TP programmation en assembleur sur microcontrôleur 8 bits

É. Carry, J.-M Friedt

2 décembre 2015

Ce TP se situe dans la continuité du premier TP de L3 disponible à `http://jmfriedt.free.fr/TP_Atmega32U4_GPIO.pdf`. Pour rappel, le schéma qui nous intéresse de la carte Olimexino32U4 est en Fig. 1

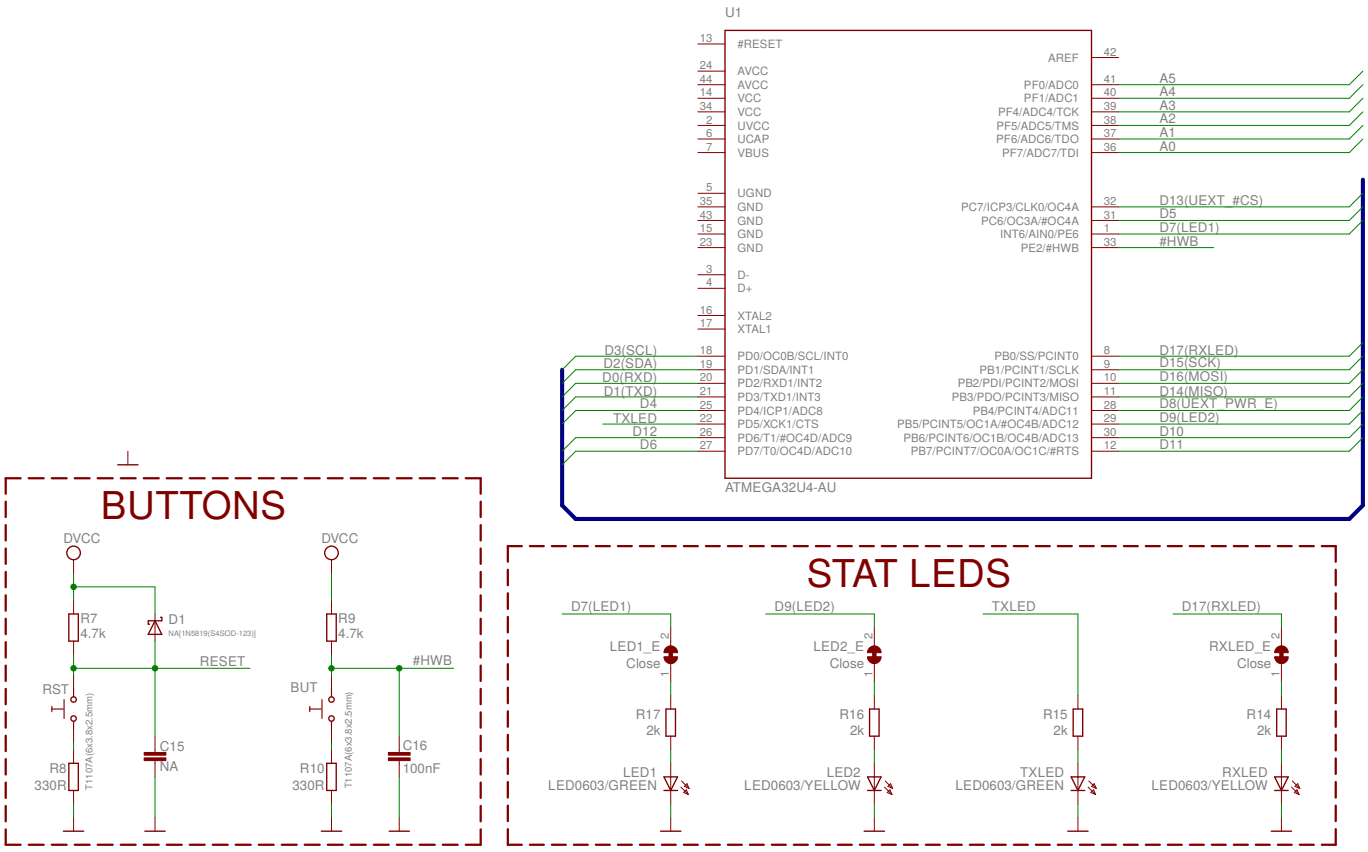

FIGURE 1 – Extrait du schéma de l'Olimexino-32U4

1 Communication sur bus USB : la bibliothèque LUFA

LUFA [1] est une bibliothèque riche de gestion du protocole USB, complexe à mettre en œuvre [1] compte tenu de la variété des conditions d'utilisation et des transferts effectués lors de l'initialisation des transactions avec le PC. Ces initialisations ont en particulier vocation à informer le système d'exploitation du PC des capacités de la liaison USB (nombre de points de communication, débit, nature du pilote susceptible de prendre en charge ces transactions sur le PC).

Nous proposons une version allgégée de la bibliothèque et un exemple simple de transactions sur port série virtuel, nommé sous GNU/Linux `/dev/ttyACM0`. L'archive est disponible à `http://jmfriedt.free.fr/LUFA_light_EEA.tar.gz`. Pour compiler ce programme :

1. aller dans `VirtualSerial_lib` et `make lib`. À l'issue de la compilation, constater l'existence de `libVirtualSerial.a`, la bibliothèque statique fournissant les fonctionnalités de LUFA,
2. aller dans le répertoire `VirtualSerial`, y copier la bibliothèque `libVirtualSerial.a`,
3. compiler l'exemple par `make`

1. *Lightweight USB Framework for AVRs*, `www.fourwalledcubicle.com/LUFA.php`

4. On pourra éventuellement coupler compilation et transfert du programme au microcontrôleur sur la carte Olinuxino32U4 par `make flash_109`.
5. Constater le bon fonctionnement du logiciel par `cat < /dev/ttyACM0`

Noter la nécessité d'inclure le fichier d'entête `VirtualSerial.h` qui définit les fonctions nécessaires au fonctionnement de l'USB, ainsi que les structures de données

```
extern USB_ClassInfo_CDC_Device_t VirtualSerial_CDC_Interface;
extern FILE USBSerialStream;
```

L'interface USB est initialisée par `CDC_Device_CreateStream(&VirtualSerial_CDC_Interface, &USBSerialStream);` tandis que le chien de garde et la cadence d'horloge sont configurés par `SetupHardware()` (lire le code source de `VirtualSerial_lib/VirtualSerialConfiguration.c`. Enfin, les trois lignes

```
CDC_Device_ReceiveByte(&VirtualSerial_CDC_Interface);
CDC_Device_USBTask(&VirtualSerial_CDC_Interface);
USB_USBTask();
```

dans la boucle infinie gèrent les messages venant du PC vers le microcontrôleur. Omettre ces lignes n'empêche par le bon fonctionnement de la liaison microcontrôleur vers PC, mais trop de caractères venant du PC vont remplir la pile d'entrée de l'USB et rendre le logiciel inopérant.
Exercice : compléter l'affichage du message par défaut par la taille des trois types d'entiers gérés par le C.

2 Programmation en assembleur

L'assembleur est un langage exprimant, selon des mnémoniques intelligibles par un humain, les instructions comprises par l'unité arithmétique et logique (ALU) du processeur. Ces instructions sont nécessairement très simples – branchement, arithmétique et logique, stockage – rendant les programmes assembleur très verbeux mais très efficaces. En particulier, la bijection entre langage machine et assembleur permet de calculer au cycle d'horloge près la durée d'exécution d'un programme. Une telle capacité s'avère souvent nécessaire dans des petits bouts précis de programme : une connaissance de l'assembleur est donc utile soit dans un contexte de déverminage de programme (lecture d'un code désassemblè depuis un exécutable), soit pour l'optimisation de quelques procédures dont les lattences doivent être précisément maîtrisées. `gcc`, et sa déclinaison pour processeur Atmel `avr-gcc`, permet évidemment de compiler un programme en assembleur [2], point de passage obligatoire pour un compilateur de langage de haut niveau tel que le C. Nous nous servirons de cet outil d'une part pour compiler des programmes écrits en assembleur pour processeur Atmega32, et d'autre part d'un émulateur pour observer les durées d'exécution indépendamment de la plateforme matérielle.

Nous nous familiariserons avec un assembleur simple d'un petit microcontrôleur 8 bits, mais tous les assembleurs se ressemblent, et la maîtrise d'un langage permet de rapidement appréhender les mnémoniques des autres processeurs à peu de différences près, généralement reflétant des différences d'architecture (notamment RISC v.s CISC).

Par défaut, un programme écrit en C commence son exécution à la fonction `main()`. Cependant, avant cette instruction, un certain nombre d'initialisations sont effectuées au moyen d'un fichier lié au programme lors de l'édition de lien, classiquement nommé `crtXX` (*C Runtime* avec `XX` l'ordre d'appel si plusieurs fichiers de démarrage existent). En compilation avec l'option `-v` de `gcc` active, nous constatons que l'éditeur de liens (*linker script*) est `avr5.x` et que le démarrage est pris en charge par `crtm32u4.o` qui se trouve dans `/usr/lib/avr/lib/avr5/` :

```
 /usr/lib/gcc/avr/4.8.1/collect2 -m avr5 -Tdata 0x800100 -o test1.out
/usr/lib/gcc/avr/4.8.1/../../../avr/lib/avr5/crtm32u4.o -L../../lib -L/usr/lib/gcc/avr/4.8.1/avr5
-L/usr/lib/gcc/avr/4.8.1/../../../avr/lib/avr5 -L/usr/lib/gcc/avr/4.8.1
-L/usr/lib/gcc/avr/4.8.1/../../../avr/lib test1.o --start-group -lgcc -lm -lc --end-group
```

Bien que le code de démarrage soit disponible dans l'archive de `avr-libc` à `http://download.savannah.gnu.org/releases/avr-libc/`, en particulier dans le fichier `avr-libc-1.8.1/crt1/gcrt1.S`, nous désirons simplement nous inspirer de cette séquence d'initialisation mais nous affranchir de l'utilisation d'un fichier dont nous ne comprenons pas pour le moment toutes les subtilités (il en sera autrement à la fin de ce TP, et nous aurons alors le droit de sereinement utiliser un fichier dont nous comprenons les opérations). `crtXX.S` effectue en particulier l'initialisation des vecteurs d'interruption et du pointeur de pile. L'appel à ce fichier est éliminé par l'option `-nostartfiles` de `gcc`.

Cependant, retirer l'initialisation fournie par la bibliothèque implique de configurer soi-même les registres garantissant le bon fonctionnement du microcontrôleur. Les deux points auxquels il faut classiquement prendre soin est de **désactiver le chien de garde** – un mode de réinitialisation automatique du microcontrôleur pour garantir son redémarrage en cas de dysfonctionnement du logiciel exécuté – et **configurer la pile**, un élément incontournable au bon fonctionnement de n'importe quel programme. La gestion du chien de garde est décrite dans [3, p.52]. Un peu plus surprenant, sur l'architecture Atmega32, la pile n'est pas contenue dans un registre spécifique mais dans un des registres généralistes [3, section 4.6]. Nous

y apprenons que la pile est definie dans le couple SPH,SPL. La RAM finit en AFF pour le 32U4 (ou 5FF respectivement pour le modèle 32U2).

```
; on commence par de la maintenance
eor r1, r1
out 0x3f, r1
out 0x34, r1 ; wdt reset
ldi r28, 0xFF ; stack ...
ldi r29, 0x05 ; ... pointer
out 0x3e, r29
out 0x3d, r28

ldi r24, 0x18 ; WDE WDCE : autorise ecriture dans WDT
cli   ; disable interrupts
sts 0x0060, r24
sts 0x0060, r1 ; WDTON=WDE=WDIE=0 => stop

sbi 0x07, 4 ; // 7 = DDRC
sbi 0x07, 5 ; // 7 = DDRC
cbi 0x08, 4 ; // clear PC4
sbi 0x08, 5 ; // set PC5
ldi r17, 0x30

boucle:
 in r24, 0x08 ; 8 // lit PORTC
 eor r24, r17
 out 0x08, r24 ; 8 // PORTC^=0x30
 call mon_delai

rjmp boucle

; cette fonction doit etre APRES son appel, pas avant ?!
mon_delai:
  ldi r19,0x8e
outer:
  ldi r18,0xff
inner:
   subi r18, 1
   brne inner
  subi r19, 1
  brne outer
ret
```

En s'inspirant du programme ci-dessus qui fait clignoter PC4 et PC5 : faire clignoter les deux LEDs verte et jaune de l'Olinuxino32U4. Pour ce faire, identifier le port de connexion, les registres associés, et modifier le programme de façon appropriée.
Comparer la taille du code ainsi généré par rapport au code généré dans le premier exemple de transactions par USB.

3 Transfert du logiciel en mémoire non-volatile du microcontrôleur

Le mécanisme de programmation et de configuration – classiquement représenté par l'acronyme ISP (*In System Programming*, donc pas d'outil matériel dédié pour flasher le microcontrôleur) est nommé par Atmel DFU (*Device Firmware Upgrade*), implémenté dans une série d'outils disponibles à `https://dfu-programmer.github.io/`. sous le nom de `dfu-programmer`. Comme d'habitude pour un microcontrôleur stockant son programme en mémoire flash, nous devons d'abord en effacer le contenu en plaçant tous les bits à 1, avant de configurer les bits qui passent à 0 lors du transfert du programme. En fin de programmation, le microcontrôleur est réinitialisé pour lancer l'exécution depuis l'instruction située à l'adresse 0x0000.

Cependant, Olimex modifie ce *bootloader* fourni par défaut par Atmel et le remplace par un logiciel respectant les consignes de la note d'application 109 d'Atmel [4], dont les sources sont disponibles à `https://github.com/arduino/Arduino/blob/master/hardware/arduino/avr/bootloaders/caterina/` et en particulier `Caterina.c`. L'utilisation d'un *bootloader* pose un problème d'initialisation des périphériques : nous constatons par exemple que le watchdog est désactivé et l'USB initialisé. Le problème survient lorsque nous désirons exécuter notre propre code sans passer par le *bootloader* : l'oubli d'initialisation d'un périphérique pris en charge par le *bootloader* se traduira par un dysfonctionnement de notre programme

particulièrement pénible à déverminer. Il est donc fondamental de bien avoir conscience des implications du *bootloader* sur le fonctionnement du microcontrôleur – un argument supplémentaire en faveur des outils *opensource* puisqu'un *bootloader* propriétaire ne peut être consulté par ses utilisateurs.

La commande pour flasher le microcontrôleur est
`avrdude -c avr109 -b57600 -D -p atmega32u4 -P /dev/ttyACM0 -e -U flash:w:fichier.elf`
(noter que la mise à jour du système permet de désormais flasher les fichiers au format ELF – la conversion explicite en format Intel hexadécimal n'est plus nécessaire).

4 Simulation

`simavr`, disponible à `https://github.com/buserror/simavr`, est un simulateur qui permet d'observer l'évolution de tous les ports/registres. Il nécessite une configuration des traces qui seront sauvegardées au moyen d'un bout de code compatible C. Dans le cas d'un programme assembleur, on utilisera la compilation séparée pour d'une part configurer `simavr` dans un code C, et d'autre part programmer le microcontrôleur en assembleur. Le code en C n'apparaîtra ni ne surchargera le code assembleur : il n'est utilisé que par `simavr` préfixe `_MMCU_`).

Ainsi, pour capturer tous les évènements liés aux changements d'état du port C, on indique dans un ficher C la macro suivante

```
1 #include <avr/io.h> //E/S ex PORTB
2 #define F_CPU 16000000UL
3
4 #include "avr_mcu_section.h"
5 AVR_MCU(F_CPU, "atmega32");
6
7 const struct avr_mmcu_vcd_trace_t _mytrace[] _MMCU_ = {
8       { AVR_MCU_VCD_SYMBOL("PORTC"), .what = (void*)(&PORTC), },
9   };
```

qui crée une entrée nommée (arbitrairement) `PORTC` dans le fichier qui sera analysé par `gtkwave`, et qui se déclenche sur tout changement du port à l'adresse `&PORTC`. Nous pouvons ajouter autant de règles que nous le voulons, pour suivre autant de signaux que nécessaire. Un résultat d'une telle simulation faisant clignoter deux LEDs sur le port C est proposé en Fig. 2.

Cette configuration nécessite un fichier au format C, avec les entêtes contenant les définitions de constantes, l'instruction `const struct` et une syntaxe en général incompatible avec l'assembleur. Pour cette raison, deux fichiers distincs sont créés, un contenant le code assembleur exécuté sur le microcontrôleur (extension `.S`) et un définissant les paramètres de simulation en C (extension `.c`). La compilation séparée de ces deux objets permet ensuite, lors de l'édition de liens, de générer un unique binaire au format ELF (qui sera converti en format Intel hexadécimal pour flasher le microcontrôleur).

Question : identifier le fichier d'entête qui définisse la constante PORTC et donner la valeur de cette constante. Est-ce que cette valeur est cohérente avec le tableau fourni en fin de [3] ?

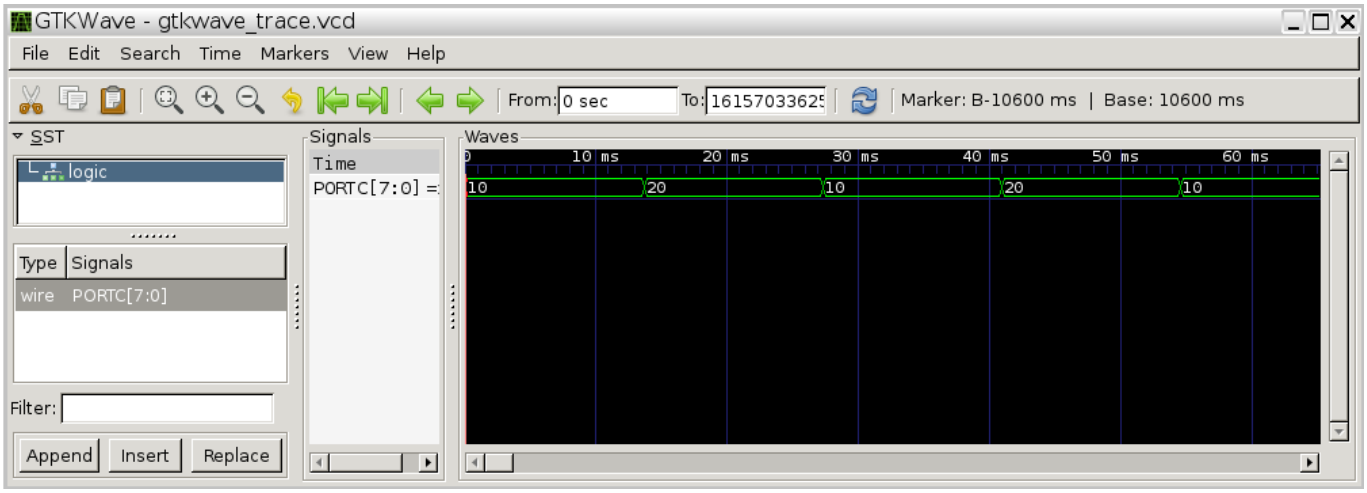

FIGURE 2 – Capture d'écran de `gtkwave` utilisé pour afficher l'évolution du port C.

Le simulateur peut s'avérer fort utile lorsqu'il informe l'utilisateur d'accès illégaux à des ressources inexistantes. Par exemple l'option `atmega32` de `simavr` suppose qu'il s'agit d'une architecture Atmega32U2.

Expliquer le message d'erreur :

```
Loaded 250 .text at address 0x0
Loaded 0 .data
avr_interrupt_init
CORE: *** Invalid write address PC=00b8 SP=0aff O=0e94 Address 0aff=5e out of ram
avr_sadly_crashed
avr_gdb_init listening on port 1234
CORE: *** Invalid write address PC=00b8 SP=0aff O=0e94 Address 0afe=00 out of ram
avr_sadly_crashed
```

lorsque le code d'initialisation correspondant, une fois désassemblé par, `avr-objdump -dSt blink.out` est

```
000000ac <__ctors_end>:
  ac:   11 24           eor     r1, r1
  ae:   1f be           out     0x3f, r1        ; 63
  b0:   cf ef           ldi     r28, 0xFF       ; 255
  b2:   da e0           ldi     r29, 0x0A       ; 10
  b4:   de bf           out     0x3e, r29       ; 62
  b6:   cd bf           out     0x3d, r28       ; 61
  b8:   0e 94 63 00     call    0xc6    ; 0xc6 <main>
```

Suivre à la trace l'évolution d'un registre permet par conséquent de suivre l'évolution de la pile, une ressource précieuse puisque si elle est corrompue, le programme crashera irrémédiablement en perdant la valeur du PC lors du saut à la sub-routine d'attente.

Question : comment modifier la consigne d'enregistrement de `gtkwave` afin de stocker le pointeur de pile à chacune de ses modifications ?

Dans la Fig. 3, une telle capture a été effectuée. Maladroits que nous sommes, nous avons empilé une valeur dans le programme principal en oubliant de la dépiler.

Question : analyser la Fig. 3 et identifier les évolutions de la pile.

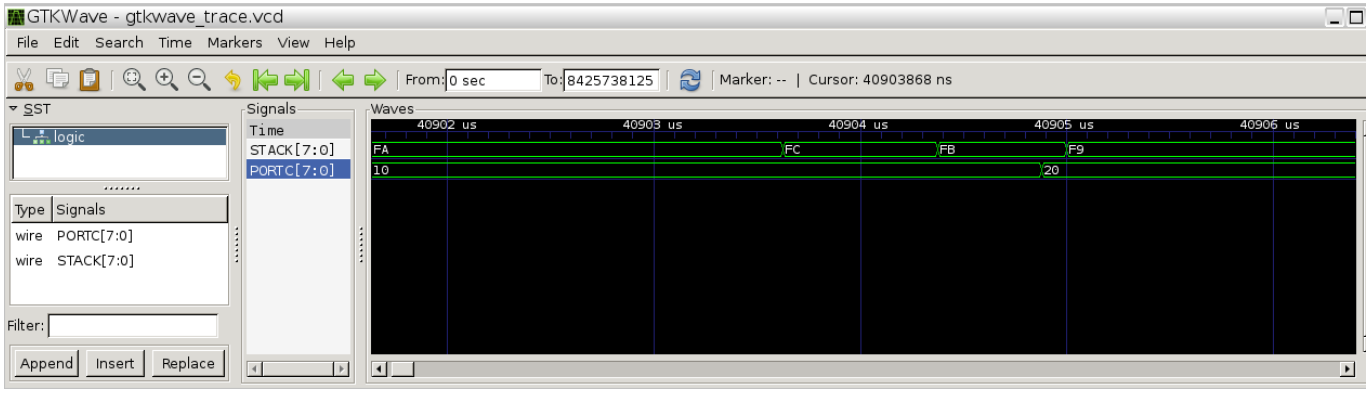

FIGURE 3 – Capture d'écran de `gtkwave` utilisé pour afficher l'évolution du port C et de la pile.

Exercice : reprendre l'exemple 6 de `http://jmfriedt.free.fr/TP_Atmega32U4_GPIO.pdf` et l'exécuter dans le simulateur, en configurant `simavr` pour afficher les caractères transmis sur port série. Pour ce faire, on recherchera l'adresse du registre UDR1 dans la documentation du microcontrôleur. Si la simulation se bloque, identifier et corriger la cause du blocage.

5 Communication entre émulateur et `gdb`

5.1 Introduction à `gdb`

Le *GNU Debugger* (`gdb`) est un outil puissant pour sonder l'évolution d'un programme en cours d'exécution, que ce soit sur plateforme matérielle si elle le supporte (le plus souvent au travers d'une interface JTAG) ou d'un simulateur. `gdb` permet d'afficher en cours d'exécution la valeur de variables, le contenu de la pile ou des registres internes au processeur (et donc inaccessibles par des affichages), ou encore induire des points d'arrêt (*breakpoint*) pour temporairement faire cesser l'évolution de *program counter* et laisser à l'utilisateur le temps d'observer l'état du programme et de la machine qui l'exécute. Le mode de communication avec `gdb` se fait le plus souvent au travers de transactions client-server, avec `gdb` agissant comment client et le simulateur (ou sonde JTAG) comme serveur.

Prenons l'exemple du programme fort simple

```c
#include <avr/io.h> //E/S ex PORTB
#define F_CPU 16000000UL
#include <util/delay.h> // _delay_ms

void ma_fonction() {volatile int i=1;i++;}

int main(void){
  DDRB |=1<<PORTB5;
  PORTB |= 1<<PORTB5;

  while (1){ /* Clignotement des LEDS */
    PORTB^=1<<PORTB5;
    _delay_ms(1000); //Attente de 500ms
    ma_fonction();
  }
  return 0;
}
```

Comme nous désirons avoir accès aux symboles du programme C, nous compilons avec l'option de déverminage `-g` résultant en

```
avr-gcc -mmcu=atmega32u2 -Os -g -Wall -o blink_simavr.out blink_simavr.c
```

Le fichier `.out` est un binaire au format ELF pour exécution sur architecture AVR :

```
$ file blink_simavr.out
blink_simavr.out: ELF 32-bit LSB executable, Atmel AVR 8-bit, version 1 (SYSV), statically linked,
not stripped
```

Nous ne pouvons donc exécuter ce programme que sur un microcontrôleur ou, en l'absence de plateforme matérielle, un émulateur. Cet émulateur, `simavr`, peut se comporter comme serveur `gdb` et ainsi devenir un outil de déverminage puissant. Par exemple,

```
$ simavr -m atmega32 -g blink_simavr.out -f 16000000
```

lance l'émulateur en cadençant l'exécution des opérations sur un oscillateur à 16 MHz et en se comportant comme serveur `gdb` (option `-g`). Nous pourrions nous connecter par `telnet` au port ainsi ouvert, mais communiquer avec le serveur gdb nécessiterait d'en étudier le protocole, une tâche qui n'est pas impossible mais ardue. Nous nous contentons donc d'utiliser un client `gdb` approprié à la plateforme matérielle, prenant comme argument le binaire ELF contenant les symboles de déverminage

```
$ avr-gdb blink_simavr.out
```

après une invitation verbeuse, `gdb` s'attend à ce que nous l'informions de l'emplacement de la cible (émulateur ou plateforme matérielle) et des commandes à exécuter. Le serveur local attend sur le port 1234 donc nous informons `gdb` par

```
(gdb) target remote localhost:1234
(gdb) continue
```

et le programme s'exécute. CTRL-C l'arrête. L'exécution n'est pas verbeuse et nous n'avons rien appris. Nous désirons arrêter l'exécution au moment de l'entrée dans `ma_fonction()`. Pour ce faire

```
(gdb) break ma_fonction
(gdb) continue
```

et après quelques secondes

```
Breakpoint 1, ma_fonction () at blink_simavr.c:13
13      {volatile int i=1;
(gdb) list
```

une fois la simulation arrêtée, nous pouvons demander la liste des instructions (`list`), l'état de la pile (`bt`), l'état d'une variable (`print i`) ou une exécution pas à pas (`step`). Une variable peut être modifiée (`set variable i=5`) et le code hexadécimal correspondant à notre fonction affiché

```
(gdb) disassemble
Dump of assembler code for function ma_fonction:
   0x0000008c <+0>:     push    r28
   0x0000008e <+2>:     push    r29
...
   0x00000098 <+12>:    ldi     r25, 0x00       ; 0
   0x0000009a <+14>:    std     Y+2, r25        ; 0x02
   0x0000009c <+16>:    std     Y+1, r24        ; 0x01
...
   0x000000ac <+32>:    pop     r29
   0x000000ae <+34>:    pop     r28
   0x000000b0 <+36>:    ret
```

Au lieu de compiler pour un Atmega32U2 comme nous l'avons fait auparavant (`-mmcu=atmega32u2`) compiler pour un Atmega32U4, et simuler le code résultant. Commenter le message fourni par l'émulateur.

5.2 Analyse du passage de paramètre par la pile

Supposons que nous ayons une structure composée de plusieurs éléments, par exemple une matrice de la forme `struct matrice {char reel[30];};`. Une fonction de calcul reçoit comme argument une telle matrice pour effectuer une opération sur ses éléments `int calcul(struct matrice) {...}`.

```
1  struct matrice {char tableau[30];};
2
3  int ma_fonction(struct matrice ma_matrice)
4  {ma_matrice.tableau[1]=3;
5   return(ma_matrice.tableau[1]);
6  }
7
8  int main(void){
9    struct matrice tt;
10   volatile i;
11   for (i=0;i<30;i++) tt.tableau[i]=i;
12   while (1){
13     i=ma_fonction(tt);
14   }
15   return 0;
16 }
```

Exercice :

1. **Remplir les éléments d'une matrice de valeurs connues, par exemple les nombres de 0 à 29 dans cet exemple,**
2. **simuler l'exécution du programme dans `avr-gdb` et placer un point d'arrêt dans l'appel à la fonction `calcul`,**
3. **afficher l'état des registres – et en particulier du pointeur de pile – par `info register`**
4. **afficher le contenu de la pile en affichant le contenu de la mémoire dans la zone occupée par la pile : on eXamine le contenu de la mémoire par l'instruction `x` suivi du nombre d'octets à afficher et du mode d'affichage, dans notre cas octet par octet : `x/70xb 0x400` pour afficher les 70 octets après l'adresse 0x400, en format hexadécimal (`x`) et par octet (`b`). Constater que deux copies de la structure sont en mémoire, la structure déclarée dans la fonction `main()` et la structure passée en argument à la fonction de calcul.**
5. **Préfixer la déclaration de la structure dans `main()` par `static` qui indique au compilateur de stocker la variable sur le tas. Que constatez vous sur l'organisation de la pile ? Afficher le contenu du début de la RAM (adresse 0x100). Que constatez vous ?**

Les diverses informations liées à un empilement de l'état du microcontrôleur sont accessibles par les diverses options de `info` : ainsi si lors de l'appel de `ma_fonction` nous savons que le *frame* correspondant est le numéro 0

```
(gdb) bt
#0  ma_fonction (ma_matrice=...) at structure.c:15
#1  0x00000116 in main () at structure.c:26
```

alors `info f 0` nous donne le détail de l'état du processeur empilé et `info args` fournit les arguments qui ont été passés par la pile.

6 Cas du PC

`gdb` peut évidemment s'utiliser sur PC, et un cas particulier de travailler sur un système d'exploitation tient en sa capacité de capturer l'état de la mémoire en cas de dysfonctionnement d'un programme. Par défaut, cette fonctionnalité est désactivée sur les versions actuelles de GNU/Linux, et doit être activée par `ulimit -c unlimited`.

De cette façon, un programme fort maladroitement écrit tel que

```
1 int main(){
2   int k=0x1234;
3   char *c;
4   c=(char*)k; *c=3;
5 }
```

se traduit par une violation de segment mémoire et induit une copie dans un fichier `core` de l'état de la mémoire au moment du crash : `Segmentation fault (core dumped)`. Ce fichier s'analyse par `gdb ./a.out` et l'état de la mémoire se charge par la commande `core` de `gdb` : `core core`. Ainsi, nous pouvons afficher la valeur de `k` (`print k`), et si nous demandons à afficher le contenu de `c` nous constatons que `Cannot access memory at address 0x1234`. Cette méthode d'analyse permet donc de connaître la séquence qui a mené au crash et donc au moins à la fonction, dans un programme complexe, défectueuse.

7 Cas de l'émulateur pour architecture ARM : `qemu`

Dans la même lignée, un émulateur de processeur 32 bits décrivant un ARM Cortex-M3 est disponible sous forme de `qemu` et en particulier sa déclinaison `qemu-arm-softmmu/qemu-system-arm`. Sans entrer dans les détails de son implémentation et des périphériques associés au cœur du processeur. Comme tout bon outil opensource, un avantage majeur est de pouvoir combler les lacunes du programme fourni par son auteur – dans notre cas par l'ajout de périphériques manquants et/ou par l'ajout de message à l'utilisateur en cas d'usage abusif de certains registres ou périphériques. Dans l'exemple qui va suivre, nous allons tenter d'accéder au port série (USART) en ayant omis de l'initialiser. Le programme que nous testons est proposé dans les exemples de test de l'émulateur disponible à `https://github.com/beckus/stm32_p103_demos.git`

```
1  #include "stm32f10x.h"
2  #include "stm32/usart.h"
3  #include "stm32/gpio.h"
4  #include "stm32/rcc.h"
5
6  void Usart1_Init(void)
7  {USART_InitTypeDef usart_i;
8   USART_ClockInitTypeDef usart_c;
9   GPIO_InitTypeDef   gpio_i;
10
11  GPIO_PinRemapConfig(GPIO_Remap_USART1,DISABLE);
12  RCC_APB2PeriphClockCmd( RCC_APB2Periph_AFIO, ENABLE);
13  RCC_APB2PeriphClockCmd( RCC_APB2Periph_USART1, ENABLE);
14  RCC_APB2PeriphClockCmd( RCC_APB2Periph_GPIOA, ENABLE);
15
16  gpio_i.GPIO_Pin = GPIO_Pin_9;
17  gpio_i.GPIO_Speed = GPIO_Speed_50MHz;
18  gpio_i.GPIO_Mode = GPIO_Mode_AF_PP;
19  GPIO_Init( GPIOA, &gpio_i);
20
21  gpio_i.GPIO_Pin = GPIO_Pin_10;
22  gpio_i.GPIO_Speed = GPIO_Speed_50MHz;
23  gpio_i.GPIO_Mode = GPIO_Mode_IN_FLOATING;
24  GPIO_Init( GPIOA, &gpio_i);
25
26  usart_i.USART_BaudRate = 115200;
27  usart_i.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
28  usart_i.USART_Mode  = USART_Mode_Rx | USART_Mode_Tx;
29  usart_i.USART_Parity  = USART_Parity_No;
30  usart_i.USART_StopBits  = USART_StopBits_1;
31  usart_i.USART_WordLength = USART_WordLength_8b;
32
33  usart_c.USART_Clock   = USART_Clock_Enable;
34  usart_c.USART_CPHA   = USART_CPHA_1Edge;
```

```
 usart_c.USART_CPOL   = USART_CPOL_Low;
 usart_c.USART_LastBit   = USART_LastBit_Disable;

 USART_ClockInit(USART1, &usart_c);
 USART_Init(USART1, &usart_i);
 USART_Cmd(USART1,ENABLE);
}

void Led_Init(void)
{GPIO_InitTypeDef GPIO_InitStructure;

 RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);
 RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);

 GPIO_WriteBit(GPIOC,0x00000000,Bit_SET);
 GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1|GPIO_Pin_2 |GPIO_Pin_8|GPIO_Pin_9;
 GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
 GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
 GPIO_Init(GPIOC, &GPIO_InitStructure);
}


void uart_putc1(char c)
{while(USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET);
 USART_SendData(USART1, c);
}

void uart_puts1(char *c) {while(*c!=0) uart_putc1(*(c++));}

int main(void)
{ int i, c = 0,k=0;
  Led_Init();
  Usart1_Init(); // usart_setup();
  while (1) {
    if (k==1) GPIO_SetBits (GPIOC, GPIO_Pin_9);
    else GPIO_ResetBits (GPIOC, GPIO_Pin_9);
    k=1-k;
    c = (c == 9) ? 0 : c + 1; // cyclic increment c
    uart_putc1(c + '0'); // USART1: send byte
    for (i = 0; i < 80000; i++) __asm__("NOP");
  }
  return 0;
}
```

On notera dans cet exemple une subtilité de mise en œuvre des périphériques du STM32 : chaque périphérique doit voir son horloge activée, faute de quoi le périphérique est inactif. Cependant sur du matériel, inactif ne signifie pas retour d'erreur – au contraire, le microcontrôleur attendra dans une boucle infinie l'activation d'un bit de statut (par exemple pour valider la disponibilité d'un périphérique tel que le port série dans la boucle

```
while(USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET);
```

et ne rendra jamais la main, donnant l'impression d'un plantage. Un émulateur peut alors être utile pour tester l'activation de chaque horloge avant l'utilisation du périphérique associé.

Ainsi,

```
$ ./arm-softmmu/qemu-system-arm -M stm32-p103 -serial tcp::7777,server -kernel ../stm32_p103_demos/demos/qemu_test/main.bin

QEMU waiting for connection on: tcp:0.0.0.0:7777,server
ADC interrupt occured
ADC interrupt occured
jmf : ADC connect
VNC server running on `::1:5900'
ADC interrupt occured
ADC interrupt occured
LED C8 Off
LED C9 Off
LED C12 Off
qemu stm32: hardware warning: Warning: You are attempting to use the UART2 peripheral while its clock is disabled.
```

```
R00=40004400 R01=00000031 R02=00000031 R03=40004400
R04=00000000 R05=00000000 R06=00000000 R07=20004fc8
R08=00000000 R09=00000000 R10=00000000 R11=00000000
R12=00000000 R13=20004fc8 R14=00002a0f R15=00001dac
PSR=20000173 --C- T svc32
qemu: hardware error: Attempted to write to USART_DR while UART was disabled.
CPU #0:
R00=40004400 R01=00000031 R02=00000031 R03=40004400
R04=00000000 R05=00000000 R06=00000000 R07=20004fc8
R08=00000000 R09=00000000 R10=00000000 R11=00000000
R12=00000000 R13=20004fc8 R14=00002a0f R15=00001dac
PSR=20000173 --C- T svc32
FPSCR: 00000000
Aborted
```

se traduit par les deux messages d'erreur lorsque nous commentons la ligne `Usart1_Init();` du code source. Dans cet exemple, nous avons lancé l'émulateur et affiché la sortie du port série sur le port 7777 d'un scoket local (adresse IP 127.0.0.1). Nous aurions pu préférer renvoyer le port série vers la sortie standard (*stdout*) au moyen de

```
.arm-softmmu/qemu-system-arm -M stm32-p103 -serial stdio -serial stdio -kernel ../stm32_p103_demos/demos/qemu_test/main.bin
```

Plus utile pour déverminer la cause du dysfonctionnement, l'exécution peut se faire sous le contrôle de `gdb` en fournissant les options `-S -s` afin de créer un port de liaison vers `gdb` sur le port 1234 du localhost.

Si les instructions d'initialisation des USARTs sont actives, nous obtenons la sortie prévue en cas de "bon" fonctionnement du programme, à savoir

```
$ ./arm-softmmu/qemu-system-arm -M stm32-p103 -serial stdio -serial stdio -kernel ./demos/qemu_test/main.bin
ADC interrupt occured
ADC interrupt occured
jmf : ADC connect
VNC server running on '::1:5900'
ADC interrupt occured
ADC interrupt occured
LED C8 Off
LED C9 Off
LED C12 Off
1LED C9 On
2LED C9 Off
3LED C9 On
4LED C9 Off
5LED C9 On
6LED C9 Off
7LED C9 On
[...]
```

Références

[1] J.-M Friedt, S. Guinot, *Programmation et interfaçage d'un microcontroleur par USB sous linux : le 68HC908JB8*, GNU/Linux Magazine France, Hors Série **23** (November/Décembre 2005)

[2] Atmel, *Atmel AVR 8-bit Instruction Set - Atmel Corporation,* à `www.atmel.com/images/atmel-0856-avr-instruction-set-manual.pdf` (version 0856J-AVR-07/2014)

[3] Atmel, *8-bit Microcontroller with 16/32K Bytes of ISP Flash and USB Controller – ATmega16U4 & ATmega32U4*, disponible à `www.atmel.com/Images/doc7766.pdf` (version 776F–AVR–11/10)

[4] Atmel, *AVR109 : Self Programming* (version Rev. 1644G–AVR–06/04), disponible à `http://www.atmel.com/images/doc1644.pdf`