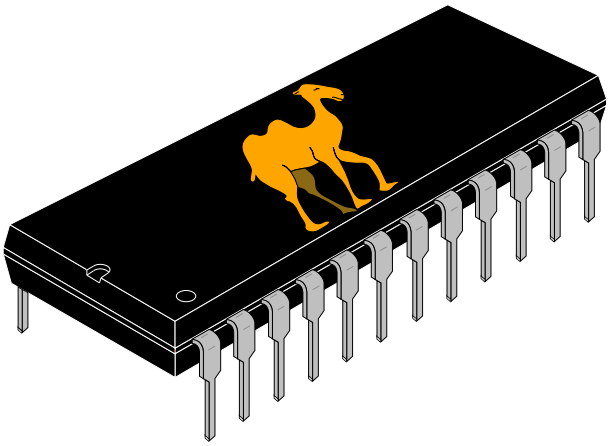
OCaPIC tutorial
Version 1.1

Programming PIC microcontrollers with OCaml

Benoît Vaugon



August 26, 2011

English translation: Zoé Faget

1 Introduction

OCaPIC is a set of programs and libraries allowing programming PIC microcontrollers in OCaml. All OCaml features are available on PIC. OCaPIC provides the major parts of the standard OCaml library as well as specific PIC libraries allowing direct interaction with hardware interruption, LCD display, serial connection...

Furthermore, OCaPIC holds a simulation mechanism in which one can improve an OCaml program before transferring it on the PIC.

For further information, please refer to OCaPIC's website

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

2 Installation

2.1 Prerequisites

- Any C-compiler (for example `gcc`).
- The standard OCaml compiler provided by INRIA (`http://ocaml.org/`). On most Linux distribution it is sufficient to install the `ocaml` package.
- A PIC assembler in order to convert OCaPIC generated assemblers files (`.asm`) into hexadecimal code (`.hex`), to be transferred on the PIC. The `gpasm` program which comes with the `gputils` Linux package is a possible option.

2.2 Downloading

From the OCaPIC website: `http://www.algo-prog.info/ocapic/`, *Download* section, download the latest available version. You should get a file named
`ocapic-X.Y.tar.bz2` (replace `X.Y` by the version number).

2.3 Installation

Here is a guideline to install OCaPIC on Linux. The Windows installation is also possible thanks to Cygwin.

In a terminal window (each time replacing `X.Y` by the version number):

- Extract the downloaded file `ocapic-X.Y.tar.bz2`:

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

- Go to the `ocapic-X.Y` directory:

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

- Configure OCaPIC. This operation checks dependencies and allows to configure installation directories and managed PIC. Only `PIC18F4620` is managed by default, because the compilation of the interface with each PIC is very long. For more details please refer to the `INSTALL` file:

  ```
  ./configure
  ```

- Compile OCaPIC :

  ```
  make
  ```

- Install OCaPIC (as super-user) :

  ```
  sudo make install
  ```

  Or:

  ```
  su
  make install
  ```

3 Use examples

All examples featured in this tutorial are adapted to the `PIC18F4620`. Although all PIC of the `18F` serie are supported, we recommend that you start with this one.

3.1 Blink a LED

We begin with a very simple example: how to blink a LED with a PIC. There is of course very little interest in using a high-level language such as OCaml to do so, nor a microcontroller. The purpose of this section is to highlight the different steps of PIC programming in OCaml.

OCaml source code:

Place the following OCaml code in a `led.ml` file:

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

write_reg TRISB 0x00; (* Configure the B port as output *)

while true do
  set_bit RB0;          (* Turn on the LED *)
  Sys.sleep 500;        (* Wait 0,5s *)
  clear_bit RB0;        (* Turn off the LED *)
  Sys.sleep 500;        (* Wait 0,5s *)
done
```

PIC configuration:

It is necessary to specify several of PIC's configuration options (programming voltage, clock selection, external reset configuration, protected code segments selection, ...). To do so, place the following code into a `config.asm` file:

```
        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:

Compile the `led.ml` file for `PIC18F4620` with the following command:

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

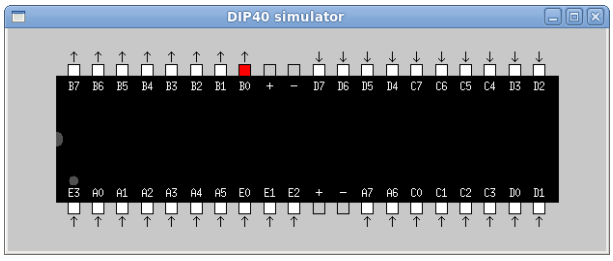
You should get, among others, the `led.asm` files (containing PIC assembler code) and `led` (executable file used for simulation).

Simulation:

Simulate the execution of your program on PIC with the following command:

```
./led ocapic_dip40_simulator
```

You should get a window representing a PIC with a blinking `B0` pin:



Since the program does not end (`while true`), kill it in the terminal window with `Ctrl-C`.

Assembling:

If you're using `gpasm`, assemble the `led.asm` file with:

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

This command generates, among others, the hexadecimal file `led.hex` to be transferred to the microcontroller.

Remark : the `-y` option enables PIC's "extended instructions set". Make sure to enable this option if you use another assembler than `gpasm`.

PIC programming:

You now have a `led.hex` file which can be directly transferred to the PIC through a "PIC programmer". If you are lucky enough to have a computer with serial port, you may use a low cost "serial programmer" (or build one yourself with a few components). Otherwise, use a USB programmer. The "USBPICPROG" project (`http://usbpicprog.org/`) helps getting and making one at a reasonable price.

Circuit:

Connect a LED between the PIC's `B0` pin and the – of the power supply (through a resistance, of course !), supply the PIC with 5V and you're done !

3.2 Hello world on an LCD display

OCaPIC provides a library named `Lcd` which allows to operate on small LCD displays with a PIC. For complete documentation, please refer to the corresponding manpage. Here is a short example:

OCaml source code :

Place the following OCaml code in a `hw.ml` file:

```
open Pic;;

module Disp = Lcd.Connect (
struct                          (* LCD/PIC connection configuration: *)
  let bus_size = Lcd.Four          (* Bus size *)
  let e  = LATD0                   (* Validation pin: D0 *)
  let rs = LATD2                   (* Instruction/data pin: D2 *)
  let rw = LATD1                   (* Read/write pin: D1 *)
  let bus = PORTB                  (* Bus: port B *)
end
);;

Disp.init ();;                     (* Display initialization *)
Disp.config ();;                   (* Display configuration *)
Disp.print_string "Hello world";; (* Display a string *)
```

PIC configuration:

As in section 3.1, place the following code of configuration in a `config.asm` file:

```
        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:

Compile `hw.ml` → `hw`, `hw.asm` :

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

Assemble `hw.asm` → `hw.hex` :

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

Simulation:

Execute the following command:

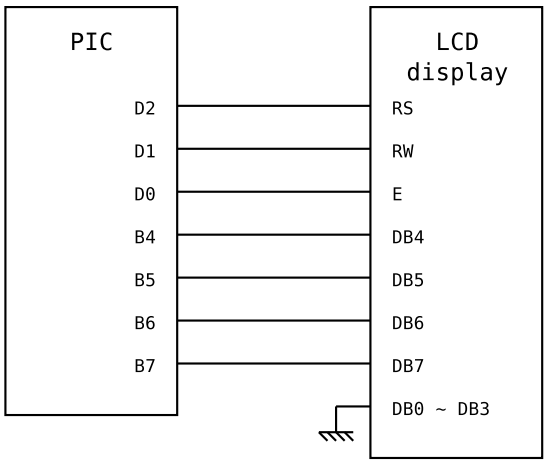
```
./hw 'ocapic_lcd_simulator 16x2 e=RD0 rs=RD2 rw=RD1 bus=PORTB'
```

You should get a window similar to this one:

Since this program does end, to stop the simulation, simply hit `Enter` in the terminal window.

Circuit:

Connect an LCD display to the PIC in the following fashion:



Supply the PIC and the LCD display with 5V.

3.3 A small timer

This short example aims at showing how to use an interesting plug-in for simulation: the "circuit" simulator. By "circuit" we mean the connection of a set of standard components to the PIC (such as switches, push buttons, LED, . . .).

We intend to create here a small *timer* with an LCD display on which runs a time counter, and several push buttons for adjusting time delay and start/stop time-out.

OCaml source code:

Place the following code in a `timer.ml` file:

```
open Pic;;

(* PIC clock speed-up *)
set_bit IRCF1;; set_bit IRCF0;; set_bit PLLEN;;

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

(* Display initialization and configuration *)
Disp.init ();; Disp.config ();;

(* Counter (reference to an Int32) *)
let counter = ref 0l;;

(* Display utility *)
let print_02d n =
  if n < 10 then Disp.print_char '0';
  Disp.print_int n;
;;

(* Displays counter's value *)
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;
;;

(* Overflow utilities *)
let is_null () = !counter <= 0l and is_full () = !counter >= 3599991;;

(* Increments the counter by n *)
let incr n =
  counter := Int32.add !counter (Int32.of_int n);
  if is_full () then counter := 359999l;
;;

(* Decrements the counter by n *)
let decr n =
  counter := Int32.sub !counter (Int32.of_int n);
  if is_null () then counter := 0l;
;;

(* Increments/Decrements the counter by 1 *)
let decr1 () = if not (is_null ()) then counter := Int32.pred !counter;;
let incr1 () = if not (is_full ()) then counter := Int32.succ !counter;;

```

```
(* Speed-up functions for time setting *)
let step n = n / 32 * n / 16 * n + n / 16 * n / 16 + n / 8 + 1;;

(* Small states machine *)
let rec run n =
  if is_null () || test_bit RB0 then stop ()
  else if test_bit RB1 then ( incr_start () ; run 0 )
  else if test_bit RB2 then ( decr_start () ; run 0 )
  else if n = 79 then ( Sys.sleep 7 ; decr1 () ; print () ; run 0 )
  else ( Sys.sleep 10 ; run (succ n) )
and pause () =
  Sys.sleep 10;
  if test_bit RB0 && not (is_null ()) then start ()
  else if test_bit RB1 then ( incr_start () ; pause () )
  else if test_bit RB2 then ( decr_start () ; pause () )
  else pause ();
and start () =
  set_bit RB3;
  Sys.sleep 10;
   if test_bit RB0 then start () else run 0;
and stop () =
  clear_bit RB3;
  Sys.sleep 10;
  if test_bit RB0 then stop () else pause ();
and incr_start () =
  incr1 ();
  print ();
  Sys.sleep 300;
  incr_loop 1;
and decr_start () =
  decr1 ();
  print ();
  Sys.sleep 300;
  decr_loop 1;
and incr_loop n =
  if test_bit RB1 then (
    incr (step n);
    print ();
    Sys.sleep 100;
    if not (is_full ()) then incr_loop (succ n);
  )
and decr_loop n =
  if test_bit RB2 then (
    decr (step n);
    print ();
    Sys.sleep 100;
    if not (is_null ()) then decr_loop (succ n);
  )
in
  write_reg TRISB 0b00000111; (* Configure button pins as input *)
  print ();                   (* Display 00:00:00 *)
  stop ();                    (* Entry point of the state machine *)
```

PIC configuration:

Create the same configuration file `config.asm` as in section 3.1.

Virtual circuit:

The plug-in which enables circuit simulation takes as parameter a file name containing the circuit description in a rather intuitive format. Place the following definitions in a `circuit.txt` file:

```
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:

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

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

Assemble `timer.asm` → `timer.hex` :

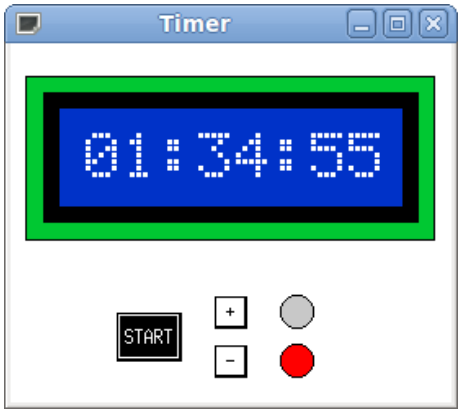
```
gpasm -y timer.asm
```

Simulation:

Execute:

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

You should get a window similar to this one:



You can now simulate a user's interaction with the timer by clicking `+`, `-` and `START`. To stop simulation, hit `Ctrl-C` in the terminal window.

3.4 Communication with a computer

Communication between a computer and a PIC can be established thanks to the two modules `Serial` of the library (one for the PIC and one for the computer). They allow the direct exchange of OCaml values between the program running on the computer and the program running on the PIC.

The exchanged OCaml values between the two programs can be of any kind as long as it makes sense. For example those values can be integers, strings or float numbers but also more complex data structures such as lists, arrays, etc. The only forbidden values are those containing code pointers such as closures. It is a true exchange of *memory graph*, values may contain cycles.

Communication is protected from transmission errors, accidental unplugging, etc. Furthermore, memory overflow and collisions in the protocol are correctly managed and translated into exceptions.

If you don't have a serial port on your computer, you may use a USB/Serial converter.

The use of this library is entirely managed by the simulator.

The precise documentation of each function is in manpages.

Here is an example of how to drive an LCD display connected to the PIC from an OCaml top-lever running on a computer:

OCaml source code for the Computer:

Place the following source code in a `sender.ml` file, adapting if needed the file path corresponding to the serial port (here `/dev/ttyUSB0`):

```
open Serial;;

(* Type of values transfered from the computer towards the PIC *)
type t = Clear | Moveto of int * int | Int of int | Text of string

(* Communication channel with the PIC via the serial port *)
let (chan : (t, unit) channel) = open_tty "/dev/ttyUSB0";;

(* Utilities *)
let clear () = send chan Clear;;
let moveto l c = send chan (Moveto (l, c));;
let print_int i = send chan (Int i);;
let print_string s = send chan (Text s);;
```

OCaml source code for the Computer (simulation mode):

Place the following code in a `simul.ml` file:

```
open Serial;;

(* Type of values transfered from the computer towards the simulator *)
type t = Clear | Moveto of int * int | Int of int | Text of string

(* Communication channel with the simulator *)
let (chan : (t, unit) channel) =
  open_prog "./pic␣'ocapic_lcd_simulator␣16x2␣e=RD0␣rs=RD2␣rw=RD1␣bus=PORTB'"
;;

(* Utilities *)
let clear () = send chan Clear;;
let moveto l c = send chan (Moveto (l, c));;
let print_int i = send chan (Int i);;
let print_string s = send chan (Text s);;
```

OCaml source code for PIC :

Place the following code in a `usart.ml` file, adapting if needed the given argument to the `open_channel` function (here 34 for 19200 baud) according to the rate of your serial port (please refer to the pic documentation for the full conversion list).

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

(* PIC clock speed-up *)
set_bit IRCF1;; set_bit IRCF0;; set_bit PLLEN;;

(* PIC/LCD display connection configuration *)
module Disp = Connect (
struct
  let bus_size = Eight
  let e  = LATD0
  let rs = LATD2
  let rw = LATD1
  let bus = PORTB
end
);;

(* Initialization and configuration of the display *)
Disp.init ();; Disp.config ~cursor:Underscore ();;

(* Type of values transfered from the computer towards the PIC *)
 type t = Clear | Moveto of int * int | Int of int | Text of string

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

(* Loop : command reception / transmission to the display *)
while true do
  match receive chan with
    | Clear -> Disp.clear ()
    | Moveto (l, c) -> Disp.moveto l c
    | Int i -> Disp.print_int i
    | Text t -> Disp.print_string t
done
```

PIC configuration :

Create the same configuration file `config.asm` as in section 3.1.

Compilation:

Remark : if you have installed the OCaPIC library somewhere else than in `/usr/lib`, modify the access path accordingly.

Compile `sender.ml` → `sender.cmo` :

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

Compile `simul.ml` → `simul.cmo` :

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

Compile `usart.ml` → `pic`, `usart.asm` :

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

Assemble `usart.asm` → `usart.hex` :

```
gpasm -y usart.asm
```

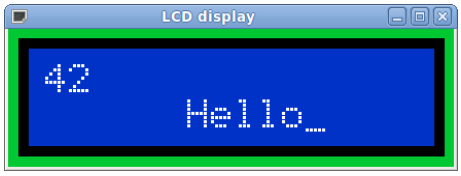
Simulation :

Launch the command:

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

This opens an OCaml top-level allowing you to call functions from the `Simul` module which you've just created, as well as a window representing the LCD display connected to the PIC. You then have control over the LCD display by writing commands in the top-level such as:

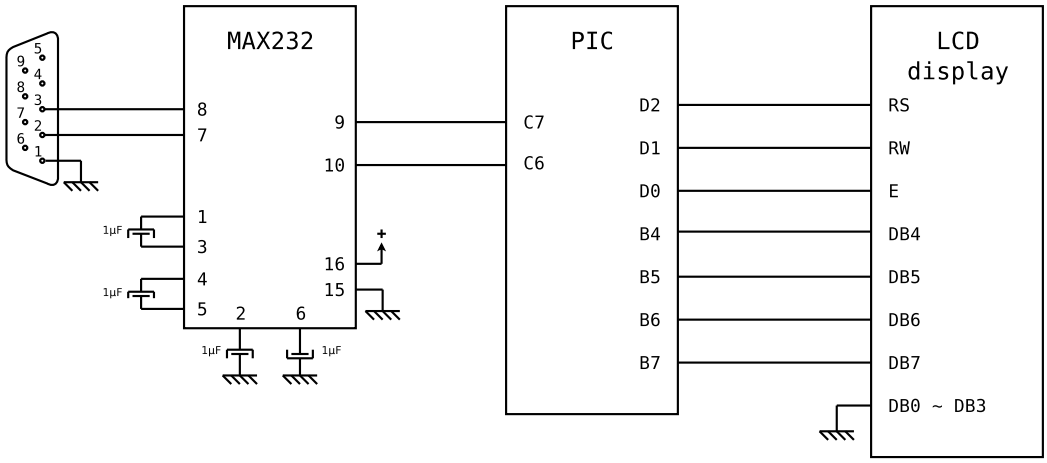
```
# open Simul;;
# print_int 42;;
# moveto 2 6;;
# print_string "Hello";;
```



To end the simulation, kill the OCaml top-level by hiting `Ctrl-D`.

Circuit:

Since the logical levels of the PIC (0V/5V) are not the same as those of the serial port (10V/-10V), use the integrated circuit `MAX232` to make conversions. The connection to the LCD display is done as in section 3.2 :



Run :

After transfering the `usart.hex` file on the PIC, connecting the serial port to the circuit and supplying the circuit with 5V, run in the terminal window:

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

You then get a new OCaml top-level with which you can control the LCD display via the `Sender` module. You should get the same behavior as in the simulator, with the difference that now you are operating the real LCD display connected to the PIC !

4 Advance use

4.1 Virtual machine settings

Several programs may need a setting for the OCaml heap's or stack's size. On the `PIC18F4620`, default settings are 174 levels in the stack and 1792 bytes for the heap.

Adjustments of the size of the heap and stack takes place during the *link* step when calling `ocapic`. The `ocapic` command has two options `-stack-size` and `-heap-size`. Each one takes as argument the number of 256 bytes segments allocated either to the heap or the stack.

For example, running:

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

defines 430 levels of stack and 1536 bytes of heap.

In the following table, we describe all possible configurations allowing the use all the PIC registers for the OCaml's stack and heap. Those values may reduce in case of interfacing with assembler code needing more than 15 bytes of memory. For further details, please refer to section 4.2.

-stack-size	-heap-size	number of stack levels	size of heap (in bytes)
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

One can note that freeing a heap segment allows to use two for the stack. This is due to the virtual memory manager based on the *Stop & Copy* algorithm, which divides by 2 the heap's usable size.

4.2 Binding with assembler code

It is possible to interface OCaml code with PIC assembler code. This assembler code may even have been generated by another compiler, provided we have some control over the labels generated by the compiler.

On the OCaml size, interfacing is done with the keyword `external`. The *e*xternal function's name is then the assembler code's label. The assembler code should be written in one (or several) `.asm` files and given to the `ocapic` command at *link* step.

A quick example:

OCaml source code:

Place the following code in a `mask.ml` file:

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

Pic.write_reg Pic.TRISB 0x00;; (* PORTB as output*)
Pic.write_reg Pic.PORTB 0x00;; (* PORTB <- 0 *)

for i = 1 to 100 do
  mask_portb i 0xF0;    (* Execute the external code *)
  Sys.sleep 1000;       (* Waits 1s *)
done
```

Assembler source code:

Place the following external function code `mask_portb` in a `ext.asm` file :

```
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
```

C source code:

In order for the simulator to work, the `mask_portb` function's assembler code must be recoded in C. Place the following code in a `ext.c` file:

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

/* Simulator's internal function */
/* Writing in a special register of the PIC */
value caml_pic_write_reg(value vreg, value vval);

/* C Version of the mask_portb function */
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;
}
```

PIC configuration:

Create the same configuration file `config.asm` as in section 3.1.

Compilation:

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

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

Assemble `mask.asm` → `mask.hex` :

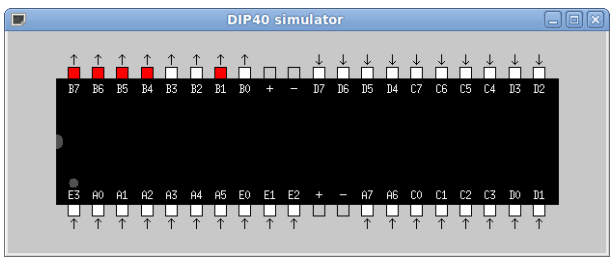
```
gpasm -y mask.asm
```

Simulation:

Run the following command:

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

You should get a window looking like this:



4.3 Interruption catching

As we can see in the Pic module's manpage of the standard library, two functions allow to operate on the PIC's hardware interruptions from OCaml :

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

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

Catching up those interruptions is similar to catching up Unix signals. The function `set_interruption_handler` allows to register an OCaml function of type `bit -> unit`. This *handler* is called each time an interruption occured with as argument the special register's bit corresponding to the interruption. You can refer to the PIC's documentation to know the correspondance between interruptions and bits. For example, the bit matching the interruption generated by a state change for the `B0` pin is `INT0F`.

As for the standard management of Unix signals in OCaml, an interruption can not be managed while an external function is running. Therefore, the wait before an interruption is managed is not equal to a machine cycle (as it is in most cases when coding in assembler for PIC), but is equal to an OCaml virtual machine cycle. This may cause reactivity problems in case of long external calls, such as a call to the `Sys.sleep` function.

Moreover, an interruption management can be interrupted by another interruption management, but not by the management of the same interruption. The behavior is once again similar to those of signals.

We notice that we can register only one interruption catcher at a time, which may first appear as a restriction. In fact it is not, as it is shown in the following example.

The following program simulate a kind of *p*arallel run. Indeed, on one hand, a small counter increments itself regularly on the display. On the other hand, a second counter is incremented at each electric impulse sent by the user on the `B0` pin. Note that a short generic library of interruption management is defined.

OCaml source code

Place the following code in a `interrupt.ml` file:

```
open Pic;;

(* Speed-up of the PIC clock *)
set_bit IRCF1;; set_bit IRCF0;; set_bit PLLEN;;

(* Configuration of the connection to the LCD display *)
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 ();;

(* Short generic library for the management of interruptions *)

type interruption_behavior =
  | Interruption_ignore
  | Interruption_handle of (bit -> unit)
;;

let handlers = ref [];;

let the_handler bit =
  try (List.assq bit !handlers) bit
  with Not_found -> ()
;;

let interruption bit ib =
```

```
  let old =
    try
      let old = Interruption_handle (List.assq bit !handlers) in
      handlers := List.remove_assq bit !handlers;
      old
    with Not_found -> Interruption_ignore
  in
    match ib with
      | Interruption_ignore -> old
      | Interruption_handle f ->
          set_interruption_handler the_handler;
          handlers := (bit, f) :: !handlers ; old
;;

let set_interruption bit ib = ignore (interruption bit ib);;

(* Using the short library *)

(* A specific handler *)
let my_handler =
  let counter = ref 0 in
    fun _ ->
      let (li, co) = current_position () in
        moveto 2 0;
        print_string "␣␣␣␣␣␣␣␣␣";
        moveto 2 0;
        Printf.fprintf output "Interruption␣%d" !counter;
        incr counter;
        moveto li co;
        Sys.sleep 100;
;;

(* Activation of the INT0 interruption for the PIC *)
set_bit INT0E;;
set_bit GIE;;

(* Registration of the handler *)
set_interruption INT0F (Interruption_handle my_handler);;

(* Regular increment of a counter *)
while true do
  for i = 0 to max_int do
    clear_bit GIE;          (* Beginning of a mutual exclusion area *)
    moveto 1 0;             (* (to avoid display collision) *)
    Printf.fprintf output "Loop␣%d" i;
    set_bit GIE;            (* End of the mutual exclusion area *)
    Sys.sleep 100;
  done
done
```

Compilation

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

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

Assemble `interrupt.asm` → `interrupt.hex` :

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

You should get the `interrupt.hex` file to transfer on the PIC. Connect an LCD display to the PIC as usual. While the program is running, you can send impulses to the `B0` pin to increment the second counter, in parallel to the first one which increments regularly.

Contents