Liaison série asynchrone sur émulateur d'Atmega32U4

É. Carry, J.-M Friedt, 13 mars 2019

Connexion internet autorisée, téléphones portables interdits, communications interdites, réflexion autorisée.

Nous avons vu que la liaison USB est complexe et requiert une bibliothèque gourmande en ressources, LUFA. Nous avons mentionné que tout circuit électronique numérique comportant un microcontrôleur est muni d'une interface de communication très simple, bien que plus lente que USB, implémentant le protocole série asynchrone RS232 [1] qui permet de démarrer le système embarqué avant qu'un système d'exploitation ou une bibliothèque lourde ne donne accès aux protocoles de communication rapides mais complexes. Nous nous proposons d'exploiter le protocole RS232 pour quelques applications de communication de base, et ce non sur microcontrôleur physique mais sur émulateur. En effet, `simavr` émule le comportement d'un Atmega(32U4) et fournit, en plus des ressources disponibles sur le "vrai" composant, accès à l'état de ses registres internes.

Nous commençons par nous familiariser avec `simavr` et les options de compilation associées, avant de nous lancer dans RS232. Un programme fonctionnel pour Atmega32U4 émulé par `simavr` est mis à disposition dans `/home/jmfriedt/exam2019_ep2.tar.gz`

1. Démontrer votre capacité à compiler ce programme et l'exécuter par `simavr -f 16000000 -m atmega32u4 programme.out` avec l'option `-f` la fréquence du processeur et `-m` le type de machine. On pourra noter qu'un fichier d'entête nécessaire à déclarer des constantes spécifiques à l'émulateur se trouvent dans `/usr/local/include/simavr/avr` : cette information peut ou ne pas être utile.

   Sachant que l'émulateur a été modifié pour indiquer quel port se voit assigner quelle valeur, indiquer quelle broche de quel port est manipulée par ce programme (on se référera à la documentation technique pour cette identification).
2. Ayant acquis la maîtrise de `simavr` et des options de compilation associées, nous voulons maintenant comprendre comment communiquer par RS232. Pour ce faire, un ensemble de fonctions sont fournies dans `uart.*` de l'archive sus-citée. Initialiser le port série et démontrer votre capacité à afficher toutes les 100 ms le caractère "A" en exécutant votre code sur `simavr`.
3. Afin de communiquer des grandeurs pertinentes, nous désirons afficher à l'écran le contenu d'une variable codée sur 16 bits. Déclarer une telle variable, l'affecter de 0x1234, et démontrer l'affichage du contenu de cette variable lorsque votre programme est exécuté sur émulateur.
4. Exploiter la fonction développée ci-dessus pour afficher le contenu d'une variable sur 32 bits. Comment est déclarée cette variable ? comment vérifier que la taille allouée est la bonne ? Initialiser le contenu de la variable avec une constante suffisamment grande pour que le nombre d'octets garantisse un état connu de chaque bit de la variable.
5. La variable qui vient d'être déclarée se trouve quelque part en mémoire. Afficher l'adresse mémoire de l'emplacement de cette variable.
6. Le compilateur nous informe d'une incohérence potentielle dans le programme. Quelle est-elle et comment analyser le commentaire du compilateur ?
7. Compte tenu de l'organisation de la mémoire décrite en page 18 de la documentation technique, cette valeur d'emplacement paraît-elle plausible ? justifier.
8. Comment faire pour placer cette variable en début de RAM au lieu de l'emplacement identifié auparavant ?
9. La liaison asynchrone fait une hypothèse sur le cadencement des deux interlocuteurs : quelle est cette hypothèse ? Dans notre exemple, nous avons proposé un certain débit de communication : quel est-il, et combien d'octets permet-il de transmettre par seconde ?
10. `simavr` fournit la capacité à générer des chronogrammes des signaux internes au microcontrôleur. À chaque simulation, un fichier `trace_file.vcd` a été généré. Ce fichier s'analyse au moyen de `gtkwave` (tout comme les chronogrammes de simulation du comportement d'un code VHDL sur FPGA). Une fois `gtkwave` lancé, cliquer sur `logic` pour faire apparaître les signaux acquis dans la fenêtre en-dessous. Faire glisser chacun des deux signaux dans la fenêtre principale des chronogrammes, et cliquer sur la loupe munie d'un carré pour se placer pleine échelle. Observer la durée de transmission d'un octet sur UDR1. Modifier alors le débit de communication pour se placer à 9600 bauds, et réitérer l'expérience. Commenter sur la durée de communication observée.

1. *Hack All The Things – 20 Devices in 45 Minutes*, DEFCON 22 (2014) à `https://www.youtube.com/watch?v=u2aKrgDtfoI`

Corrections

1. La compilation s'obtient par `make`. L'exécution se traduit par

```
SIMAVR: IOPORT @0x25<-0x20
SIMAVR: IOPORT @0x25<-0x0
SIMAVR: IOPORT @0x25<-0x20
SIMAVR: IOPORT @0x25<-0x0
```

   La lecture de la datasheet nous informe que 0x25 est l'emplacement de PORTB, et 0x20=32 est le bit 5 de ce registre, donc nous manipulons la LED sur PORTB5.

2. Nous ajoutons au programme

```c
#include <avr/io.h>      // voir /usr/lib/avr/include/avr/iom36u4.h
#include <util/delay.h>  // _delay_ms
#include "uart.h"

int main(void){
  (*(volatile unsigned char*)(0x24))=32; //@24=DDRB, 32=1<<5
  (*(volatile unsigned char*)(0x25))=0;  //@25=PORTB
  init_uart();
  while (1)
   {(*(volatile unsigned char*)(0x25))^=32;
    uart_transmit('A');
    _delay_ms(100);
   }
  return 0;
}
```

   qui fait appel aux fonctions fournies dans `uart.c`, et complétons la ligne de compilation dans le Makefile par une dépendance à uart.c

```
$(EXEC).out: $(EXEC).o fonctions_simavr.o uart.o
$(CC) $(CFLAGS) -o $(EXEC).out $(EXEC).o fonctions_simavr.o uart.o

uart.o: uart.c
$(CC) $(CFLAGS) -c uart.c
```

   L'exécution de ce programme se traduit par

```
SIMAVR: IOPORT @0x25<-0x20
A
SIMAVR: IOPORT @0x25<-0x0
A
SIMAVR: IOPORT @0x25<-0x20
A
SIMAVR: IOPORT @0x25<-0x0
A
```

   qui affiche bien le symbole attendu entre deux clignotements de LED.

3. Nous ajoutons nos propres fonctions dans un nouveau fichier afin de bien distinguer notre contribution du code existant

```c
#include "uart.h"

void send_byte(unsigned char c)
{unsigned char tmp;
 tmp=c>>4;      if (tmp<10) uart_transmit(tmp+'0'); else uart_transmit(tmp+'A'-10);
 tmp=(c&0x0f); if (tmp<10) uart_transmit(tmp+'0'); else uart_transmit(tmp+'A'-10);
}

void send_short(unsigned short s) {send_byte(s>>8);send_byte(s&0xff);}

void send_string(char *c) {while (*c!=0) {uart_transmit(*c);c++;}}
```

   De cette façon, le contenu d'une variable de type `short` est initialisée à 0x1234 par `short s=0x1234;` affichée par `send_short(s);` qui se traduit par

```
SIMAVR: IOPORT @0x25<-0x20
1234
SIMAVR: IOPORT @0x25<-0x0
1234
```

4. un `long` est une variable sur 32 bits, tel qu'en atteste `send_short(sizeof(long));` qui renvoie 0004. Ainsi, une variable sur 32 bits est la concaténation de deux variables sur 16 bits, dont le contenu s'affiche dont trivialement par `send_short(s>>16); send_short(s&0xffff);` On prendra soit d'initialiser `long s=0x12345678;` avec un argument de 8 quartets pour bien démontrer la fonctionnalité de l'affichage.

5. Le pointeur vers la variable fournit l'adresse de son emplacement. La ligne
`unsigned long ad; ad=(&s); send_short(ad>>16); send_short(ad&0xffff);`
affiche l'emplacement en mémoire de `s`, soit 0x0A8F puisque

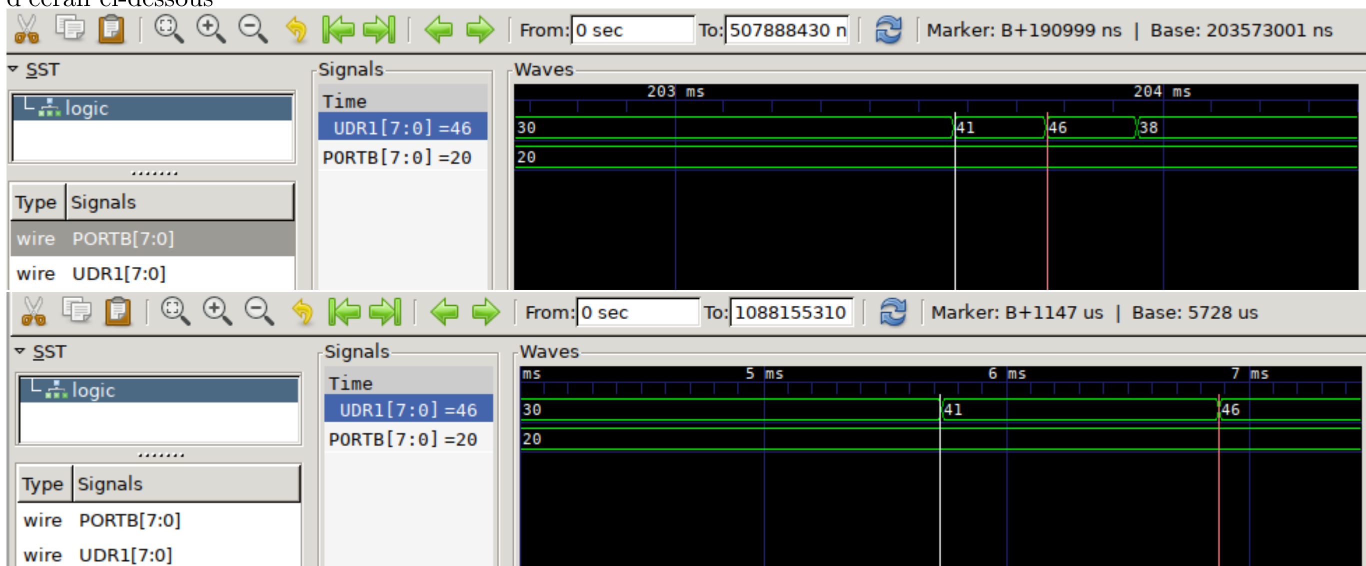
```
SIMAVR: IOPORT @0x25<-0x20
00000AF8
SIMAVR: IOPORT @0x25<-0x0
00000AF8
```

6. `warning: assignment makes integer from pointer without a cast [-Wint-conversion]` nous informe que nous assignons un scalaire avec un pointeur (emplacement mémoire d'une variable), erreur courante dont nous sommes informés par un message d'alerte (warning) pour s'assurer qu'il s'agit bien de l'opération escomptée.
7. Une telle adresse est cohérente avec la carte mémoire de l'Atmega32U4 qui nous informe que la RAM s'étend jusqu'à 0xAFF. La pile étant placée en fin de RAM, une variable qui y est allouée se voit placée dans un emplacement proche de cette borne supérieure de la plage d'adresses.

Memory		Mnemonic	ATmega32U4	ATmega16U4
Flash	Size	Flash size	32KB	16KB
	Start Address	- 0x0000		
	End Address	Flash end	0x7FFF[1] 0x3FFF[2]	0x3FFF[1] 0x1FFF[2]
32 Registers	Size	-	32 bytes	32 bytes
	Start Address	-	0x0000	0x0000
	End Address	-	0x001F	0x001F
I/O Registers	Size	-	64 bytes	64 bytes
	Start Address	-	0x0020	0x0020
	End Address	-	0x005F	0x005F
Ext I/O Registers	Size	-	160 bytes	160 bytes
	Start Address	-	0x0060	0x0060
	End Address	-	0x00FF	0x00FF
Internal SRAM	Size	ISRAM size	2,5KB	1.25KB
	Start Address	ISRAM start	0x100	0x100
	End Address	ISRAM end	0x0AFF	0x05FF

8. Si nous préfixons le nom de la variable de `static`, la durée de vie de la variable est la durée d'exécution du programme puisque placée sur le tas. Le tas est localisé en début de RAM dont l'adresse est ici 0x100 (les 0x100 premiers emplacements sont dédiés aux registres contrôlant les périphériques matériels).

L'émulateur `simavr` est capable de générer des chronogrammes exacts en temps sur des signaux que nous avons sélectionné. Dans ces exemples, nous avons observé `UDR1`, le registre de données lié au périphérique UART1. Ainsi, chaque transition de ce signal est mémorisée dans un fichier qui est analysé au moyen de `gtkwave`. Les deux captures d'écran ci-dessous



indiquent le temps de communication (changement d'état de `UDR1`) tel que indiqué par `simavr`. À 57600 bauds et un protocole 8N1 (10 symboles/octet transmis), nous transmettons 5760 octets/s ou 173,6 μs/octet. À 9600 bauds (bas), nous transmettons 960 octets/s ou 1,042 ms/octet. Ces résultats se comparent avantageusement aux 190 μs et 1,147 ms indiqués par `simavr` dans ces chronogrammes, indiquant que les simulations sont exactes en durée estimée d'exécution.