
**Prise en main des outils et
de la carte de développement LPC1769.
Exercice sur la manipulation de bits
Semestre hiver**

Introduction

Ce laboratoire va vous permettre de prendre en main les outils de travail utilisés dans le cadre des laboratoires. Les outils se composent de :

- L'environnement de développement (IDE)
 - L'éditeur
 - Le compilateur
 - Le débogueur
- La carte à base de micro-contrôleur LPCxpresso LPC1769

La carte LPC1769 dispose d'un micro-contrôleur avec un processeur ARM Cortex M3. Il n'est pas possible de compiler nativement sur ce processeur et de ce fait tous les développements se feront en compilation croisée. A savoir que nous devons compiler sur un PC un exécutable ARM pouvant être exécuté par la cible ARM.

L'objectif de ce laboratoire est dans un premier temps de réaliser quelques exercices simples sur la carte LPC1769 en utilisant l'éditeur, le compilateur et le débogueur afin de prendre en main les outils de travail utiles à la programmation matérielle. Dans un second temps, vous devrez développer une application en langage C. Au travers de ce langage couramment utilisé dans le domaine des systèmes embarqués, vous serez amené à effectuer des opérations bit à bit (opérations qui se retrouvent très souvent utilisées en programmation matérielle).

1 Cyberlearn

Avant de commencer inscrivez-vous sur le site du cours

<http://cyberlearn.hes-so.ch/course/view.php?id=3545>

clé d'inscription : hepia

2 Workspace

Le branchement de la carte LPC1769 se fait via un câble mini USB. Lancez ensuite l'environnement de développement LPCxpresso.

CHOISISSEZ OBLIGATOIREMENT UN RÉPERTOIRE DANS LE LECTEUR **Z** : POUR VOTRE WORKSPACE. N'UTILISEZ PAS DE CARACTÈRES ACCENTUÉS NI D'ESPACE DANS LE CHEMIN D'ACCÈS.

3 Exercices

Vous importerez dans votre *workspace* les trois projets contenus dans l'archive. Le fichier **labo1.c** du projet Labo1 contient la fonction **main** qui à son tour contient les exercices.

Build : Pour compiler votre projet, utilisez le bouton **Build 'labo1' [Debug]** situé dans la fenêtre Quickstart Panel (par défaut en bas à gauche) de l'écran.

Debug : Pour télécharger le programme sur la cible, utilisez le bouton **Debug 'labo1' [Debug]** aussi dans le Quickstart Panel.

Plus d'information sur comment déboguer le programme sur <http://goo.gl/B7M9BN>

3.1 Affichage des variables globales

Notre programme contient des variables globales (tab1, tab2) et des variables locales (i,j, texte). Une fois le programme téléchargé sur la cible et lancé, la vue en mode debug contient la fenêtre « Variable » permettant de visualiser l'état des variables.

Question 1 : Voit-on les variables globales ?

Question 2 : Faites en sorte que l'on puisse les voir et observer les différents octets de ces tableaux (sans modifier le code évidemment, mais en utilisant la vue Expression).

Pour les différentes variables, observez leur contenu en binaire, hexadécimal et décimal.

3.2 Points d'arrêts

Notre programme contient, par défaut, un point d'arrêt (breakpoint) situé à la première ligne de la fonction `main()`, autrement dit, au point d'entrée de notre programme. Celui-ci nous permet de s'arrêter au début de notre programme avant l'exécution du programme. L'état initial peut ainsi être vérifié avec le débogueur.

Notre programme contenant plusieurs exercices, nous aimerions nous arrêter à la fin de la première boucle afin de voir le résultat du passage dans celle-ci et, le cas échéant, faire des modifications ou des corrections de bugs.

Question 3 : Ajoutez un point d'arrêt entre les deux boucles.

Question 4 : Observez les valeurs des variables et vérifiez si elles correspondent à ce que vous attendiez

3.3 Exercice 1 : HelloWorld

Tout le monde connaît le fameux « HelloWorld » initiée par le livre *The C Programming Language* de Brian Kernighan et Dennis Ritchie. (Le premier exemple de ce livre affiche `hello, world` sur la sortie standard (généralement la console) en utilisant un `printf()`).

Dans notre cas, aucune sortie standard n'est actuellement définie, car notre cible (Carte LPC1769) n'a pas d'écran pour afficher un message, et de plus, il n'y a pas d'OS fournissant cette sortie standard.

N'ayant pas beaucoup de moyens de communication pour l'instant, la première boucle de notre programme se contente de recopier 10 fois **HelloWorld** dans le tableau `Tab1`.

Question 5 : Le texte de 10 lettres **HelloWorld** doit être copié 10 fois successivement dans le tableau (sans espaces entre eux). En observant le contenu de Tab1 vous constaterez un décalage entre les **HelloWorld** successifs.

Pour comprendre la cause de ce décalage, vous pouvez re-exécuter le programme et le faire s'exécuter en pas à pas. Revenez à l'éditeur, étudiez le code et corrigez le problème dans la première boucle. Vérifiez le fonctionnement correct en exécutant à nouveau le programme. La correction doit pouvoir fonctionner quelque soit la longueur de la chaîne de caractère.

N'oubliez pas de re-compiler et de re-télécharger le code dans la cible...

Rappel : Une fin de chaîne de caractère se termine par la valeur hexa 0x00.

3.4 Exercice 2 : Vue mémoire

La seconde boucle de notre programme se contente de recopier 10 fois **HelloWorld** dans le tableau Tab2 , mais nous aimerions maintenant changer le texte **HelloWorld** de Tab2 en **Coucou!**

Question 6 : Lorsque le programme s'est arrêté au niveau du point d'arrêt situé entre les deux boucles, utilisez la vue *Memory* pour voir où se situe la variable `texte` en mémoire. Indice : Ajoutez `texte` avec le plus vert. Le contenu de la mémoire est alors visible sous forme hexadécimal.

La vue hexadécimal n'étant pas très adaptée à l'édition de caractères, trouvez comment afficher la correspondance ASCII de ces caractères dans la vue *Memory*.

Question 7 : Grâce à cette vue ASCII, modifiez le contenu de la variable `texte` par **Coucou!** (avec le point d'exclamation à la fin). Donc directement depuis la mémoire, et non pas depuis le code!

Info : Vous constaterez que ceci n'est pas possible. En effet la variable `texte` pointe sur une chaîne de caractère qui est définie en mémoire FLASH (mémoire non volatile ne pouvant être modifiée que par le programmeur de l'environnement LPCxpresso). Nous allons redéfinir la variable en mémoire RAM (volatile) afin de pouvoir par la suite la modifier.

Pour ceci remplacez la ligne `char *texte = "HelloWorld";`
par la ligne `char texte[11] = "HelloWorld\0";`

Faites tourner la boucle 2 et vérifiez :

Question 8 : Est ce que **Coucou!** a bien été copié plusieurs fois dans le tableau Tab2 ?

3.5 Vérification

Le cahier des charges de notre programme stipule les choses suivantes :

1. La première boucle doit remplir Tab1 avec **HelloWorld**
2. La seconde boucle doit remplir Tab2 avec **Coucou!**

Question 9 : Est-ce que les deux tableaux sont correctement remplis ? Vérifiez attentivement le début du tableau Tab1.

Revenez à l'éditeur, étudiez le code et corrigez le programme de la seconde boucle afin que le contenu de la variable `texte` soit copié successivement un maximum de fois sans être tronqué ni corrompre le contenu de Tab1.

3.6 Boucle while(1)

Notre programme se termine par une boucle infinie `while (1)`

Question 10 : Pourquoi ? Testez en enlevant la boucle `while (1)` Quelle est votre conclusion ? Mettez en commentaire la nouvelle boucle `while (1)` située dans le fichier qui vient de s'ouvrir. Tester à nouveau. Qu'elle est la signification de la fonction dans laquelle vous arrivez ?


4 Ce que je dois avoir retenu à la fin de cette introduction

À la fin de cet exercice vous devrez avoir compris le fonctionnement de l'environnement de développement LPCXpresso ainsi que l'exécution sur la carte LPC1769 d'un programme ainsi que son débogage.

De façon plus détaillée :

- L'utilisation de l'environnement de développement
- La compilation d'un programme
- Le téléchargement d'un programme dans la carte LPC1769
- L'utilisation des points d'arrêts (breakpoints)
- L'exécution en mode pas à pas
- L'observation des variables locales et globales
- L'observation du contenu de la mémoire
- La modification d'une information en mode débogage
- Le fait que dans ce laboratoire et tous les prochains, vous exécutez un programme depuis le point de départ (il n'y a pas de système d'exploitation pour lancer votre programme, ni reprendre la main à la fin)

5 Pour aller plus loin

- Vous pouvez essayer de comprendre le fonctionnement de la vue *watch* (Expressions)
- Comprendre la différence entre *step into*, *step over* et *step return*
- La fonction *Run to line*
- Le mode *Instruction stepping mode* (dans la fenêtre debug, )
- Les points d'arrêt conditionnels (par exemple s'arrêter uniquement lorsque `i=42`)
- Accéder à la vue *Registers*

6 Manipulation des bits d'une variable

Pour cet exercice, vous allez devoir utiliser le tableau `image` qui est un tableau de 128 mots de 32 bits. Pour information l'écran LCD à votre disposition est un écran monochrome de 128x32 pixels. Chaque pixel pouvant soit être allumé, soit éteint.

La fonction `Display_Picture32(image)` permet d'afficher le contenu du tableau `image` sur l'écran.

Vous constaterez que le tableau `image` est à une dimension et est composé de 128 mots de 32 bits. Il vous aurait peut-être paru plus aisé de disposer d'un tableau de `BOOLEEN` à deux dimensions de 128x32. Cependant la représentation du Booléen n'est pas un type optimisé pour les processeurs et chaque `BOOL` sera au mieux stocké dans un octet. 7 bits de perdus pour 1 bit utile ! Compte tenu des ressources mémoires limitées du micro-contrôleur, cette méthode de représentation n'est pas souhaitable et le choix préconisé ne gaspille pas de mémoire mais vous oblige à manipuler des bits.

Votre tableau `image` contient un Smiley. Pour cet exercice voici les opérations à effectuer :

- Écrire un test (`if`) permettant de vérifier l'état d'une case. À savoir isoler un bit dans le mot de 32 bits correspondant à la colonne que vous voulez tester et vérifier si ce bit est à 1 ou 0
- Modifier l'état d'une case à noir en mettant un bit à 1. Ceci sans modifier les autres bits de la colonne. Il faut pour cela lire le contenu de la colonne où vous devez effectuer la modification et à l'aide d'un `OU` bit à bit ajouter un 1 à la bonne position (il faudra utiliser un décalage pour cette dernière opération).
- Modifier l'état d'une case à blanc en mettant un bit à 0. Ceci sans modifier les autres

bits de la colonne. Il faut pour cela lire le contenu de la colonne où vous devez effectuer la modification et à l'aide d'un `ET` bit à bit forcer un 0 à la bonne position (Pour cette opération, il faut mettre un 1 à la bonne position à l'aide d'un décalage, puis inverser bit à bit le tout).

- Tester ces trois fonctionnalités en faisant clignoter un œil du Smiley.

7 Conception d'une application : Fourmi de Langton

Vous allez devoir développer une application utilisant l'écran LCD de la carte d'extension My-Lab. Il s'agit dans le cadre de cet exercice, d'essentiellement effectuer des manipulations binaires. L'utilisation des outils de debuggage s'avérera essentielle pour développer rapidement votre application.

Le jeu de la fourmi de Langton fait parti de la catégorie des automates cellulaires. Ce programme informatique décrit une fourmi se déplaçant sur les cases d'une grille. Les règles qui régissent le mouvement de la fourmi sont d'une grande simplicité, cependant son comportement global s'avère fort complexe.

7.1 Règles du jeu

A chaque tour, la fourmi se déplace selon les règles suivantes :

- Si la fourmi est sur une case blanche, elle effectue une rotation vers la gauche ; si elle est sur une case noire, elle effectue une rotation vers la droite ;
- La fourmi inverse la couleur de la case sur laquelle elle se trouve (blanc devient noir et réciproquement) ;
- La fourmi avance d'une case dans la direction de son orientation.

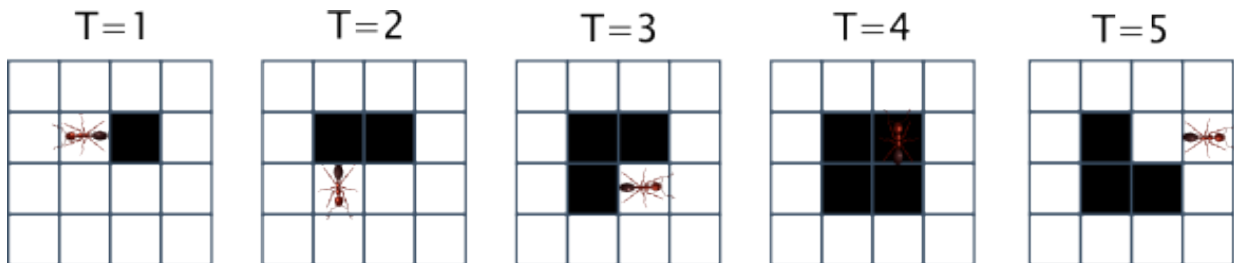


FIGURE 1. Fourmi de Langton

Implémentez l'algorithme de Langton en modifiant, à chaque itération, les bits nécessaires dans le tableau `image` à l'aide des opérations mises en place dans l'exercice précédent.

Au démarrage du jeu, vous positionnerez la fourmi à un endroit quelconque de l'écran (de préférence vers le centre).

Vous devez également gérer l'écran de façon toroïdale, à savoir que lorsque la fourmi arrive à l'extrémité droite elle continuera sur la gauche de l'écran et qu'elle quittera le bas de l'écran en revenant par le haut (et réciproquement dans les autres directions).

Afin de permettre un affichage fluide et non trop rapide, pensez à insérer un délai (au travers d'une boucle d'attente faite à l'aide d'un `for`) entre chaque itération.

Vous déclarez trois variables, qui pourront être `x`, `y`, `dir`. Testez la valeur de la case où se trouve la fourmi et modifiez l'état de la case. Vous avez par la suite quatre cas à gérer, correspondant aux quatre directions possibles. Selon la direction actuelle choisissez la nouvelle position de la fourmi. Il faudra peut être jouer avec des modulus ou des tests afin de gérer les bords de l'écran.

7.2 Pour aller plus loin

Vous pouvez ajouter d'autres fourmis à votre automate cellulaire qui évolueront en même temps sur la grille.

8 Conception d'une application : Jeu de la Vie (option)

Développez le fameux automate cellulaire du jeu de la vie de J.H. Conway (Game of Life). Vous pouvez envisager d'initialiser votre grille avec plusieurs milliers d'itérations de la fourmi de Langton.

9 Travail à présenter

Pour ce laboratoire, il vous est demandé de présenter, à la fin de la séance, le code de l'exercice ainsi qu'effectuer une démonstration.

Conseil : Il vous est vivement suggéré, dans le code, d'ajouter des commentaires ainsi que vos réponses aux questions.

Bon travail.

Annexes

A Types prédéfinis

Ces types sont donnés dans le fichier `stdint.h` fournit par l'environnement LPCxpresso pour l'utilisation avec le LPC1769.

Attention : La taille de ces types peut être différentes avec d'autres compilateurs ou processeurs. Il est important de toujours se documenter sur l'environnement utilisé si vous utilisez directement `short`, `int`, `long`.

Nous vous recommandons d'utiliser les types définis dans `stdint.h`

Occupation en mémoire (octet)	type de donnée C	Type définit dans <code>stdlib.h</code>	Domaine de définition
1	unsigned char	<code>uint8_t</code>	0 à 255
1	signed char	<code>int8_t</code>	-128 à 127
2	unsigned short	<code>uint16_t</code>	0 à 65'535
2	signed short	<code>int16_t</code>	-32'768 à 32'767
4	unsigned int	<code>uint32_t</code>	0 à $2^{32} - 1$
4	signed int	<code>int32_t</code>	-2^{31} à $2^{31} - 1$
4	unsigned long	<code>uint32_t</code>	0 à $2^{32} - 1$
4	signed long	<code>int32_t</code>	-2^{31} à $2^{31} - 1$
8	unsigned long long	<code>uint64_t</code>	0 à $2^{64} - 1$
8	signed long long	<code>int64_t</code>	-2^{63} à $2^{63} - 1$

Pour en savoir plus sur les types :

https://fr.wikibooks.org/wiki/Programmation_C/Types_de_base