Émulation d'un circuit comportant un processeur Atmel avec `simavr`

J.-M Friedt, 27 mars 2020

1 Introduction

Il existe de nombreux cas où le matériel n'est pas disponible pour développer un système embarqué, que ce soit parce que la carte commandée n'a pas encore été livrée, parce que le collègue chargé de la conception du circuit imprimé a fait une erreur ou est en retard, ou parce qu'un virus interdit l'accès aux salles de travaux pratiques de l'Université (Fig. 1). Pour toutes ces raisons, nous désirons appréhender le développement d'un système embarqué sur un émulateur, c'est à dire un logiciel capable de fournir une représentation fidèle du comportement du dispositif réel, incluant ses latences et temporisations.

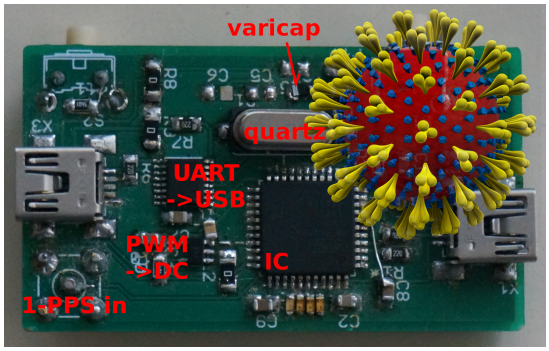

FIGURE 1: Circuit de contrôle d'un oscillateur à quartz sur GPS. Contaminé par un virus, il nous a fallu recourir à une solution d'émulation pour pallier l'absence de matériel pour achever le développement logiciel.

Nous avions proposé [1] l'utilisation d'émulateurs pour appréhender le développement du code embarqué (*firmware*) sur divers microcontrôleurs, de la petite architecture 8 bits de l'Atmega de Atmel (maintenant Microchip) aux gros ARM, RISC-V et MIPS capables d'exécuter GNU/Linux. Dans tous ces cas, nous nous étions arrêtés au *firmware* sans considérer l'émulation du circuit imprimé qui entoure le microcontrôleur et lui injecte potentiellement des stimuli, que ce soit sous forme de signaux binaires (GPIO pour *General Purpose Input Output*) ou messages (série sur bus USART ou parallèle sur un port de GPIO). Nous allons combler cette lacune en appréhendant l'émulateur de petits microcontrôleurs de la gamme Atmega `simavr` non plus au niveau du cœur du processeur mais des périphériques qui l'entourent. En particulier, nous constaterons que l'utilisation des interruptions devient naturelle dès qu'il y a interaction du microcontrôleur avec son environnement. Par ailleurs, nous verrons que l'accès au temps émulateur – et non pas au temps du système d'exploitation de l'hôte sur lequel s'exécute l'émulateur – permet d'injecter des signaux avec une grande précision temporelle *dans le référentiel de l'émulateur*, une propriété fondamentale pour résoudre le problème qui nous intéressera, à savoir l'asservissement de l'oscillateur cadençant le microcontrôleur sur la référence supposée exacte qu'est le signal de temporisation 1-PPS issu d'un récepteur GPS (virtuel) [2].

Nous supposerons que le lecteur, travaillant sous GNU/Linux, a acquis les sources de `simavr` à `https://github.com/buserror/simavr` et l'aura compilé par `make` dans sa racine après les quelques modifications suivantes qui seront l'occasion de voir que les sources de l'émulateur sont limpides à analyser. Afin de nous faciliter la visualisation de l'état des ports, nous complétons la fonction `avr_ioport_write()` de `simavr/sim/avr_ioport.c` avec

```
static void avr_ioport_write(struct avr_t * avr, avr_io_addr_t addr, uint8_t v, void * param)
{printf("\nSIMAVR: IOPORT @0x%x<-0x%x\n",addr,v);fflush(stdout);
 avr_ioport_t * p = (avr_ioport_t *)param;
 [...]
```

et de la même façon nous pouvons observer l'initialisation du port série dans `simavr/sim/avr_uart.c` en complétant

```
static void avr_uart_baud_write(struct avr_t * avr, avr_io_addr_t addr, uint8_t v, void * param)
{printf("SIMAVR: BAUDRATE 0x%x\n",v);fflush(stdout);
 p->io = _io;
 [...]

void avr_uart_reset(struct avr_io_t *io)
{printf("\nSIMAVR: UART RESET\n");fflush(stdout);
 avr_uart_t * p = (avr_uart_t *)io;
 [...]

void avr_uart_init(avr_t * avr, avr_uart_t * p)
{printf("\nSIMAVR: UART INIT\n");fflush(stdout);
 p->io = _io;
 [...]
```

Aucune subtilité de compilation si ce n'est d'avoir installé la version de développement de `libelf` pour compiler cet outil, et éventuellement les bibliothèques OpenGL pour profiter des interfaces graphiques des exemples. La compilation des *firmware* des exemples de `simavr` impose d'avoir installé le cross-compilateur `avr-gcc` (paquet `gcc-avr` dans Debian GNU/Linux avec sa dépendance `avr-libc` qui peut être pratique).

2 GPIO

Le cas du GPIO en sortie ne présente à peu près aucun intérêt puisque `simavr` affiche l'état des registres manipulés, y compris ceux chargés de définir le statut des ports d'entrée-sortie, mais fournit l'opportunité d'introduire l'arborescence du projet, le fichier de définition d'un circuit imprimé et les nomenclatures à respecter pour exploiter les `Makefile` de `simavr`.

Soit le programme trivial suivant pour faire clignoter périodiquement une LED connectée à la broche 5 du port B :

```
1  #define F_CPU 16000000UL
2  #include <avr/io.h>
3  #include <util/delay.h>  // _delay_ms
4
5  int main(void)
6  {DDRB |=1<<PORTB5;
7   PORTB |= 1<<PORTB5;
8   while(1)
9    {PORTB ^= (1<<PORTB5);
10    _delay_ms(200);
11   }
12 }
```

qui se compile avec `avr-gcc` par `avr-gcc -mmcu=atmega32u4 -O2 prog.c` en supposant que le code source est stocké dans le fichier `prog.c` et s'exécute sur un Atmega32U4. Ce programme définit le bit 5 du port B en sortie (`DDRB`) et fait clignoter la LED connectée à ce port (`PORTB`) avec une période de 400 ms. Nous n'avons pas besoin de rechercher dans la *datasheet* l'emplacement des ports, le fichier d'entête `io.h` et ses dérivées dans `/usr/lib/avr/include/avr` définissant un certain nombre de constantes en cohérence avec le nomenclature d'Atmel dans ses documentations.

Cependant, en tentant d'exécuter ce programme dans `simavr` par `simavr/run_avr a.out`, nous nous faisons insulter : la nature du processeur n'a pas été renseignée, `avr_make_mcu_by_name: AVR '' not known`, car `simavr` ne sait pas quel cœur émuler. Nous pouvons tricher en explicitant ces paramètres par `run_avr -m atmega32u4 -f 16000000`, mais ce n'est pas l'approche sélectionnée dans cet article qui visera à décrire un circuit imprimé équipé d'un processeur de modèle connu.

Que faut-il ajouter pour exécuter ce fichier depuis un circuit imprimé embarquant le microcontrôleur et décrit selon les préceptes de `simavr` ? Premièrement, si le circuit imprimé est décrit dans le fichier `intro.c` – et ce programme sera exécuté sur le processeur exécutant `simavr`, donc probablement un processeur compatible Intel x86 et non pas une architecture AVR – alors la structure des `Makefiles` des exemples de `simavr` impose d'appeler le firmware correspondant – qui lui s'exécutera sur le microcontrôleur – avec un nom de la forme `atXXX_intro.c`. Nous constatons ceci en lisant un des `Makefile` des exemples contenus dans `examples/board*` et en observant la règle `firm_src = ${wildcard at*${target}.c}` avec `target= intro` dans ce cas. Ainsi, le nom `firm_src` du logiciel embarqué se déduit de `target` préfixé de `at` et, pour faciliter la lecture, le nom du processeur cible. Nous reprenons donc notre `prog.c` pour le renommer `atmega32u4_intro.c` : contrairement à l'habitude sous unix, cette fois les noms de fichiers ont une importance (on verra que ce n'est pas la dernière fois). Ensuite, il faut compléter le programme avec quelques définitions qui seront exploitées par le compilateur pour savoir quel processeur appréhender : ces instructions n'affecteront *pas* le firmware flashé à terme dans le vrai microcontrôleur mais sont contenues dans l'exécutable au format ELF fourni à `simavr`. Nous allons préciser que nous travaillons sur Atmega32U4 en ajoutant, après les entêtes, les lignes

```
1  #include "avr_mcu_section.h"
2  AVR_MCU(F_CPU, "atmega32u4");
```

Ce programme se compile toujours de la même façon en ajoutant (option `-I`) le répertoire de `simavr` contenant le fichier d'entête, soit
`avr-gcc -mmcu=atmega32u4 -O2 -Isimavr/sim/avr atmega32u4_intro.c`
et s'exécute de la même façon, en faisant appel à `simavr` au travers de la commande
`simavr/run_avr a.out`
pour cette fois convenablement s'exécuter et indiquer

```
Loaded 232 .text at address 0x0
Loaded 0 .data
SIMAVR: UART INIT
SIMAVR: UART RESET
SIMAVR: IOPORT @0x25<-0x20
SIMAVR: IOPORT @0x25<-0x0
SIMAVR: IOPORT @0x25<-0x20
```

Ça y est, la simulation est bien partie, mais est toujours faite en explicitant `run_avr` et non pas en passant par la description d'un circuit imprimé.

Nous rédigeons donc un fichier `intro.c` qui contient

```c
#include <stdlib.h>
#include <stdio.h>
#include "sim_elf.h"

int main(int argc, char *argv[])
{avr_t * avr = NULL;
 elf_firmware_t f;
 const char * fname =  "atmega32u4_intro.axf"; // a.out if compiled manually
 elf_read_firmware(fname, &f);
 printf("fw␣%s␣f=%d␣mmcu=%s\n", fname, (int)f.frequency, f.mmcu);
 avr = avr_make_mcu_by_name(f.mmcu);
 if (!avr) {fprintf(stderr, "%s:␣AVR␣'%s'␣not␣known\n", argv[0], f.mmcu);exit(1);}
 avr_init(avr);
 avr_load_firmware(avr, &f);
 while (1) {avr_run(avr);}
}
```

qui décrit le circuit imprimé ... qui n'est ici fait de rien d'autre qu'un microcontrôleur dont le firmware est chargé après avoir été lu au format ELF. Nous avons choisi de conserver la nomenclature par défaut des makefiles proposés dans `examples/board*` de nommer le fichier ELF du nom du code source avec l'extension `axf`, mais il est tout à fait envisageable de compiler à la main et fournir `a.out` comme nom d'exécutable.

La simulation s'exécute dans la boucle infinie faisant appel à `avr_run()` : cela sera important pour la suite de la discussion et la génération des stimuli.

Évidemment nous désirons automatiser la compilation par `Makefile` : partant de l'exemple fourni dans `examples/board_ledramp` que nous copions dans le répertoire de travail, nous modifions le nom de `target` et tentons de compiler par `make`.

> Ici nous devons passer par une étape fort peu élégante pour contourner un dysfonctionnement du `Makefile`. Soit nous créons à la main le répertoire `obj-x86_64-linux-gnu`, et dans ce cas `make` est satisfait, soit nous remontons au sommet de l'arborescence de `simavr`, nous nettoyons le projet par `make clean`, et nous le re-créons par `make` qui fabrique le répertoire manquant dans notre exemple *avant* de peupler le répertoire `examples/parts`, cause de l'erreur `fatal error: opening dependency file obj-x86_64-linux-gnu/intro.d: No such file...`
> Dans ce second cas, nous aurons pris soin de nommer le répertoire contenant le projet d'un nom commençant par `board`, par exemple `board_intro`, pour qu'il soit compilé, puisque c'est selon ce critère que le `Makefile` de `examples` balaie les répertoires d'après la règle `for bi in ${boards}; do $(MAKE) -C $$bi; done`

Tout ceci étant fait, nous émulons *le circuit imprimé* (sans appeler explicitement `run_avr`) par `./obj-x86_64-linux-gnu/intro.elf` qui donne le même résultat que précédemment.

Qu'avons nous gagné si le résultat est le même qu'avant ? Nous avons maintenant la possibilité d'instancier des périphériques ! Ajoutons par exemple un bouton poussoir connecté au port C3 :

```c
#include <stdlib.h>
#include <stdio.h>
#include "sim_elf.h"
#include "avr_ioport.h"
#include "button.h"
volatile int appui=0;

int main(int argc, char *argv[])
{avr_t * avr = NULL;
 elf_firmware_t f;
 button_t button;               // add a button
 const char * fname = "atmega32u4_intro.axf";
 elf_read_firmware(fname, &f);
 avr = avr_make_mcu_by_name(f.mmcu);
 avr_init(avr);
 avr_load_firmware(avr, &f);
 // ajout pour la definition du bouton
 button_init(avr, &button, "button"); // inits our 'peripheral'
 avr_connect_irq(                     // connect the button to the port pin
    button.irq + IRQ_BUTTON_OUT,
    avr_io_getirq(avr, AVR_IOCTL_IOPORT_GETIRQ('C'), 3));
 avr_raise_irq(button.irq + IRQ_BUTTON_OUT, 1); // raise = pull up
 while (1) {avr_run(avr);
            if (appui==1) {printf("PCB:␣pushed\n"); // level to 0
                           appui=0;button_press(&button, 300000);}
```

```
  }
}
```

Ce programme définit la structure `button_t`, l'initialise et lui associe des événements liés à `C3`, se compile et s'exécute pour donner strictement le même résultat qu'avant, puisque évidemment `appui` ne change jamais d'état. On notera que dans la nomenclature de `simavr`, appuyer sur un bouton signifie le passer à l'état bas, et son état par défaut (équivalent à une résistance de tirage en pull-up) est défini par `avr_raise_irq(button...,1)`. L'utilisation des termes "irq" est quelque peu trompeuse, car fait référence au transfert d'événements asynchrones – qui peuvent se déclencher à tout moment – entre le circuit imprimé (PCB – *Printed Circuit Board*) et le firmware exécuté par le cœur, mais la détection de l'état du port se fera en mode *polling* sans activer les interruption de changement d'état de broche PCINT, ou de niveau INT (nous ferons cela plus tard). Cette affirmation est confirmée par le commentaire "IRQ stands usually for Interrupt Request, but here it has nothing to do with AVR interrupts" de `http://fabricesalvaire.github.io/simavr/doxygen/group__sim__irq.html`

C'est ici que nous découvrons l'importance d'avoir la simulation cadencée dans la boucle infinie appelant `avr_run()` : il nous faut une seconde boucle infinie pour cadencer les événements extérieurs. Il y a plusieurs religions :

— sûrement par soucis d'égayer leurs simulations avec des interfaces graphiques (nous ferons de même à la fin de ce document), les auteurs de `simavr` ont choisi de faire appel à GLUT, l'OpenGL Utility Toolkit. Dans ce cas, OpenGL se charge de générer les événements (appui de touche, de souris ou de timer) de façon asynchrone, et `avr_run()` est relégué à son propre thread,

— fort de cet enseignement, nous pouvons créer notre propre POSIX-thread (`pthread_create()`), y placer soit `avr_run()`, soit le séquenceur d'événements, et ainsi voir les deux boucles infinies tourner en parallèle,

— soit faire appel aux signaux d'unix qui se déclencheront en arrière plan de la boucle infinie qui exécute `avr_run()`.

Bien qu'aucune de ces solutions ne soit la bonne tel que nous le découvrirons plus tard (section 2.1) en terme d'exactitude temporelle, nous poursuivons cette description car elle permet de générer des stimuli complexes depuis l'émulateur du circuit imprimé vers le microcontrôleur si la temporisation relative au quartz qui cadence le microcontrôleur n'a pas d'importance.

Nous choisissons ici le dernier cas, qui nécessite le moins d'investissement intellectuel puisque SIGALRM se déclenche toutes les secondes par `signal(SIGALRM, handle_alarm); alarm(1);` et que le gestionnaire d'alarme `handle_alarm()` ré-enclenche le compte à rebours. Ainsi, nous ajoutons au code précédent

```
#include <signal.h>
void handle_alarm(int xxx) { appui=1; alarm(1); }     // 1 PPS on PC side (board)
void init_alarm(void) { signal(SIGALRM, handle_alarm); alarm(1);} // 1 PPS signal
```

Comme nous ne savons pas passer d'argument au gestionnaire de signal `handle_alarm()`, nous appliquons les préceptes des gestionnaires d'interruptions d'exceptionnellement nous autoriser une variable globale (`appui`), de type `volatile` pour interdire au compilateur de l'optimiser, qui partage l'information entre le *handler* et la boucle exécutant `avr_run()`.

Finalement, il reste à voir si le bouton est bien appuyé : en l'absence de communication pour le moment, nous modifions le masque d'allumage du port B en fonction de l'appui sur C3 :

```
int main(void)
{int masque=(1<<PORTB5);
 DDRB =0xff;  // port B output
 PORTB=0;     // port C default config = input
 while(1)
  {PORTB ^= masque;
   if ((PINC&(1<<3))==0)  // pressed: button goes low
     {// PORTB=0;         // reset PORTB (later ...)
      if (masque<0x80) masque=masque<<1;
         else masque=0x01;
     }
   _delay_ms(100);
  }
}
```

Ainsi, le bouton poussoir reste appuyé 300 ms (pour rappel, `if (appui==1) {button_press(b, 300000);appui=0;}`) et se ré-enclenche toutes les secondes (`alarme(1);`), tandis que le firmware change l'état du port B toutes les 100 ms. Nous nous attendons à voir deux (ou trois) changements consécutifs du masque, puis qu'il reste au même état pendant le reste de la seconde qui s'écoule. Cependant :

```
Loaded 250 .text at address 0x0
Loaded 0 .data
fw atmega32u4_intro.axf f=16000000 mmcu=atmega32u4
SIMAVR: UART INIT
SIMAVR: UART RESET
SIMAVR: IOPORT @0x25<-0x0
[... efface 35 fois la meme ligne ...]
SIMAVR: IOPORT @0x25<-0x0
PCB: pushed
SIMAVR: IOPORT @0x25<-0x20
```

```
SIMAVR: IOPORT @0x25<-0x60
SIMAVR: IOPORT @0x25<-0xe0
button_auto_release
SIMAVR: IOPORT @0x25<-0xe1
SIMAVR: IOPORT @0x25<-0xe0
SIMAVR: IOPORT @0x25<-0xe1
SIMAVR: IOPORT @0x25<-0xe0
[... efface 25 lignes alternant e0 et e1 ...]
SIMAVR: IOPORT @0x25<-0xe0
SIMAVR: IOPORT @0x25<-0xe1
SIMAVR: IOPORT @0x25<-0xe0
PCB: pushed
SIMAVR: IOPORT @0x25<-0xe1
SIMAVR: IOPORT @0x25<-0xe3
SIMAVR: IOPORT @0x25<-0xe7
```

```
button_auto_release
SIMAVR: IOPORT @0x25<-0xef
SIMAVR: IOPORT @0x25<-0xe7
SIMAVR: IOPORT @0x25<-0xef
[... efface 25 lignes alternant ef et e7 ...]
SIMAVR: IOPORT @0x25<-0xef
SIMAVR: IOPORT @0x25<-0xe7
SIMAVR: IOPORT @0x25<-0xef
SIMAVR: IOPORT @0x25<-0xe7
PCB: pushed
SIMAVR: IOPORT @0x25<-0xef
SIMAVR: IOPORT @0x25<-0xff
SIMAVR: IOPORT @0x25<-0xdf
button_auto_release
SIMAVR: IOPORT @0x25<-0x9f
SIMAVR: IOPORT @0x25<-0xdf
```

donc nous constatons que le bouton est appuyé (notre gestionnaire de temps sur l'émulation du PCB nous en informe par le message "PCB : pushed") puis relâché (message de simavr "button_auto_release") mais le nombre de messages laisse cependant à désirer. La temporisation imposée par le programme émulant le PCB qui contrôle simavr est fausse, il y a clairement plus d'une seconde écoulée entre la fin de l'appui et l'appui suivant.

En effet, comme attendu, nous avons 3 messages entre "pushed" et "auto_release" correspondant à trois intervalles de temps de 100 ms : les 300 ms de l'émulateur de PCB sont comptabilisées comme 3 fois 100 ms par le microcontrôleur. Cependant, nous avons une trentaine de messages de transitions d'état entre la fin de l'appui et l'appui suivant, soient environ 3 s dans le temps interne du microcontrôleur. Cette valeur est incohérente, notre 1-PPS ne dure pas du tout 1-seconde.

2.1 Du temps faux au temps juste

La source de notre erreur apparaît à la lecture de `examples/parts/button.c` et sa fonction `button_press()` dont la temporisation est observée correcte. Nous constatons que la durée d'appui est enregistrée auprès d'un *timer* géré par `simavr` qui appelle, à expiration, la fonction de *callback* qui relâche le bouton. Dans notre approche, la temporisation est prise en charge par le système d'exploitation exécutant le simulateur de PCB, dont le temps n'a aucune raison de s'écouler au même rythme que le temps interne du simulateur (sans parler des fluctuations induites par l'absence de contraintes temps-réel de GNU/Linux). Nous reprenons donc l'émulateur de PCB en modifiant l'appel au signal `alarm()` par un appel au gestionnaire de timer interne à `simavr` grâce aux fonctions `avr_cycle_timer_cancel();` et `avr_cycle_timer_register_usec();`. L'émulateur de PCB `intro.c` devient donc :

```
1  static avr_cycle_count_t PPS(avr_t * avr, avr_cycle_count_t when, void * param)
2  {button_t * b = (button_t *)param;
3   button_press(b, 300000);       // pressed = low
4   printf("PCB: pushed\n");
5   avr_cycle_timer_cancel(b->avr, PPS, b);
6   avr_cycle_timer_register_usec(b->avr, 1000000, PPS, b);
7   return 0;
8  }
9
10 int main(int argc, char *argv[])
11 {[... initialisation du PCB ...]
12  avr_raise_irq(button.irq + IRQ_BUTTON_OUT, 1); // raise = pull up
13  // 1-PPS timer init
14  avr_cycle_timer_cancel(button.avr, PPS, &button);
15  avr_cycle_timer_register_usec(button.avr, 1000000, PPS, &button);
16  while (1) {avr_run(avr);}
17 }
```

qui a été épuré au maximum en éliminant le code redondant avec l'exemple précédent. Nous n'avons plus besoin de `signal()` ou `alarm()`, cette fois tous les événements d'appui (*callback* nommé `PPS()`) et de relâchement du bouton sont pris en charge par les timers de `simavr`.

Fort de ces modifications, le résultat devient cohérent avec la temporisation attendue puisque

```
Loaded 250 .text at address 0x0
Loaded 0 .data
fw atmega32u4_intro.axf f=16000000 mmcu=atmega32u4
SIMAVR: UART INIT
SIMAVR: UART RESET
```

```
SIMAVR: IOPORT @0x25<-0x0
[... 8 transitions retirees ...]
SIMAVR: IOPORT @0x25<-0x20
SIMAVR: IOPORT @0x25<-0x0
PCB: pushed
```

```
SIMAVR: IOPORT @0x25<-0x20
SIMAVR: IOPORT @0x25<-0x60
SIMAVR: IOPORT @0x25<-0xe0
button_auto_release
SIMAVR: IOPORT @0x25<-0xe1
SIMAVR: IOPORT @0x25<-0xe0
SIMAVR: IOPORT @0x25<-0xe1
SIMAVR: IOPORT @0x25<-0xe0
SIMAVR: IOPORT @0x25<-0xe1
SIMAVR: IOPORT @0x25<-0xe0
SIMAVR: IOPORT @0x25<-0xe1
PCB: pushed
```

```
SIMAVR: IOPORT @0x25<-0xe0
SIMAVR: IOPORT @0x25<-0xe2
SIMAVR: IOPORT @0x25<-0xe6
button_auto_release
SIMAVR: IOPORT @0x25<-0xee
SIMAVR: IOPORT @0x25<-0xe6
SIMAVR: IOPORT @0x25<-0xee
SIMAVR: IOPORT @0x25<-0xe6
SIMAVR: IOPORT @0x25<-0xee
SIMAVR: IOPORT @0x25<-0xe6
SIMAVR: IOPORT @0x25<-0xee
PCB: pushed
SIMAVR: IOPORT @0x25<-0xe6
```

avec au début 10 transitions de 100 ms chacune avant que le bouton soit appuyé une première fois, puis un appui pendant 3 transitions correspondant à 300 ms, et un bouton relâché pendant 7 transitions ou les 700 ms restantes. Il ne reste plus qu'à décommenter l'ordre `PORTB=0x00` lors de l'appui du bouton pour obtenir un masque cohérent qui avance d'un bit à chaque appui du bouton.

Cette longue introduction sur le cas du GPIO nous a armé pour la suite des hostilités : nous savons créer un PCB, lui assigner une architecture de processeur et son firmware, exciter des signaux extérieurs et constater la cohérence avec les temporisations attendues. Passons aux choses sérieuses que sont les communications et interruptions sur événements timers.

Nous nous proposons de tester `simavr` sur un cas pratique que nous avions exposé dans [2] – moins impressionnant que `https://github.com/buserror/simreprap` qui émule une imprimante additive 3D complète et sera source d'inspirations pour appeler certains périphériques – mais qui sert encore de prétexte à appréhender les divers périphériques de processeurs, à savoir l'asservissement d'un oscillateur à quartz sur le signal horaire de référence issu d'un récepteur GPS.

Un récepteur GPS est avant tout un instrument dédié à reproduire une horloge locale suffisamment stable pour dater avec précision les signaux horaires transmis par la constellation de satellites, et éventuellement par trilatération de positionner son propriétaire dans l'espace. L'information de base fournie par un récepteur GPS, avant cette information de position, est une impulsion périodique générée chaque seconde – 1-PPS (1-*Pulse Per Second*) – dont le front montant indique, avec une précision de ±100 ns environ [4, Figs.2–4], le début de la seconde GPS.

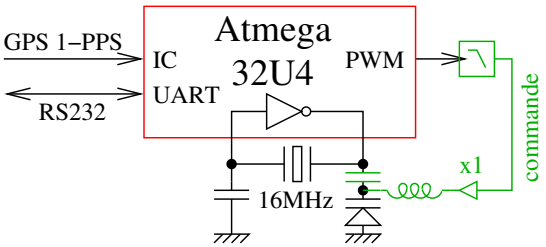

Figure 2: Schéma de principe, avec le 1-PPS issu du récepteur GPS qui alimente une entrée *input capture* (IC) connectée à un compteur (*timer*). L'ecart de fréquence à la valeur nominale est corrigée en polarisant une varicap faisant office de condensateur de pieds d'un oscillateur à quartz comprenant un résonateur et la porte inverseuse se comportant comme amplificateur introduisant un déphasage de 180°. Le filtre passe-bas chargé de lisser la PWM, l'amplificateur suiveur, l'inductance bloquant la fuite du signal radiofréquence et le condensateur de blocage de la commande DC vers l'amplificateur (en vert) ne seront pas explicités dans ce document qui se focalise sur les éléments numériques de l'asservissement.

Toute fluctuation de l'oscillateur local qui cadence le microcontrôleur entre deux fronts montants du 1-PPS s'observe comme une variation du compteur dont une interruption *input capture* est déclenchée par ce front. Trop lent, le décompte sera inférieur à la fréquence nominale du résonateur à quartz, trop rapide et le décompte sera supérieur à cette valeur. En plus d'observer cette dérive de la fréquence du résonateur à quartz face au 1-PPS de référence supposé parfait, nous pourrons corriger cette erreur de fréquence en ajustant les conditions d'oscillations dites de Barkhausen : la rotation de phase le long d'un oscillateur doit être multiple de 2π. Étant donné que dans un microcontrôleur l'amplificateur est formé d'une porte inverseuse (NOT) qui introduit donc une rotation de phase de 180°, et que le résonateur n'introduit pas de rotation de phase à la résonance (le circuit résonant formé d'une inductance et d'une capacité en série voit les deux réactances s'annuler à la résonance) et il reste les deux condensateurs de pieds – typiquement quelques dizaines de pF – reliant chaque borne du résonateur à quartz à la masse pour vérifier la condition de Barkhausen sur la phase. En jouant sur ces valeurs de condensateurs, nous pouvons abaisser la fréquence d'oscillations de quelques parties par million (ppm) et ainsi ajuster la fréquence d'oscillation. Un condensateur commandé en tension est une varicap. Comme le microcontrôleur Atmega32U4 ne possède pas de convertisseur analogique-numérique, le signal de commande sera généré par une sortie en modulation de largeur d'impulsion PWM suivie d'un filtre passe-bas pour la lisser et fournir une tension continue égale à la valeur moyenne : la commande se traduira donc par une modification du rapport cyclique de la PWM (en vert sur Fig. 2).

Résumons donc les périphériques dont nous aurons besoin (Fig. 2) et que nous simulerons sur `simavr` :

— la **communication** pour interagir avec l'utilisateur au travers du port série de communication asynchrone compatible RS232,

— un timer avec une résolution aussi bonne que possible pour **mesurer la fréquence** du quartz entre deux fronts montants du 1-PPS détéctés par *input capture* qui se charge de mémoriser l'état du compteur au moment de l'événement,

— un timer en PWM pour **commander** la varicap et de ce fait la fréquence de l'oscillateur cadençant le microcontrôleur, variable à laquelle nous aurons accès dans la configuration du microcontrôleur équipant le PCB,

— pour être complet, un **bouton poussoir déclenché par interruption** pour laisser l'opportunité à l'utilisateur d'interagir avec son instrument.

Nous allons aborder chacun de ces périphériques dans les sections qui suivent.

3 Communication asynchrone du monde extérieur vers le processeur

Commençons par communiquer. Le port de communication série asynchrone, compatible RS232, s'initialise en activant les divers paramètres que sont le débit de communication (*baudrate*), nombre de bits/symbole transmis, présence ou non de bits de parité, des choses très standard :

```
#define USART_BAUDRATE 115200  // avoid wasting too much time talking

void usart_setup()
{unsigned short baud;
 baud  = (((( F_CPU / ( USART_BAUDRATE * 16UL))) - 1));
 DDRD  |= 0x18;
 UBRR1H = (unsigned char)(baud>>8);
 UBRR1L = (unsigned char)baud;
 UCSR1C = (1<<UCSZ11) | (1<<UCSZ10); // async, no parity, 1 stop, 8 data
 UCSR1B = (1<<RXEN1 ) | (1<<TXEN1 ); // enable TX and RX
}
```

Écrire sur le port série s'obtient en vérifiant que le périphérique est libre, et en stockant dans le registre adéquat l'octet à communiquer, le matériel se chargeant ensuite de générer les signaux :

```
void mytransmit_data(uint8_t data)
{while ( !( UCSR1A & (1<<UDRE1)) );
 UDR1 = data;
}
```

L'émission d'un message d'un périphérique vers le microcontrôleur n'étant pas déterministe, la "bonne" façon de gérer la communication n'est pas d'attendre que le message arrive mais de déclencher une interruption sur la réception du caractère, le stocker dans un tampon afin de prévenir la boucle du programme principal de gérer le message quand il en a le temps. Le vecteur d'interruption `USART1_RX_vect` correspondant à l'UART1 est appelé quand un tel événement survient sous réserve d'avoir activé la fonctionnalité par le bit `RXCIE1` de `UCSR1B`. Nous proposons par ailleurs d'activer le traçage de toutes les interruptions pour en observer l'évolution dans le temps :

```
#include "avr_mcu_section.h"
AVR_MCU(F_CPU, "atmega32u4");
AVR_MCU_VCD_FILE("trace_file.vcd", 1000);
const struct avr_mmcu_vcd_trace_t _mytrace[]  _MMCU_ = {
        { AVR_MCU_VCD_SYMBOL("PORTB"), .what = (void*)&PORTB, },
        { AVR_MCU_VCD_SYMBOL("UDR1"), .what = (void*)&UDR1, },
};
AVR_MCU_VCD_ALL_IRQ();  // also show ALL irqs running

volatile char rx,flag;
ISR(USART1_RX_vect) {rx=UDR1;flag=1;}

int main(void)
{char c='.';
 DDRB=0xff;
 usart_setup();
 UCSR1B |= (1<<RXCIE1); // add RX interrupt
 flag=0;
 sei();
 while(1)
  {mytransmit_uart(c);           // transmits current symbol
   if (flag!=0) {flag=0;c=rx+1;} // if received, update c
   PORTB^=0x10;
   _delay_ms(200);
  }
}
```

Du point de vue du PCB, communiquer du microcontrôleur vers l'utilisateur ne présente aucune difficulté puisque `simavr` affiche sur la console tout caractère transmis sur le port de communication asynchrone. Plus difficile, envoyer un

symbole de l'utilisateur vers le microcontrôleur, qui impose de connecter un terminal à `simavr` pour en capturer les messages. Ceci s'obtient dans le code d'émulation du PCB par

```
#include "uart_pty.h"
uart_pty_t uart_pty;

int main(int argc, char *argv[])
{[... initialisation AVR et firmware ELF ...]
 uart_pty_init(avr, &uart_pty);
 uart_pty_connect(&uart_pty, '1');
  while (1) {avr_run(avr);}
}
```

pour connecter un terminal à UART1 (dernier argument de `uart_pty_connect()`. Selon le programme proposé plus haut, nous affichons le dernier symbole mémorisé `c` toutes les 200 ms, et mettons à jour le symbole si une communication de l'utilisateur vers le microcontrôleur a eu lieu afin de vérifier le bon fonctionnement l'interruption.

À l'exécution, `simavr` nous informe que

```
uart_pty_init bridge on port *** /dev/pts/9 ***
uart_pty_connect: /tmp/simavr-uart1 now points to /dev/pts/9
note: export SIMAVR_UART_XTERM=1 and install picocom to get a terminal
```

et en nous exécutant, nous obtenons le résultat de la Fig. 3.

FIGURE 3 – Communication depuis `picolog` vers `simavr` émulant un Atmega32U4 et répondant avec le caractère suivant de celui fourni. Lorsque nous tapons 'a' dans picolog (rouge), le microcontrôleur répond 'b' qui s'affiche dans le terminal exécutant simavr ainsi que dans picolog. Nous répétons l'opération jusqu'à 'e' qui se traduit par la réponse 'f' (vert).

Très intéressant grâce à la fonction de traçage des signaux internes au microcontrôleur, nous observons avec `gtkwave` la trace au format VCD (*Value Change Dump*) qui contient l'état des interruptions (Fig. 4) et constatons que l'interruption 0x19=25 s'est déclenchée, en accord avec la documentation technique [3, p.57] qui, en indexant les interruptions à partir de 1 et non de 0, indique que "USART1 Rx Complete" est l'interruption numéro 26.

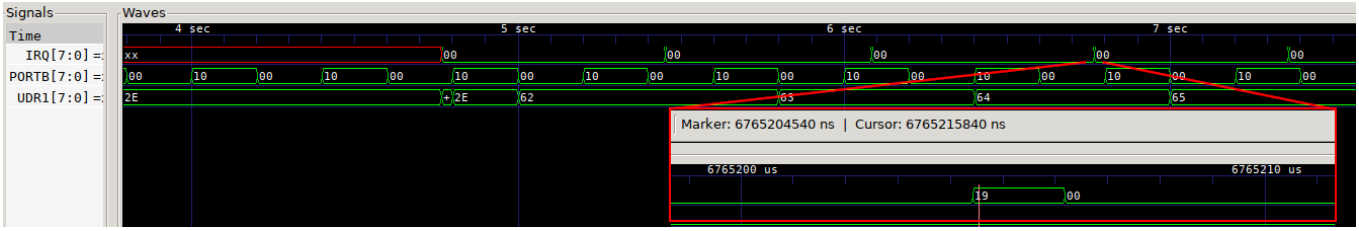

FIGURE 4 – Trace des signaux internes au microcontrôleur, incluant le registre de transmission des caractères du port série UDR et l'état des interruptions. Le cadran du bas est un zoom sur la courbe du haut pour identifier le numéro d'interruption déclenchée.

4 Interruptions : input capture, PWM et GPIO

Passons désormais aux *timers*. Il s'agit de périphériques forts simples mais forts utiles : un compteur tourne continuellement, atteint sa valeur maximale qui peut être source d'interruption (*overflow*) pour revenir à 0, et nous pouvons potentiellement déclencher des événements le long du décompte, par exemple un change d'état de broche pour moduler le rapport cyclique en mode *PWM* en maintenant une période constante. Plus intéressant dans notre cas, ce compteur peut être mémorisé lors du déclenchement d'un événement externe tel que la transition d'état sur une broche : il s'agit du mode *input capture* qui nous permettra de compter le nombre d'oscillations du circuit cadençant le microcontrôleur entre deux fronts montants du 1-PPS.

4.1 PWM

La PWM est sous le contrôle du microcontrôleur, donc notre seul objectif ici est de récupérer depuis l'émulateur du PCB l'information que la tension de commande de la varicap a été modifiée, entraînant une modification de la fréquence d'oscillation du circuit cadençant le microcontrôleur, et modifier les variable d'émulation en conséquent. Nous nous inspirons pour cela de `examples/board_timer_64led/timer_64led.c` qui module l'intensité lumineuse de l'afficheur par le rapport cyclique de la PWM.

```
void pwm_changed_hook(struct avr_irq_t * irq, uint32_t value, void * param)
     { display_pwm = value; pwm_flag=1; }
int main()
{[...]
 avr_irq_t* i_pwm= avr_io_getirq(avr, AVR_IOCTL_TIMER_GETIRQ('0'), TIMER_IRQ_OUT_PWM0);
 avr_irq_register_notify(i_pwm, pwm_changed_hook, NULL);
 [...]
 while (1)
 { avr_run(avr);  // avr->frequency: see ../../simavr/sim/sim_avr.c: avr_init()
   if (pwm_flag==1) {avr->frequency=16000000+display_pwm*10;
                     printf("PWM:%d -- freq: %d\n",display_pwm,avr->frequency);pwm_flag=0;
                    }
 }
}
```

Le pendant dans le firmware exécuté sur le microcontrôleur est de périodiquement modifier la valeur de la PWM qui a été initialisée par

```
void mypwm0_setup(void)
{TCCR0A =  (1<<COM0A1)|(0<<COM0A0)|(0<<COM0B1)|(0<<COM0B0)
             |(1<<WGM01) |(1<<WGM00);
 TCCR0B =  (0<<FOC0A) |(0<<FOC0B) |(0<<WGM02)
             |(0<<CS01)  |(1<<CS00);
 TCNT0 = 0;
 OCR0A = 127; // initial value
}

int main(void)
{mypwm0_setup();
 while(1) {OCR0A++;_delay_ms(200);}
}
```

Ce code affirme que la PWM s'incrémente toutes les 200 ms et se traduit, dans le code gérant le PCB, par un incrément de la fréquence de cadencement du microcontrôleur (`avr->frequency`) de (arbitrairement) 10 fois la valeur de la PWM, soit quelques kHz sur la gamme d'ajustement du timer, une valeur typique d'un résonateur à quartz. Comment pouvons nous vérifier si ces affirmations sont exactes? En analysant la valeur du timer configuré en *input capture* pour mesurer l'intervalle de temps entre deux fronts montants du 1-PPS.

4.2 Input Capture

Contrairement à la PWM où seule la simulation du circuit imprimé doit déclarer la gestion de l'événement de variation du rapport cyclique puisque le matériel gère cette configuration du timer dans le microcontrôleur, le cas *input capture* doit être pris en compte et dans le firmware, et dans le PCB.

Côté firmware, une gestion classique d'input capture sans originalité

```
volatile int flag_icp,value_icp;   // globar var for exchanging with ISR

ISR(TIMER3_OVF_vect) {flag_ovf++;} // timer overflow

ISR(TIMER3_CAPT_vect)
{flag_icp  = 1;    // informs main() of an event
 value_icp = ICR3; // remember timer value
 TCNT3 = 0;        // resets counter to 0 (timer diff)
}

void myicp_setup() // inits (16 bit) timer3
{TCCR3B = 1<<ICES3 | 1<<CS30;    // prescaler = 1
 TIMSK3 = (1<<ICIE3)|(1<<TOIE1); // IC & OVF interrupts
```

```
 TIFR3  = 1<<ICF3;
}

int main(void)
{int compteur=0;
 myicp_setup();
 sei();
 flag_icp=0;
 while(1)
  {if (flag_icp!=0)
      {write_short(value_icp); mytransmit_data('␣');
       write_short(flag_ovf);  mytransmit_data('\n');
        compteur++;
        if (compteur==5)
           {OCR0A++;compteur=0;} // update PWM duty cycle
            // -> triggers an event on the PCB
            flag_ovf=0;flag_icp=0;}
  }
}
```

Dans cet exemple, nous comptons grossièrement l'intervalle de temps entre deux fronts montants du 1-PPS en mémorisant le nombre d'*overflows* (interruption `TIMER3_OVF_vect` indiquant que le compteur a atteint sa borne maximale – TOP dans la nomenclature Atmel) et avec précision cet intervalle de temps en mémorisant la valeur du compteur au moment de l'événement par *input capture* par l'interruption `TIMER3_CAPT_vect`. Le compteur est remis à 0 dans ce gestionnaire d'interruption pour mesure un *écart* de temps entre deux fronts montants, et un drapeau est validé pour informer le programme principal que l'événement a eu lieu. Afin de caractériser la dépendance en boucle ouverte de la fréquence du quartz avec la valeur de la PWM, nous maintenons 5 fois de suite le seuil de transition de la broche correspondante à la même valeur, avant de l'incrémenter pour mesure la fréquence avec cette nouvelle commande.

Côté PCB, nous générons une fois par seconde un événement représentatif du 1-PPS du GPS. Nous avons vu (section 2) que nous ne devions pas exploiter le temps hôte (ordinateur sur lequel s'exécute `simavr`) mais les timers de `simavr` pour induire une datation cohérente des événements. Qu'à cela ne tienne : nous utilisons deux *timers* de `simavr`, un pour déclencher le front montant toutes les secondes, et l'autre pour redescendre le signal une centaine de millisecondes après sa montée

```
volatile int pwm_flag,icp_flag;

void icp_changed_hook(struct avr_irq_t * irq, uint32_t value, void * param) { icp_flag=1; }

static avr_cycle_count_t PPSlo(avr_t * avr, avr_cycle_count_t when, void * param)
{avr_irq_t* irq=(avr_irq_t*) param;
 printf("PPS:␣lo\n");
 avr_raise_irq(irq,1);
 return 0;
}

static avr_cycle_count_t PPSup(avr_t * avr, avr_cycle_count_t when, void * param)
{avr_irq_t* irq=(avr_irq_t*) param;
 printf("PPS:␣up\n");
 avr_raise_irq(irq,0);
 // avr_cycle_timer_cancel(avr, PPSup, irq); // useless cf ../../simavr/sim/sim_cycle_timers.c
 avr_cycle_timer_register_usec(avr, 1000000, PPSup, irq);   // which alread takes care of that
 // avr_cycle_timer_cancel(avr, PPSlo,NULL);
 avr_cycle_timer_register_usec(avr, 200000, PPSlo, irq);
 return 0;
}

int main(int argc, char *argv[])
{[ ... inits AVR and firmware provided in ELF format ...]
 avr_irq_t* irq= avr_io_getirq(avr, AVR_IOCTL_TIMER_GETIRQ('3'), TIMER_IRQ_IN_ICP);
 // avr_irq_t* irq= avr_get_interrupt_irq(avr, 32); // same effect than TIMER3+TIMER_IRQ_IN_ICP
 avr_irq_register_notify(irq, icp_changed_hook, NULL);
 // 1-PPS timer init
 avr_cycle_timer_cancel(avr, PPSup, irq);
 avr_cycle_timer_register_usec(avr, 1000000, PPSup, irq);

```

```
32  icp_flag=0;
33  while (1)
34  { avr_run(avr);
35    if (pwm_flag==1) {avr->frequency=16000000+display_pwm*10; // change oscillator frequency
36                      printf("PWM:%d -- freq: %d\n",display_pwm,avr->frequency);pwm_flag=0;}
37    if (icp_flag==1) {printf("ICP\n");icp_flag=0;}
38  }
39  return NULL;
40 }
```

Le résultat de l'exécution de ce code est de la forme

```
$ ./obj-x86_64-linux-gnu/PPScontrol.elf > t
23E3 00F4
23E3 00F4
23E3 00F4
23ED 00F5
23ED 00F4
23EB 00F4
23ED 00F4
23EF 00F4
23F6 00F5
23F6 00F4
```

qui indique donc que 244 ou 245 overflows se déclenchent, et que le résidu entre deux fronts montants est de l'ordre (première ligne) de 9187. Ainsi, la fréquence du quartz est de l'ordre du nombre d'overflows multiplié par la valeur maximale du compteur (ici 16 bits donc 65536) auquel on ajoute le résidu, soit $244 \times 65536 + 9187 = 15{,}999971$ MHz qui est raisonnablement proche des 16 MHz visés. Plus important, le PPS résultant de l'utilisation des timers de `simavr` est stable tel que le démontre la Fig. 5 dans laquelle l'incrément d'une unité de la PWM toutes les 5 mesures se traduit bien par une croissance de 10 unités de la fréquence.

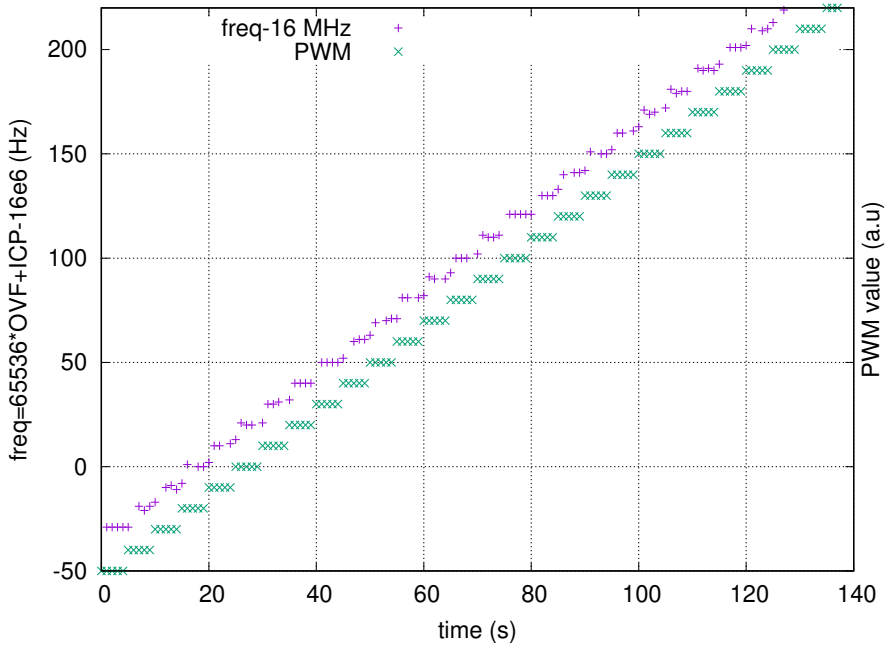

FIGURE 5 – Évolution de la fréquence de l'oscillateur cadençant le microcontrôleur (violet), et donc ses timers, en fonction de la tension de commande issue de la PWM filtrée par un passe-bas (vert).

5 Interruption GPIO

Afin de proposer un petit exercice sur les pointeurs, nous nous proposons de gérer un dernier type d'événement que sont les interruptions sur appui de bouton connecté à un GPIO. Nous n'avons pas d'application pratique dans ce contexte, mais nous allons voir que cela donne l'opportunité de nous amuser avec les cast.

Ainsi, nous ajoutons à notre séquence d'interruptions gérées par le firmware un bouton connecté (arbitrairement) à PD2, qui déclenche l'interruption sur niveau INT2 :

```
volatile int flag_int2;

ISR(INT2_vect) {flag_int2=1;}

int main(void)
{MCUCR &=~(1<<PUD);      // pull up resistor on the real microcontroller
 DDRD  &= ~(1<<PORTD2); // int2 on PD2 printed as D0
 PORTD |= (1<<PORTD2);  // pullup on, cf doc p62
 EICRA=0xff;            // rising edge on all pin interrupts
 EIMSK = 1<<INT2;       // enable int2 on PD2
 while (1)
   {if (flag_int2!=0) {mytransmit_data('2'); flag_int2=0;}
   }
}
```

dans lequel nous ne mentionnons que les lignes ajoutées au cas précédent, puisque nous désirons à la fois gérer l'input capture du timer et cette interruption GPIO.

Côté simulation du PCB, nous déclarons la fonction de callback liée à l'événement de changement de niveau du GPIO, mais comme nous voudrons passer les *deux* gestionnaires d'interruptions (input capture et GPIO) en argument aux gestionnaires de temporisation de `simavr` ou au thread que nous allons déclarer plus tard pour gérer de façon asynchrone `simavr` tandis que le gestionnaire d'interface graphique monopolise le processeur par sa boucle infinie, il nous faut une unique structure à passer comme argument. Nous allons donc déclarer un tableau de gestionnaires d'interruptions `irq[2]`

```
int main()
{avr_irq_t* irq[2];
 irq[0]= avr_io_getirq(avr, AVR_IOCTL_IOPORT_GETIRQ('D'), 2);
 irq[1]= avr_io_getirq(avr, AVR_IOCTL_TIMER_GETIRQ('3'), TIMER_IRQ_IN_ICP);
 avr_irq_register_notify(irq[1], icp_changed_hook, NULL);
 avr_irq_register_notify(irq[0], pd2_changed_hook, NULL);
 avr_cycle_timer_register_usec(avr, 1000000, PPSup, irq);
}
```

Ainsi grâce à cette structure, nous pouvons passer l'unique pointeur comme argument aux fonctions appelées, par exemple `avr_cycle_timer_register_usec()` ici, et comprendre la puissance du typage `void*` qui détermine la nature de ce dernier argument. Un `void*` est un pointeur sur n'importe quoi, une zone mémoire contenant donc aussi bien un scalaire (`char`, `short` ou `int`) qu'une structure complexe. Il nous suffit, dans la fonction appelée, de caster ce pointeur vers la nature de la structure passée en argument pour expliciter l'organisation de la mémoire :

```
static avr_cycle_count_t PPSup(avr_t * avr, avr_cycle_count_t when, void * param)
{avr_irq_t** irq=(avr_irq_t**) param;
 avr_raise_irq(irq[0],0);
 avr_raise_irq(irq[1],0);
 avr_cycle_timer_register_usec(avr, 1000000,PPSup, irq); // next rising edge event
 avr_cycle_timer_register_usec(avr, 200000, PPSlo, irq); // next falling edge event
 return 0;
}

static avr_cycle_count_t PPSlo(avr_t * avr, avr_cycle_count_t when, void * param)
{avr_irq_t** irq=(avr_irq_t**) param;
 avr_raise_irq(irq[0],1);
 avr_raise_irq(irq[1],1);
 return 0;
}
```

La première fonction est un callback d'un timer chargé de déclencher le signal 1-PPS. Lors de son déclenchement, le 1-PPS s'enregistre pour se re-déclencher une seconde plus tard, et enregistre un second timer pour s'abaisser 200 ms plus tard en appelant la seconde fonction. Dans ces deux fonctions, nous avons choisi de manipuler simultanément PD2 (connectée à INT2) et input capture afin d'illustrer le passage du tableau d'interruptions. Dans les deux cas, l'organisation de la mémoire est obtenue par le cast `avr_irq_t** irq=(avr_irq_t**) param;` qui indique que la zone mémoire sans type `void* param` doit être interprétée comme un tableau de pointeurs vers les gestionnaires d'interruptions. Cette méthode de passage de paramètre est très générale puisque nous la retrouvons dans FreeRTOS (prototype des tâches crées par `xTaskCreate()` de la forme `void vATaskFunction( void *pvParameters)` tel que décrit à `https://www.freertos.org/implementing-a-FreeRTOS-task.html`, ou dans le noyau Linux avec le prototype du gestionnaire d'interruptions

```
int request_irq(unsigned int irq, irqreturn_t (*handler)(int, void *, struct pt_regs *),
    unsigned long flags, const char *dev_name, void *dev_id);
```

décrit à `https://www.oreilly.com/library/view/linux-device-drivers/0596005903/ch10.html`.

6 Ajout de l'interface graphique

Maintenant que les bases du programme fonctionnent, avec la PWM, input capture, communication et interruption GPIO actives, nous pouvons retourner aux aspects esthétiques en ajoutant une illustration afin de mettre cette démonstration au niveau des autres projets de `examples/board*`, et profiter de ce fait du passage de paramètres tel que nous venons de l'expliciter lors de la création du POSIX thread qui exécute l'émulateur.

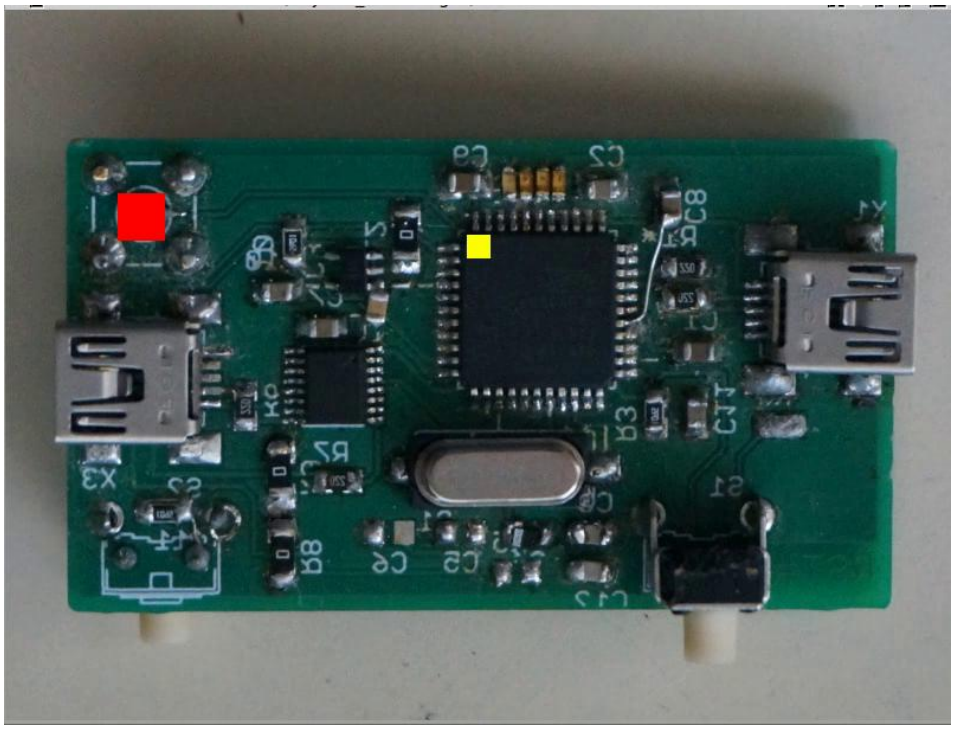

FIGURE 6 – Interface graphique permettant de visualiser les stimuli issus du circuit imprimé vers le microcontrôleur. En rouge le 1-PPS, en jaune l'input capture (timer 3) résultant. Un second input capture (timer 1) peut apparaître en cyan à côté du carré jaune s'il est connecté.

Nous n'entrons pas dans les détails des étapes, nous étant contenté de glaner des informations sur divers forums faute d'expérience sur la programmation en OpenGL, mais GLUT simplifie considérablement la gestion des événements graphiques tel que nous laisserons le lecteur le constater en consultant l'archive `https://github.com/jmfriedt/l3ep/blob/master/board_project/PPScontrol.c`. Nous voulions présenter à l'utilisateur les divers signaux transmis depuis le PCB au microcontrôleur et visualiser les réactions à ces stimuli. Ainsi, le 1-PPS est visualisé par un carré rouge sur le connecteur SMA auquel nous serions susceptibles de connecter la sortie du signal horaire d'un récepteur GPS, tandis que des carrés de diverses couleurs (jaune pour input capture du timer 3 dans l'exemple de la Fig. 6) sur le microcontrôleur représentent le déclenchement de l'interruption correspondante. Il ne reste qu'à rajouter autant de symboles dans la fonction d'affichage de l'image que de signaux à présenter, en conditionnant l'affichage du symbole sur le drapeau correspondant mis en place dans la fonction de callback (Fig. 6). La lecture de l'image JPEG en arrière plan s'appuie sur `libdevil` telle que décrite à `https://community.khronos.org/t/how-to-load-an-image-in-opengl/71231/6`.

Comme nous l'avions mentionné dans [1], un des avantages d'un tel outil de simulation est la génération de conditions de mesure difficilement reproductibles expérimentalement, voir de systématiser les tests en balayant les signaux physiques comme le fait tout bon développeur avec les tests unitaires. Dans ce cas, il reste donc à induire une variation brusque de la fréquence du quartz telle que nous l'observerions en plaçant la panne d'un fer à souder dessus, et boucler l'asservissement sur le 1-PPS pour corriger cette dérive de fréquence en ajustant de façon appropriée le condensateur de pieds de l'oscillateur. La démarche suivie dépasse le cadre du présent article mais est largement discutée dans le manuscrit qui accompagne le projet de Licence 3 qui a motivé cette étude [5] : nous sautons ici au résultat qui vise à démontrer le bon fonctionnement de la boucle d'asservissement fonctionnant sur le principe d'un contrôle proportionnel-intégrale (PI [6, 7]).

Le concept de commande proportionnelle, intégrale et dérivée (PID) issue des première réflexions sur les pilote automatiques [8, chap.1.6] se base sur l'idée de générer une commande sur un actuateur formée de la somme d'un terme proportionnel à la différence entre une observable et une consigne, d'un terme intégrale (somme des valeurs passées) de cette différence, et éventuellement de la dérivée de cette différence. Nombre d'ouvrages décrivent la contribution de chaque terme – correction, annulation de l'erreur statique et éviter de trop brusquer la commande, mais ici nous nous intéressons uniquement à l'implémentation. La commande $c(t)$ en temps continu t s'obtient à partir de l'erreur ε entre observable et consigne par $c(t) = K_p\varepsilon(t) + K_i\int_0^t \varepsilon(\tau)d\tau + K_d\frac{d\varepsilon}{dt}$ avec K_p, K_i et K_d des constantes à identifier, pour devenir en temps discret n : $c_n = K_p\varepsilon_n + K_i\sum_{k=0}^{n}\varepsilon_k + K_d(\varepsilon_n - \varepsilon_{n-1})$ en considérant la période d'échantillonnage unitaire. Afin d'éviter de voir la somme de l'intégrale diverger quand n devient grand, il est classique d'utiliser l'équation de récurrence sur c_n en écrivant c_{n+1} et en constatant que $c_{n+1} - c_n = K_p(\varepsilon_{n+1} + \varepsilon_n) + K_i\varepsilon_{n+1} + K_d(\varepsilon_{n+1} - 2\varepsilon_n + \varepsilon_{n-1})$ puisque les termes de la somme $\sum_{k=0}^{n}$ s'annulent. Ainsi, nous ne devrions pas écrire la solution de gauche mais celle de droite

```
err=0;
integrale=0;
err_1=0;
while(1) {
  mesure(&freq_mesure);
  erreur=consigne-freq_mesure;
  integrale+=err;                 // *dt
  sature_integrale(&integrale);  // evite depassement
  derivee=(err-err_1);           // /dt
  cmde_pwm=Kp*err+Ki*integrale+Kd*derivee;
  sature_commande(commande_pwm); // protege cmd
  err_1=err;
}
```

```
err=0;
err_1=0;
err_2=0;
commande_pwm=0;
while(1) {
  mesure(&freq_mesure);
  err_2=err_1;
  err_1=err;
  err=consigne-freq_mesure;
  cmde_pwm+=Kp*(err-err_1)+Ki*err+Kd*(err+err_2-2*err_1);
  sature_commande(&commande_pwm); // noter ci-dessus "+="
}
```

Nous nous imposons de travailler sur des entiers uniquement, l'Atmega n'étant pas équipé d'une unité de calcul flottante, et donc chosirons une expression fractionnaire de K_p, K_i et K_d qui nous permet d'exprimer ces coefficients inférieurs à 1 dans le cas de notre système de gain supérieur à 1 (pour rappel, nous avons choisi de bouger la fréquence de 10 Hz pour une commande qui varie d'une unité). Lors de la mise en œuvre de l'algorithme de droite, nous avons constaté que l'erreur statique n'était pas annulée mais que la commande faisant converger vers une valeur proche de la consigne mais biaisée. En effet dans la solution itérative, le terme intégrale $K_i\varepsilon_n$ s'annule si ε_n est plus petit que $1/K_i$ (dont nous rappelons qu'il est l'inverse d'un entier). Au contraire si l'intégrale est conservée explicitement dans la solution de gauche, alors le terme $\sum_{k=0}^{n}\varepsilon_k$ accumule tout l'historique des erreurs de l'asservissement et sa multiplication par K_i ne s'annule pas même si à un instant ε_n devient plus petit que $1/K_i$: la commande continuera à annuler le biais statique jusqu'à ce que *la somme* devienne plus petite que $1/K_i$, un cas qui se produit tard après que le biais ait été compensé. C'est donc la solution de gauche qui a été implémentée pour fournir les solutions de la Fig. 7.

Dans ce contexte, une erreur ε_n entre l'observable (la fréquence de l'oscillateur) à l'instant n et sa consigne est utilisée pour produire une commande c_n formée de la somme de cette erreur multipliée par une constante K_p (terme proportionnel) et l'intégrale de cette erreur multipliée par une constante K_i (terme intégral) avec toute la subtilité du réglage des constantes pour minimiser le temps de convergence tout en évitant de trop bousculer la commande. Alors qu'en temps continu la méthode dite de Ziegler et Nichols [9] est classique, en temps discret l'approche est un peu différente avec un point de départ pour régler K_p et K_i fournie par Takahashi [10, 11]. Dans ce contexte, sachant que nous avons choisi dans notre émulateur un facteur $a = 10$ entre le commande (sortie de PWM) et la variation de fréquence, le gain est connu et le retard L supposé nul car la commande agit immédiatement sur la fréquence dans devoir attendre une période T_e de la boucle d'asservissement. Dans ce contexte, Takahashi prévoit donc $K_p = \frac{0,27}{a\cdot(L+0,5T_e)^2} \simeq 0,11$ et $K_i = \frac{0,9}{a\cdot(L+0,5T_e)} - 0,5K_i \times T_e \simeq 0,125$. Comme nous travaillons sur microcontrôleur 8 bits sans unité de calcul en virgule flottante, nous nous imposons de ne travailler qu'avec des entiers et prendrons les inverses de ces coefficients dans l'implémentation de la loi de commande, soit $1/K_p \simeq 9$ et $1/K_i \simeq 8$. De ce point de départ, nous observons l'évolution de la fréquence de l'émulateur sous commande de la loi que nous venons de proposer : les divers régimes sont observés en Fig. 7.

Nous constatons sur cette figure 7 que Takahashi est un peu optimiste et induit un régime d'oscillations qui, sur un système mécanique, se traduirait rapidement par un risque de rupture. En atténuant un peu la loi de commande en abaissant K_p ou K_i (*i.e.* en augmentant leur inverse), nous retrouvons des régimes soit excessivement atténués, soit proche de l'optimum avec une légère dépassement de consigne avant de converger vers la valeur recherchée (rouge sur la Fig. 7). La boucle d'asservissement est donc complètement fonctionnelle dans l'émulateur.

7 Conclusion

Un projet conçu pour être aussi expérimental que possible d'asservissement de la fréquence d'un oscillateur à quartz cadençant un microcontrôleur sur le 1-PPS de référence issu d'un récepteur GPS a été rendu virtuel en nous appuyant sur `simavr` pour émuler le comportement du microcontrôleur Atmega32U4 mais surtout des divers périphériques qui le stimulent. Nous avons vu que grâce aux *timers* proposés par `simavr`, nous obtenons une simulation avec une base de temps commune à celle du microcontrôleur, tandis que les divers stimuli issus du microcontrôleur (gestion d'interruption, changement d'état de broche, communication ou variation de l'état d'une PWM) sont restitués à l'outil de simulation, permettant

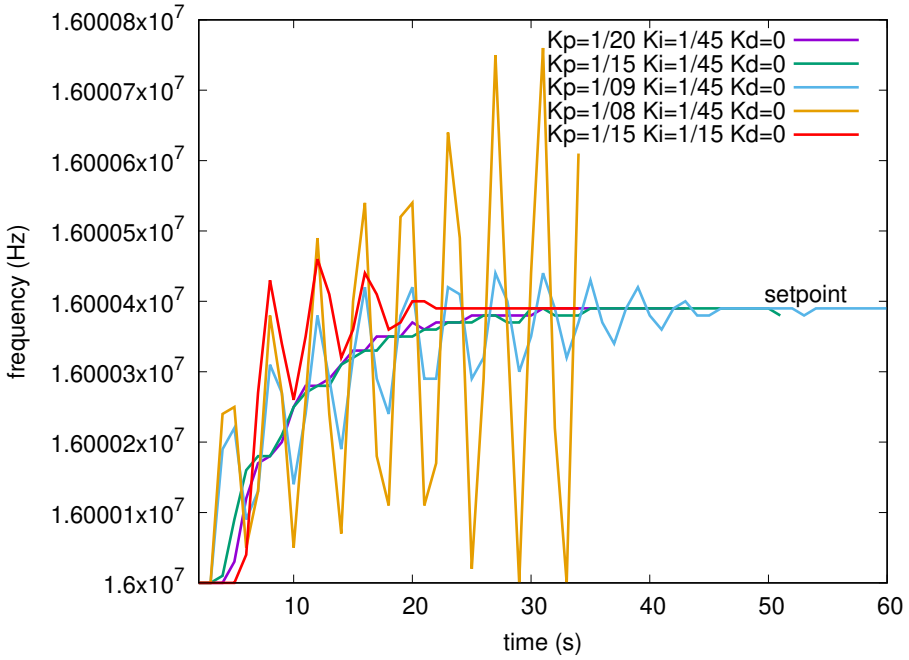

FIGURE 7 – Évolution de la fréquence de l'oscillateur en fonction des paramètres de la loi de commande proportionnelle-intégrale tels que discutés dans le texte. Les courbes violette et verte sont excessivement lentes à converger vers la consigne de 16 MHz+400 Hz, la courbe cyan présente des oscillations dangereuses mais finit par se stabiliser, tandis que la courbe orange présente un gain excessif et donc des oscillations divergentes. La courbe rouge est un optimal alliant rapidité de convergence sans présenter d'oscillation excessive.

de fermer la boucle entre le traitement embarqué dans le *firmware* et l'environnement physique émulé. On notera dans ce contexte la capacité à enregistrer des séquences de mesures et les rejouer comme stimuli tel que nous informe l'option `[--input|-i <file>] A .vcd file to use as input signals` de `run_avr`, malgré quelques contraintes quand à la nature des signaux injectés si l'on en croît `https://github.com/buserror/simavr/issues/267`.

Peu de déficiences sont apparues à l'usage de `simavr`, si ce n'est l'absence – pour le microcontrôleur Atmega32U4 qui nous concerne – du support du timer 4 dont nous avons facilement pu nous passer dans ce projet. Il est judicieux de vérifier quels périphériques sont émulés en cas de dysfonctionnement : ne voyant pas le compteur TCNT 3 du timer 3 s'incrémenter lorsque nous le placions en mode 11 (WGM3=WGM1=WGM0=1), il est apparu dans `simavr/cores/sim_mega32u4.c` (entrées `.timer1` et `.timer3`) que ce mode n'est pas implémenté (tout comme les modes 8 à 10).

Nous n'avons par ailleurs pas abordé les multiples fonctionnalités additionnelles que sont la génération dynamique de fichiers de traces de changement d'état des signaux (format VCD) depuis l'émulateur de PCB, ou l'interaction avec `gdb` qui est très bien décrite dans [12]. L'auteur de simavr nous met cependant en garde que les valeurs des compteurs de timers ne sont calculées qu'à leur utilisation par le cœur de processeur, et que `gdb` sondant l'état de ces registres observera une valeur erronée qui ne s'incrémente pas en l'absence de sollicitation par un périphérique.

Le résultat de ce travail est disponible à `https://github.com/jmfriedt/l3ep/` dans le sous-répertoire `board_project` à côté des divers exemples de code qui sont proposés en cours d'introduction à la programmation des microcontrôleurs 8 bits de Licence 3 de l'Université de Franche-Comté à Besançon.

8 Remerciements

M. *BusError* Pollet, auteur de `simavr`, a amélioré ce manuscrit par ses relectures des versions préliminaires, et motivé la finalisation de l'étude par sa disponibilité sur IRC. G. Cabodevila (enseignant-chercheur à l'École Nationale Supérieure de Mécanique et des Microtechniques) m'a enseigné la mise œure en temps discret de la loi de commande proportionnelle, intégrale et dérivée, et l'identification des coefficients de pondération par la méthode de Takahashi ainsi que l'implémentation de la méthode itérative. Toutes les références bibliographiques qui ne sont pas librement disponibles sur le web ont été acquises auprès de Library Genesis à `gen.lib.rus.ec`, une ressource indispensable à nos activités de recherche et d'enseignement.

Références

[1] J.-M. Friedt, *Développer sur microcontrôleur sans microcontrôleur : les émulateurs*, GNU/Linux Magazine **HS 103** (2019), à `https://connect.ed-diamond.com/GNU-Linux-Magazine/GLMFHS-103/Developper-sur-microcontroleur-sans-microcontroleur-les-emulateurs`

[2] J.-M. Friedt *& al.*, *Les microcontroleurs MSP430 pour les applications faibles consommations – asservissement d'un oscillateur sur le GPS*, GNU/Linux Magazine France **98** (2007)

[3] Atmel, *ATmega16U4/ATmega32U4 datasheet*, révision 7766H (06/2014)

[4] U-Blox, *GPS-based Timing Considerations with u-blox 6 GPS receivers – Application Note* (2011) à `https://www.u-blox.com/sites/default/files/products/documents/Timing_AppNote_(GPS.G6-X-11007).pdf`

[5] Guide orientant l'asservissement d'un oscillateur à quartz sur un signal de référence horaire issu d'un récepteur GPS : `http://jmfriedt.free.fr/projet_atmega.pdf`

[6] N. Minorsky *Directional stability of automatically steered bodies*, J. American Society for Naval Engineers **34**(2), pp. 280–309 (1922)

[7] K. Åström & T. Hägglund, *PID controllers – 2nd Ed.*, Instrument Society of America (1995)

[8] S. Bennett. *A History of Control Engineering 1930-1955*, IET (1993)

[9] J.G. Ziegler & N.B. Nichols, *Optimum settings for automatic controllers*, Trans. ASME **64** pp. 759-–768 (1942)

[10] Y. Takahashi, C.S. Chan & D.M. Auslander, *Parametereinstellung bei linearen DDC-Algorithmen*, Automatisierungstechnik (1971), 237-244

[11] A. Besançon-Voda & S. Gentil, *Régulateurs PID analogiques et numériques*, Tech. de l'ingénieur R7416 (1999), ou sans les fautes, le cours de Gonzalo Cabodevila disponible à `http://jmfriedt.free.fr/Gonzalo_cours1A.pdf`

[12] manuel de `simavr` par J. Gruber dans le répertoire `doc` du projet, ou L. Kellogg-Stedman, *Debugging attiny85 code* (2019) à `https://blog.oddbit.com/post/2019-01-22-debugging-attiny-code-pt-1/`