Arduino Lab 4 - 2020 - Home startup

Pre-installation (Arduino stuff)

Figure 1: Arduino Uno Platform

- You may have to install the following (Linux) packages: `arduino` , `gcc-avr` (or `avr-gcc`) and `avrdude` (`avr-binutils` et `avr-libc` if they are not included)
- `avrdude.conf` must be in `/usr/share/arduino/hardware/tools/` (if not you have to modify Makefiles).
- The user should have the right to write on the USB port : `usermod -aG dialout <username>` (and re-login).

Some Arduino general information

Arduino/Genuino cards are free cards build around a Atmel AVR microntroler. In this lab we will use Arduino UNOs (atmega328p) like in the following picture. The platform has a few numerical and analogic I/Os where we will connect LEDs, buttons, seven segment led displays...

The microcontroler itself is programmed with a bootloader so that a dedicated system is not necessary. The Makefile we give you will use `avrdude` doc to load the binaries into the microcontroler memory.

You will be given a whole platform with an arduino UNO, some leds, a breadboard,

wires ... **You will be responsible for them for the duration of the lab** On the breadboards, all points in a given supply line (blue/black, red) are connected. Same for the columns.

Getting started with the breadboard (raw C stuff).

First of all, make a simple circuit to test the board itself:

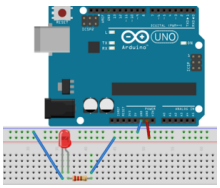


Figure 2: Simple led

The board should be powered : blue/black line to arduino ground (GND) and red line to arduino +5V. Plug the arduino to the USB port of your laptop, the led should shine.

Now, let us program a blinking LED.

From now, shematics implicitely contain the wire required to power the board (link from red lines to 5V, link from blue/black lines to GND)

LED on Digital 13

- Led on digital 13 with 220 ohm resistor. (long leg to digital 13)

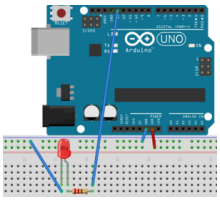


Figure 3: Arduino Uno + Led on Port Digital 13

- TODO : Test the low level code in `_code/blinkingled` (make, make upload)

Understand Arduino Programming style

Here is the pattern we will use to program the Arduino :

```
#include <avr/io.h>
#include <util/delay.h>
```

```
void setup(void) {}

int main(void)
{
  setup();
  while(1)
  {
    // Business code goes here
    _delay_ms(1000);
  }
}
```

The program to embed on the board implements an infinite loop, where the main behavior will be called. The `setup` procedure is used to initialize the board, //e.g.//, specify which pin is used as input or output.

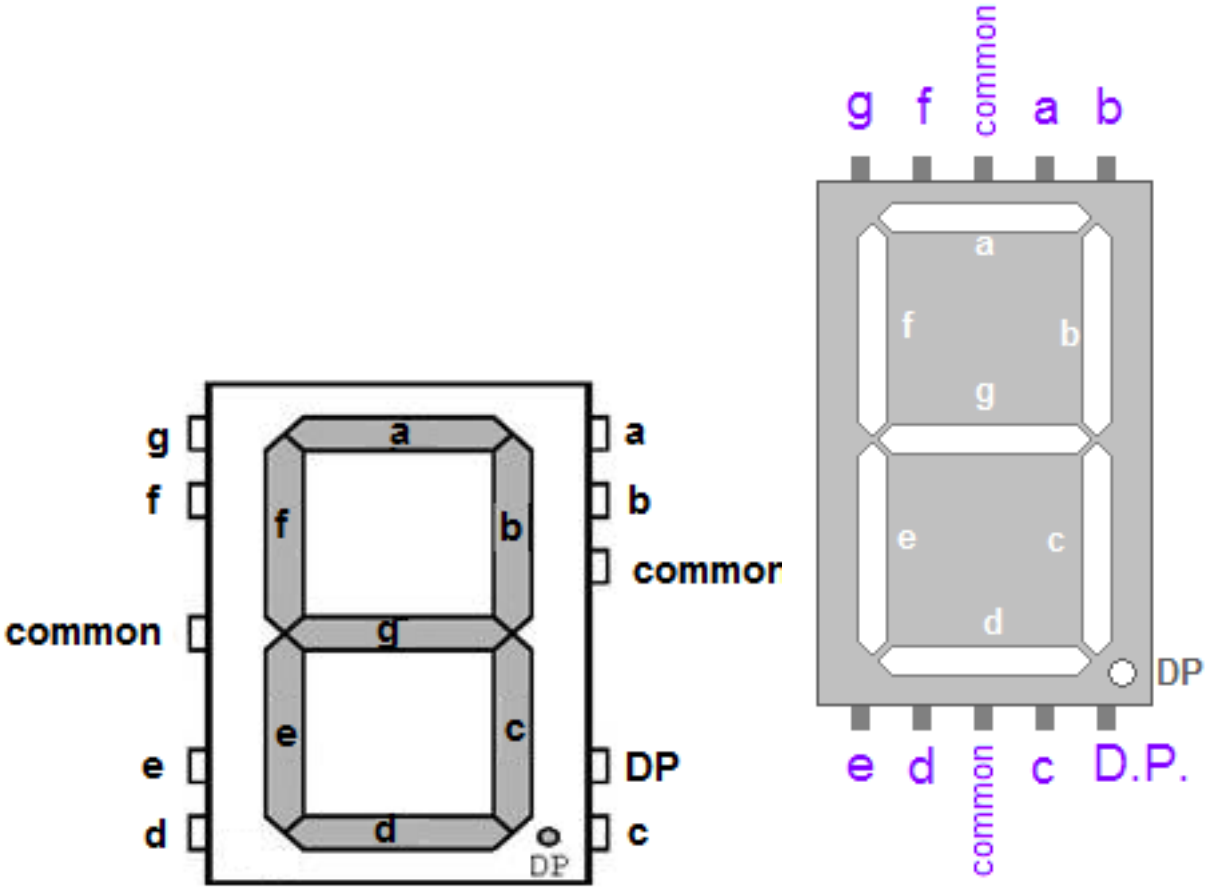
To switch a led on or off on a Arduino, we need to * configure the port/pin where the led is connected into *reading mode* (in the setup procedure) * write a 0 (off) or a 1 (on) into the same port when required.

Some information: * As the led is linked to digital pin 13, the port to be manipulated is DDRB, here every single pin from 8 to 13 is set to "input" (0) excepting bit 5 which gives an "output" access to the led (1).

- To make the led blink, use the xor operator to toggle the 5th bit from 0 to 1 or 1 to 0 each time we enter a different loop: 'PORTB ^= 0bxxxxxxxx;' (replace x with bits values).

7 segments display . (Common Anode OR Common Cathode)

- 7 segments labels:



- 7 segment : a on digital 1 via resistor 220 ohm, b on digital 2, ... g on digital 7, according to the preceding numbering.

Warning, the 7 segment display should be connected though resistors (from 220 to 400 ohm) – NEW PICTURE

- Plug your 7 segment as if it was Common Anode (most probably it is the case) like in the following picture:

(comon anode can be horizontal or vertical, be careful).

- Test your wiring with the test file in `_code/test7segV2` (you have a Readme file) Understand this code. From now, you should know if you have a CC or CA 7-segment display.

Additional documentation (please have a look)

Other links: * port manipulation. Warning, the documentation is for the Arduino Lib format. In raw C, you should use `0b11111110` (rather than

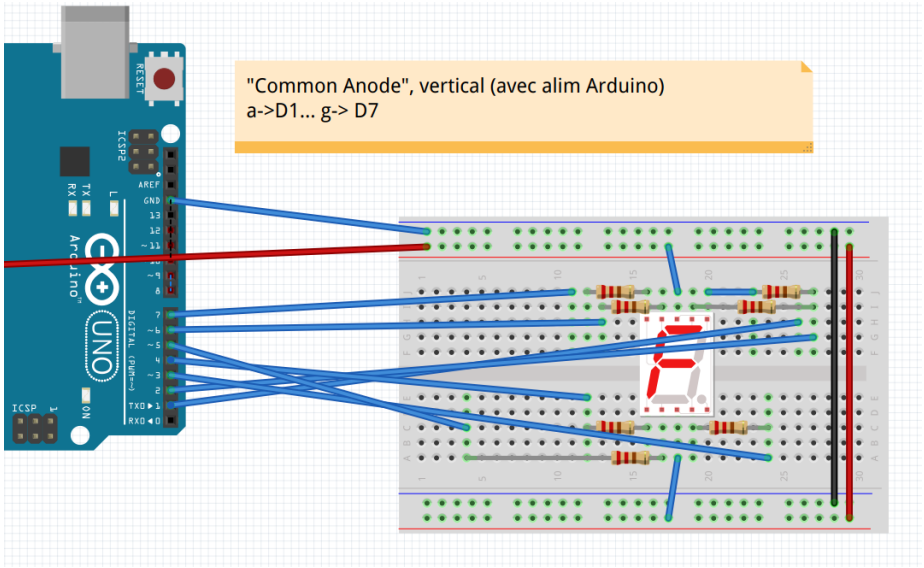


Figure 4: Arduino Uno + 7 seg Common Anode Vertical

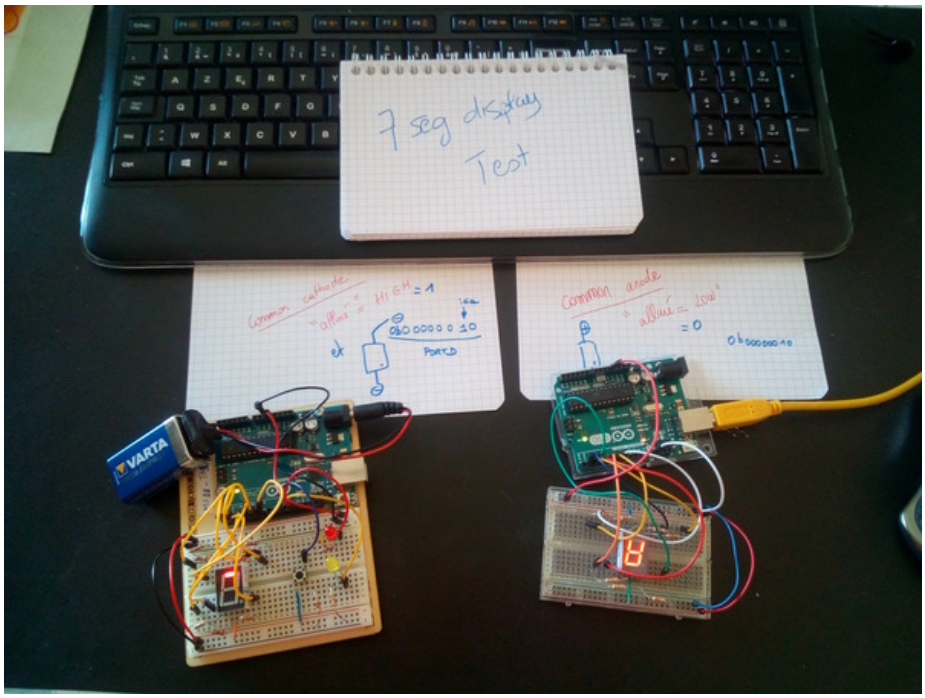


Figure 5: 7seg test with test7segv2

`B11111110`). * Boolean operators, * AVR libc doc for delays. * Common error `avrdude:stk500_recv(): programmer is not responding` may be solved by removing the wire connected to digital 1 before uploading.

MIF18-Lab 4: Arduino 101

- Laure Gonnord, Université Lyon 1, LIP email

- Version: 2020.01
- Inspired by a lab with Sebastien Mosser, UQAM, Montreal.
- Other collaborators: Lionel Morel (Insa Lyon), Julien Forget (Lille).
- Deadline : Monday, April 20th, 6pm strict, on Tomuss.
- General Instructions for MIF18
- Special instructions for this Lab : deliver code from step1 to step4, and your report with your answers to questions (in French or English)

Problem Description

In this lab you will be asked to write simple Arduino programs that interact with sensors/actuators. The objective is double: * Manipulate the platform and the compilation chain * Experiment two variants of Arduino programming and compare their pros and cons.

Prerequisite : Test your arduino setting follow this link

All given code is in this code directory

Step 1: Led on Digital 13 + Button on Digital 10 (ex1/)

- Button on digital 10, and 220 ohm resistor

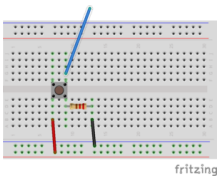


Figure 1: Button on Port Digital 10

- Write a `int get_button_state()` function that reads on digital 10 (use `PINB` value and some boolean operators). Use it in the `main` to control the led (switch it on to off or off to on if the button is pressed).

Step 2: Two LEDs with different frequencies (ex2/)

Now make two leds blink at different frequencies (the second one uses PIN 12). Make your code as generic as possible.

In the README, explain your solution for different values of the frequencies. Explain why is it satisfactory or not ? What would you like as a developper?

Step 3: Led, Button, 7 segment V1 (ex3/)

Here is the new component assembly:

- The seven segment displays can be "common cathode" or "common anode" (common anode for most of us, but please test with the code provided in `_code/test7seg`). See the startup before making this exercice.

The objective is to build the following behavior: 1. Switch on or off a LED based on a button sensor; 2. Building a simple counter using a 7-segment display; 3. Compose the two behaviors so that the button control both the display and the LED.

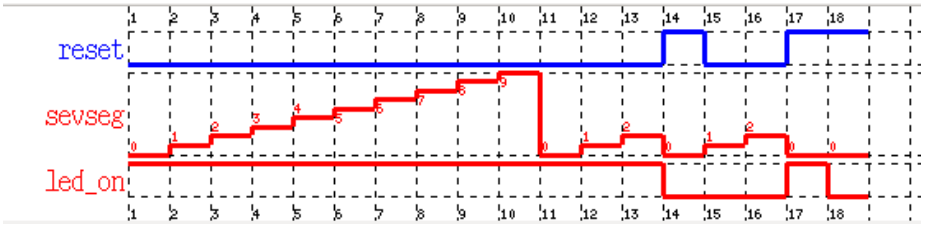


Figure 2: Expected behavior

Expected work

- We give you a starting code.
- Add functionality for the button (see `ex1`). Do not forget to update the `setup` function if required. Test.
- Add functionality for the seven-segment display: write a `void display_7seg(int value)` function to display a given number. As an example, our version begins with:

```
void display_7seg(int value){
  switch (value) {
  case 0: //a,b,c,d,e,f
    PORTD = 0b01111110;
    break;
    //todo: implement the rest!
}
```

And use it to increment the 7 segment value each time you enter the loop. If the button is pressed, the 7-segment should reset to 0. Test.

Questions (TBD in README)

- What can we say about readability of this code? What are the skills required to write such a program?

- Regarding embedded systems, how could you characterize the expressivity (can all applications be written in that way) ? The configurability of the code to change pins or behavior? Its debugging capabilities?
- Regarding the performance of the output code, what kind of parallelism is expressed by the use of the DDRx registers?
- What if we add additional tasks in the micro-controller code, with the same frequency? With a different frequency?

Step 4: Led, Button, 7 segment V2 (ex4/)

A little journey into the Arduino library.

The LED example

- Include `Arduino.h` and link with the lib (the Makefile does this job):

```
#include <avr/io.h>
#include <util/delay.h>
#include <Arduino.h>
```

- Now each pin has his own configuration and can be set independently of the others: `pinMode(led, OUTPUT);` for the led setup and `digitalWrite(led, LOW);`, `digitalWrite(led, HIGH);` to set the led value.
- We have to store the led state in the `led_on` variable.

Documentation & Bibliography

- The Arduino C++ library reference. See the `pinMode` and `digitalRead` documentation.

Expected Work

- Implement the Button functionality. Test it!
- Implement the Seven Segment display functionnality: first implement `displayDigit`:

```
void displayDigit(int digit)
{
  turnOff();
  //Conditions for displaying segment a
  if(digit!=1 && digit != 4)
    digitalWrite(a,HIGH);
    //continue
    }
```

and use it in the main. Test it!

Feedback Questions

- Is the readability problem solved?
- What kind of parallelism can still be expressed? (task parallelism, instruction parallelism, hardware parallelism) ?
- Who is the public targeted by this "language"? It it ok for (real-time) system programmers ?
- Is this language extensible enough to support new features?
- What is the price for the developer?

Arduino Lab 6 - 2020 - Home startup

Pre-installation (Arduino stuff)

- Schéma du montage : 2 leds avec résistance 220ohm~: rouge sur analog 0, jaune sur analog1, et un écran LCD sur Digital 5 à 9 comme sur le dessin ci-dessous.

Les deux LEDs sont câblées comme précédemment. * Le LCD est relié aux ports "digital' ' suivant lcd(4)-> D8, lcd 6-> D9, lcd11 -> D4, lcd 12 -> D5, lcd13 -> D6, lcd14 -> D7. * Le reste du branchement sert à assurer l'alimentation, la stabilité, et le contraste.

Test de l'écran et du montage

- C'est dans `_code/testLCD`.
- Test: Après upload du .hex, l'écran LCD doit afficher "Hello" et les deux LED sont allumées.

MIF18-Lab 6: Round Robin Scheduler on Arduino.

- Laure Gonnord, Université Lyon 1, LIP email

- Version: 2020.04 Last modif 10/5.
- Inspired by a lab with Julien Forget and Thomas Vantroys, Lille
- Other collaborators: Thierry Excoffier (Lyon1), Grégoire Pichon.
- May, 15th, 2020, 3 hours, remotely (discord)

TP delivery

- cf les instructions
- Give a tgz after make clean (no binary file)
- One directory per version for the scheduler, please.
- No not forget the report!
- Deadline on Monday 18/5, 6pm SHARP on TOMUSS

LAB STARTUP ! TODO at home before !

- Do the startup here : startup

Problem Description

In this lab you will implement a simple scheduler for Arduino with timers and interruptions.

The kick-off code of this lab is available under the code directory

Step 1 : 2 leds in parallel, with timers

Dans cet exercice, au lieu d'utiliser l'attente active (wait) en bidouillant pour réaliser le parallélisme, on se propose d'utiliser l'un des timers du micro-contrôleur pour faire clignoter les deux LEDs à deux fréquences différentes, la LED rouge clignotera toutes les 200ms et la LED jaune toutes les 400 ms en parallèle. Le montage est le suivant: led rouge sur analog 0, led jaune sur analog 1.

Afin de faire clignoter la LED jaune toutes les 400 ms en parallèle de la LED rouge, nous allons utiliser un timer~: * Le main continue d'allumer la LED rouge comme dans le TP précédent. * Toutes les 400 ms, une fonction d'interruption sera appelée. Elle réalisera le changement d'état de la LED jaune, puis rendra la main.

Documentation

Explication des timers/interruptions

Expected work

Dans le fichier `main.c` du répertoire `ex1`: * Écrivez le code de la fonction `task_led_red` qui fait clignoter la LED rouge toutes les 200 ms (i.e., la LED est allumée pendant 200 ms, puis éteinte pendant 200 ms). Cette fonction est appelée dans la boucle de la fonction `main`. Testez le bon fonctionnement. * Déterminez la valeur du nombre NB_TICK pour initialiser le registre OCR1A, afin que la fonction d'interruption soit appelée toutes les 400 ms. * Remplissez la fonction `ISR` pour réaliser le changement d'état de la LED jaune.

Grâce à l'utilisation du timer, vous avez pu réaliser ainsi deux tâches qui s'exécutent en parallèle.

Step 2: Scheduler

Vous allez maintenant réaliser un véritable ordonnanceur utilisant un algorithme *Round Robin* avec un intervalle de temps de 40 ms. Votre micro-contrôleur réalisera trois tâches : * faire clignoter la LED rouge toutes les 300 ms. * envoyer en boucle un message sur le port série (chaque envoi de caractère sera espacé de 100 ms). * envoyer en boucle un message sur un mini écran LCD (avec une API fournie).

Circuit

On a toujours les deux LEDs (rouge sur Analog 0, jaune sur Analog 1), et on ajoute un écran LCD comme sur le dessin ci dessous:

Les deux LEDs sont câblées comme précédemment. * Le LCD est relié aux ports "digital' ' suivant lcd(4)-> D8, lcd 6-> D9, lcd11 -> D4, lcd 12 -> D5, lcd13 -> D6, lcd14 -> D7. * Le reste du branchement sert à assurer l'alimentation, la stabilité, et le contraste.

Écriture série, stty et screen.

L'écriture port série s'effectue en fait sur `/dev/ttyACM0`. Pour visualiser ce qui est écrit sur le port série, nous utilisons `stty` pour configurer la lecture (voir le Makefile), et le logiciel `screen`(voir https://www.gnu.org/software/screen/), dans un terminal différent de celui utilisé pour compiler/uploader:

```
screen /dev/ttyACM0 9600
```

Pour quitter proprement screen, on fera `CTRL-a`, puis `k` (kill).

Alternativement vous pouvez directement faire :

```
make && make upload && (stty 9600 ; cat) </dev/ttyACM0
```

Travail préliminaire

Ici chaque tâche effectuera une boucle infinie (et les `delay_ms` aussi). Le code fourni se trouve dans le répertoire `scheduler`: * Testez la tâche LED rouge (de période 300 ms). * À l'aide de la fonction `send_serial` fournie (qui envoie un caractère sur le port série), écrivez une tâche qui écrit le message de votre choix sur le port série en envoyant caractère par caractère toutes les 100 ms. Testez cette tâche. * Écrivez le code de la tâche écrivant une chaîne sur l'écran LCD toutes les 400 ms. Vous utiliserez : * `lcd_write_string_4d(s)` pour imprimer une chaîne. Attention cette tâche elle même prend du temps (allez regarder le code de la librairie). * `lcd_write_instruction_4d(lcd_Clear)` pour effacer le texte courant. * Testez individuellement chacune de ces tâches.

Ordonnanceur RR sans sauvegarde de contexte (V1, 30% de la note)

Maintenant que vos différentes tâches fonctionnent, vous pouvez réaliser l'ordonnanceur. Vous devez commencer par créer une structure (tableau de structs) représentant les différentes tâches (ici 3). Chaque structure de tâche sauvegardera ici son état (`RUNNING`, `NOT_STARTED`): `C typedef struct task_s { volatile uint8_t state; // RUNNING ou NOT_STARTED void (*start)(void); //code for the task } task_t;` L'ordonnanceur suivra donc l'algorithme suivant :

```
currentask <- nexttask()
if currenttask.state == RUNNING //la tâche a déjà été interrompue
   then
        currenttask.relaunch(); // en vrai, utilise la fonction start
   else  // premier lancement de la tâche
        currenttask.state = RUNNING ;
        sei();                 //permettre les interruptions.
        currenttask.launch();
endif
```

Codez cet ordonnanceur (répertoire scheduler/) * Vous utiliserez toujours un timer qui lancera une interruption régulièrement. * La période de l'ordonnanceur sera de 40 ms. Pour faciliter le debug, chaque fois que l'ordonnanceur s'exécute, vous allumerez la LED jaune. * Ajout 20/03: évidemment, comme il y a une interruption toutes les 40ms, les périodes des tâches n'ont plus vraiment de sens, on ne demande pas ici de modifier ce comportement. * Mettez en évidence le problème de non-restoration de contexte : les tâches reprennent toujours au début et * Après avoir testé, sauvez cette v1 dans un répertoire à part `scheduler/V1/` * (dans V1, il y aura un code complet qui compile, commenté, avec un readme, et quelques lignes d'explication des fonctionnalités et ce que vous observez.

Ordonnanceur avec sauvegarde de contexte (V2, 40% de la note)

Pour pouvoir reprendre l'exécution d'une tâche `au bon endroit'' après interruption, il faut sauvegarder`le contexte'', ie l'ensemble des valeurs

des registres à une certaine adresse dans la pile (SP) au moment de son interruption. On modifiera la structure de tâche en:

```
typedef struct task_s {
        volatile uint16_t stack_pointer; // variable pour stocker SP.
        volatile uint8_t state; // toujours RUNNING ou NOT_STARTED
        // here you can add the associated task
} task_t;
```

Pour une explication du volatile clic

Un point sur le changement de contexte Comme on l'a déjà vu en architecture, pour réaliser le changement de contexte, il est nécessaire de sauvegarder la valeur des différents registres du micro-contrôleur (Lire aussi multitasking)

Pour cela, on vous fournit deux macros `SAVE_CONTEXT` et `RESTORE_CONTEXT`. Pour la gestion de la mémoire des différents processus, nous allons découper l'espace mémoire. Dans un AVR atmega328p, la mémoire est utilisée en commençant par les adresses les plus élevées. La tâche "écriture série" débutera son utilisation mémoire à l'adresse 0x0700, la tâche "écriture écran LCD" débutera à l'adresse 0x0600 et la tâche "LED" débutera à l'adresse 0x0500. Ces adresses représentent le départ du pointeur de pile (Stack Pointer). Ce dernier est rendu directement accessible par `gcc-avr` via la "variable" SP.

L'ordonnanceur réalisera donc l'algorithme suivant :

```
SAVE_CONTEXT();       // sur la pile
currenttask.saveSP(); // sauver le SP dans la tâche
sei();          // permettre les interruptions
currenttask <- nexttask()
SP = currenttask.getSP();
si currenttask.state == RUNNING //la tâche a déjà été interrompue
   alors
        RESTORE_CONTEXT()
   sinon  // premier lancement de la tâche
        currenttask.state = RUNNING ;
        currenttask.launch();
finsi
```

TODO : Dans le main:

- Observez les deux macros de stockage du contexte. Attention, dans ces macros, on appelle la fonction `cli` pour stopper les interruptions. La conséquence est qu'il faudra relancer `sei` à chaque reprise en main de l'ordonnanceur.
- Modifier l'ordonnanceur Round Robin précédent (avec un intervalle de 40 ms) et testez-le.

- Sauvegardez cette version dans `scheduler/V2/`

Ordonnanceur avec ressource partagée. (30% de la note)

On souhaite maintenant rajouter une tâche supplémentaire qui envoie en boucle le caractère `'@'` sur le port série. Ceci nécessite donc de gérer l'accès concurrent au port série entre deux tâches.

Vous allez mettre en place un mécanisme simplifié de sémaphores permettant d'assurer l'accès en exclusion mutuelle au port série.

Voici une solution possible: chaque tâche a désormais trois statuts possibles: `CREATED` (la tâche n'a encore jamais été exécutée), `ACTIVE` (la tâche est disponible) et `SUSPENDED` (la tâche attend l'accès à une ressource partagée).

TODO

- Faites une étude sérieuse du problème dans un README. Si vous manquez de temps, je préfère que cette étude soit faite, avec les primitives suivantes en pseudo code plutôt qu'une implementation buguée. (le tiers des points de la partie sera donné sur cette étude. Des dessins scannés peuvent aider.
- Écrivez une primitive `take_serial` (similaire au P/Wait des sémaphores), qui vérifie si le port série est disponible, si oui le prend, si non la tâche l'exécutant est suspendue.
- Écrivez une primitive `release_serial` (similaire au V/Signal des sémaphores), qui rend le port série et, si besoin, rend active la tâche suspendue.
- Modifiez le code des tâches et de l'ordonnanceur en conséquence. Notez bien que les primitives `take_serial` et `release_serial` doivent être ininterruptibles !
- Commentez correctement votre code, ajoutez dans le README le statut de votre implementation. Sauvegardez cette version dans `scheduler/V3/`

Arduino et timers

L'initialisation du timer est fournie.

À chaque cycle de l'horloge, un compteur est incrémenté et il est comparé à la valeur se trouvant dans un registre (registre OCR1A dans notre cas). Si ils sont égaux, une interruption est générée. Le programme est alors dérouté pour exécuter la fonction correspondante et le compteur est réinitialisé.

Un exemple http://www.avrbeginners.net/architecture/timers/timers.html

Interruptions en arduino?

Les fonctions d'interruptions, dans le cas d'un micro-contrôleur AVR et de l'utilisation de avr-gcc sont nommées ISR et prennent en paramètre le vecteur d'interruption correspondant (dans notre cas, il s'agit de TIMER1_COMPA_vect). Le timer est initialisé avec un prescaler de 1024, cela signifie que le compteur est incrémenté tous les 1024 cycles. L'arduino est cadencé à 16 MHz.

Pour activer les interruptions vous devez appeler la fonction `sei()`. Pour désactiver les interruptions, la fonction est `cli()`.

Doc : http://www.nongnu.org/avr-libc/user-manual/group___avr___interrupts.html