utilisé pour s'interfacer avec celui-ci, les deux la logique nécessaire pour exécuter les différentes procédures du programmateur.

Mots clés : Programmateur PIC, microcontrôleurs PIC, Java, JavaFX, base de données H2.

Abstract

A programmer is an instrument that allows the content of a HEX file to be loaded to the memory of the target microcontroller, it is also used to read the content of the memory and to erase it. It uses the USB protocol to communicate with the software that provides a graphical user interface to the user.

This project aims to provide the basic concepts for programming PIC microcontrollers. During the development process, a firmware was created along with the software used to interface with it, both of which provide the necessary logic to execute the different procedures of the programmer.

Key words : PIC programmer, PIC microcontrollers, Java, JavaFX, H2 database.

ملخص

المبرمج هو أداة تسمح بتحميل محتوى ملف HEX على ذاكرة المتحكم الهدف ، كما تستخدم لقراءة محتوى الذاكرة ومحوها. يستخدم بروتوكول USB للتواصل مع البرنامج الذي يوفر واجهة رسومية للمستخدم.

يهدف هذا العمل إلى توفير المفاهيم الأساسية لبرمجة ميكروكنترولر PIC. أثناء عملية التطوير ، تم إنشاء برنامج ثابت مع البرنامج المستخدم للتفاعل معه ، وكلاهما يوفر المنطق الضروري لتنفيذ الإجراءات المختلفة للمبرمج.

الكلمات المفتاحية : مبرمج PIC ، ميكروكنترولر PIC ، جافا، JavaFX، قاعدة البيانات H2