Utilisation d'un simulateur de processeur

É. Carry, J.-M Friedt, 16 mars 2020

Nous allons profiter de l'incapacité de nous rencontrer pour exploiter un simulateur de processeur pour aboutir dans la réalisation du projet de commande d'un oscillateur à quartz pour l'asservir sur le 1-PPS de GPS.

1 Introduction

Nous avions discuté dans [1] de l'utilisation d'un émulateur d'Atmega32U4 libre nommé `simavr` [1], outil capable d'exécuter un code compilé pour une cible – Atmega, ARM ou RISC-V dans le cas de cet article – autre que l'architecture de l'hôte sur lequel nous travaillons – généralement un PC basé sur un processeur compatible x86. Cette méthode de travail a l'avantage de permettre d'appréhender du matériel que nous ne possédons pas, d'émuler le comportement de périphériques en cours de développement ou simplement de pallier l'absence de matériel comme c'est le cas ici.

Dans [1], nous avions été dans l'incapacité d'émuler le comportement du matériel autour du cœur du microcontrôleur et nous étions contenté d'observer le comportement du processeur exécutant des instructions, communiquant par UART ou manipulant ses GPIO par l'affichage de chronogrammes. Nous allons ici compléter cette émulation en ajoutant les périphériques entourant l'Atmega32U4 pour permettre d'achever le projet que nous avons commencé. Il nous faut donc émuler un signal 1-PPS (issu normalement du récepteur GPS) pour alimenter l'entrée *input capture* d'un timer, et récupérer la sortie PWM pour ajuster la fréquence du quartz.

> La compilation de `simavr` s'obtient trivialement en clonant le dépôt `git` et par `make` dans le répertoire ainsi créé. On prendra cependant soin de cloner l'exemple proposé dans `https://github.com/jmfriedt/l3ep/` et de copier `board_project` à côté des autres exemples de `examples/` **avant** cette première compilation. Noter que le nom du répertoire importe car le `Makefile` de `examples` ne compile que le contenu des répertoires dont le nom commence par `board` (`for bi in ${boards}; do ...`).

Une fois `simavr` compilé, nous trouvons dans `simavr/` le programme `run_avr` qui prend comme argument le firmware (exécutable binaire au format ELF) tel que nous avions l'habitude de le compiler pour l'Atmega32U4. Seule subtilité, `simavr` impose de décrire la nature du processeur et sa fréquence de fonctionnement : les programmes émulés par `simavr` doivent tous contenir, en plus de ce que nous faisions habituellement en travaux pratiques d'Électronique Programmable, les lignes suivantes

```
1 #define F_CPU 16000000UL
2 #include "avr_mcu_section.h"
3 AVR_MCU(F_CPU, "atmega32u4");
```

Ces lignes ne seront pas compilées dans le programme à destination du vrai microcontrôleur, mais informent `simavr` de la nature du processeur (ici atmega32u4) et sa fréquence de cadencement (ici 16 MHz). La liste des processeurs supportés s'obtient par `simavr/run_avr --list-cores`. Cet entête peut aussi contenir des directives de mémorisation de l'état de registres (et donc des périphériques associés) :

```
1 const struct avr_mmcu_vcd_trace_t _mytrace[]  _MMCU_ = {
2       { AVR_MCU_VCD_SYMBOL("PORTB"), .what = (void*)&PORTB, },
3       { AVR_MCU_VCD_SYMBOL("UDR1"), .what = (void*)&UDR1, },
4 };
```

mémorise les changements d'état du port B et du port de communication asynchrone compatible RS232. En effet, il nous sera plus aisé de communiquer par ce biais que par le port USB tel que nous l'avons fait en travaux pratiques afin de nous affranchir de la complexité de ce protocole complexe.

Le programme ci-dessous démontre l'initialisation de la communication asynchrone, ainsi que le résultat des trames stockées par `simavr` et visualisées par `gtkwave` (Fig. 1).

```
1 #define F_CPU 16000000UL
2 #include <avr/io.h>
3 #include <util/delay.h>  // _delay_ms
4
5 // simulator variables: trace file + CPU type (not used in the firmware)
6 #include "avr_mcu_section.h"
7 AVR_MCU(F_CPU, "atmega32u4");
8 AVR_MCU_VCD_FILE("trace_file.vcd", 1000);
9 const struct avr_mmcu_vcd_trace_t _mytrace[]  _MMCU_ = {
10      { AVR_MCU_VCD_SYMBOL("PORTB"), .what = (void*)&PORTB, },
11      { AVR_MCU_VCD_SYMBOL("UDR1"), .what = (void*)&UDR1, },
12 };
```

1. `https://github.com/buserror/simavr` et son excellente documentation à `https://polprog.net/papiery/avr/simavr.pdf`

```
13
14 #define USART_BAUDRATE 9600
15
16 void transmit_data(uint8_t data)
17 {while ( !( UCSR1A & (1<<UDRE1)) );
18  UDR1 = data;
19 }
20
21 void usart_setup()              // initialisation UART
22 {unsigned short baud;
23  baud  = (((( F_CPU / ( USART_BAUDRATE * 16UL))) - 1));
24  DDRD  |= 0x18;
25  UBRR1H = (unsigned char)(baud>>8);
26  UBRR1L = (unsigned char)baud;
27  UCSR1C = (1<<UCSZ11) | (1<<UCSZ10); // asynch mode,no parity, 1 stop bit, 8 bit size
28  UCSR1B = (1<<RXEN1 ) | (1<<TXEN1 ); // enable Transmitter and Receiver
29 }
30
31 int main(void)
32 {char symbole='.';
33  MCUCR &=~(1<<PUD);
34  DDRB |=1<<PORTB5;      // blinking LED
35  PORTB |= 1<<PORTB5;
36  usart_setup();
37  while(1)
38   {transmit_data(symbole);
39    PORTB^=(1<<PORTB5);
40    _delay_ms(200);
41    symbole++;
42   }
43 }
```

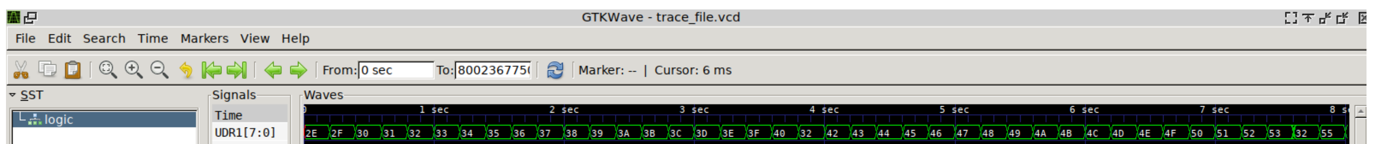

FIGURE 1 – Chronogramme issu de l'exécution du code incrémentant à chaque itération de la boucle infinie l'octet transmis sur le port série.

2 Émulation du matériel autour du processeur

`simavr` propose un certain nombre d'exemples de circuits émulés dans `examples/` dont nous nous inspirons pour notre projet. Nous constatons que chaque répertoire contient la description d'un circuit imprimé avec ses périphériques, à côté duquel est fourni le programme qui devra être flashé dans le microcontrôleur. Si par exemple nous analysons `examples/board_timer_64led`, nous constatons que le comportement du circuit imprimé est décrit dans `timer_64led.c` tandis que le logiciel à embarquer dans le microcontrôleur se trouve dans `atmega168_timer_64led.c`. Ici encore les noms ont une importante, tel que nous le constatons en lisant le `Makefile` : il faut que le nom du firmware *soit le même que le nom du circuit*, préfixé du nom du processeur supporté, commençant par `atmega`. En effet dans le `Makefile` nous constatons `target=timer_64led` et `firm_src=${wildcard at*${target}.c}` qui signifie de rechercher tous les fichiers commençant par `at` et contenant le nom de la cible qu'est le nom du circuit imprimé.

Ainsi, chaque répertoire de projet contient deux fichiers, le firmware qui peut s'exécuter par `run_avr`, et un exécutable pour l'ordinateur hôte, généralement au format Intel x86 :

```
$ file atmega32u4_PPScontrol.axf obj-x86_64-linux-gnu/PPScontrol.elf
atmega32u4_PPScontrol.axf:           ELF 32-bit LSB executable, Atmel AVR 8-bit, ...
obj-x86_64-linux-gnu/PPScontrol.elf: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), ...
```

Le premier fichier s'exécute sur l'émulateur : `../../simavr/run_avr atmega32u4_PPScontrol.axf` se traduit par la séquence

```
Loaded 390 .text at address 0x0
Loaded 0 .data
SIMAVR: UART INIT
SIMAVR: UART RESET
SIMAVR: IOPORT @0x25<-0x20
SIMAVR: IOPORT @0x2b<-0x4
SIMAVR: BAUDRATE 0x67
0x18.
SIMAVR: IOPORT @0x25<-0x0
/
SIMAVR: IOPORT @0x25<-0x20
0
SIMAVR: IOPORT @0x25<-0x0
1
SIMAVR: IOPORT @0x25<-0x20
2
SIMAVR: IOPORT @0x25<-0x0
```

En effet, en l'absence de stimulation externe, ce programme se contente de commuter l'état du port B et afficher un caractère sur le port série selon

```
1  flag_int2=0;
2  while(1)
3   {transmit_data(symbole);
4    if (flag_int2!=0) {transmit_data('2'); flag_int2=0;}
5    PORTB^=(1<<PORTB5); // bouge PORTB
6    _delay_ms(200);
7    symbole++;
8   }
9 }
```

mais plus intéressant, exécuter ce même programme dans le contexte de la plateforme par `./obj-x86_64-linux-gnu/PPScontrol.elf` se traduit par le message périodique contenant le symbole "2" transmis par le microcontrôleur (ici après le symbole "A") indiquant qu'une interruption matérielle a été déclechée sur le microcontrôleur

```
SIMAVR: IOPORT @0x25<-0x20
@
SIMAVR: IOPORT @0x25<-0x0
PPS:PD2
PD2
ICP
A2
SIMAVR: IOPORT @0x25<-0x20
B
SIMAVR: IOPORT @0x25<-0x20
```

et par ailleurs l'émulation du circuit imprimé a aussi pris connaissance du déclenchement de l'interruption en affichant "PD2" tel que proposé dans la fonction de callback liée à ce gestionnaire d'interruption. En effet dans le fichier d'émulation du circuit imprimé `PPScontrol.c`, nous trouvons

```
1 void pd2_changed_hook(struct avr_irq_t * irq, uint32_t value, void * param) {printf("PD2\n");}
2 ...
3 avr_irq_t* pd2= avr_io_getirq(avr, AVR_IOCTL_IOPORT_GETIRQ('D'), 2);
4 avr_irq_register_notify(pd2, pd2_changed_hook, NULL);
```

tandis que le signal 1-PPS est émulé par

```
1 if (alarm_flag==1) {
2   avr_raise_irq(pd2,0);
3   avr_raise_irq(icp,0);
4   usleep(100);
5   avr_raise_irq(icp,1);
6   avr_raise_irq(pd2,1);
7 }
```

avec `alarm_flag` un signal (au sens Unix, `man 7 signal`) qui se déclenche toutes les secondes. La seconde occurrence de "PD2" correspond au front descendant car l'émulateur du circuit imprimé n'est pas au courant de notre configuration sur le micronctrôleur du déclenchement de l'interruption sur front montant uniquement, tandis que "PPS" est un message qui indique que la condition d'alarme a été atteinte. Ces deux messages sont transmis par `PPScontrol.c` (circuit imprimé) et ne doivent pas être confondus avec les messages transmis par `atmega32u4_PPScontrol.c` (firmware exécuté sur l'émulateur de microcontrôleur).

3 Mise en œuvre pour la détection du PPS et asservissement du quartz

Fort de ces connaissances, nous pouvons décrire le comportement du circuit imprimé utilisé pour réaliser le projet : nous devons réagir aux modifications sur la PWM qui contrôle la varicap, qui doit donc changer la fréquence de cadencement du microcontrôleur, et recevoir les impulsions 1-PPS sur une entrée timer adéquate. Nous ne rentrons pas dans les détails de `https://github.com/jmfriedt/l3ep/blob/master/board_project/PPScontrol.c` mais c'est le rôle de ce fichier que de capturer ces informations et les restituer sur la console à l'utilisateur. On notera que l'émulation du 1-PPS s'obtient par le mécanisme des alarmes qui est classique sous unix mais ne permet probablement pas la compilation de ce programme sous MS-Windows. Si ce dernier système d'exploitation doit être utilisé, il y a peut être moyen de déclencher l'interruption timer toutes les secondes grâce à *OpenGL utility toolkit* (GLUT), mais nous ne savons pas faire.

Ainsi, nous compilerons `PPScontrol.c` (qui ne devrait normalement pas avoir à être modifié, sauf erreur de description du comportement du circuit) qui fait appel à `atmega32u4_PPScontrol.c`. Nous avons fourni une version minimaliste de ce programme embarqué pour démontrer le bon fonctionnement de l'approche, mais tout le projet reste à implémenter, à savoir

- interruption *input capture* sur le timer 1
- PWM sur le timer 0
- caractérisation en boucle ouverte de la fréquence du quartz en fonction de la commande (virtuelle) du polarisation de la varicap

Votre travail devrait donc se limiter à ajouter des fonctionnalités à `atmega32u4_PPScontrol.c` tandis que nous (Émile + JMF) nous chargerons de rendre l'émulation du circuit imprimé aussi réaliste que possible dans `PPScontrol.c`

En l'absence d'un oscilloscope pour observer les signaux, profitez des traces pour observer l'état des registres internes au processeur (dont vous ne pourriez même pas observer l'état avec du "vrai" matériel dans certains cas).

4 Émulation du timer

Nous mentionnons en appendice A un cas où le timer ne semble pas fonctionner selon nos attentes. Nous illustrons ici un cas qui semble fonctionner correctement : le timer 3 est configuré pour déclencher une interruption sur *overflow* avec un pre-scaler de 256 et sur *input capture*. Le signal d'input capture est généré environ chaque seconde. Nous constatons la sortie

```
SIMAVR: IOPORT @0x2e<-0x0
.........i05E2 0CDC 0002.
2....
SIMAVR: IOPORT @0x2e<-0x40
......iF1BE 05C1 0001.
2.....
SIMAVR: IOPORT @0x2e<-0x0
...
..........i98BD C1F6 0001.
2.
SIMAVR: IOPORT @0x2e<-0x40
.....
SIMAVR: IOPORT @0x2e<-0x0
..........i3DC8 4D58 0002.
2
```

Avec un pre-scaler de 256, le timer3 en mode normal se déclence toutes les 1.05 s. Nous constatons que ce rythme est à peu près respecté, avec parfois deux déclenchements entre deux input capture dont le signal n'a pas prétention d'exactitude. Chaque message "SIMAVR :" indique un changement d'état du port B (interruption overflow) et la dernère valeur de chaque message qui suit un "i" (input capture) est le nombre d'overflows (1 ou 2). Finalement, les deux autres valeurs sont celles de ICR3 (valeur du timer au moment de l'input capture) et TCNT3, la différence entre les deux étant le temps de quitter l'ISR et effectuer les affichages. Ce résultat est donc cohérent avec nos attentes.

De la même façon, en passant le pre-scaler à 1, nous constatons que deux input capture déclenchées par le 1-PPS émulé par le signal sont séparées de 199 à 248 overflows, en accord avec la théorie de $16 \cdot 10^6/65536 \simeq 244$. Nous ne pourrons donc cependant pas compter sur l'exactitude du signal pour fournir le 1-PPS idéal que devrait fournir GPS.

Références

[1] J.-M Friedt, *Développer sur microcontrôleur sans microcontrôleur : les émulateurs*, GNU/Linux Magazine France HS **103** (Juillet-Aout 2019), disponible à `http://jmfriedt.free.fr/glmf_emulateur.pdf`

A Émulation du timer

Attention : il y a sûrement des problèmes d'implémentation de l'émulation du timer sous `simavr`. Le code simple ci-dessous ne se comporte pas comme prévu :

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>  // _delay_ms

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

ISR(TIMER3_OVF_vect) {PORTE^=1<<PORTE6;}

void myicp_setup() // initialisation timer3
{TCCR3A = 1<<WGM31 | 1<<WGM30;
 TCCR3B = 1<<ICES3 | 1<<WGM33 | 1<<CS31;  // -> CS30 to fail triggering overflow
 OCR3A  = 32000;
 TIMSK3 = (1<<TOIE1);
 TIFR3  = 1<<ICF3;
}

int main(void)
{char symbole='.';
 MCUCR &=~(1<<PUD);
 DDRE |=1<<PORTE6;      // blinking LED
 myicp_setup();
 sei();
 while(1) {}
}
```

Ce programme déclenche une multitude d'interruptions par overflow mais si nous passons le bit CS30 à 1 dans TCCR3B il n'y a plus d'interruption déclenchée, alors que nous sommes supposés avoir accéléré l'horloge d'un facteur 8. Il serait intéressant de corriger ce dysfonctionnement de l'émulateur en en identifiant la cause.

Dans l'état actuel de notre compréhension, il *faut* initialiser l'horloge du timer par

`TCCR3B = 1<<ICES3 | 1<<WGM33 | 1<<CS30;`

alors que

`TCCR3B = 1<<ICES3 | 1<<WGM33 | 1<<CS31;`

qui devrait simplement abaisser d'un facteur 8 la fréquence de cadencement, se traduit par l'absence d'incrément du timer tel que observé en affichant `TCNT3`. Nous atteignons donc ici les limites de l'émulation, dont il faut corriger les dysfonctionnements.

Comme avec du vrai matériel, il faudra peut etre revenir à quelques expériences de base pour vérifier quel mode te timer fonctionne dans quelles conditions.