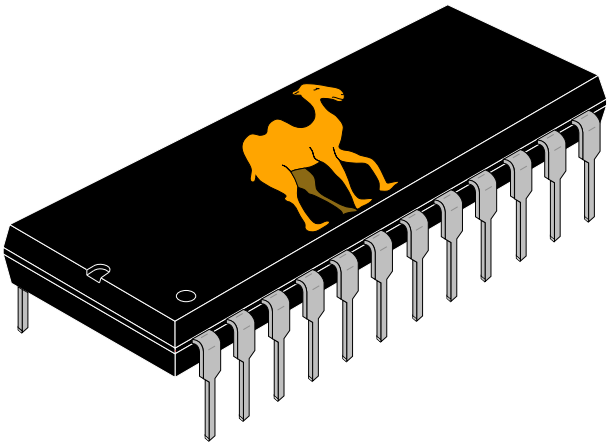
Tutoriel sur OCaPIC
Version 1.0

OCaml sur microcontrôleur PIC

Benoît Vaugon



6 août 2011

1 Introduction

OCaPIC est un ensemble d'applications et de bibliothèques permettant de programmer les microcontrôleurs PIC en OCaml. Il permet de bénéficier de l'intégralité du langage OCaml sur PIC. Il fournit la majeure partie de la bibliothèque standard OCaml ainsi que des bibliothèques spécifiques aux PIC permettant de manipuler directement les interruptions matérielles, des afficheurs LCD, une connection série, ...

Il possède de plus un mécanisme de simulation permettant de mettre au point les programmes OCaml sur un ordinateur avant de les transférer sur le PIC.

Plus de documentation est disponible à partir de son site web :

`http://www.algo-prog.info/ocapic/`

2 Installation

2.1 Prérequis

- Un compilateur C (par exemple `gcc`).
- Le compilateur standard OCaml distribué par l'INRIA (`http://ocaml.org/`). Sous Linux, il suffit d'installer le paquet `ocaml`.
- Un assembleur PIC pour convertir les fichiers assembleurs (`.asm`) générés par OCaPIC en code hexadécimal (`.hex`) à transférer sur le PIC. Utilisez par exemple le programme `gpasm` fourni par le paquet `gputils` sous Linux.

2.2 Téléchargement

À partir du site d'OCaPIC : `http://www.algo-prog.info/ocapic/`, section *Download*, téléchargez la dernière version disponible. Vous devez obtenir un fichier nommé `ocapic-X.Y.tar.bz2` (remplacez `X.Y` par le numéro de version).

2.3 Installation

Voici comment installer OCaPIC sous Linux. L'installation sous Windows est également possible grâce à Cygwin.

Dans un terminal (en remplacant chaque fois `X.Y` par le numéro de version) :

- Décompressez le fichier téléchargé `ocapic-X.Y.tar.bz2` :

  ```
  tar jxf ocapic-X.Y.tar.bz2
  ```

- Allez dans le répertoire `ocapic-X.Y` :

  ```
  cd ocapic-X.Y/
  ```

- Configurez l'application. Cette opération vérifie les dépendances et permet de configurer les répertoires d'installation et les PIC gérés. Par défaut, seul le `PIC18F4620` est géré car la compilation de l'interfaçe avec chaque PIC est très longue. Se référer au fichier `INSTALL` pour plus de détails :

  ```
  ./configure
  ```

- Compilez OCaPIC :

  ```
  make
  ```

- Installez OCaPIC (en tant que super-utilisateur) :

  ```
  sudo make install
  ```

  Ou :

  ```
  su
  make install
  ```

3 Exemples d'utilisation

Tous les exemples présentés dans ce tutoriel sont adaptés au `PIC18F4620`. Bien que tous les pics de la série `18F` soient normalement supportés, il est recommandé de commencer avec ce PIC.

3.1 Faire clignoter une LED

Commençons par un premier exemple très simple : faire clignoter une LED avec un PIC. Il n'y a, bien entendu, absolument aucun intérêt à utiliser un langage de haut niveau comme OCaml, ni même un microcontrôleur, pour un exemple si simple. Le seul but de cette section est de montrer les différentes étapes de la programmation du PIC en OCaml.

Code source OCaml :
Placez le code OCaml suivant dans un fichier `led.ml` :

```
open Pic;;              (* Module contenant write_reg, set_bit, RB0, ... *)

write_reg TRISB 0x00; (* Configurer le port B en sortie *)

while true do
  set_bit RB0;          (* Allumer la LED *)
  Sys.sleep 500;        (* Attendre 0,5s *)
  clear_bit RB0;        (* Eteindre la LED *)
  Sys.sleep 500;        (* Attendre 0,5s *)
done
```

Configuration du PIC :
Il est nécessaire de spécifier un certain nombre d'options de configuration du PIC (tension de programmation, sélection de l'horloge, configuration du reset externe, sélection des segments de code protégés, ...). Pour ce faire, placez le code suivant dans un fichier `config.asm` :

```
        config OSC     = INTIO7
        config PWRT    = ON
        config BOREN   = OFF
        config WDT     = OFF
        config MCLRE   = OFF
        config PBADEN  = OFF
        config STVREN  = OFF
        config LVP     = OFF
```

Compilation :
Compilez le fichier `led.ml` pour le `PIC18F4620` grâce à la commande :

```
ocapic 18f4620 config.asm led.ml
```

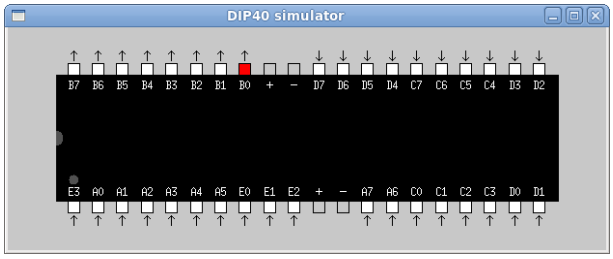
Vous devez obtenir, entre autres, les fichiers `led.asm` (contenant le code assembleur PIC) et `led` (un exécutable pour ordinateur utilisé pour la simulation).

Simulation :

Simulez l'exécution de votre programme sur le PIC grâce à la commande :

```
./led ocapic_dip40_simulator
```

Vous devez obtenir une fenêtre représentant un PIC ayant la patte `B0` qui clignotte :



Le programme ne se terminant jamais (`while true`), tuez-le avec `Ctrl-C` dans le terminal.

Assemblage :

Si vous utilisez `gpasm`, assemblez le fichier `led.asm` en exécutant :

```
gpasm -y led.asm
```

Cette commande génère (entre autres) le fichier hexadécimal `led.hex` à transférer sur le microcontrôleur.

Remarque : l'option `-y` permet d'activer le « jeu d'instructions étendu » du PIC. Attention à bien activer cette option si vous utilisez un autre assembleur que `gpasm`.

Programmation du PIC :

Vous disposez maintenant d'un fichier `led.hex` qui peut être transféré directement sur le PIC via un « programmateur de PIC ». Si vous avez la chance de disposer encore d'un ordinateur avec port série, vous pouvez utiliser un « programmateur série » à bas prix (ou vous en fabriquer un avec quelques composants). Sinon, utilisez un programmateur USB. Le projet « USBPICPROG » (`http://usbpicprog.org/`) permet d'en obtenir et d'en fabriquer à prix raisonnable.

Circuit :

Connectez une LED entre la patte `B0` du PIC et le − de l'alimentation (via une résistance, bien entendu !), alimentez le PIC en 5V et c'est gagné !

3.2 Hello world sur un afficheur LCD

OCaPIC fournit une librairie nommée `Lcd` permettant de manipuler de petits afficheurs LCD avec un PIC. Vous pouvez vous référer à la page de manuel de ce module pour obtenir sa documentation complète. Voici un petit exemple d'utilisation :

Code source OCaml :

Placez le code OCaml suivant dans un fichier `hw.ml` :

```
open Pic;;

module Disp = Lcd.Connect (
struct                          (* Configuration de la connection LCD/PIC : *)
  let bus_size = Lcd.Four          (* Taille du bus *)
  let e  = LATD0                   (* Pin validation : D0 *)
  let rs = LATD2                   (* Pin instruction/donnees : D2 *)
  let rw = LATD1                   (* Pin lecture/ecriture : D1 *)
  let bus = PORTB                  (* Bus : port B *)
end
);;

Disp.init ();;                     (* Initialisation de l'afficheur *)
Disp.config ();;                   (* Configuration de l'afficheur *)
Disp.print_string "Hello world";; (* Affichage d'une chaine de caracteres *)
```

Configuration du PIC :

Comme dans la section 3.1, placez le code suivant de configuration du PIC dans un fichier `config.asm` :

```
        config OSC     = INTIO7
        config PWRT    = ON
        config BOREN   = OFF
        config WDT     = OFF
        config MCLRE   = OFF
        config PBADEN  = OFF
        config STVREN  = OFF
        config LVP     = OFF
```

Compilation :

Compilez `hw.ml` → `hw`, `hw.asm` :

```
ocapic 18f4620 config.asm hw.ml
```

Assemblez `hw.asm` → `hw.hex` :

```
gpasm -y hw.asm
```

Simulation :

Exécutez la commande :

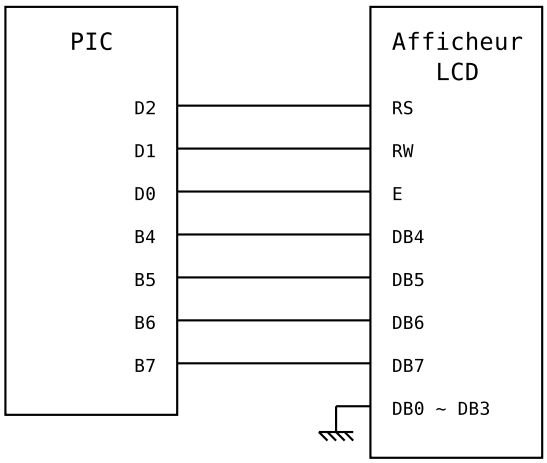
```
./hw 'ocapic_lcd_simulator 16x2 e=RD0 rs=RD2 rw=RD1 bus=PORTB'
```

Vous devez obtenir une fenêtre semblable à celle-ci :

Ce programme se terminant, pour arrêter la simulation, tapez simplement `Entrée` dans le terminal.

Circuit :

Connectez un afficheur LCD au PIC de la manière suivante :



Alimentez alors le PIC et l'afficheur LCD en 5V.

3.3 Une petite temporisation

Ce petit exemple a principalement pour but de montrer l'utilisation d'un plug-in intéressant pour la simulation : le simulateur de "circuit". On entend ici par "circuit" la connection au PIC d'un ensemble de composants standards comme des interrupteurs, des boutons poussoirs, des LED, des afficheurs 7-segments, des afficheurs LCD, ...

On souhaite créer ici une petite *temporisation* composée d'un afficheur LCD sur lequel défile un compteur de temps, et de quelques boutons poussoirs permettant d'ajuster le retard de la temporisation et de démarrer/arrêter la temporisation.

Code source OCaml :

Placez le code suivant dans un fichier `timer.ml` :

```
open Pic;;

(* Configuration de la connection PIC/Afficheur LCD *)
module Disp = Lcd.Connect (
struct
  let bus_size = Lcd.Four
  let e  = LATD0
  let rs = LATD2
  let rw = LATD1
  let bus = PORTC
end
);;

(* Initialisation et configuration de l'afficheur *)
Disp.init ();;
Disp.config ();;

(* Compteur (reference sur un Int32) *)
let counter = ref 0l;;

(* Utilitaire pour l'affichage *)
let print_02d n =
  if n < 10 then Disp.print_char '0';
  Disp.print_int n;
;;

(* Affiche la valeur du compteur *)
let print () =
  let t = !counter in
  let t60 = Int32.div t 60l in
  let s = Int32.to_int (Int32.rem t 60l) in
  let m = Int32.to_int (Int32.rem t60 60l) in
  let h = Int32.to_int (Int32.div t60 60l) in
    Disp.moveto 0 0;
    print_02d h;
    Disp.print_char ':';
    print_02d m;
    Disp.print_char ':';
    print_02d s;
;;

(* Utilitaires de gestion des overflow *)
let is_null () = !counter <= 0l and is_full () = !counter >= 359999l;;

(* Incremente le compteur de n *)
let incr n =
  counter := Int32.add !counter (Int32.of_int n);
  if is_full () then counter := 359999l;
;;
```

```
50
51 (* Decremente le compteur de n *)
52 let decr n =
53   counter := Int32.sub !counter (Int32.of_int n);
54   if is_null () then counter := 0l;
55 ;;
56
57 (* Incremente/Decremente le compteur de 1 *)
58 let decr1 () = if not (is_null ()) then counter := Int32.pred !counter;;
59 let incr1 () = if not (is_full ()) then counter := Int32.succ !counter;;
60
61 (* Fonction d'acceleration pour le reglage du temps *)
62 let step n = n / 32 * n / 8 * n + n / 16 * n / 16 + n / 8 + 1;;
63
64 (* Petite machine a etat *)
65 let rec run n =
66   if is_null () || test_bit RB0 then stop ()
67   else if test_bit RB1 then ( incr_start () ; run 0 )
68   else if test_bit RB2 then ( decr_start () ; run 0 )
69   else if n = 100 then ( decr1 () ; print () ; run 0 )
70   else ( Sys.sleep 10 ; run (succ n) )
71 and pause () =
72   Sys.sleep 10;
73   if test_bit RB0 && not (is_null ()) then start ()
74   else if test_bit RB1 then ( incr_start () ; pause () )
75   else if test_bit RB2 then ( decr_start () ; pause () )
76   else pause ();
77 and start () =
78   set_bit RB3;
79   Sys.sleep 10;
80    if test_bit RB0 then start () else run 0;
81 and stop () =
82   clear_bit RB3;
83   Sys.sleep 10;
84   if test_bit RB0 then stop () else pause ();
85 and incr_start () =
86   incr1 ();
87   print ();
88   Sys.sleep 300;
89   incr_loop 1;
90 and decr_start () =
91   decr1 ();
92   print ();
93   Sys.sleep 300;
94   decr_loop 1;
95 and incr_loop n =
96   if test_bit RB1 then (
97     incr (step n);
98     print ();
99     Sys.sleep 100;
100    if not (is_full ()) then incr_loop (succ n);
101  )
102 and decr_loop n =
103   if test_bit RB2 then (
104    decr (step n);
105    print ();
106    Sys.sleep 100;
107    if not (is_null ()) then decr_loop (succ n);
108  )
109 in
110   write_reg TRISB 0b00000111; (* Pattes connectees au boutons en entree *)
111   print ();                   (* Affichage de 00:00:00 *)
112   stop ();                    (* Point d'entree de la machine a etat *)
113 ;;
```

Configuration du PIC :

Créez le même fichier de configuration `config.asm` que dans la section 3.1.

Circuit virtuel :

Le plug-in permettant la simulation du circuit prend en paramètre le nom d'un fichier contenant la description du circuit dans un format assez intuitif. Placez les définitions suivantes dans un fichier `circuit.txt` :

```
window width=270 height=220 title="Timer"

button x=85 y=40 width=40 color=black label="START" pin=RB0
button x=135 y=55 width=20 height=20 label="+" pin=RB1
button x=135 y=25 width=20 height=20 label="-" pin=RB2

led x=175 y=55 pin=RB3 color=green
led x=175 y=25 pin=RB3 inverse=true

lcd x=10 y=100 column_nb=8 line_nb=1 e=RD0 rs=RD2 rw=RD1 bus=PORTC
```

Compilation :

Compilez `timer.ml` → `timer`, `timer.asm` :

```
ocapic 18f4620 config.asm timer.ml
```

Assemblez `timer.asm` → `timer.hex` :

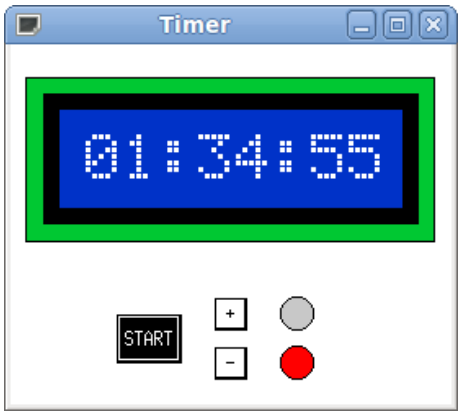
```
gpasm -y timer.asm
```

Simulation :

Exécutez la commande :

```
./timer 'ocapic_circuit_simulator circuit.txt'
```

Vous devez obtenir une fenêtre semblable à celle-ci :



Vous pouvez alors simuler l'interaction d'un utilisateur avec la temporisation en cliquant sur les boutons `+`, `-` et `START`. Pour arrêter la simulation, tapez `Ctrl-C` dans le terminal.

3.4 Communication avec un ordinateur

La communication entre un Ordinateur et un PIC peut se faire grâce aux deux modules `Serial` de la librairie (un module pour le PIC et un pour l'ordinateur). Ils permettent d'échanger directement des valeurs OCaml entre un programme s'exécutant sur un ordinateur et le programme s'exécutant sur le PIC.

Les valeurs OCaml échangées entre les deux programmes peuvent être de toute nature tant que cela a du sens. Cela peut être, par exemple, des entiers, des chaînes de caractères ou des flottants ; mais aussi des structures de données plus complexes comme des listes, des tableaux, etc. Les seules valeurs interdites sont celles contenant un pointeur de code comme, par exemple, les fermetures. Il s'agit d'un véritable échange de *graphe mémoire*, les valeurs peuvent contenir des cycles.

La communication est protégée contre les erreurs de transmission, les débranchements accidentels, etc. De plus, les débordements mémoire et les collisions dans le protocole de communication sont gérés correctement et traduites en exceptions.

Si vous ne possédez pas de port série sur votre ordinateur, vous pouvez utiliser un convertisseur USB/Série.

L'utilisation de cette librairie est entièrement gérée par le simulateur.

La documentation précise de chaque fonction se trouve dans les pages de manuel.

Voici un exemple d'utilisation permettant de piloter un afficheur LCD connecté au PIC depuis un top-level OCaml lancé sur un ordinateur :

Code source OCaml pour Ordinateur :

Placez le code suivant dans un fichier `sender.ml` en adaptant éventuellement le chemin du fichier correspondant au port série (ici `/dev/ttyUSB0`) :

```
1  open Serial;;
2
3  type t = Clear | Moveto of int * int | Int of int | Text of string
4
5  let (chan : (t, unit) channel) = open_tty "/dev/ttyUSB0";;
6
7  let clear () = send chan Clear;;
8  let moveto l c = send chan (Moveto (l, c));;
9  let print_int i = send chan (Int i);;
10 let print_string s = send chan (Text s);;
```

Code source OCaml pour Ordinateur (mode simulation) :

Placez le code suivant dans un fichier `simul.ml` :

```
1  open Serial;;
2
3  type t = Clear | Moveto of int * int | Int of int | Text of string
4
5  let (chan : (t, unit) channel) =
6    open_prog "./pic␣'ocapic_lcd_simulator␣16x2␣e=RD0␣rs=RD2␣rw=RD1␣bus=PORTB'"
7  ;;
8
9  let clear () = send chan Clear;;
10 let moveto l c = send chan (Moveto (l, c));;
11 let print_int i = send chan (Int i);;
12 let print_string s = send chan (Text s);;
```

Code source OCaml pour PIC :

Placez le code suivant dans un fichier `pic.ml` en adaptant éventuellement l'argument donné à la fonction `open_channel` (ici 34 pour 19200 bauds) en fonction du débit de votre port série (se référer à la documentation du pic pour la convertion à effectuer) :

```
open Pic;;
open Lcd;;
open Serial;;

set_bit IRCF1;;
set_bit IRCF0;;
set_bit PLLEN;;

module Disp = Connect (
struct
  let bus_size = Four
  let e  = LATD0
  let rs = LATD2
  let rw = LATD1
  let bus = PORTB
end
);;
open Disp;;

init ();;
config ~cursor:Underscore ();;

type t = Clear | Moveto of int * int | Int of int | Text of string

let (chan : (unit, t) channel) = open_channel 34;; (* 19200 bauds *)

while true do
  match receive chan with
    | Clear -> clear ()
    | Moveto (l, c) -> moveto l c
    | Int i -> print_int i
    | Text t -> print_string t
done
```

Configuration du PIC :

Créez le même fichier de configuration `config.asm` que dans la section 3.1.

Compilation :

Remarque : si vous avez installé la librairie OCaPIC ailleurs que dans `/usr/lib`, adaptez le chemin d'accès à votre installation.

Compilez `sender.ml` → `sender.cmo` :

```
ocapic -I /usr/lib/ocapic/extra -c sender.ml
```

Compilez `simul.ml` → `simul.cmo` :

```
ocamlc -I /usr/lib/ocapic/extra -c simul.ml
```

Compilez `pic.ml` → `pic`, `pic.asm` :

```
ocapic 18f4620 config.asm pic.ml
```

Assemblez `pic.asm` → `pic.hex` :

```
gpasm -y pic.asm
```

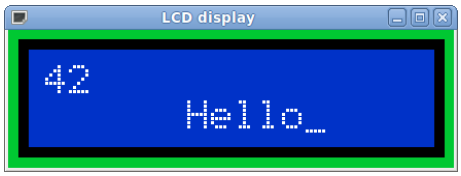
Simulation :

Lancez la commande :

```
ocaml -I +threads -I /usr/lib/ocapic/extra unix.cma threads.cma
serial.cmo simul.cmo
```

Ceci ouvre un top-level OCaml vous permettant d'appeler des fonctions du module `Simul` que vous venez de créer, ainsi qu'une fenêtre représentant l'afficheur LCD connecté au PIC. Vous pouvez alors contrôler l'afficheur LCD en tapant dans le top-level des commandes comme, par exemple :

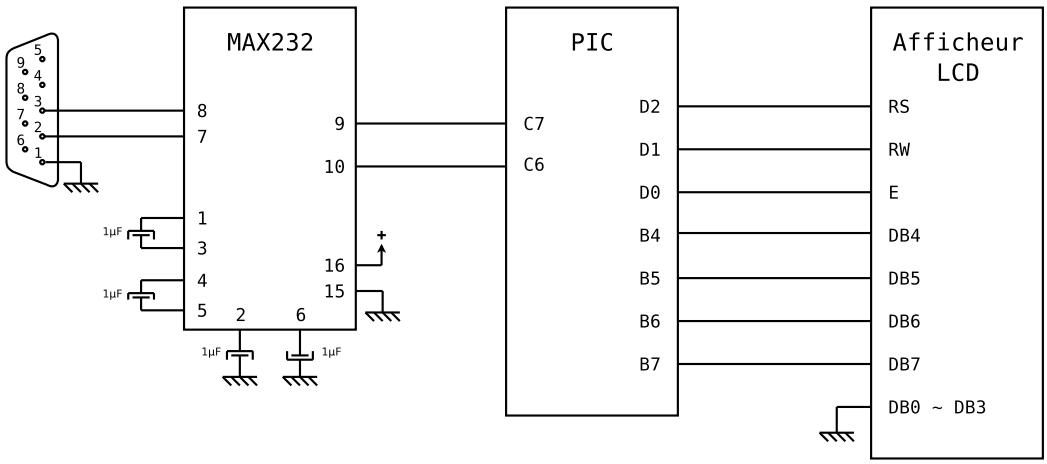
```
# open Simul;;
# print_int 42;;
# moveto 2 6;;
# print_string "Hello";;
```



Pour terminer la simulation, il suffit de tuer le top-level OCaml en tapant `Ctrl-D`.

Circuit :

Les niveaux logiques du PIC (0V/5V) n'étant pas les mêmes que ceux du port série (10V/-10V), utilisez le circuit intégré `MAX232` pour les convertir. La connection à l'afficheur LCD se fait comme dans la section 3.2 :



Exécution :

Après avoir transféré le fichier `pic.hex` sur le PIC, connecté le port série au circuit et alimenté le circuit en 5V, exécutez dans le terminal :

```
ocaml -I +threads -I /usr/lib/ocapic/extra unix.cma threads.cma
serial.cmo sender.cmo
```

Vous obtenez alors un nouveau top-level OCaml grâce auquel vous pouvez contrôler l'afficheur LCD via le module `Sender`. Vous devez obtenir le même comportement que dans le simulateur, sauf que cette fois ci, vous contrôlez le vrai afficheur LCD connecté au PIC !

4 Utilisation avancée

4.1 Paramétrage de la machine virtuelle

Certains programmes peuvent nécessiter un paramétrage de la taille de la pile ou du tas. Par défaut, sur le `PIC18F4620`, la pile fait 174 niveaux et le tas 1792 octets.

L'ajustement de la taille de la pile et du tas se fait au moment du *link* lors de l'appel à `ocapic`. La commande `ocapic` prend deux options `-stack-size` et `-heap-size`. Chacune prend en argument le nombre de segments de 256 octets alloués soit à la pile soit au tas.

Par exemple, l'exécution de :

```
ocapic 18f4620 -stack-size 3 -heap-size 6 config.asm hw.ml
```

définit 430 niveaux de pile et 1536 octets de tas.

Les différentes configurations possibles permettant d'utiliser tous les registres du PIC pour la pile et le tas OCaml sont décrites dans le tableau suivant. Ces valeurs peuvent être réduites en cas d'interfaçage avec du code assembleur nécessitant plus de 15 octets de mémoire, voir la section 4.2 pour plus de détails à ce sujet.

-stack-size	-heap-size	nombre de niveau de pile	taille du tas (en octets)
1	7	174	1792
3	6	430	1536
5	5	686	1280
7	4	942	1024
9	3	1198	768
11	2	1454	512
13	1	1710	256

On peut remarquer que la libération d'un segment pour le tas permet d'en utiliser deux pour la pile. Ceci est dû au gestionnaire mémoire basé sur l'algorithme *Stop & Copy* qui divise la taille utilisable du tas par 2.

4.2 Interfaçage avec du code assembleur

Il est possible d'interfaçer du code OCaml avec du code assembleur PIC. Ce code assembleur peut même avoir été généré par un autre compilateur sour réserve d'avoir un certain contrôle sur les labels générés par le compilateur.

Du côté OCaml, l'interfaçage se fait grâce aux mot clé `external`. Le nom de la *fonction externe* correspond alors au label du code assembleur. Le code assembleur doit être écrit dans un (ou plusieurs) fichier `.asm` et être donné au moment du *link* à la commande `ocapic`.

Voici un petit exemple :

Code source OCaml :

Placez le code suivant dans un fichier `mask.ml` :

```
external mask_portb : int -> int -> unit = "mask_portb";;

Pic.write_reg Pic.TRISB 0x00;; (* PORTB en sortie *)
Pic.write_reg Pic.PORTB 0x00;; (* PORTB <- 0 *)

for i = 1 to 100 do
  mask_portb i 0xF0;    (* Execution du code externe *)
  Sys.sleep 1000;       (* Attend 1s *)
done
```

Code source Assembleur :

Placez le code de la fonction externe `mask_portb` dans un fichier `ext.asm` :

```
mask_portb:
        rrcf    ACCUH, W        ; PORTB <- Long_val(ACCU)
        rrcf    ACCUL, W
        movwf   PORTB
        rrcf    [0x2], W        ; PORTB <- PORTB ^ STACK[0]
        rrcf    [0x1], W
        xorwf   PORTB, F
        clrf    ACCUH           ; return ()
        movlw   0x1
        movwf   ACCUL
        return
```

Code source C :

Pour que le simulateur fonctionne, le code assembleur de la fonction `mask_portb` doit être recodé en C. Placez le code suivant dans un fichier `ext.c` :

```
#include <caml/mlvalues.h>

/* Fonction interne au simulateur */
/* Ecriture dans un registre special du PIC */
value caml_pic_write_reg(value vreg, value vval);

/* Version C de la fonction mask_portb */
value mask_portb(value v1, value v2){
  caml_pic_write_reg(Val_long(0x01), Val_long(Long_val(v1) ^ Long_val (v2)));
  return Val_unit;
}
```

Configuration du PIC :

Créez le même fichier de configuration `config.asm` que dans la section 3.1.

Compilation :

Compilez `mask.ml`, `ext.c`, `ext.asm` → `mask`, `mask.asm` :

```
ocapic 18f4620 config.asm mask.ml ext.c ext.asm
```

Assemblez `mask.asm` → `mask.hex` :

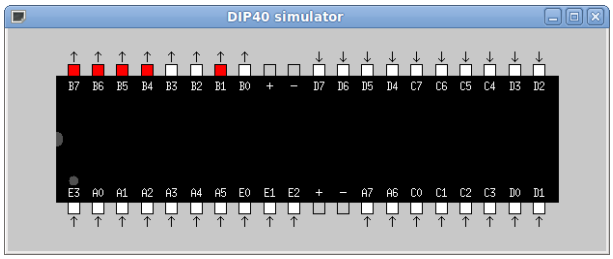
```
gpasm -y mask.asm
```

Simulation :

Exécutez la commande :

```
./mask 'ocapic_dip40_simulator'
```

Vous devez obtenir une fenêtre comme :



4.3 Rattrapage des interruptions

Comme on peut le voir dans la page de manuel du module Pic de la librairie, deux fonctions permettent de manipuler depuis OCaml les interruptions matérielles du PIC :

```
(* Registers the interruption handler. *)
val set_interruption_handler : (bit -> unit) -> unit

(* Removes the interruption handler. *)
val clear_interruption_handler : unit -> unit
```

Le rattrapage des interruptions est similaire au rattrapage des signaux Unix. La fonction `set_interruption_handler` permet d'enregistrer une fonction OCaml de type `bit -> unit`. Ce *handler* est appelé à chaque fois qu'une interruption est levée avec en argument le bit du registre spécial correspondant à l'interruption. Vous pouvez vous référer à la documentation du PIC pour connaître la correspondance interruption/bit. Par exemple, le bit correspondant à une interruption générée par un changement d'état de la patte `B0` est `INT0F`.

Tout comme pour la gestion standard des signaux Unix en OCaml, une interruption ne peut pas être gérée au cours de l'exécution d'une fonction externe. Le temps d'attente avant la gestion d'une interruption n'est donc pas d'un cycle machine (comme c'est presque toujours le cas lorsque l'on code en assembleur pour PIC), mais d'un cycle de la machine virtuelle OCaml. Ceci peut poser des problèmes de réactivité en cas d'appel externe long, comme par exemple l'appel à la fonction `Sys.sleep`.

D'autre part, la gestion d'une interruption peut être interrompue par la gestion d'une autre interruption mais pas par la gestion de la même interruption. Le comportement est alors encore une fois similaire à celui des signaux.

Comme on peut le remarquer, on ne peut enregistrer qu'un seul rattrapeur d'interruption à la fois et cela peut paraître restrictif au premier abord. En réalité, il n'en est rien comme le montre le petit exemple suivant.

Le programme suivant permet de simuler une sorte d'*exécution parallèle*. En effet, d'un côté, un petit compteur s'incrémente régulièrement sur l'afficheur. D'un autre côté, un second compteur est incrémenté à chaque impulsion électrique que l'utilisateur envoie sur la patte `B0`. Vous remarquerez qu'une petite librairie générique de gestion des interruptions y est définie.

Code source OCaml

Placez le code suivant dans un fichier `interrupt.ml` :

```
open Pic;;

(* Acceleration de l'horloge du PIC. *)
set_bit IRCF1;; set_bit IRCF0;; set_bit PLLEN;;

(* Configuration de la connection a l'afficheur LCD. *)
module Disp = Lcd.Connect (
struct
  let bus_size = Lcd.Eight
  let e  = Pic.LATD0
  let rs = Pic.LATD2
  let rw = Pic.LATD1
  let bus = Pic.PORTC
end
);;
open Disp;;

init ();;
config ();;
```

```
20
21 (* Petite librairie generique de gestion des interruptions. *)
22
23 type interruption_behavior =
24   | Interruption_ignore
25   | Interruption_handle of (bit -> unit)
26 ;;
27
28 let handlers = ref [];;
29
30 let the_handler bit =
31   try (List.assq bit !handlers) bit
32   with Not_found -> ()
33 ;;
34
35 let interruption bit ib =
36   let old =
37     try
38       let old = Interruption_handle (List.assq bit !handlers) in
39       handlers := List.remove_assq bit !handlers;
40       old
41     with Not_found -> Interruption_ignore
42   in
43     match ib with
44       | Interruption_ignore -> old
45       | Interruption_handle f ->
46           set_interruption_handler the_handler;
47           handlers := (bit, f) :: !handlers ; old
48 ;;
49
50 let set_interruption bit ib = ignore (interruption bit ib);;
51
52 (* Utilisation de la mini librairie. *)
53
54 (* Handler. *)
55 let my_handler =
56   let counter = ref 0 in
57     fun _ ->
58       let (li, co) = current_position () in
59         moveto 2 0;
60         print_string "␣␣␣␣␣␣␣␣␣";
61         moveto 2 0;
62         Printf.fprintf output "Interruption␣%d" !counter;
63         incr counter;
64         moveto li co;
65         Sys.sleep 100;
66 ;;
67
68 (* Activation de l'interruption pour le PIC. *)
69 set_bit INT0E;;
70 set_bit GIE;;
71
72 (* Enregistrement du handler. *)
73 set_interruption INT0F (Interruption_handle my_handler);;
74
75 (* Incrementation reguliere d'un petit compteur. *)
76 while true do
77   for i = 0 to max_int do
78     clear_bit GIE;          (* Debut de la zone d'exclusion mutuelle *)
79     moveto 1 0;             (* (pour eviter les collisions dans l'affichage). *)
80     Printf.fprintf output "Loop␣%d" i;
81     set_bit GIE;            (* Fin de la zone d'exclusion mutuelle. *)
82     Sys.sleep 100;
83   done
84 done
```

Compilation

Compilez `interrupt.ml` → `interrupt`, `interrupt.asm` :

```
ocapic 18f4620 config.asm interrupt.ml
```

Assemblez `interrupt.asm` → `interrupt.hex` :

```
gpasm -y interrupt.asm
```

Vous obtenez le fichier `interrupt.hex` à transférer sur le PIC. Connectez alors un afficheur LCD au PIC comme d'habitude. Lors de l'exécution, vous pouvez envoyer des impulsions sur la patte `B0` pour incrémenter le second compteur en parallèle avec le premier qui, pour sa part, s'incrémente régulièrement.

Table des matières