Rapport de PSTL

Simulateur & IDE génériques pour OMicrob

Author :
Qiwei XIAN
Ruiwen WANG

Professeur :
Prof.Emmanuel Chailloux

4 juin 2020

Table des matières

1 Introduction

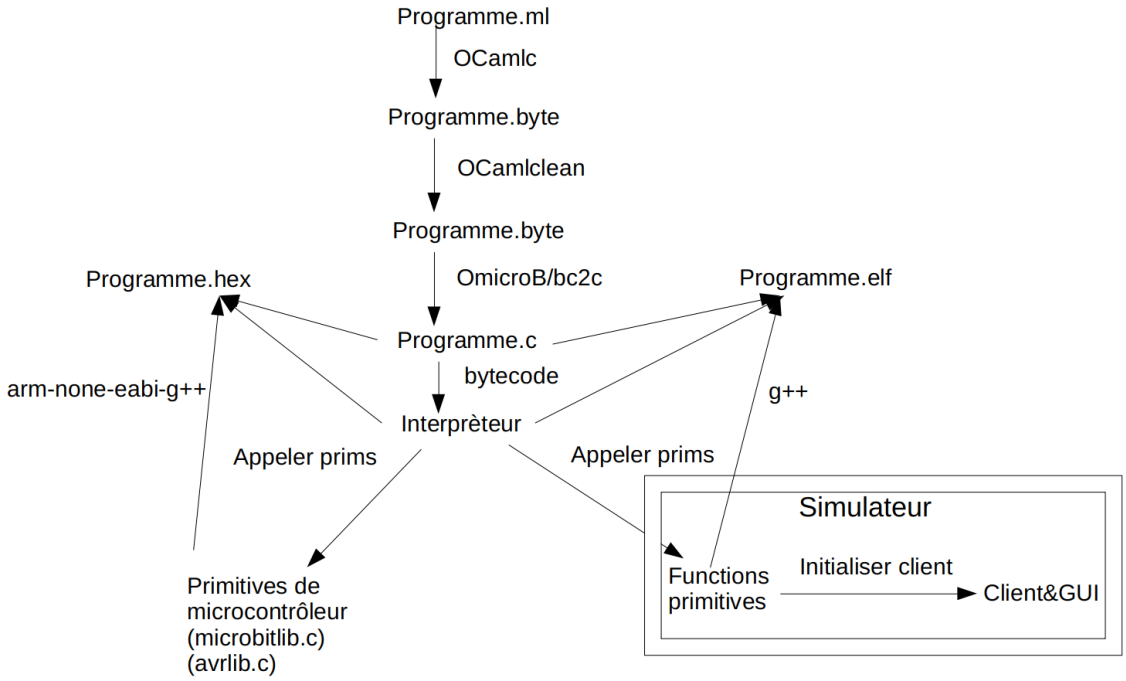
lien vers les codes sources : https://github.com/XIANQw/OMicroB/tree/microbit

La programmation des architectures à base de microcontrôleurs est difficile tant par les ressources limitées accessibles que par les modèles de programmation proposés. L'intégration électronique poussée d'un micro-contrôleur permet de diminuer la taille, la consommation électrique et le coût de ces circuits. La taille des programmes et la quantité de mémoire vive sont "faibles", le tas peut être de quelques kilo-octets seulement. Ils doivent communiquer directement avec les dispositifs d'entrées/sorties (capteurs, effecteurs, ...) via les pattes du circuit principal, et ne possèdent pas les périphériques classiques (souris, clavier, écran). La mise au point d'un programme devient plus difficile de par ce manque d'interaction classique.

On s'intéresse ici à une nouvelle version portable de la machinerie OCaml, appelée OMicroB, qui engendre un programme C contenant la version byte-code du programme OCaml ainsi que l'interprète de ce byte-code OCaml. Omicrob vient avec un environnement de développement incluant un simulateur permettant de décrire un montage et d'exécuter le programme sur l'ordinateur hôte avant de le transférer sur le microcontrôleur. Bien que portable, la version initiale de l'environnement de développement dont le simulateur a été principalement testée pour l'architecture Arduino. Le portage d'OMicrob vers d'autres architectures (Micro :bit Arm Corex-M0, PIC32) nécessite maintenant d'adapter son environnement de développement, principalement le simulateur. L'idée est d'ajouter une couche d'abstraction aux circuits utilisés pour pouvoir facilement passer d'un microcontrôleur à un autre. Par ailleurs une extension synchrone à flots de données, appelée OCaLustre, est particulièrement appropriée pour décrire les interactions externes et la concurrence interne à l'application. Le couple OCaLustre+OMicroB permet une programmation mixte (synchrone et multi-paradigme classique) avec une consommation parcimonieuse des ressources. L'intérêt de ces couches d'abstractions est de faciliter le développement d'applications fiables sur micro-contrôleurs

Ce projet cherche à améliorer cette mise au point de programmes en utilisant d'une part un langage de haut niveau, ici OCaml et son extension synchrone OCaLustre, et d'autre part en fournissant un simulateur et un IDE simple pour la mise au point de tels programmes.

2 OMicroB



Cet image montre chaque étape de compilation d'un programme OCaml par OMicroB.

1. **ocamlc** compile le fichier **.ml** avec la bibliothèque et génére un fichier **.byte**.
2. **ocamlclean** traite le fichier généré **.byte**.
3. **bc2c** est le compilateur de **OMicroB**, il permet de transférer le code binaire à un programme **.c**.
4. **g++** compile le **programme.c** et la bibliothèque de simulateur **sf-regs** puis génère le fichier exécutable **.elf**.
5. **arm-none-eabi-g++** compile le **programme.c** avec la bibliothèque de microcontrôleur (avrlib, microbitlib, etc) puis génére le fichier exécutable **.hex**.

OMicroB produit deux fichiers exécutables après de compiler un programme **OCaml**.
- **.elf** est exécutable en mode simulation, il permet de démarrer le simulateur et montrer le changement des états de pin et les effets de programme sur une interface graphique.
- **.hex** est exécutable sur un microcontrôleur.

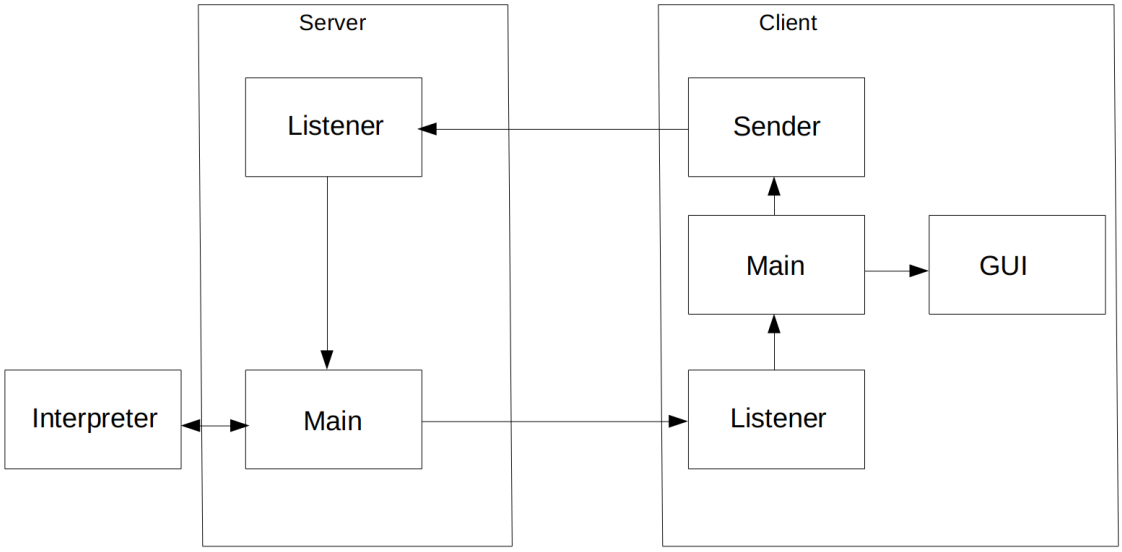
3 Simulateur

Pour afficher les effets de programme exécuté sur un microcontrôleur, on propose un simulateur qui sert à afficher simultanément l'état de chaque pin et chaque composant pendant l'exécution du programme. Donc on peut considérer l'interprèteur de **OMicroB** comme un producteur qui produit les demandes de visualisation. La simulateur est un consommateur, il satisfait chaque demande en visualisant les effets sur une interface graphique. Donc on peut implémenter un architecture **producteur-consommateur** pour réaliser le simulateur, un producteur transfère les demandes de l'interprèteur au consommateur, le consommateur visualise sur l'interface graphique finalement.

Considérons la facilité du développement, nous séparons la simulateur en deux parties indépendantes, **serveur** et **client**, le serveur est producteur et client est consommateur. Ses avantages est que :

1. Séparer le logiciel en deux parties indépendantes, nous pouvons développer paralèllement, c'est plus facile de travailler en groupe.
2. On peut mieux maintenir la simulateur. Lorsqu'il y a des erreurs générées pendant l'exécution, il plus pratique de localiser où produit l'erreur.

Par conséquence, on a un simulateur dont la structure est décrite par cette image, il exécute deux processus en concurrence. Chaque processus exécute plusieurs threads et travaillent asynchroniquement.



(a) la structure de simulateur

3.1 Serveur

La partie serveur est codée en C dans "/src/byterun/", il contient des fichiers :

1. **/src/byterun/vm**. Ce dossier contient les fichiers de l'interprèteur d'OMicroB, il à été réalisé en OMicroB existant. par exemple le format des type basiques de données, le garbage collector de machine virtuelle ainsi que les définition des instructions de code binaire. Il exécute le programme OCaml et envoie les instructions des codes binaires au serveur.
2. **/src/byterun/simul/sf-regs.c/.h**. C'est la bibliothèque des primitives de simulateur, la simulateur propose beaucoup d'API aux programmeurs d'OCaml. Ils peuvent appeler ces API dans un programme OCaml qui exécute sur un microcontrôleur, par exemple **set_pixel 0 0 true** sert à allumer un LED de microbit. Lorsque ce API est utilisé, la fonction définite dans sf-regs.c est appelé. Elle envoie une instruction au client, puis le client allume LED sur l'interface graphique.
3. **src/byterun/shared.c/.h**. Ce fichier définit des structures de données partagées entre le processus client et serveur, elles sont stockées dans une mémoire partagée au fur et mesure de l'exécution de programme.

Processus Serveur exécute deux threads, **listener** et **Main**.

1. **Main** traite simplement l'instruction qui vient de l'interprèteur et l'envoyer au thread **listener** de client.
2. **listener** reçoit l'instruction venant de **Sender** et informe **Main**.

3.2 Client

La partie client est programmée en C dans **src/byterun/client**, se compose de deux fichiers.

1. **client.c** définit les fonctions du travail par exemple traiter le message qui vient du côté serveur.
2. **gui.c** implémente l'interface graphique par **gtk3.0**. il nous montre un UI de simulateur permet d'afficher le résultat de programme et les états des Pins pendant la simulation, ainsi que l'interaction par les boutons.

Processus Client lance simultanément quatre threads, **Main**, **GUI**, **Sender** et **listener**.

1. **Main** est le premier thread principal, il traite les arguments venant de processus serveur et connecte aux mémoires partagés, par exemple la mémoire partagée pour la communication et pour le montage. Il est responsable d'exécuter les autres threads.
2. **GUI** crée les composants de l'interface graphique en fonction de l'information du montage. Il actualise l'interface dans un boucle infini afin d'afficher le changement des états de chaque composant simultanément.
3. **Sender** sert à traiter input de l'utilisateur et envoyer l'instruction au serveur, par exemple appuyer les boutons.
4. **listener** reçoit l'instruction venant de serveur, en plus manipuler les composants de l'interface graphique en fonction de l'instruction.

3.3 Montage

Pour démarrer le processus client et implémenter l'interface graphique, le client a besoin de savoir quels composants du microcontrôleur il doit afficher sur l'interface graphique. Donc nous devons

proposer un fichier **circuit.txt** qui décrit les composants du microcontrôleur et les associations entre chaque composant et les Pins correspondants. Avant de démarrer le processus client, le processus analyse est exécuté pour évaluer ce fichier. Une structure qui contient ces informations est stockée dans une mémoire partagée après l'évaluation. Le client peut recevoir les informations des composants du microcontrôleur depuis cette mémoire partagée.
Cette fonctionnalité est réalisée dans le dossier **src/byterun/montage**, la structure est définie dans **src/byterun/simul/share.h**.

4 Mécanisme de communication

4.1 Protocole de communication

Pour décrire bien l'instruction, on a implémenté le protocole en plusieurs versions diverses. (S -> C) représente la direction de l'envoie est du serveur au client, (C -> S) est la direction inversée.

4.1.1 Chaine de caractère

Dans la version de protocole initiale, Nous avons choisi la chaine de caractère comme le type du protocole. La raison est que c'est plus pratique de les afficher sur le terminal, on peut facilement vérifier si le protocole est correct pour chaque instruction.
La structure du protocole est (**numéro de primitive, argument1, argument2, ...**). Pour cette version, nous avons réalisé les protocoles.

1. **PRINT_IMAGE** : (0, 25 carcatères de 0 ou 1)
   (S -> C) Il sert à afficher une image en fonction de 25 caractères, chaque caractère représente l'état de LED. Envoyé par **microbit_print_image(char* ima)**.

- **SET_PIXEL(x,y,val)** : (1, x, y, val)
  (S -> C) Modifier l'état du pixel des coordonnées x et y à val, envoyé par **microbit_write_pixel(int x, int y, int val)**.
- **CLEAR_SCREEN** : (2)
  (S -> C) Mettre les états de tous les pixels à 0, envoyé par **microbit_clean_screen()**.
- **SET_PIN(p,n)** : (3, numéro de pin, niveau)
  (S -> C) Modifier le pin p au niveau n. (n = 0 ou 1), envoyé par **microbit_digital_write**.
- **WRITE_PIN(p,v)** : (5, p, v)
  (S -> C) Modifier le pin p à la valeur v (0 <= v <= 1024), envoyé par **microbit_analog_write**.

- **INVERSE_PIN(p,n)** : (0, p)
  (C -> S) Inverser l'état de pin qui associéte avec un bouton, lorsque l'utilisateur appuie sur un bouton, ce protocole est envoyé au serveur et **serveur listener** inverse l'état de pin correspondant.

4.1.2 Entier de 32 bits

Après tester tous les primitives, on est sûr que le protocole est correctement décrite chaque instruction. On commence d'améliorer ce protocole, parceque la formalisation et décodage du protocole consomment beaucoup de temps, ainsi que la chaine de caractère consomment aussi l'espace de

mémoire.
Commme un integer du langage C se compose de 32 bits, ainsi que 32 bits pour le protocole est suffisant sauf pour PRINT_IMAGE. Il ne suffit pas à afficher une image dont le nombre de pixel est supérieur à 31. Donc on supprime ce protocole dans cette version, et on réalise la primitive **microbit_print_image(char* ima)** en utilisant le protocole **SET_PIXEL(x,y,val)**, il ne suffit d'envoyer le nombre de pixel fois le protocole **SET_PIXEL(x,y,val)**.
Par conséquence, la structure du nouveau protocole est **(numéro de méthode 7bits) :: arguments(25 bits)**.
7 bits pour le numéro de fonction, on peut étendre jusqu'à 128 primitives maximum, c'est très suffisant.
25 bits suffit de représenter tous les arguments pour l'instant. Il y à des protocoles comme ci-dessous

- **SET_PIXEL(x,y,val)** : 1(7bits) ::x(12 bits) ::y(12 bits) ::val(1 bits), (S -> C)
- **CLEAR_SCREEN** : 2(32 bits), (S -> C)
- **SET_PIN(p,n)** : 3(7 bits) ::p(8 bits) ::n(17 bits), (S -> C)
- **WRITE_PIN(p,v)** : 4(7 bits) ::p(8 bits) ::v(17 bits), (S -> C).

- **INVERSE_PIN(p,n)** : 0(7 bits) ::p(25 bits) (C->S).

4.2 Synchronization

Deux processus exécutent en concurrence, le plus important est que comment synchroniser leur tâche ? Nous avons testé le simulateur sans synchronization, et nous avons observé qu'il y a des protocoles qui ne sont pas visualisées sur l'interface graphique. C'est parceque la visualisation de quelques protocoles prend du temps. Par exemple éteindre tous les LEDs ou afficher une image. Donc on a implémenté un mécanisme de communication, il sert à garantir que chaque protocle peut être bien visualisée. Ce sont les étapes pour la communication entre le processus serveur et client.
Etapes pour afficher d'un protocole

1. **Client listener** attend le protocole de serveur.
2. **Server main** vérifie si le protocole précédente est bien visualisé.
   - Si oui, il envoie la nouveau protocole au client.
   - Sinon, **Server main** attend .
3. **Server main** réveille le thread **client listener**.
4. **Client listener** vérifie si la nouvelle instruction arrive.
   - Si oui, il informe au **GUI** afin d'afficher le nouvel état de composant.
   - Sinon, c'est un déblocage fausse, il continue à attendre.
5. **Client listener** réveille le thread **server main**.
6. retour à l'étape 1.

Pour garantir que ces deux processus exécutent étape par étpae. Nous avons essayé plusieurs manières diverses au fur et mesure du développement.

4.2.1 Pipe + Signal

Au début, nous utilisions le pipe et le signal pour transmettre l'instruction. Pour la direction du serveur au client, le protocole est transmis par le pipe, et la direction inverse, on a utilisé le signal

pour représenter le protocole, parce qu'il n'y a que événements, soit appuyer le bouton A soit le boutton B, donc on a attaché deux signaux SIGUSR1 et SIGUSR2 au ces deux bouttons, si le boutton est appuyé, le serveur reçoit SIGUSR1 ou SIGUSR2, et la fonction **handler** inverse l'état de pin correspondant.
Son avantage est que l'on n'a plus besoin exécuter un thread **listener** dans le processus **serveur**. Comme la fonction **handler** exécute dès que le processus serveur reçoit le sigal depuis le processus client.
Mais le mécanisme de signal est très limité, le signal utilisable est divers pour le système différent, par exemple pour linux on a droit d'utiliser les signaux qui sont supérieur à 32, mais pour macOS, il y a seulement quatres signaux qu'on peut utiliser.

4.2.2 Pipe

Après qu'on a constaté la limite du signal, on a essayé d'utiliser deux pipes pour la communication en deux direction. Pipe possède le mécanisme de protection, et il peut synchroniser automatiquement la communication, mais il génère un fichier temporaire pour la lecture et l'écriture, ainsi que il est moins efficace que la mémoire partagée. Donc on a finalement décidé d'utiliser la mémoire partagée.

4.2.3 Mémoire partagée

La mémoire partagée sert à communiquer entre le serveur et client, puisque c'est la plus rapide manière du transport de données. Mais son inconvénient est que l'on doit implémenter un mécanisme de synchronization afin de protéger les données partagées pour la lecture et l'écriture. Donc pour deux directions du transport (du serveur au client et du client au serveur), on a besoin de deux mémoires partagées, **shm1** est écrit par le serveur, le client le lit. **shm2** est inversé.

Afin de synchroniser la lecture et l'écriture, on met un mutex et une variable conditionnelle pour chaque mémoire partagée. Il garant que chaque instruction de serveur peut être bien traiter.

5 Méthodologie de réalisation

Selon ce qui précède, notre objectif est de simuler les informations d'entrée et de sortie du microcontrolleur monopuce, pour réaliser les fonctionnalité du simulateur, il y a deux versions, en **haut niveau** et en **bas niveau**.

5.1 Version haut niveau

haut niveau est que l'on simule abstraitement un microcontrôleur, la simulation haut niveau ne reflète pas les intéractions réelles entre le microcontrôleur et son environement. On implémente les tableau pour stocker les états de chaque composant. Dès que l'état de composant est modifié par la primitive, on modifie le tableau directement.
Par exemple le serveur appelle la primitive **microbit_write_pixel 0 0 true**, il ne suffit de modifier le tableau qui stock l'état des LEDs. pixels[0][0] = true.

Avantage
La version est plus simple, il a juste besoin de modifier le tableau correspondant.

Inconvénient
Cette méthode n'est pas générale, parce que les composants sont différents pour les microcontôleurs divers, on ne peut pas implémenter les tableau pour chaque microcontrôleur. Si le microcontrôleurs se change, nous devons réimplémenter le protocole et les tableaux des composants.

5.2 Version bas niveau

bas niveau est que l'on simule réellement un microcontrôleur, afin de mieux simuler l'état de fonctionnement réel du microcontrôleur à partir de l'architecture du microcontrôleur, nous espérons modifier directement l'état du **tableau de pins**, sans avoir besoin d'autres appels de fonctions, et refléter directement les effets de ces modifications sur **l'interface graphique**. Cela fera quelques changements basés sur le haut niveau précédent.
Lorsque nous voulons allumer un led, nous devons d'abord chercher son **pin_row** et **pin_col** correspondant, puis mettons **pin_row** à haut niveau et **pin_col** à bas niveau.

Avantage La simulateur de bas niveau peut s'adapter à un grand nomrbe de composants électroniques externes au microcontrôleur puisque les interactions sont réalisées par la modification des états des pins. Donc le protocole **SET_PIN** est général pour le plupart des primitives, on n'a plus besoin de protocle spécifique comme **SET_PIXEL**. Même si le microcontôleur branche un composant spécifique, par exemple le moteur, l'écran LCD, etc, on a juste besoin de changer un fichier du montage, et rajouter un protocole spécifique mais pas tout le simulateur.

Inconvénient Le traitement d'une instruction est plus compliqué, l'interaction entre le serveur et le client est plus fréquente.
Par exemple, pour allumer un **LED**, le serveur envoie une seule instruction **SET_PIXEL(X,Y,True)** au client dans la version haut niveau. Mais dans le bas niveau, le serveur doit d'abord chercher le **pin_row** et **pin_col**, puis envoyer deux instructions **SET_PIN(pin_row, 1)** et **SET_PIN(pin_col, 0)** au client.

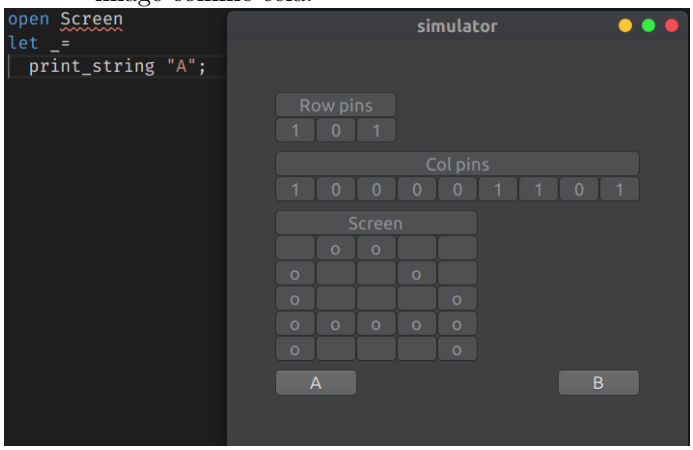
6 Fonctionalités Implémentées

6.1 Visualiser du texte sur la matrice des leds

Tout d'abord,nous convertissons les informations de chaque caractère en forme des graphiques raster dans une table en code hexadécimal 5x8 bits en bien tirer par order du code ASCII.Lorsqu'on recevoir un command d'imprimer une caractère à l'écran, on va calculer ASCII code correspond à cette caractère,et puis en fonction de ce ASCII code ,on peut calculer le décalage dans la table de code hexadécimal correspond à cette caractère. Sortir enfin les 5 * 8 bits d'information dont il dispose. Prendre 5 * 5 bits d'entre eux et les imprimer pixel par pixel sur l'écran.

Une exemple de visualiser la caractère 'A' sur la matrice des leds.

1. calcule le code ASCII de 'A', c'est 65.
2. calcule son décalage c'est 65 * 5 = 325.(parce que l'écran est de 5 lignes et chaque ligne est composé de 2 chiffre hexadécimal)
3. cherche son table de code hexadécimal correspondant dans le tableau, c'est {30,28,38,28,28}.
4. nous convertissons code hexadécimal en forme code binaire.
   00110000
   00101000
   00111000
   00101000
   00101000
5. modifier les Pins de chaque LED en fonction de cette code binaire. On a finalement une image comme cela.



6.2 Analyser le ficher du montage

Pour le montage de microcontrôleur, il a besoin d'un fichier qui stocke les informations de chaque composant de microcontrôleur comme le nombre des pins, les association entre chaque composant et le pin correspondant. Une fois que le simulateur connait ces informations, il peut implémenter l'interface graphique et contrôler les composants par modifier l'état des pins.
Donc on a utilisé Flex, Yacc et langage C afin de réaliser un langage descriptif.

1. On défini les **tokens** dans le fichier **parser.y**.
2. Lier les mots prédéfinis avec ces tokens dans **lexer.lex**
3. Créer les noeuds de **AST**(l'arbre d'analyse syntaxe) dans fichier **AST.c** et **AST.h**.
4. Réaliser les fonctions de l'évaluation dans le fichier **get_env.c**.

```
toto{
    nb_pins(2,2), nb_leds 2, nb_buttons 2, screen(1, 2)
    [
        led 0: (0, 0), pin 0, pin 0;
        led 1: (0, 1), pin 0, pin 1;
        button 0: A pin 0;
        button 1: B pin 1;
    ]
}
```

(b) cf.exmple **circuit.txt**

Le grammaire et les mots prédéfinis

1. **nom du simulateur{l'information du montage}** : C'est la structure du langage.
2. **nb_pins (num_row, num_col)** : **num_row** est le nombre de pins de ligne(row), **num_col** est celui de colonne.
3. **nb_leds num** : **num** est le nombre de leds.
4. **nb_buttons num** : **num** est le nombre de buttons.
5. **screen(row, col)** : Cela indique le nombre de ligne et colonne de matrice des LEDs.
6. **led id : (row, col), pin id1, pin id2** : Cela représente le numéro du LED et ses coordonnées dans la matrice, ainsi que l'association entre le led et deux pins correspondant. id1 est l'identifiant de pin row, id2 est celui de pin col.
7. **button id : étiquette pin id** : Cette une déclaration du bouton dont l'identifiant est id et son étiquette, ainsi que le pin correspondant.

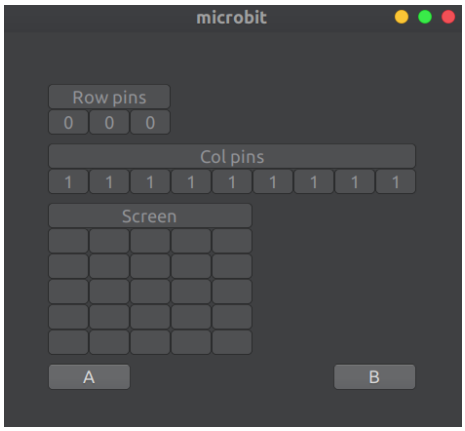
7 Sénarios et Tests

Ce sont les étapes de simuler un programme ocaml par la simulateur du microbit.

1. Charger l'information du montage par un fichier **circuit.txt**.
2. Le processus montage analyse ce fichier et génère une structure du montage, en plus envoie au serveur et au client par le mémoire partagée **envid**.
3. Le processus client implémente une interface graphique en fonction de structure du montage.

```
microbit{
    nb_pins(3,9), nb_leds 25, nb_buttons 2, screen(5, 5)
    [
        led 0: (0, 0), pin 0, pin 0;
        led 1: (0, 2), pin 0, pin 1;
        led 2: (0, 4), pin 1, pin 3;
        led 3: (3, 4), pin 1, pin 4;
        led 4: (3, 3), pin 0, pin 2;
        led 5: (3, 2), pin 2, pin 3;
        led 6: (3, 1), pin 2, pin 4;
        led 7: (3, 0), pin 2, pin 5;
        led 8: (2, 1), pin 2, pin 6;
        led 9: (2, 4), pin 2, pin 7;
        led 10: (2, 0), pin 1, pin 1;
        led 11: (2, 2), pin 0, pin 8;
        led 12: (0, 1), pin 1, pin 2;
        led 13: (0, 3), pin 2, pin 8;
        led 14: (4, 3), pin 1, pin 0;
        led 15: (4, 1), pin 0, pin 7;
        led 16: (4, 2), pin 0, pin 6;
        led 17: (4, 4), pin 0, pin 5;
        led 18: (0, 4), pin 0, pin 4;
        led 19: (1, 0), pin 0, pin 3;
        led 20: (1, 1), pin 2, pin 2;
        led 21: (1, 2), pin 1, pin 6;
        led 22: (1, 3), pin 2, pin 0;
        led 23: (1, 4), pin 1, pin 5;
        led 24: (2, 3), pin 2, pin 1;
        button 0: A pin 0;
        button 1: B pin 1;
    ]
}
```

(c) la description du circuit

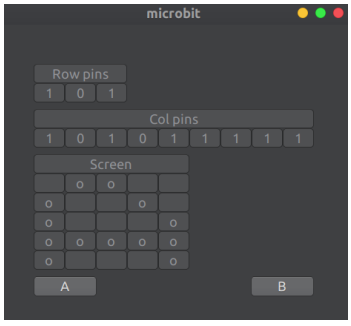


(d) l'interface graphique de microbit

4. Le serveur interprète le programme OCaml, puis envoie des protocoles au client.
5. Le client visualise les effets du programme.

```
open Screen
let _=
  print_string "A";
```

(e) un programme ocaml



(f) L'effet de la visualisation

8 Problème et Solution

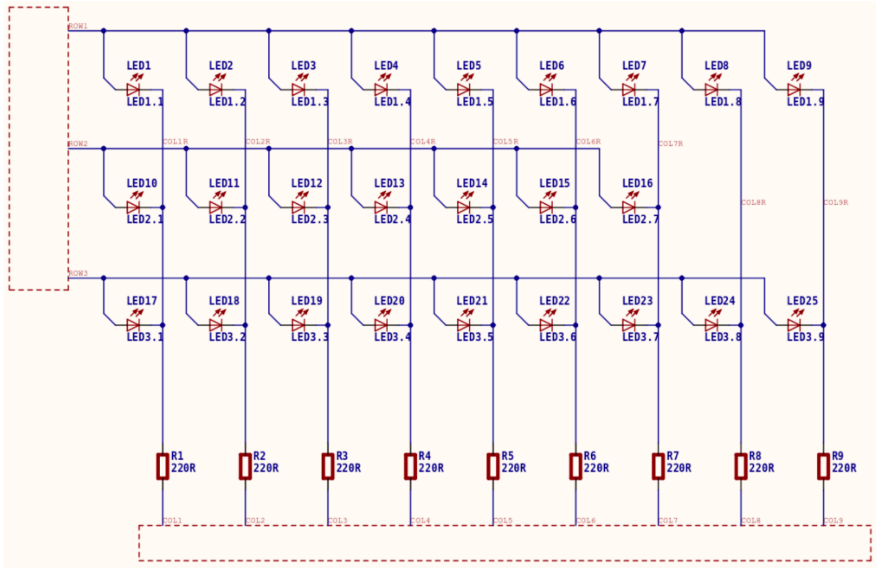
8.1 Choix de processus ou de thread

Quand on désigne l'architecture du simulateur, on a deux choix pour le côté du serveur et client. Soit on les sépare en deux threads, soit deux processus. Les avantages du thread est que c'est plus facile de partager les données et réaliser la synchronization. Néanmoins, la partie serveur est compilé par **ocamlc** dans une étape de la compilation, et nous utilisons **gtk+3.0** pour réaliser l'interface graphique, le compilateur **ocamlc** ne peut pas compiler la bibliothèque de **gtk+3.0**, donc on sépare le serveur et le client en deux processus.

8.2 Synchronization

Comme ce que l'on a présenté, le serveur envoie les protocoles au client, et le client visualise simultanément les états de chaque composant, mais pour quelque protocole par exemple **CLEARSCREEN**, il prend du temps parce qu'il faut éteindre tous les LEDs. Donc s'il n'y a pas de mécanisme de synchronization afin de garantir que le nouveau protocole arrive après le traitement du protocole précédent, il risque de perdre quelques instructions.
Pour synchroniser le transfert du protocole entre le serveur et le client, nous avons essayé plusieurs méthodes

8.3 Problème de conflit



(g) cf.le circuit réel de microbit

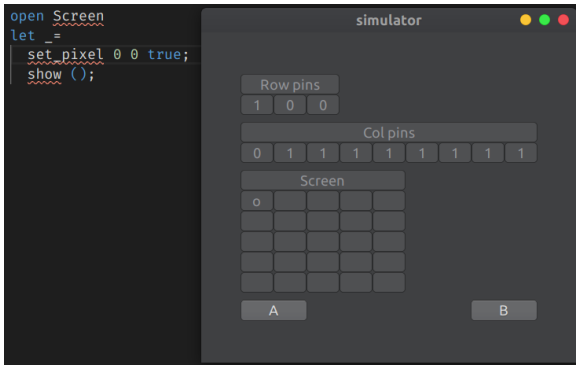
Cet image montre le circuit du **microbit**, la matrice de LED se compose d'une matrice de pins de 3 * 9. Pour allumer un LED, on doit mettre le **pin_row** correspondant au niveau **HIGH**, et mettre son **pin_col** au niveau **LOW**.

1.1	2.4	1.2	2.5	1.3
3.4	3.5	3.6	3.7	3.8
2.2	1.9	2.3	3.9	2.1
1.8	1.7	1.6	1.5	1.4
3.3	2.7	3.1	2.6	3.2

(h) cf.le tableau des associations entre le LED et les pins correspondant

Ce tableau montre la relation de position entre le circuit du **microbit** et l'éran du **microbit**.

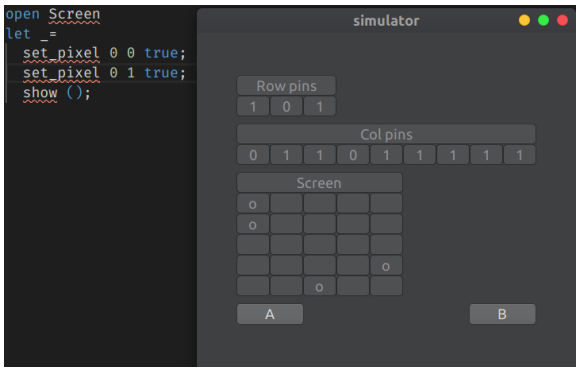
Lorsque nous voulons allumer un seulement LED, il n'y a pas de conflit. Par exemple, pour **LED 0 0**, on met **pin_row1** au niveau **HIGH**, et **pin_col1** au niveau **LOW**. Les effets de la simulation est comme cet image.



(i) cf.exmple **allumer LED 0 0**

Néanmoins, si on veut allumer deux LEDs dont le pin_row n'est pas la même, il y aura le conflit. Par exemple on allume **LED 0 0** et **LED 0 1**. On doit mettre **pin_row1** et **pin_row3** au niveau **HIGH**, mettre **pin_col1** et **pin_col4** au niveau **LOW**. Dans ce cas-là, il y aura quatre LEDs allumé.

1. **LED 0 0** correspondant à **pin_row1, pin_col1**.
2. **LED 4 3** correspondant à **pin_row1, pin_col4**.
3. **LED 0 1** correspondant à **pin_row3, pin_col4**.
4. **LED 2 4** correspondant à **pin_row3, pin_col1**.



(j) cf.exmple **allumer LED 0 0 et LED 0 1**