



RAPPORT DE PSTL

Trac-Simulateur & IDE génériques pour OMicrob

Author :

Qiwei XIAN
Ruiwen WANG

Professeur :

Prof. EMMANUEL CHAILLOUX

3 juin 2020

Table des matières

1	Introduction	2
2	Structure de compilation	3
3	Composition du simulateur	4
3.1	Montage	4
3.2	Client	4
3.3	Serveur	4
4	Architecture du simulateur	5
4.1	Tâche du thread	5
4.2	Mécanisme de communication	6
4.3	Protocole	7
5	Fonctionnalités Implémentées	8
5.1	Visualiser les phrases sur la matrice des leds	8
5.2	Analyser le fichier du montage	9
6	Méthodologie de réalisation	10
6.1	Version haut niveau	10
6.2	Version bas niveau	10
7	Sénarios et Tests	11
8	Problème et Solution	13
8.1	Choix de processus ou de thread	13
8.2	Synchronisation	13
8.2.1	Pipe + Signal	13
8.2.2	Pipe	13
8.3	Problème de conflit	14

1 Introduction

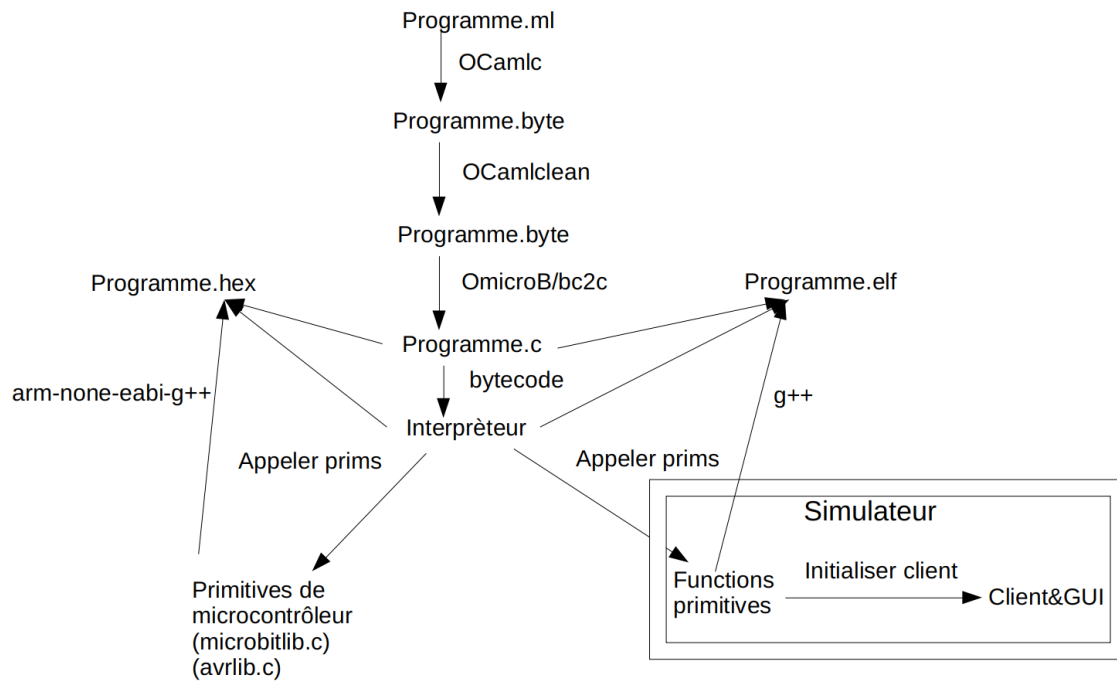
lien vers les codes sources : <https://github.com/XIANQw/OMicroB/tree/microbit>

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 Structure de compilation



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

1. **ocamlc** compile le fichier **.ml** avec la bibliothèque et génère 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 (**avr-lib**, **microbitlib**, etc) puis génère 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 Composition du simulateur

Le simulateur est composé du côté de client et du côté de serveur, ainsi que un langage de description pour le montage des périphériques sur un microcontrôleur.

3.1 Montage

Pour démarrer le processus client et implémenter l'interface graphique, le client a besoin de savoir quels composants du microcontrôleur il doit visualiser 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 un mémoire partagé après l'évaluation. Le client peut recevoir les informations des composants du microcontrôleur depuis ce mémoire partagé.

Cette fonctionnalité est réalisée dans le dossier **src/byterun/montage**, la structure est définie dans **src/byterun/simul/share.h**.

3.2 Client

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

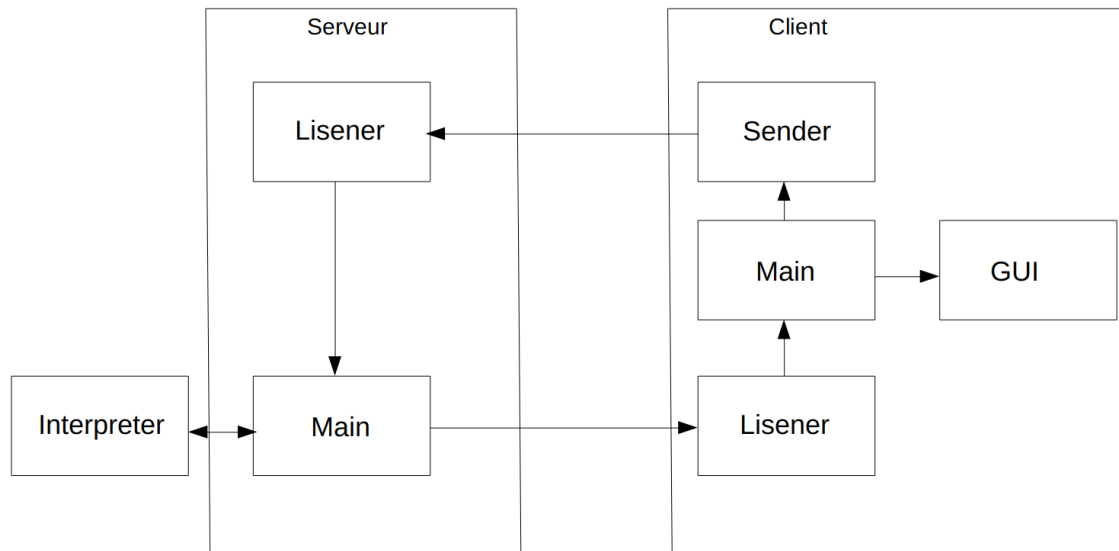
1. **client.c** définit les fonctions du travail par exemple traiter le message qui vient du côté server.
2. **gui.c** implémente l'interface graphique par **gtk3.0**. il nous montre un UI de simulateur permet de visualiser le résultat de programme et les états des Pins pendant la simulation, ainsi que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

3.3 Serveur

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

1. **/src/byterun/vm**. Ce dossier contient les fichiers de l'interpréteur d'OMicroB, par exemple le format des type basiques de données, le gc 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, lorsque le programme utilise un primitive par exemple **set_pin**, l'interpréteur va appeler le primitive correspondant dans la bibliothèque, le serveur envoie l'instruction au côté client après le calcul.
3. **src/byterun/shared.c/.h**. Ce fichier définit des structures de données partagées entre le processus client et serveur, elles sont stockées dans un mémoire partagé au fur et mesure de l'exécution de programme.

4 Architecture du simulateur



Cet image décrit globalement la structure du simulateur. On l'a réalisé par **l'architecture serveur-client** et il exécute deux processus en concurrence. Chaque processus exécute plusieurs threads et travaille asynchroniquement.

4.1 Tâche du thread

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

1. **Main** traite simplement l'instruction qui vient de l'interpréteur et envoyer au thread **lisener** de client.
2. **lisener** reçoit l'instruction venant de **Sender** et informer **Main**.

Processus Client tourne simultanément quatre threads, **Main**, **GUI**, **Sender** et **Lisener**.

1. **Main** est le premier thread principal, il traite les arguments venant de processus serveur et connecte aux mémoires partagés, par exemple le mémoire partagé 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 de visualiser 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. **Lisener** reçoit l'instruction venant de serveur, en plus manipuler les composants de l'interface graphique en fonction de l'instruction.

4.2 Mécanisme de communication

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 synchronisation 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 garantit que chaque instruction de serveur peut être bien traitée.

Sénaire du traitement d'une instruction

1. **client lisener** se bloque pour attendre l'instruction de serveur.
2. **serveur main** demande le mutex
3. **serveur main** vérifie la variable conditionnelle.
 - Si la condition se satisfait, il passe à l'étape suivante. - Sinon, cela veut dire que l'instruction dernière n'est pas encore traitée par client, **serveur main** se bloque.
4. **serveur main** envoie l'instruction au client.
5. **serveur main** débloque le thread **client lisener**.
6. **serveur main** rend le mutex.
7. **client lisener** est débloqué, demande le mutex.
8. **client lisener** vérifie si la condition se satisfait. - Si oui, cela veut dire que la nouvelle instruction arrive, il passe à l'étape suivante. - Sinon, c'est un déblocage fautive, il se bloque.
9. **client lisener** informe au **GUI** afin de visualiser le nouvel état de composant.
10. **client lisener** débloque le thread **serveur main**.
11. **client lisener** rend le mutex.
12. retour à l'étape 1.

4.3 Protocole

Le protocole est un rôle de transférer les informations entre coté client et coté serveur. Il est un entier de 32 bits. la structure est (numéro de méthode 7bits) :: arguments(25 bits). Afin de représenter le protocole en un entier de 32 bits. on a utilisé 7 bits pour le nom de fonction, alors on peut étendre jusqu'à 128 primitives maximum.

(S -> C) indique que la direction de l'envoi est du serveur au client.

- **SET_PIXEL(x,y,val)** : 1(7bits) ::x(12 bits) ::y(12 bits) ::val(1 bits)
(S -> C) Modifier l'état du pixel des coordonnées x et y à val.
Dans la version de haut niveau, ce protocole sert à modifier directement l'état de pixel. Dans la version de bas niveau, ce protocole est remplacé par SET_PIN.
- **CLEAN_SCREEN** : 2(32 bits)
(S -> C) Mettre les états de tous les pixels à 0, envoyé par **microbit_clean_screen**.
- **SET_PIN(p,n)** : 3(7 bits) ::p(8 bits) ::n(17 bits)
(S -> C) Modifier le pin p au niveau n. (n = 0 ou 1), envoyé par **microbit_digital_write**.
- **WRITE_PIN(p,v)** : 4(7 bits) ::p(8 bits) ::v(17 bits)
(S -> C) Modifier le pin p à la valeur v (0 <= v <= 1024), envoyé par **microbit_analog_write**.
- **APPUIE_A** : 0(32 bits)
(C -> S) Inverser l'état du bouton A.
- **APPUIE_B** : 1(32 bits)
(C -> S) Inverser l'état du bouton B.

5 Fonctionnalités Implémentées

5.1 Visualiser les phrases sur la matrice des leds

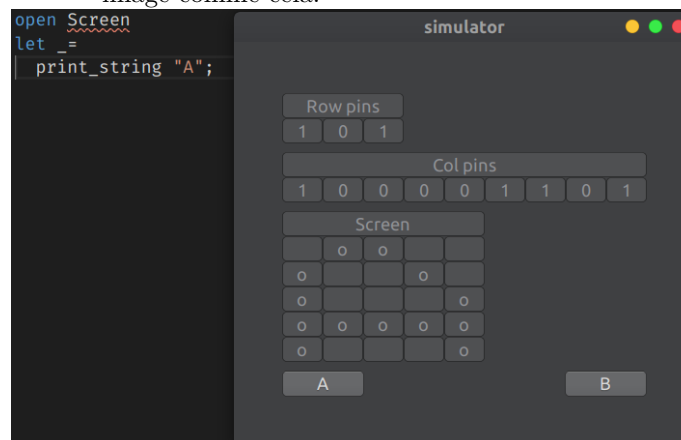
Tout d'abord, nous convertissons les informations de chaque caractère en forme des graphiques raster dans une table en code hexadécimal 5x8 bits en bien tirer par order du code ASCII. Lorsqu'on recevoir un command d'imprimer une caractère à l'écran, on va calculer ASCII code correspond à cette caractère, 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.



5.2 Analyser le fichier 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;
    ]
}
```

(a) cf.exemple **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.

6 Méthodologie de réalisation

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

6.1 Version haut niveau

haut niveau est que l'on simule abstraitement un microcontrôleur, on implémente les tableaux 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 a juste besoin de modifier le tableau qui stocke 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 microcontrôleurs divers, on ne peut pas implémenter les tableaux pour chaque microcontrôleur.

6.2 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.

Par exemple, Un **LED** est contrôlé par deux **pins** (**pin_row** et **pin_col**) dans le circuit du microcontrôleur réel. 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 Cette version est générale pour tous les microcontrôleurs, parce que on manipule directement les états de **pins** au lieu de composant. Même si le microcontrôleur est changé, on a juste besoin de changer un fichier du montage, mais pas 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.

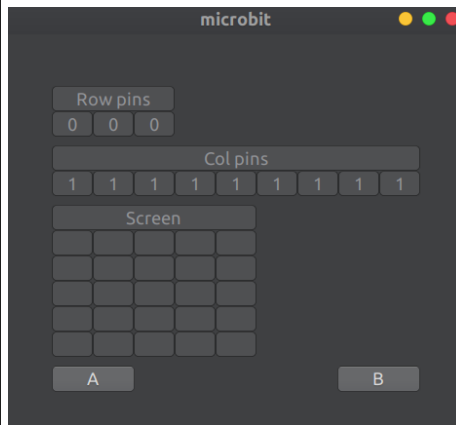
7 Scénarios et Tests

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;
  ]
}
```

(b) la description du circuit

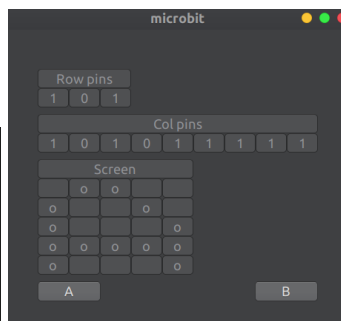


(c) l'interface graphique de microbit

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";
```

(d) un programme ocaml



(e) 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 synchronisation. 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 synchronisation 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 diverses dans le processus du développement.

8.2.1 Pipe + Signal

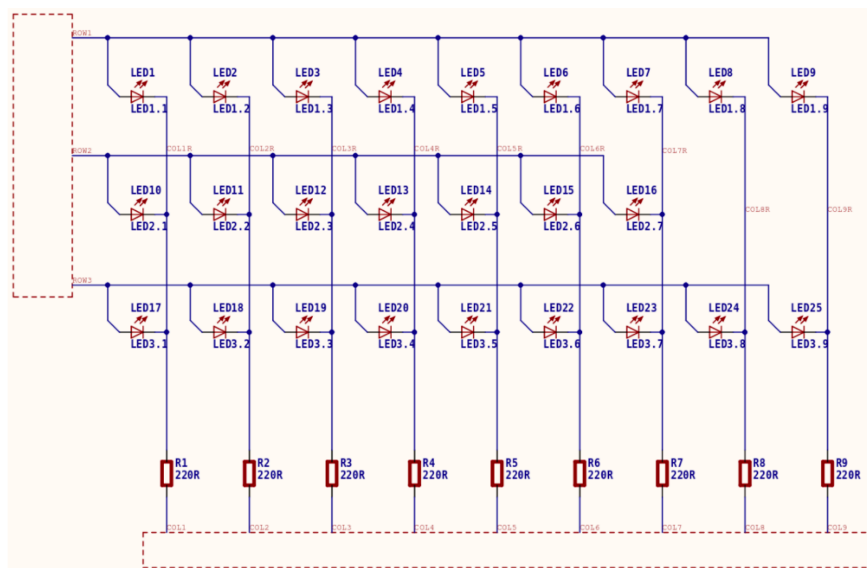
Au début, nous utilisons le pipe et le signal pour transmettre le protocole. Pour la direction du serveur au client, le protocole est transmis par le pipe, et la direction inverse, on a utilisé le signal pour représenter le protocole, parce qu'il n'y a que deux protocoles(**APPUIE_A** et **APPUIE_B**). Le serveur traite le protocole par la fonction **handler**. Son avantage est que l'on n'a plus besoin exécuter un thread **lisener** dans le processus **serveur**. Comme la fonction **handler** exécute dès que le processus serveur reçoit le sigal depuis le processus client.

Mais la 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.

8.2.2 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 la 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.

8.3 Problème de conflit



(f) cf.le circuit réel de microbit

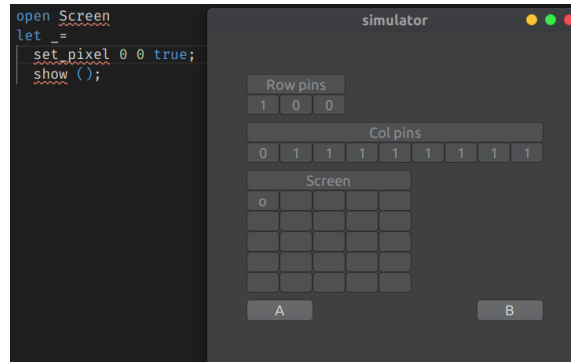
Cet image montre le circuit du **microbit**, la matrice de LED se compose d'une matrice de pins de $3 * 9$. Pour allumer un LED, on doit mettre le **pin_row** correspondant au niveau **HIGH**, et mettre son **pin_col** au niveau **LOW**.

	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	

(g) 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'écran du **microbit**.

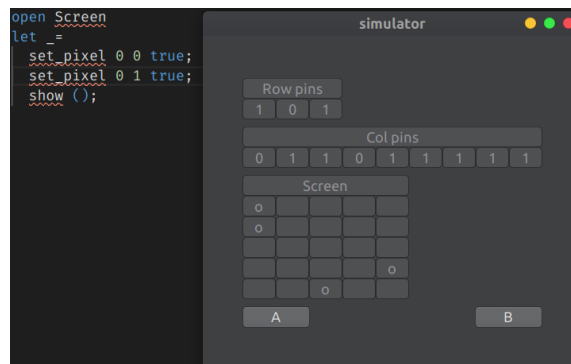
Lorsque nous voulons allumer un seulement LED, il n'y a pas de conflit. Par exemple, pour **LED 0 0**, on met **pin_row1** au niveau **HIGH**, et **pin_col1** au niveau **LOW**. Les effets de la simulation est comme cet image.



(h) cf.exemple 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 allumés.

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**.



(i) cf.exemple allumer **LED 0 0** et **LED 0 1**