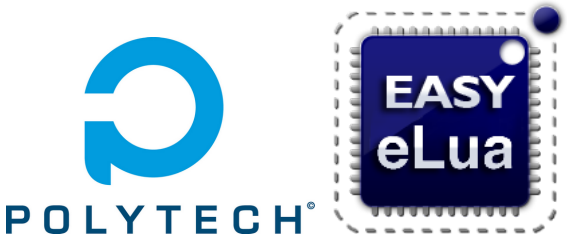


*Elizabeth PAZ*
*Salem HARRACHE*

# Easy eLua

## eLua et approche Arduino sur STM32F4-DISCOVERY

Projet Innovant, Avril 2012

RICM 4A

# Table des matières

# Table des figures

# Liste des tableaux

# Introduction

Après un quatrième semestre très chargé, l'heure est aux projets innovants. Aujourd'hui vient l'heure de l'expérience et du premier vrai test sur nos capacités à concevoir, à développer et à innover.

Parmi ces projets, certains, et le nôtre en particulier, s'adressent avant tout aux développeurs et aux entreprises désireuses de gagner du temps, et donc de l'argent. En effet, il est question ici de mettre en place une API, plus particulièrement, de porter l'API Arduino dans un autre environnement et un autre langage.

Arduino est l'entreprise italienne qui a innové dans le monde de l'embarqué par son matériel simple, peu cher, et open source mais également par sa librairie qui fait passer la programmation sur embarqué pour un jeu d'enfants. Cette approche, qui consiste d'une part à offrir une structure et API très simple et à proposer l'ensemble en open source a aussi bien séduit les novices que les experts. Nous avons été séduits par cette approche en novices que nous sommes.

D'autre part, un autre projet open source dans l'embarqué prend de l'ampleur depuis quelques années : eLua. Ce projet propose un environnement Lua pour beaucoup de cartes microélectroniques (MicromintEagle 100, Texas instruments LM3S6965 etc.) qui permet d'écrire des programmes avec le très puissant langage Lua.

Notre projet repose sur ces deux projets existant pour faciliter la programmation sur les cartes STM32F4-DISCOVERY de ST-Links. Il consiste en le développement et portage d'une partie de l'API Arduino sur l'environnement eLua. Le but souhaité est de permettre à n'importe quel développeur Arduino de s'y retrouver rapidement dans l'environnement eLua avec toutes les fonctions qu'il connait, déjà utilisables et exploitables.

Ce rapport s'articule autour de trois parties. En premier lieu, nous allons revenir sur l'approche Arduino et l'environnement eLua. Ensuite, nous détaillerons le déroulement du projet, la méthode de travail et le travail réalisé. Enfin on terminera par le bilan sur l'ensemble du projet.

# Première partie

# Introduction aux projets existants

Chapitre 1

# Approche Arduino

## 1.1 Qu'est-ce qu'Arduino ?


**Figure 1.1** – Logo Arduino

Le système Arduino [1] est une plateforme *open-source* d'électronique programmée basée sur une carte à microcontrôleur de la famille AVR [2], et sur un environnement de développement intégré qui permet d'écrire, compiler et transférer un programme vers la carte. Ce logiciel utilise la technique du Processing/Wiring [3].

Avec le système Arduino on peut réaliser divers tâches, par exemple le développement des objets interactifs indépendants : le prototypage rapide. Aussi, la domotique, qui grâce aux différents interrupteurs/capteurs permet de contrôler plusieurs sorties matérielles : contrôle des appareils domestiques, pilotage de robot, etc ...ou même charger des batteries par l'analyse et la production des signaux éléctriques. De plus, il peut communiquer avec des logiciels tournant sur l'ordinateur tel que Macromedia Flash, Processing, MaxMSP, etc ....

Le langage de programmation Arduino est une implémentation de Wiring, une plateforme de développement similaire, qui est basée sur l'environnement multimédia de programmation Processing.

Les cartes électroniques sont accesibles a tous, elles peuvent être achetées préassemblées ou être fabriquées manuellement, tout en ayant la totalité des informations nécessaire à l'assemblage.

1. Le projet Arduino a reçu un titre honorifique à l'Ars Electronica 2006, dans la catégorie Digital Communities
2. Cœur du processeur et famille de microcontrôleurs
3. Processing est un langage de programmation et un environnement de développement

## 1.2 Avantages de l'utilisation de Arduino

Le système Arduino a simplifié la façon de travailler avec les microcontrôleurs, en offrant plusieurs avantages pour les enseignants, les étudiants et les amateurs intéressés par d'autres systèmes. Ce système prend en charge des détails compliqués de la programmation des microcontrôleurs et les intègrent dans une présentation facile à utiliser. Voici, plusieurs avantages qui propose Arduino :

**Peu coûteux :** En comparaison a d'autres plateformes les cartes Arduino sont peu coûteuses, les moins chères sont les versions qui peuvent être assemblées à la main.

**Multi-plateforme :** Le logiciel de programmation des modules Arduino est une application Java, libre et qui peut être tourné dans plusieurs systèmes d'exploitation tel que Linux, Windows et Mac.

**Environnement de programmation clair et simple :** Le logiciel est facile à utiliser pour les débutants ( nous même l'ayant utiliser suite au cours d'initiation à l'Arduino) ; de plus, il reste flexible pour des utilisateurs avancés.

**Logiciel Open Source et extensible :** Le logiciel et le langage Arduino sont publiés sous licence Open Source qui est donc disponible à tous, ce qui permet donc la possibilité d'être complété et amélioré par des programmateurs plus experimentés. Le langage Arduino, qui utilise le langage C++, peut être étendu grâce à l'aide des librairies du C++.

**Matériel Open Source et extensible :** La version sur plaque d'essai de la carte Arduino peut être acheté à très bas coût et peut être réalisé par tout utilisateur, elle a pour but comprendre comment la carte fonctionne. De plus, tous les schèmas de modules Arduino sont publiés alors les utilisateur plus experimentés en circuits peuvent apporter des améliorations à leur cartes.

## 1.3 Utilisation d'Arduino

Dans cette partie on va décrire certains points importants pour l'utilisation du logiciel, de la carte et du langage.


**Figure 1.2** – Carte Arduino Uno

Un programme en langage Arduino doit obligatoirement être composé de ces deux fonctions :
– la fonction d'initialisation *setup*() qui est exécutée une seule fois au démarrage.
– la fonction "boucle sans fin" loop() qui est exécutée en boucle une fois que la fonction setup() a été exécutée une fois.
Puis, si besoin, autres fonctions peuvent être crées en suivant ce schèma :

```
type nom_fonction (arguments) {
   // ici le code de la fonction
}
```

**Table 1.1** – Création d'une nouvelle fonction en Arduino

```
/*
  Blink
  Turns on an LED on for one second, then off for one second, ...
      repeatedly.
*/
void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);   // set the LED on
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // set the LED off
  delay(1000);              // wait for a second
}
```

**Table 1.2** – Exemple Blink pour Arduino

L'utilisation du logiciel est facile et très clair, ce qui est très important a rétenir c'est de bien vérifier avant d'envoyer le programme à la carte, si on utilise le bon port et la bonne carte.

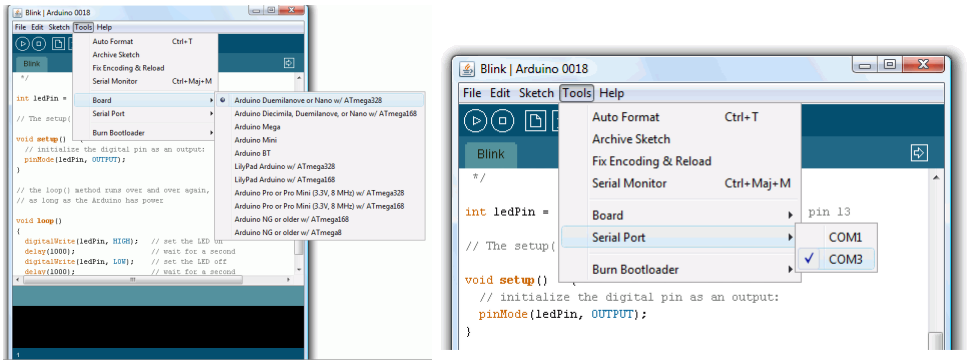


**Figure 1.3** – Environnement de travail : choix de la carte et du port

Chapitre 2

# elua sur STM32F4-DISCOVERY

## 2.1 Qu'est-ce que Lua ?

Lua est un langage de script libre, réflexif et impératif. Il a été conçu afin de pouvoir être embarqué au sein d'autres applications et les étendre Lua (qui signifie lune en portugais) a été développé par des membres du groupe de recherche TeCGraf, de l'université de Rio de Janeiro au Brésil. Il est écrit en langage C ANSI strict, et grâce a cela il est compilable sur une grande variété de systèmes ; le plus souvent est utilisé dans des systèmes embarqués, dont sa compacité est très appréciée, de plus, il possède la compatibilité du langage Cpour s'intégrer facilement dans la plupart des projets. Il profite de la compatibilité que possède le langage C avec un grand nombre de langages pour s'intégrer facilement dans la plupart des projets. Lua a déjà été utilisé pour le développement de jeux vidéo, entre eux, par exemple l'interface du jeu World of Warcraft de Blizzard Entertainment, SimCity4, entre autres.


**Figure 2.1** – Logo Lua

## 2.2 Qu'est-ce qu'eLua ?

elua adopte le langage de programmation Lua pour faire une complète implémentation dans le monde de l'embarqué, elua ajoute des caractéristiques spécifiques pour une

efficacité, portabilité et développement de logiciels embarqués. eLua propose la totalité des caractéristiques de la version de bureau de Lua, et c'est important de remarqué que elua utilise les mécanismes de base pour pouvoir l'étendre avec des fonctionnalités de développement de l'embarqué optimisés et spécifiques.

### 2.2.1 Architecture de eLua

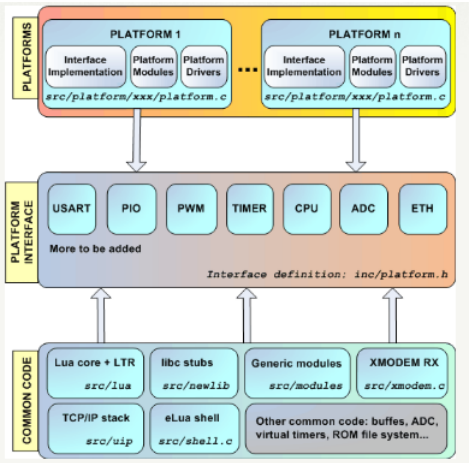


**Figure 2.2** – Structure logique de eLua

Dans la documentation de eLua on utilise la notion de "plateforme" pour designer un groupe de CPU qui partagent la même structure du noyau ; pourtant ils peuvent différer dans le nombre de péripheriques integrés, la mémoire interne, et toute autres attributs. Un port eLua implemente un ou plusieurs CPU d'une même plateforme. Dans la figure **??**, on observe que eLua essaie d'être aussi portable que possible dans différents plateformes, pour cela plusieurs règles ont été mises en place, entre elles :

- Le code qui est indépendant du type de plateforme doit être écrit en ANSI C dans toutes les parties ou cela est possible, afin qu'il soit portable parmi plusieurs architectures et compilateurs, comme Lua.
- Le code qui ne peut pas être générique (par exemple le péripheriques et le code spécifique au CPU) doit être fait aussi portable que possible en utilisant une interface en commun qui doit être implementé par toutes les plateformes dans lequelles eLua peut être executé. On va l'appeler "interface de plateforme" (à complèter avec la documentation).
- Toutes les plateformes et leur péripheriques ne sont pas crées de la même façon et possèdent donc des fonctionnalités très variés. Pour accèder a une fonctionnalité spécifique à une plateforme on peut utiliser un module (voir dans la documentation). Ces modules ont été crées afin de complèter l'écart entre l'interface de la plateforme et toutes les caractéristiques proposés par la plateforme. toutes les fonct

### 2.2.2 Code en commun

La liste suivante montre quelques éléments qui sont classifiés comme du code en commun :

- Le code Lua
- Tous les composants eLua (par exemple le ROM file system, le shell eLua, et autres)
- Tous les modules génériques, qui sont des modules exportés de Lua
- Le code générique des péripheriques

C'est important de remarquer que les parties génériques doivent être la plupart du code. Par exemple, lorsqu'on veut ajouter un nouveau fichier du système celui-ci doit être un code générique, sinon le code aura des dependances par rapport où il reside. On peut corriger cela en utilisant des fonctions définies dans l'interface de la plateforme, mais si cela n'est pas possible il faudra séparer les fonctions spécifiques dans une interface separé qui va devoir être implementé par toutes les plateformes qui veulent utiliser ce nouveau fichier. Ceci donne un maximum de portabilité au code.

L'utilisateur ne doit jamais oublier le but principal de eLua : la flexibilité. Il doit donc être dans la capacité de savoir quelles composants font partie de son binaire eLua, de même pour les modules, il doit savoir quels modules il a besoin. L'utilisateur est demandé de faire leur code pour différents scénarios possible, afin de promouvoir la portabilité.

## 2.3 Avantages de eLua

**Contrôle total de la plateforme :** il n'existe pas un systèmee d'exploitation entre les programmes et le microcontrôleur.

**Portabilité du code :** Comme Lua, le programme peut être executée dans un grand groupe varié de plateformes et architectures.

**Facilité de transformation :** Le code et le design des produits pour eLua peuvent être conçu indépendament du matériel, ainsi toute changement ou amélioration dans le future peut être fait facilement et gagner du temps.

**Développement d'objectifs :** Lua est complètement fonctionnel avec la possibilité d'avoir un shell dans le microcontrôleur, il n'y a pas besoin de rien installer côté ordinateur a part la connexion du port ou ethernet. Les programmes sont utilisable directement dans les plateformes.

**Flexibilité des produits :** L'utilisation de ce langage script de très haut niveau dans un projet rend celui-ci très adaptable, facilement reprogrammable et reconfigurable. Les systèmes sont très efficients pour une future évolution.

**Embarqué RAD :** Prototype et experimenté dans un modèle "Rapid Aplication Develop". Les idées peuvent être testé directement dans les plateformes avec les kits de développement ; l'utilisateur n'a pas besoin des simulateurs ou des futures modification du code pour le rendre adaptable.

**Les kits sont prêts a être utilisés :** Grand nombre des logiciels open source et plateformes sont utilisables.

**Long cycle de vie :** Add user configuration and scripting capabilities to your projects, making them adaptable to the always changing contexts of industrial processes, evolving engineering, automation standards, field optimizations etc...

**Apprendre l'embarqué :** L'experimentation est très simple et interactive. L'utilisateur est invité a utiliser ses compètences en programmation pour devenir un développeur des systèmes embarqués rapidement et de façon amusante.

**Open Source :** elua est libre, gratuit, et open source logiciel, comme Lua, elle possède une licence MIT qui permet l'utilisation de eLua dans des codes "closed" source. Il n'y a aucune permission a demander, ils demandent juste de faire circuler l'information au monde : on utilise eLua !

# Deuxième partie

# Travail réalisé

Avant de nous attarder sur le travail réalisé, nous allons tout d'abord aborder les méthodes de travail et notre organisation tout au long pour mener à bien ce projet. Puis nous détaillerons la structure du projet et les différentes composantes de celui-ci. Nous détaillerons enfin les fonctions portées, les fonctions qu'ils restent à écrire et les concepts nouveaux, propres à Easy-eLua. Nous terminerons par des exemples de programme développés avec Easy-eLua

Chapitre 3

# Organisation du travail

Sur ce projet, il y a eu deux phases distinctes. La première phase concernait la recherche d'information et l'évaluation de la faisabilité du projet. Cette phase était assez longue et peu productive en code. C'est également dans cette phase où l'on a commencé l'initiation à la programmation de l'embarqué et l'évaluation des différentes librairies disponibles pour linux (stlink, libopenstm32...). Nous nous concertions avec le tuteur pour jauger la voie à emprunter, car il faut bien le dire, beaucoup de sprint n'aboutissait pas forcément à un résultat espéré.

Avec le recul, on peut voir cette phase comme une longue formation aux différents outils et à la programmation de l'embarqué. Elle est d'autant plus importante, qu'elle a permis à cette d'avoir une deuxième phase très rapide.

Pour la deuxième phase, nous avons utilisé une méthode de travail qui s'inspire des méthodes agiles. Cette méthode s'appuie sur des valeurs fondamentales :

- Les interactions entre individus priment sur les processus et outils, ceci permet de développer davantage le travail en groupe, et favorise la communication en face à face.
- Le fonctionnement prime sur le reste : il est vital que l'application fonctionne. Le reste, et notamment la documentation technique, est une aide précieuse mais non un but en soi.
- Accepter le changement plutôt que de suivre un plan. En effet, il faut considérer le changement comme une opportunité car effectuer un changement au plus tôt permet de réduire le coût.

Ceci se traduit concrètement, par des objectifs courts, les "sprints" qu'on se fixe en fin de semaine pour la semaine à venir. Chaque fin de journée, on se regroupe durant 15-20 minutes pour une réunion hebdomadaire. À tour de rôle, chaque développeur prend la parole pour faire part de l'avancement de son projet et peut aborder les problèmes rencontrés. Cela nous permet de connaître l'état d'avancement de l'autre développeur et de proposer des suggestions et un nouveau point de vue. Ça permet aussi de voir si quelqu'un est bloqué, mais généralement si c'est le cas, l'aide survient plus tôt.

À la fin de chaque sprint, on effectue une réunion où l'on dresse le bilan de ce qui a été réalisé puis l'on fixe les objectifs du sprint suivant. Il n'y a pas réellement de scrum master, vu qu'on est deux et avec le même niveau de connaissances. On décide tous les deux des sprints de chacun.

Chapitre 4

# Arborescence du projet

L'arborescence du projet est la suivante :

**arduino_wrapper :** les sources de nos portages.

**docs :** contient l'ensemble des documents utiles des projets.

**elua :** contient les sources eLua.

**env :** le dossier contient les outils indispensables pour la compilation (sourcery-toolchain g++) et le flash de la carte.

**examples :** contient les exemples documentés prêts à l'emploi.

L'une des fonctionnalités intéressantes du projet, est l'installation automatique de l'environnement de développement. En effet, le script "install.sh" permet d'automatiser cette tâche rendant plus simple le premier contact avec Easy-eLua.

Nous avons également développé d'autres scripts pour automatiser d'autres tâches, comme le flash de la carte avec un programme Easy-eLua ou encore l'envoi de celui-ci sur une carte possédant déjà un environnement eLua.

Chapitre 5

# Fonctions portées

L'environnement eLua est très riche. La plupart des portages qu'on a réalisés sont assez courts et dépassent rarement les 10 lignes. Voici les principales fonctions portées.

## 5.1 Entrées/Sorties numériques

Ce sont les fonctions de base d'entrée sortie qui vont permettre d'interagir avec l'extérieur. Trois fonctions ont été réalisées :

**pinMode() :** Cette fonction spécifie le mode (INPUT ou OUTPUT) d'une broche. Cette fonction existe déjà dans eLua, mais elle porte un autre nom, le portage est donc très court :

```
1 function pinMode(pin, mode)
2             if mode == OUTPUT or mode == INPUT then
3                 pio.pin.setdir(mode, pin)
4             end
5 end
```

**Table 5.1 –** Fonction *pinMode*

**digitalWrite() :** Envoie la valeur HIGH ou LOW (respectivement 1 ou 0) sur une broche. De même que *pinMode*() le portage ne pose pas de problème puisque cette fonction (comme on peut s'en douter) est disponible dans l'environnement eLua.

```
1 Function digitalWrite(pin, value)
2             if value == HIGH or value == LOW then
3               pio.pin.setval( value, pin )
4             end
5 end
```

**Table 5.2 –** Fonction *digitalWrite*

**digitalRead() :** DigitalRead permet de lire la valeur de la broche. Cette valeur est soit de 0 soit de 1.

```
1 functiondigitalRead(pin)
2           return pio.pin.getval( pin )
3 end
```

**Table 5.3 –** Fonction *digitalRead*

## 5.2 Communication série

Pour l'entrée/sortie série, nous avons implémenté la classe Serial proposée par Arduino.

```
1 -- Serial communication object
2 SerialPort=Class:new()
3
4 function SerialPort:__new(uartid, baud,databits, parity,stopbits)
5     -- Initialize SerialPort
6     self.uartid = uartid
7     self.baud = baud or 115200
8     self.databits = databits or 8
9     self.parity= parity or uart.PAR_NONE
10    self.stopbits=stopbits or uart.STOP_1
11 end
```

**Table 5.4 –** Entrée/Sortie

La carte STM32F4-DISCOVERY dispose de 6 modules UART (Universal Asynchronous Receiver Transmitter) dont deux synchrones (USART). Dans la librairie Arduino, ces modules peuvent être utilisés par des objets nommés **Serialx** où x est l'identifiant de l'UART. Nous disposons donc dans Easy-eLua de 6 objets Serial accessibles directement depuis le programme, puisqu'ils sont initialisés directement dans le fichier *arduino_wrapper.lua*.

```
-- Define Serials object
Serial0 =SerialPort:new(0)
Serial1 =SerialPort:new(1)
Serial2 =SerialPort:new(2)
Serial3 =SerialPort:new(3)
Serial4 =SerialPort:new(4)
Serial5 =SerialPort:new(5)
```

**Table 5.5 –** Objets Serial

La classe dispose bien évidemment d'une méthode *begin*() qui initialise la connexion série. Dans ce cas les broches ne peuvent plus être utilisées avec *digitalWrite*() et *digitalRead*().

```
function SerialPort:begin(baud)
    -- Setup the serial port
    -- Returns: The actual baud rate set on the serial port.
    -- Depending on the hardware, this might have a different value ...
        than
    -- thebaud parameter
    self.baud = baud or self.baud
    return uart.setup(self.uartid,self.baud,self.databits,self.parity...
        ,
                                                    self.stopbits)
end
```

**Table 5.6 –** Classe *begin*

La fonction la plus intéressante (nous ne détaillerons pas les autres ici) c'est bien évidemment la fonction *print*() qui permet d'envoyer des données avec différents formatages possibles :

Nous sommes restés le plus fidèle possible à la fonction initiale de Arduino.

```
function SerialPort:print(value,format)
    -- Prints data to the serial port as human-readable ASCII text.
    -- This method can take many forms. Numbers are printed using
        an ASCII
    -- character for each digit. Floats are similarly printed
        as ASCII digits,
    -- defaulting to two decimal places. Bytes are sent as a
        single character.
    -- Characters and strings are sent as is. For example:
    -- SerialPort.print(78) gives "78"
    -- SerialPort.print(1.23456) gives "1.23"
    -- SerialPort.print('N') gives "N"
    -- SerialPort.print("Hello world.") gives "Hello world."

    -- An optional second parameter specifies the base (format)
        to use;
    -- permitted values are BIN (binary, or base 2), OCT (octal,
        or base 8),
    -- DEC (decimal, or base 10), HEX (hexadecimal, or base 16).
    -- For floating point numbers, this parameter specifies
    -- the number of decimal places  to use. For example:
    -- SerialPort.print(78, BIN) gives "1001110"
    -- SerialPort.print(78, OCT) gives "116"
    -- SerialPort.print(78, DEC) gives "78"
    -- SerialPort.print(78, HEX) gives "4E"
    -- SerialPort.println(1.23456, 0) gives "1"
    -- SerialPort.println(1.23456, 2) gives "1.23"
    -- SerialPort.println(1.23456, 4) gives "1.2346"

    if value == nil then
        return
    end

    if type(value) == "string" then
        uart.write(self.uartid, value)
    end

    if type(value) == "number" then
        if format ~= nil then
            if format == BIN then
                value = numberstring(value,2)
            elseif format == OCT then
                value = numberstring(value,8)
            elseif format == DEC then
                value = numberstring(value,10)
            elseif format == HEX then
                value=string.upper(numberstring(value,16))
            end
        elseif type(format) == "number" and format>=0 then
            value=string.format("%."..format.."f", value)
        else
            -- if value is float
            if value ~= floor(value)then
                value = string.format("%.2f", value)
            else
                value = string.format("%d", value)
            end
        end
        uart.write(self.uartid, value)
    end
end
```

**Table 5.7** – Fonction *print*

Chapitre 6

# Nouveaux concepts

La structure générale d'un programme Arduino repose sur les méthodes setup() et loop(). Le méthode loop() qui constitue le cœur du programme sera appelée indéfiniment, tandis que la méthode setup() elle, sert seulement à initialiser les variables et le contexte d'exécution au début du programme.

Partant de ce constat, on a décidé de donner une orientation objet à nos programmes lua. En effet ce choix peut être contestable, mais il nous est paru important pour mettre en place une API efficace.

Lua n'est pas pourvu de type "class", mais il permet de programmer le comportement de ses propres objets/types. Autrement dit, il est certes très léger, mais c'est un langage tout désigné pour la méta-programmation. Le mécanisme de classe, et d'héritage est défini par les lignes suivantes :

```
function Class:new(super)
    local class,metatable, properties = {},{},{}
    class.metatable=metatable
    class.properties=properties

    function metatable:__index(key)
        local prop = properties[key]
        if prop then
            return prop.get(self)
        elseif class[key] ~=nil then
            return class[key]
        elseif super then
            return super.metatable.__index(self, key)
        else
            return nil
        end
    end

    function metatable:__newindex(key, value)
        local prop = properties[key]
        if prop then
            return prop.set(self, value)
        elseif super then
            return super.metatable.__newindex(self, key, value)
        else
            rawset(self, key, value)
        end
    end

    functionclass:new(...)
        local obj=setmetatable({},self.metatable)
        if obj.__new then
            obj:__new(...)
        end
        return obj
    end
    return class
end
```

**Table 6.1 –** Classe *new*

Grâce à ce nouveau type, nous avons défini la classe série (vu plus haut) mais également la classe centrale : App

```
1 -- App
2 App = Class:new()
3
4 function App:__new(name)
5     -- Initialize App
6     self.name = name
7 end
8
9 function App:setup()
10     return
11 end
12
13 function App:loop()
14     return
15 end
16
17 function App:run()
18     self:println("Run : ".. self.name)
19     self:setup()
20     while self:condition()do
21         self:loop()
22     end
23     collectgarbage()
24 end
```

**Table 6.2 –** Classe centrale *App*

Un programme utilisant Easy-eLua n'aura qu'à redéfinir le comportement de setup() et de loop() (voir les exemples plus loin). Ce qu'il faut faire en plus par rapport au modèle Arduino, est l'instanciation d'un nouvel objet *App*, puis on l'appelle à la méthode run(). Le lancement est donc explicite. L'idée sous-jacente est la possibilité d'avoir plusieurs applications dans un même programme. Et le choix de laquelle lancer pourra se faire sur certaines conditions.

## 6.1 Fonctions relatives au temps

Deux fonctionnalités intéressantes pour un programme ont été mises au point. La première est la durée d'exécution du programme. Depuis combien de temps il tourne. Ça permet généralement de différer dans le temps certaines tâches. La seconde est la possibilité d'attendre passivement (delay()), ce qui est vital si on ne veut pas surcharger le CPU (attente active avec un while).

Ces fonctionnalités sont plus au moins disponibles dans eLua.

```
function delay(period)
    -- period in milliseconds
    tmr.delay(0,period*1000)
end

function delayMicroseconds(period)
    -- period in us
    tmr.delay(0, period)
end
```

**Table 6.3 –** Fonction *delay*

L'implémentation utilise le module timer de eLua. Lors du lancement de l'application, on mémorise le compteur d'un timer, pour que l'on puisse retrouver, par différence, la durée d'exécution :

```
function App:run()
    self:println("Run : ".. self.name)
    self.start_counter=tmr.start(self.timerid)
    [...]
end
function App:micros()
    -- Number of microseconds since the program started
    time_now=tmr.read(self.timerid)
    return tmr.gettimediff(self.timerid,self.start_counter,time_now)
end
```

**Table 6.4 –** Utilisation du module timer

## 6.2 Le Shell

Le Shell est un des avantages indéniables de l'environnement lua. On peut l'utiliser pour exécuter un interprète lua qui tourne directement sur la carte. On peut ainsi écrire des programmes dynamiquement, et par extension utiliser Easy-eLua par ce moyen.

Voici un exemple “HelloWorld” utilisant Easy-eLua :

```
eLua#lua
Press CTRL+Z to exitLua
Lua5.1.4   Copyright(C)1994-2011 Lua.org, PUC-Rio
>require("arduino_wraper")
>app=App:new("Hello World!")
>app:run()
Run: Hello World
```

**Table 6.5 –** Exemple “Helllo World”

Chapitre 7

# Exemples

## 7.1 Blink

### 7.1.1 Version Arduino

```
1 /*
2   Blink
3 Turns on an LED on for one second, then off for one second, ...
      repeatedly.
4
5   This example code is in the public domain.
6  */
7
8 // Pin 13 has an LED connected on most Arduino boards.
9 // give it a name:
10 int led =13;
11
12 // the setup routine runs once when you press reset:
13 void setup(){
14     // initialize the digital pin as an output.
15     pinMode(led, OUTPUT);
16 }
17
18 // the loop routine runs over and over again forever:
19 void loop(){
20     digitalWrite(led, HIGH);// turn the LED on (HIGH is the voltage ...
          level)
21     delay(1000);// wait for a second
22     digitalWrite(led, LOW);// turn the LED off by making the voltage ...
          LOW
23     delay(1000);// wait for a second
24 }
```

**Table 7.1** – Blink : version Arduino

### 7.1.2 Version Easy-eLua

```
1 --  Blink
2 --  Turns on an LED on for one second, then off for one second, ...
      repeatedly.
3 require("arduino_wraper")
4
5 functionApp:setup()
6     self.ledpin= ORANGE_LED -- Pin PD_13 has a LED connected
7     pinMode(self.ledpin, OUTPUT)-- Initialize the digital pin as an ...
          output.
8 end
9
10 functionApp:loop()
11     digitalWrite(self.ledpin, HIGH)-- set the LED on
12     delay(1000)-- wait for a second
13     digitalWrite(self.ledpin, LOW)-- set the LED off
14     delay(1000)-- wait for a second
15 end
16
17 app=App:new("Blink led")
18 app:run()
```

**Table 7.2 –** Blink : version Easy-eLua

## 7.2 DigitalRead

### 7.2.1 Version Arduino

```
1 /*
2   DigitalReadSerial
3  Reads a digital input on pin 2, prints the result to the serial ...
       monitor
4
5  This example code is in the public domain.
6  */
7
8 // digital pin 2 has a pushbutton attached to it. Give it a name:
9 intpushButton=2;
10
11 // the setup routine runs once when you press reset:
12 void setup(){
13     // initialize serial communication at 9600 bits per second:
14     Serial.begin(9600);
15     // make the pushbutton's pin an input:
16     pinMode(pushButton, INPUT);
17 }
18
19 // the loop routine runs over and over again forever:
20 void loop(){
21     // read the input pin:
22     intbuttonState=digitalRead(pushButton);
23     // print out the state of the button:
24     Serial.println(buttonState);
25 }
```

**Table 7.3 –** DigitalRead : version Arduino

### 7.2.2 Version Easy-eLua

```
1 -- DigitalReadSerial
2 -- Reads a digital input on pin PA0 (B1 user), prints the result to ...
     the serial
3 -- monitor
4 require("arduino_wraper")
5
6 -- the setup routine runs once when you press reset
7 functionApp:setup()
8     Serial2:begin(9600)-- initialize serial com at 9600 bits per ...
          second
9     pinMode(USER_BTN, INPUT)-- Initialize the digital pin as an ...
          output.
10 end
11
12 functionApp:loop()
13     buttonState=digitalRead(USER_BTN)-- read the input pin
14     Serial2:print(buttonState)-- print out the state of the button
15 end
16
17 app=App:new("DigitalRead Serial")
18 app:run()
```

**Table 7.4 –** DigitalRead : version Easy-eLua

Chapitre 8

# Conclusion - Bilan de l'expérience

Ce projet aura été pour nous un premier contact concret avec le monde de l'embarqué. Malgré un démarrage lent, dû à une grande phase de formation, le projet est au final opérationnel et c'est ce qui compte. L'expérience était vraiment enrichissante et a été propice aux nouvelles réflexions sur le rôle de l'ingénieur dans la recherche, l'analyse et la conception d'API utilisable par les programmeurs. Pour une fois, on ne se situe non pas dans le rôle de l'ingénieur-technicien, mais dans celui de l'ingénieur simplement.

Lors du développement, notre principale règle était de coder proprement et efficacement avec le minimum de lignes de code. Plus un code est petit et lisible, plus il est facile à maintenir et à garder cohérent (ne pas avoir de débordement, de comportements inattendus, etc...).

Ce projet ne nous a pas seulement appris de nouvelles notions de programmation et conceptions, mais également à rechercher efficacement l'information, à prendre contact avec des personnes diverses qui travaillent à l'autre bout du monde, à jauger de la viabilité d'un projet et savoir si on l'utilise ou pas dans le nôtre.

Concernant la démarche de travail, on a pu mettre en place une organisation plus flexible par rapport à celle des TPs classiques. En effet, si l'autonomie dans un TP est un plus indéniable qui contribue à la réussite, ici elle se trouve au cœur de la gestion et du développement du projet. Les différentes phases du projet n'étaient pas figées. L'essentiel était d'atteindre l'objectif et par conséquent d'être capable de prendre en charge les différentes tâches tout en étant le plus productif possible.