Développer sur microcontrôleur sans microcontrôleur : les émulateurs

J.-M Friedt, 15 mai 2019

Le monde fascinant du développement sur systèmes embarqués – des petits microcontrôleurs 8 bits aux processeurs capables d'exécuter GNU/Linux – est accessible même sans posséder de plateforme matérielle dédiée grâce aux émulateurs. Nous proposons un cheminement progressif du petit Atmega32U4 avec le tracé des chronogrammes des signaux internes, au STM32 programmé en C ou exécutant NuttX, pour aborder le tout nouveau RISC-V exécutant FreeRTOS et finalement voir GNU/Linux tourner sur RISC-V, MIPS ou ARM.

Il est bien connu que les développeurs de matériel n'aboutissent jamais à temps à faire fonctionner leur circuit. Que ce soit une erreur de routage, un problème d'approvisionnement ou de réalisation de circuit, le matériel n'est jamais disponible à temps. En attendant, le développeur de logiciel attend ... ou pas. Nous allons présenter quelques émulateurs de processeurs dans un contexte un peu plus serein que la lutte fratricide des hardeux contre les softeux : un émulateur permet de sonder l'état de registres difficilement accessibles sinon (traceur de signaux dans l'émulateur Atmega), voir de conseiller le développeur dans sa stratégie d'initialisation des signaux (horloges dans l'émulateur `qemu` appliqué au STM32). Cependant, un émulateur n'est jamais aussi bon que sa capacité à convenablement émuler les signaux du microprocesseur, et l'opensource prend une fois de plus tout son sens pour corriger des dysfonctionnements. Finalement, nous verrons que l'émulateur permet d'appréhender des architectures encore balbutiantes (RISC-V) ou moins courantes (MIPS) que les x86/ARM dont nous sommes abreuvés quotidiennement.

L'objectif de cet article est d'illustrer l'utilisation d'émulateurs pour permettre au lecteur ne possédant pas de matériel de se familiariser avec le développement sur microcontrôleurs. Les trois cibles logicielles sont la programmation bas-niveau en C (*baremetal*), la sur-couche FreeRTOS qui permet d'appréhender les concepts de gestion de tâches et de la cohérence de l'accès concurrent aux ressources, et finalement à GNU/Linux embarqué sur cibles MIPS et ARM. Nous suivons ainsi une évolution "logique" des niveaux d'abstractions en partant du plus bas niveau pour élever l'abstraction vers les environnements exécutifs puis le système d'exploitation.

Appréhender une nouvelle architecture prend un peu de temps et de patience pour se familiariser avec les divers registres et subtilités d'une architecture donnée. Alors qu'un petit microcontrôleur 8-bits tel que les Atmel (maintenant Microchip) Atmega n'ont pas beaucoup de surprise à livrer autre que leur architecture Harvard, un ST Microelectronics STM32 est bien plus surprenant avec sa granularité fine de distribution d'horloges désactivées par défaut. RISC-V vient d'exploser aux yeux du grand public tandis que ARM continue à lutter avec MIPS et SPARC : maîtriser ces architectures est gage de liberté et de capacité à sélectionner la meilleure plateforme pour un objectif donné. Ces diverses architectures peuvent se tester sur émulateur avant d'investir dans du matériel, tel que nous le verrons ici.

1 Microcontrôleur 8 bits Atmega32U4

1.1 Le simulateur `simavr`

`simavr` (`github.com/buserror/simavr`, aussi disponible en paquet Debian mais d'un intérêt discutable si on n'en consulte pas les sources) est un simulateur de microcontrôleurs Atmel d'architecture Harvard (Atmega) incluant l'Atmega32U4 qui équipe les plateformes Olimexino32U4 que nous utilisons pour l'enseignement introductif aux microcontrôleurs en Licence3. Ainsi, l'émulateur offre un outil pédagogique indéniable car, sans prétendre pallier la manipulation sur plateformes expérimentales openhardware au coût modeste (13 euros/unité), il permet de reproduire chez soi les travaux pratiques ne faisant pas appel à une interaction avec le matériel. `simavr` est de plus capable de générer des traces exactes en temps pour suivre l'évolution de signaux internes au microcontrôleur – fonction supportée par les sondes JTAG pour les microcontrôleurs équipés de cette fonction (broches partagées avec le port F sur l'Atmega32U4) mais avec une souplesse additionnelle de pouvoir intercepter une manipulation erronée de registres non-initialisés par exemple.

Prenons un programme trivial chargé de faire clignoter une LED connectée à une broche de port d'entrée-sortie généraliste (GPIO), ici la broche 5 du port C :

```
1 #include <avr/io.h> //E/S ex PORTB
2 #define F_CPU 16000000UL
3 #include <util/delay.h>
4 int main(void){
5   DDRC |=1<<PORTC5;
6   PORTC|=1<<PORTC5;
7   while (1){PORTC^=1<<PORTC5;_delay_ms(10000);}
8   return 0;
9 }
```

Générer les traces indiquant l'évolution d'un signal interne nécessite l'ajout, dans le code source du programme à destination du microcontrôleur, d'un entête déclarant les signaux à observer de la forme

```
#include "avr_mcu_section.h"
AVR_MCU(F_CPU, "atmega32");
const struct avr_mmcu_vcd_trace_t _mytrace[]  _MMCU_ = {
  { AVR_MCU_VCD_SYMBOL("PORTC"), .what = (void*)&PORTC, }, };
```

qui ne sera pas compilé dans le binaire final à destination du microcontrôleur. Ces lignes de code informent `simavr` que nous désirons observer le port C, et effectivement la simulation du programme proposé auparavent au moyen de la commande `simavr -f 16000000 -m atmega32u4 programme.out` se traduit par un chronogramme dans lequel le PORT C change périodiquement d'état (Fig. 1). Les arguments de `simavr` sont `-f` pour définir la fréquence du processeur et `-m` le type de processeur.

À chaque simulation, un fichier `trace_file.vcd` est 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 signaux dans la fenêtre principale des chronogrammes, et cliquer sur la loupe munie d'un carré pour se placer pleine échelle.

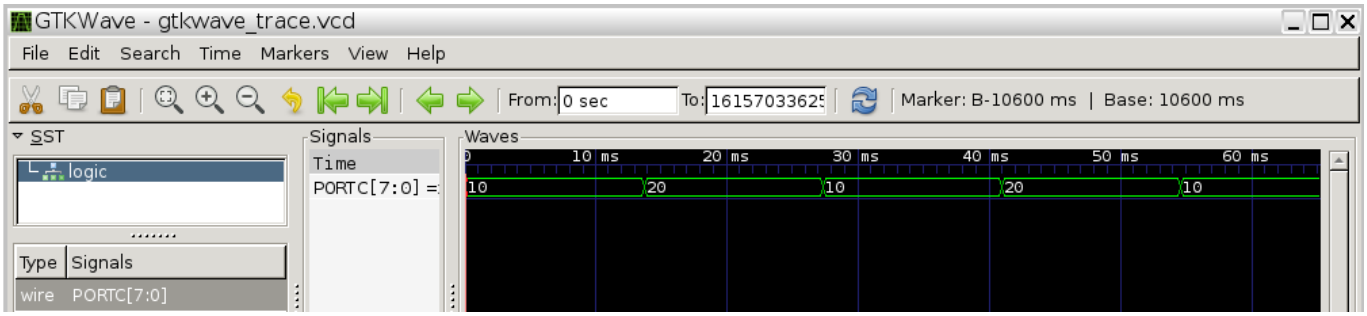


FIGURE 1 – Chronogramme issu d'une simulation `simavr` et affiché dans `gtkwave`, ici un simple clignotement toutes les 10 ms sur un GPIO.

1.2 Communications asynchrones (RS232)

Une liaison USB est complexe et requiert une bibliothèque gourmande en ressources pour déclarer ses interfaces de communication (*endpoints*). Néanmoins, tout circuit électronique numérique comportant un microcontrôleur est muni d'une interface de communication très simple et facile à déverminer, bien que plus lente que USB, implémentant le protocole série asynchrone compatible 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 donc d'observer les transaction selon 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.

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

#include "avr_mcu_section.h"
AVR_MCU(F_CPU, "atmega32");
AVR_MCU_VCD_FILE("trace_file.vcd", 1000);
const struct avr_mmcu_vcd_trace_t _mytrace[]  _MMCU_ = {
     { AVR_MCU_VCD_SYMBOL("UDR1"), .what = (void*)&UDR1, },
};
```

Ces premières lignes, en plus de définir quelques fonctions fournies par `avr-libc`, indiquent à `simavr` d'exporter la trace représentant le registre de transmissions de données de l'interface asynchrone `UDR1` situé à l'adresse 0xCE [2, p.388] : écrire une valeur dans ce registre se traduit par l'activation de la machine à états qui va générer les niveaux successifs sur la broche TXD1 (PD3) pour transmettre la donnée sans supervision explicite du logiciel.

Suivent les fonctions de communication : comme tout protocole asynchrone, nous devons déterminer le débit de transfert (*baudrate*) que nous sélectionnerons entre deux valeurs standard que sont 57600 bauds ou 9600 bauds.

```
#define UART_BAUDRATE (57600)
//#define UART_BAUDRATE (9600)

void init_uart(void)
{unsigned short baud;
 UCSR1A = (1<<UDRE1);                // importantly U2X1 = 0
```

```
 UCSR1B = (1 << RXEN1)|(1 << TXEN1); // enable receiver and transmitter
 UCSR1C = (1<<UCSZ11)|(1<<UCSZ10);   // 8N1
 // UCSR1D = 0;                      // no rtc/cts
 baud  = (((( F_CPU / ( UART_BAUDRATE * 16UL))) - 1));
 UBRR1H = (unsigned char)(baud>>8);
 UBRR1L = (unsigned char)baud;
}

void uart_transmit (unsigned char data)
{while (!( UCSR1A & (1<<UDRE1)));    // wait while register is free
 UDR1 = data;                        // load data in the register
}

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);}
```

Le port série est configuré pour communiquer 8 bits/donnée, sans bit de parité et 1 bit d'arrêt auquel on pensera à ajouter le bit implicite de départ, soient au total 10 bits par donnée transmise. Nous devrions donc attendre 1000/960=1,042 ms (9600 bauds) ou 1000/5760=0,174 ms (57600 bauds) par donnée transmise.

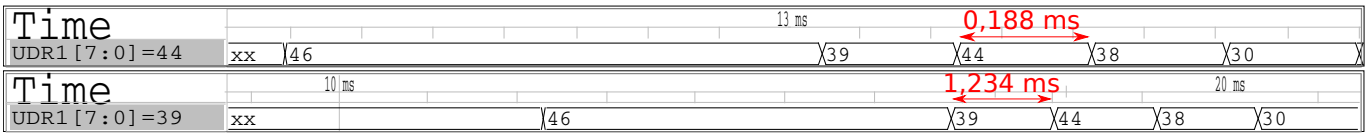


FIGURE 2 – Chronogrammes des communications asynchrones – changement d'état de `UDR1` – pour un débit de communication de 57600 bauds (haut) nécessitant 0,188 ms pour transmettre un octet, ou 9600 bauds (bas) nécessitant 1,234 ms pour transmettre un octet.

`simavr` sauve donc l'évolution du registre, dont le chronogramme est tracé en Fig. 2. Le point intéressant ici est le respect des contraintes temporelles de la communication, considérablement plus lente que tout calcul sur le microcontrôleur. Nous constatons qu'à 9600 bauds un octet est transmis en 1,234 ms (soit près de 20000 cycles d'horloge du processeur cadencé à 16 MHz) ou à 57600 bauds à 0,188 ms. Ces durées correspondent à 874 octets/seconde (contre 960 octets/seconde en théorie) ou 5313 octets/seconde (contre 5760 octets/seconde en théorie), un résultat tout à fait acceptable compte tenu des activités annexes de maintenance exécutées par ailleurs par le microcontrôleur.

1.3 Pointeur de pile

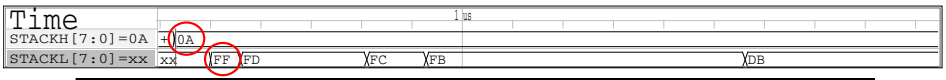
Simuler un GPIO est un peu trivial et sans intérêt pratique. Simuler un port de communication est mieux. Plus intéressant encore, les microcontrôleurs de la gamme Atmega ont la propriété de stocker le pointeur de pile dans deux registres (deux registres 8 bits pour pointer sur une adresse de 16 bits pour adresser plus de 256 octets de mémoire) qui peuvent donc être observés par `simavr` comme tout port. Pour rappel, la pile est la zone mémoire utilisée comme brouillon par le processeur, et la corruption de pile est un problème classique de plantage de programme, ne serait-ce que parce-que le processeur empile l'adresse de retour du compteur de programme pour se rappeler où continue l'exécution d'un programme qui a sauté dans une fonction ou une procédure. Comme la pile sert aussi au stockage des variables locales aux fonctions (au contraire du tas qui stocke les variables globales ou à portée de la durée de l'exécution du programme – préfixe `static` lors de la déclaration), une pile qui sort de l'espace adressable de mémoire se traduira par un plantage irrémédiable du programme : connaître l'état de sa pile est donc un élément clé pour garantir le bon fonctionnement d'un programme.

Nous nous intéressons désormais au pointeur de pile sur 16 bits défini comme deux registres de 8 bits situés aux adresses 0x3D et 0x3E [2, p.388]. Ainsi, nous modifions la configuration de la sauvegarde des traces par `simavr` en complétant le code d'initialisation par

```
#include "avr_mcu_section.h"
AVR_MCU(F_CPU, "atmega32");
AVR_MCU_VCD_FILE("trace_file.vcd", 1000);
const struct avr_mmcu_vcd_trace_t _mytrace[]  _MMCU_ = {
    { AVR_MCU_VCD_SYMBOL("UDR1"), .what = (void*)&UDR1, },
    { AVR_MCU_VCD_SYMBOL("STACKL"), .what = (void*)(0x5D), }, // stack=0x3{DE}+0x20
```

```
    { AVR_MCU_VCD_SYMBOL("STACKH"), .what = (void*)(0x5E), }, // stack=0x3{DE}+0x20
};
```

qui ajoute à UDR1 les deux octets du pointeur de pile (*stack pointer*) STACKL et (octet de poids faible) et STACKH (octet de poids fort). Nous observons (Fig. 3) que la pile est initialisée en *haut* (fin) de RAM puisque empiler une variable correspond à *décrémenter* le pointeur de pile.



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

FIGURE 3 – Initialisation du pointeur de pile à 0xAFF par `avr-gcc`, au sommet de la pile comme l'indique la carte des plages mémoires [2, p.18] de l'Atmega32U4 émulé ici.

Un exemple classique d'occupation de pile qui explose est un programme récursif dans lequel une fonction est appelée une multitude de fois, par exemple dans le cas du calcul du factoriel d'un nombre $N! = \prod_{i=1}^{N} i$.

Une telle fonction, `factoriel()`, est implémentée dans le programme qui suit. À titre pédagogique, nous attendons un peu entre deux itérations (`_delay_ms()`) pour bien séparer les appels de la fonction dans les chronogrammes, et allouons un tableau inutilisé pour illustrer l'occupation de la pile pour conserver les variables temporaires locales à une fonction.

```
int factoriel(int n)
{volatile char inutile[0x32];
 _delay_ms(1);
 PORTB^=0xff;
 if (n==1) return(1);
   else return (n*factoriel(n-1));
}

int main()
{long p;
 DDRB=0xff;
 _delay_ms(2);
 p=factoriel(8);
 _delay_ms(2);
 init_uart();
 send_short(p>>16);
 send_short(p&0xffff);
 uart_transmit('\n');
 while (1){}
}
```

Afin d'interdire à `gcc` d'éliminer le tableau inutilisé nous préfixons sa définition du mot clé `volatile` qui interdit au compilateur de faire une hypothèse sur l'utilisation de cette variable, et nous compilons avec l'option d'optimisation `-O0` pour interdire à `gcc` de transformer la fonction récursive en une itération au sein de la même fonction (remplacement de l'appel à la fonction par `call`, qui empile l'adresse du pointeur de programme, par une boucle itérée par un simple saut

`jmp` par `gcc` lors de son optimisation du code). Nous passons outre le message avertissant que la fonction `__delay_ms()` ne sera pas exacte en durée avec cette option d'optimisation, ce point n'étant pas le sujet de la démonstration.

Les chronogrammes des premières itérations sont fournis en Fig. 4. Chaque appel à la fonction `factoriel()` se traduit par un décrément du pointeur de pile de 0xD7-0x95=66 octets, un peu plus que les 0x32=48 octets nécessaires à stocker le tableau `inutile`. Cette allocation se répète au cours des trois premières itérations de l'appel à `factoriel`, occupant petit à petit la RAM disponible (l'Atmega32U4 ne propose que 2,5 KB de RAM compris entre 0x100 et 0xAFF, Fig. 3). À la quatrième itération de `factoriel`, l'octet de poids fort du pointeur de pile est décrémenté de 0x0A à 0x09.

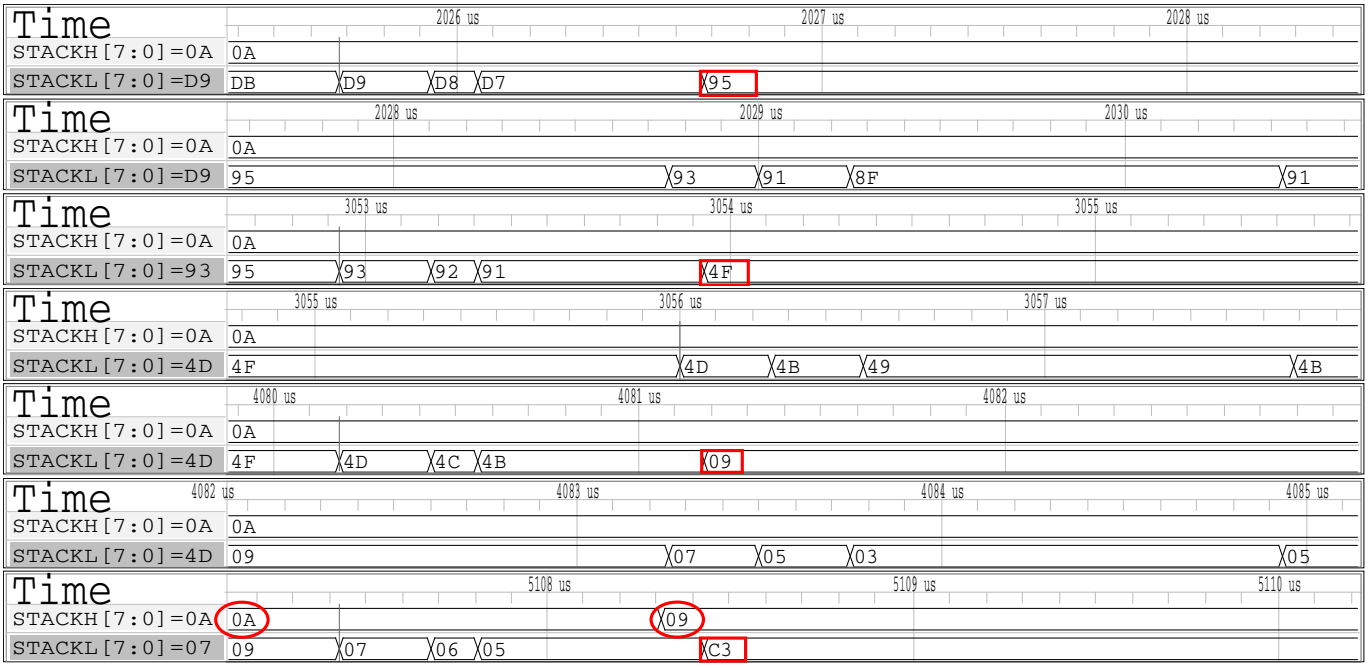


FIGURE 4 – Évolution de la pile lors des appels à la fonction `factoriel()`, avec décrément du pointeur de pile pour allouer l'espace de la variable locale `inutile` à chaque nouvel itération du calcul (rectangles rouges) jusqu'à atteindre la modification de l'octet de poids fort du pointeur de pile (rond rouge).

Finalement, les sources de `simavr` sont limpides et leur manipulation est aisée pour ajouter ses propres fonctionnalités. Par exemple pour imposer un rythme plus soutenu de communication qui ne se contente pas d'afficher les messages transmis sur port de communication asynchrone à chaque retour chariot ou lorsque le tampon est plein, nous avons décidé de modifier `simavr/sim/avr_uart.c` avec des instructions du type

```
1 if ((v>' ') && (v<127)) printf("%c",v); else printf("0x%02x",v); fflush(stdout);
```

dans les fonctions `avr_uart_udr_write()` et `avr_uart_write()`. De la même façon, si on ne veut pas passer par l'analyseur de chronogrammes mais afficher l'état des GPIOs en cours d'exécution, on manipulera `simavr/sim/avr_ioport.c` dans `avr_ioport_write()` avec

```
1 printf("\nSIMAVR: IOPORT @0x%02x<-0x%02x\n",addr,v);fflush(stdout);
```

pour fournir l'état du GPIO. Ce faisant, l'exécution de la simulation dans `simavr` se traduit par

```
$ simavr -f 16000000 -m atmega32u4 simulation_stack.out
Loaded 1956 .text at address 0x0
Loaded 0 .data
SIMAVR: IOPORT @0x25<-0xff
SIMAVR: IOPORT @0x25<-0x00
SIMAVR: IOPORT @0x25<-0xff
SIMAVR: IOPORT @0x25<-0x00
SIMAVR: IOPORT @0x25<-0xff
SIMAVR: IOPORT @0x25<-0x00
SIMAVR: IOPORT @0x25<-0xff
SIMAVR: IOPORT @0x25<-0x00
0x200x18
FFFF9D80
```

La solution 0x9d80=40320 est bien 8! tandis que l'état du port B – dont l'emplacement est à l'adresse 0x25 [2] – est immédiatement indiqué en cours d'exécution du simulateur sans devoir passer par `gtkwave`. Le lecteur peut ainsi agrémenter le simulateur de diverses conditions de validité du programme, par exemple vérifier qu'une broche a bien été configurée en sortie si son état est manipulé, ou vérifier que le débit de communication a été configuré si une liaison asynchrone est engagée.

`simavr` est donc parfaitement fonctionnel pour simuler le comportement de petits microcontrôleurs 8-bits d'architecture Harvard. Il s'agit là de cas bien particuliers ne proposant pas beaucoup de fonctionnalités au-delà de petits automates relativement simples. Plus général et surtout supportant des architectures bien plus puissantes, `qemu` ouvre un horizon bien plus vaste d'activités.

2 qemu pour STM32

L'émulateur `simavr` est fort sympathique pour un petit microcontrôleur 8 bits tels que la série des Atmega mais manque de généralité : `qemu` supporte multitude d'architectures dont le STM32 de ST Microelectronics grâce à l'effort d'André Beckus. Nous avons déjà largement présenté cette version de `qemu` – et sa déclinaison pour `eclipse` – dans [3] et ne le mentionnons ici que pour insister sur le fait que la qualité d'un émulateur ne vaut que le détail de son émulation. Nous avons récemment rencontré une erreur d'implémentation du convertisseur analogique-numérique qui avait pourtant toujours donné satisfaction mais dont l'émulation ne répondait pas aux attentes de la bibliothèque libre pour ARM Cortex `libopencm3`. Ce dysfonctionnement fut l'occasion de corriger `qemu` en conséquence : `github.com/beckus/qemu_stm32/issues/24`

Notre attrait pour `qemu` porté au STM32 tient dans les messages d'erreurs inclus lors de l'utilisation de périphériques non-initialisés. En effet, le STM32 désactive par défaut l'horloge cadençant tout périphérique, et une erreur classique de débutant sur cette plateforme est d'oublier d'initialiser la source d'horloge adéquate, se traduisant par un plantage du programme tentant de sonder le statut d'un bit qui ne changera jamais d'état si le périphérique n'est pas cadencé. L'analyse du code de l'instrument commercialisé par la société SENSeOR sur `qemu` a ainsi permis de corriger un certain nombre d'erreurs d'ordre d'initialisation de périphériques qui finissaient par tout de même fonctionner, mais au risque de rencontrer un dysfonctionnement si un événement se déclenchait au cours de cette initialisation. Prenons de nouveau un exemple très simple pour STM32F103 qui se lie avec `libopencm3` (`github.com/libopencm3/libopencm3`), que nous découpons étape par étape pour faire le lien avec l'emulateur.

2.1 C avec libopencm3

Après bien des évolutions de son arborescence, compte tenu de la multitude de plateformes ARM-Cortex supportées (M3 mais aussi M4), une solution stable semble avoir été trouvée dans laquelle les fonctions associées aux divers périphériques sont annoncées en préfixant le chemin du répertoire par le nom du processeur concerné

```
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/adc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/usart.h>
```

L'initialisation des GPIO est classique mais doit répondre aux broches simulées dans `qemu`. Le STM32 exécutera

```
void Led_Init(void) {gpio_set_mode(GPIOC,GPIO_MODE_OUTPUT_2_MHZ,GPIO_CNF_OUTPUT_PUSHPULL,GPIO12);}
void Led_Hi(void) {gpio_set  (GPIOC, GPIO12);}
void Led_Lo(void) {gpio_clear(GPIOC, GPIO12);}
```

qui fait le pendant, dans `qemu`, à la déclaration d'un GPIO par (`github.com/beckus/qemu_stm32/blob/stm32/hw/arm/stm32_p103.c#L122`)

```
led_irq = qemu_allocate_irqs(led_irq_handler, NULL, 1);
qdev_connect_gpio_out(gpio_c, 12, led_irq[0]); // GPIO C12
```

Il est ainsi extrêmement aisé de rajouter ses propres GPIO et les messages associés indiquant leur état. De la même façon, l'initialisation des ports de communication asynchrone par `libopencm3` selon

```
void Usart1_Init(void)
{// Setup GPIO pin GPIO_USART1_TX/GPIO9 on GPIO port A for transmit. */
 gpio_set_mode(GPIOA, GPIO_MODE_OUTPUT_50_MHZ,
      GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, GPIO_USART1_TX);
 usart_set_baudrate(USART1, 115200);
 usart_set_databits(USART1, 8);
 usart_set_stopbits(USART1, USART_STOPBITS_1);
 usart_set_mode(USART1, USART_MODE_TX);
 usart_set_parity(USART1, USART_PARITY_NONE);
 usart_set_flow_control(USART1, USART_FLOWCONTROL_NONE);
```

```
 usart_enable(USART1);  // PA9 & PA10 for USART1
}
```

font le pendant de l'initialisation des ports de communication dans `qemu` (`github.com/beckus/qemu_stm32/blob/stm32/hw/arm/stm32_p103.c#L129`) par

```
stm32_uart_connect((Stm32Uart *)uart2,serial_hds[0],STM32_USART2_NO_REMAP);
```

qui permet donc ici encore de facilement déclarer des ports de communication additionnels si nécessaires. Quelques fonctions de base sur le microcontrôleur simplifieront les échanges entre le programme et le terminal chargé d'afficher les messages :

```
void uart_putc(char c)  {usart_send_blocking(USART1, c);} // USART1: send byte

void affchar(char c)
{char b;
 b=((c&0xf0)>>4); if (b<10) uart_putc(b+'0'); else uart_putc(b+'A'-10);
 b=(c&0x0f);      if (b<10) uart_putc(b+'0'); else uart_putc(b+'A'-10);
}

void affshort(short s) {affchar((s&0xff00)>>8); affchar(s&0xff);}
```

Finalement le périphérique le plus intéressant mais qui a posé le plus de soucis dans son émulation : le convertisseur analogique-numérique. Son initialisation dans `libopencm3` se fait sans grande surprise :

```
void adc_setup(void)
{volatile int i;
 gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_ANALOG, GPIO0);
 gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_ANALOG, GPIO1);

 adc_power_off(ADC1); // Make sure the ADC doesn't run during config
 adc_disable_scan_mode(ADC1);
 adc_set_single_conversion_mode(ADC1);
 adc_disable_external_trigger_regular(ADC1);
 adc_set_right_aligned(ADC1);
 adc_set_sample_time_on_all_channels(ADC1, ADC_SMPR_SMP_28DOT5CYC);
 adc_power_on(ADC1);

 for (i=0;i<800000; i++) // Wait for ADC starting up.
     __asm__("nop");

 adc_reset_calibration(ADC1);
 adc_calibrate(ADC1);
}
```

alors que le convertisseur analogique-numérique n'est pas explicitement instancié par la plateforme `stm32-p103` mais fait partie intégrante du cœur du microcontrôleur décrit dans `github.com/beckus/qemu_stm32/blob/stm32/hw/arm/stm32.c` par `stm32_create_adc_dev(stm32_container, STM32_ADC1, 1, rcc_dev, gpio_dev, 0x40012400, 0);` Nous avons fait le choix, dans `github.com/beckus/qemu_stm32/blob/stm32/hw/arm/stm32_adc.c#L700` d'émuler divers comportements de convertisseur analogique-numérique selon le canal sélectionné, par exemple en fournissant une sinusoïde qui se veut exacte en période d'échantillonnage grâce à l'utilisation de la représentation du temps par `qemu` (`qemu_clock_get_ns(QEMU_CLOCK_VIRTUAL)`).

Notre contribution récente à `qemu` pour STM32 a porté sur l'analyse du code de conversion analogique numérique proposé par `libopencm3` implémenté sous forme

```
unsigned short read_adc_naiive(unsigned char channel)
{int c;
 unsigned char channel_array[16];
 channel_array[0] = channel;
 adc_set_regular_sequence(ADC1, 1, channel_array);
 adc_start_conversion_regular(ADC1);
 while (! adc_eoc(ADC1));
 return adc_read_regular(ADC1);
}
```

qui ne rendait jamais de résultat de mesure. En effet dans notre implémentation initiale de l'émulation du convertisseur, nous nous étions contenté d'annoncer la fin de conversion (`s->ADC_SR&=~ADC_SR_EOC;`) mais avions omis de relever le drapeau testé par `libopencm3` pour respecter les consignes du manuel d'utilisateur du STM32F103 [4, p.231], à savoir `s->ADC_CR2&=~ADC_CR2_SWSTART;`. Ce faisant, nous respectons bien le test de `libopencm3` qui vérifie la fin de

conversion non pas en testant le bit du même nom (EOC) mais par (github.com/libopencm3/libopencm3/blob/master/lib/stm32/common/adc_common_v1.c#L695)

```
while (ADC_CR2(adc) & ADC_CR2_SWSTART);
```

Le programme sur le microcontrôleur se conclut par l'initialisation des horloges des périphériques utilisés – problème classique de plantage d'un programme sur STM32 qui par défaut ne cadence pas ses périphériques

```
void clock_setup(void)
{rcc_clock_setup_in_hse_8mhz_out_72mhz();  // STM32F103
 rcc_periph_clock_enable(RCC_GPIOC); // Enable GPIOC clock
 rcc_periph_clock_enable(RCC_GPIOD); // Enable GPIOD clock for F4 (LEDs)
 rcc_periph_clock_enable(RCC_GPIOA); // Enable GPIOA clock
 rcc_periph_clock_enable(RCC_ADC1);
 rcc_periph_clock_enable(RCC_USART1);
}

int main(void)
{volatile int i;
 short res;
 clock_setup();
 Usart1_Init();
 Led_Init();
 adc_setup();

 while (1) {
    res=read_adc_naiive(1);
    uart_putc('0');uart_putc('x');
    affshort(res); uart_putc('\n'); uart_putc('\r');
    Led_Hi2();
    for (i = 0; i < 800000; i++) __asm__("NOP");
    Led_Lo2();
 }
 return 0;
}
```

Ce programme se traduit lors de sa simulation par

```
$ qemu-system-arm -M stm32-p103 -serial stdio -serial stdio -kernel programme.bin
LED Off
0x045B
LED On
LED Off
0x04AA
LED On
LED Off
0x04F1
```

qui démontre bien l'émulation du convertisseur analogique-numérique, de la communication asynchrone et d'une broche de GPIO. Si cependant nous sommes suffisamment maladroits pour compléter la fonction principale par un affichage sur le second port de communication asynchrone `usart_send_blocking(USART2, 'a');` nous nous faisons insulter par

```
qemu stm32: hardware warning: Warning: You are attempting to use the ADC2 peripheral while its clock is disab

R00=40004800 R01=00000061 R02=40012400 R03=10000000
R04=08000974 R05=08000974 R06=00000000 R07=200067d8
R08=00000000 R09=00000000 R10=00000000 R11=00000000
R12=00000002 R13=200067d8 R14=08000243 R15=0800089e
PSR=80000173 N--- T svc32
qemu: hardware error: Attempted to write to USART_DR while UART was disabled.
CPU #0:
```

nous indique donc que nous tentons de manipuler les registres d'un périphérique dont l'horloge n'est pas initialisée, puis que nous tentons de communiquer par un port asynchrone qui n'a pas été configuré.

De la même façon, omettre l'initialisation du convertisseur analogique-numérique en commentant `adc_setup();` se traduit par

```
qemu: hardware error: Attempted to start conversion while ADC was disabled
```

qui est suffisamment explicite pour corriger le problème. L'émulateur peut donc amener un réel bénéfice d'analyse de code en sondant l'état des registres par rapport à JTAG qui se contente de fournir le statut du matériel sans en analyser le comportement.

2.2 NuttX

Nous avions présenté NuttX [5] comme environnement exécutif visant la compatibilité POSIX avec une empreinte mémoire compatible avec les plus petits STM32. La video de `www.youtube.com/embed/4t5pO8cJU9k` présente l'émulation de NuttX dans `qemu`, qui s'avère simple à mettre en œuvre sur une plateforme ne requérant pas le support de l'horloge temps-réelle (RTC). En effet nous constatons qu'en clonant le dépôt git de NuttX à la date de rédaction de cette prose

```
git clone https://bitbucket.org/nuttx/apps.git
git clone https://bitbucket.org/nuttx/nuttx.git
```

et en configurant pour une version minimaliste du STM32F103

```
cd nuttx/
./tools/configure.sh configs/stm32f103-minimum/nsh
make
```

l'exécution dans la version d'André Beckus de `qemu` se solde immédiatement par une invite de commande du shell `nsh` :

```
$ [...]/qemu_stm32/arm-softmmu/qemu-system-arm -M stm32-p103 -serial stdio -serial stdio -serial stdio -kerne
NuttShell (NSH)
nsh>
```

En l'état nous ne pouvons pas fournir de commande à `nsh` en l'absence de gestion des entrées du clavier, mais le principe de base fonctionne.

3 `qemu` pour RISC-V

Alors qu'il semblait acquis que l'architecture ARM allait dominer le monde de l'électronique numérique embarquée avec sa panoplie de processeurs répondant aux besoins allant du petit automate aux téléphones mobiles, 2018 s'avère peut-être marquer la fin de cette hégémonie avec la mise sur le marché d'une architecture qui couvait depuis 8 ans à Berkeley. Cette architecture libre (sous licence BSD), initialement proposée comme *softcore* sur FPGA, a été présentée dans une puce silicium contrôlant un ordinateur pour la première fois au FOSDEM en 2018 (`archive.fosdem.org/2018/schedule/event/riscv/`) par SiFive, et a donné lieu à une session dédiée en 2019 (`fosdem.org/2019/schedule/track/risc_v/`). Non content de remettre en cause l'hégémonie d'ARM en lui imposant de libérer l'implémentation de certains de ses cœurs (`www.arm.com/resources/designstart/designstart-fpga`), cette nouvelle architecture libre impose à d'autre architectures de suivre la tendance, en particulier MIPS (`wavecomp.ai/mipsopen`). Les années à venir promettent donc une compétition excitante entre les "anciennes" architectures (ARM, MIPS, SPARC) et les nouveaux venus de la gamme RISC-V (Fig. 5) [6], avec laquelle il est certainement judicieux de se familiariser, d'autant plus qu'une version combinant CPU et FPGA, dans la lignée du Zynq, est proposée par Microsemi (ex-Lattice : `www.microsemi.com/product-directory/soc-fpgas/5498-polarfire-soc-fpga`).

Figure 5: Logo du projet RISC-V

En l'absence de circuit matériel implémentant un des cœurs RISC-V, nous allons nous familiariser avec le travail sur cette architecture sur l'émulateur `qemu`. RISC-V est évidemment supporté par `gcc`. Bien que GNU/Linux soit supporté sur cette architecture, nous nous focaliserons dans un premier temps sur sa programmation en C (*baremetal*) et au moyen de FreeRTOS [7] (Fig. 6).

3.1 Programmation en C (*baremetal*)

L'utilisation en C d'un microcontrôleur nécessite d'accéder directement aux registres de communication pour interagir avec l'utilisateur, puisque les bibliothèques de plus haut niveau telles que `newlib` ne peuvent pas savoir vers quel périphérique une instruction de type `printf()` vise : cette fonction fait appel à `write()` qui est implémentée, dans le cas de `newlib`, par les *stubs* qui font pointer les écritures vers le bon périphérique.

FIGURE 6 – R. Barry présente FreeRTOS, désormais acquis par Amazon Web Service, exécuté sur RISC-V. Cette session du FOSDEM a une fois de plus fait salle comble.

En l'absence de ces *stubs*, une bibliothèque se charge pour nous de transmettre les écritures et lectures vers le périphérique émulé par l'entrée-sortie standard (stdio) : `libfemto`, disponible à `github.com/michaeljclark/riscv-probe`. L'implémentation des fonctions d'entrée-sortie permettra notamment d'interagir avec le programme émulé par `qemu` qui supporte officiellement (`github.com/qemu/qemu`) RISC-V lors de la compilation par `./configure --enable-debug --target-list="riscv64-softmmu riscv32-softmmu" --disable-gtk --disable-sdl --disable-werror`.

À titre d'échauffement, on pourra valider l'exécution de l'exemple "Hello World" nommé `hello` dans `riscv-probe/build/bin/rv` en fournissant comme format de machine à `qemu` l'option `-M sifive_e` et en prenant soin de demander `-nographic` afin que l'affichage s'effectue sur le terminal courant. Une fois l'exécution achevée, nous quitterons l'émulateur du terminal comme nous le ferions depuis `minicom` par `CTRL-a` puis `x`, nous rappelant ainsi que `qemu` affiche les liaisons sur le port série et non sur un écran qui pourrait équiper le système embarqué. On s'en rappelera lors de l'émulation de GNU/Linux pour lequel nous redirigerons la console sur un port série virtuel.

Le résultat s'obtient en se plaçant dans le répertoire `riscv-probe` et en compilant toute l'arborescence

```
riscv-probe$ make
...
riscv-probe$ [...]/riscv-qemu/riscv32-softmmu/qemu-system-riscv32 -M sifive_e -nographic \
  -kernel build/bin/rv32imac/qemu-sifive_e/hello
```

qui se traduit par

```
hello world
```

Basé sur cet exemple, nous pouvons commencer à analyser le comportement de la nouvelle architecture avec par exemple la taille d'un `long` et l'organisation des octets en mémoire :

```
1 #include <stdio.h>
2 int main()
3 {long x=0x12345678;
4  char *c=(char*)&x;
5  printf("\n%d\n",sizeof(long));
6  printf("%hhx␣%hhx␣%hhx␣%hhx\n",c[0],c[1],c[2],c[3]);
7 }
```

donne après compilation, dont nous avons identifié les options en exécutant `make V=2` pour rendre Makefile verbeux, par

```
FEMTO=[...]/riscv-probe/
riscv64-unknown-elf-gcc -Os -march=rv32imac -mabi=ilp32 -mcmodel=medany -c hello.c
riscv64-unknown-elf-gcc -Os -march=rv32imac -mabi=ilp32 -mcmodel=medany -nostartfiles \
  -nostdlib -nostdinc -static -lgcc -T $(FEMTO)/env/qemu-sifive_e/default.lds \
```

```
  $(FEMTO)/build/obj/rv32imac/env/qemu-sifive_e/crt.o \
  $(FEMTO)/build/obj/rv32imac/env/qemu-sifive_e/setup.o hello.o \
  $(FEMTO)/build/lib/rv32imac/libfemto.a -o hello32
```

le résultat qu'un long est codé sur cette architecture 32 bits sur 4 octets tandis que l'affichage des octets individuels compris dans un mot codé sur plusieurs octets indique que, puisque 0x78 apparaît en premier, l'octet de poids faible est à l'adresse la plus faible. Nous sommes donc dans un modèle de processeur *little endian*, organisation des données similaires de celles adoptées par Intel sur architecture x86 afin de charger en premier les octets de poids faible et donc calculer, lors des opérations arithmétiques, les retenues en même temps que les octets de poids plus fort sont chargés dans l'unité arithmétique et logique. Cependant, en compilant en 64 bits par (noter les options `-march=` et `-mabi`

```
riscv64-unknown-elf-gcc -Os -march=rv64imac -mabi=lp64 -mcmodel=medany -c hello.c
riscv64-unknown-elf-gcc -Os -march=rv64imac -mabi=lp64 -mcmodel=medany -nostartfiles \
  -nostdlib -nostdinc -static -lgcc -T $(FEMTO)/env/qemu-sifive_e/default.lds \
  $(FEMTO)/build/obj/rv64imac/env/qemu-sifive_e/crt.o \
  $(FEMTO)/build/obj/rv64imac/env/qemu-sifive_e/setup.o hello.o
  $(FEMTO)/build/lib/rv64imac/libfemto.a -o hello64
```

nous obtenons sous la version 64 bits de qemu le résultat

```
$ [...]/riscv-qemu/riscv64-softmmu/qemu-system-riscv64 -M sifive_e -nographic -kernel hello64
8
00000078 00000056 00000034 00000012
```

qui indique cette fois qu'une variable de type `long` occupe 8 octets (64 bits) mais l'organisation reste évidemment *little endian*.

3.2 FreeRTOS

La version V10.2.0 de FreeRTOS, dont les évolutions sont décrites à `www.freertos.org/History.txt`, publiée le 25 Février 2019 supporte officiellement l'architecture RISC-V de processeur, et en particulier sa déclinaison 32 bits.

L'archive se télécharge à `sourceforge.net/projects/freertos/files/latest/download` et la démonstration pour plateforme RISC-V se trouve dans le répertoire `FreeRTOS/Demo/RISC-V-Qemu-sifive_e-FreedomStudio`. Nous y trouvons les scripts nécessaires pour compiler un exemple qu'il faut cependant modifier un peu pour les adapter à la chaîne de compilation fournie par `github.com/riscv/riscv-gnu-toolchain` – la chaîne de compilation s'obtient par `make newlib` pour éviter de compiler toutes les dépendances inutiles. Nous remplaçons en effet dans `BuildEnvironment.mk` l'appel à `riscv32-unknown-elf` par `riscv64-unknown-elf` et ajoutons dans `Makefile` l'ordre de lier le binaire sur la version 32 bits de la bibliothèque `libc` en ajoutant `LDFLAGS += -march=rv32imac -mabi=ilp32` après la première occurrence de la variable d'environnement `LDFLAGS` autour de la ligne 128. En effet, le nom du compilateur ne présume pas de la taille des registres de la cible mais c'est l'option `-march` qui détermine la nature de la cible (tout comme pour ARM lorsque nous choisissons entre Cortex M3 ou M4). Le fichier résultant de la compilation de FreeRTOS est bien un binaire à destination d'un RISC-V :

```
RISC-V-Qemu-sifive_e-FreedomStudio$ file build/FreeRTOS-simple.elf
build/FreeRTOS-simple.elf: ELF 32-bit LSB executable, UCB RISC-V, version 1 (SYSV),
 statically linked, with debug_info, not stripped
```

Lors de son exécution sous `qemu`, ce programme affiche périodiquement le même message :

```
[...]/RISC-V-Qemu-sifive_e-FreedomStudio$ riscv-qemu/riscv32-softmmu/qemu-system-riscv32 \
  -M sifive_e -nographic -kernel build/FreeRTOS-simple.elf
core freq at 8628832 Hz
Blink
Blink
Blink
[...]
```

Au-delà de cet exemple trivial de démonstration de la capacité à compiler et exécuter un programme lié à l'ordonnanceur fourni par FreeRTOS, nous avons ici l'opportunité de découvrir les subtilités du développement collaboratif et en particulier le partage des ressources. D'une part la mémoire est une denrée rare qu'il faut partager efficacement en l'absence de gestionnaire de mémoire capable de virtualiser les domaines d'adresses accessibles par chaque tâche, et d'autre part plusieurs tâches peuvent vouloir accéder à la même ressource qu'il faut arbitrer par le mécanisme classique des MutEx (accès Mutuellement Exclusif aux ressources).

L'exemple qui suit reprend intégralement le code source fourni dans [3, section 4] pour faire appel à une émulation des accès aux ressources matérielles et ainsi rendre les codes portables – un méthode classique de séparation de la

partie algorithmique du code (portable) et la partie accédant aux périphériques (dépendante de chaque architecture). Dans notre cas, `common.c` et `common.h` qui implémentent ces fonctions sont disponibles à `github.com/jmfriedt/tp_freertos/tree/master/common` et se contentent d'afficher un message sur la console pour indiquer qu'une ressource matérielle est sollicitée. Nous ne reproduisons pas ici le code dans son intégralité (voir [3]) mais précisons qu'il faut penser à fournir les diverses fonctions de maintenance requises par le portage au RISC-V de FreeRTOS :

```
1  void vApplicationMallocFailedHook( void );
2  void vApplicationIdleHook( void );
3  void vApplicationStackOverflowHook( TaskHandle_t pxTask, char *pcTaskName );
4  void vApplicationTickHook( void );
5
6  void vApplicationStackOverflowHook( TaskHandle_t pxTask, char *pcTaskName )
7  {taskDISABLE_INTERRUPTS();
8   for( ;; );
9  }
10
11 void vAssertCalled( void )
12 {volatile uint32_t ulSetTo1ToExitFunction = 0;
13  taskDISABLE_INTERRUPTS();
14  while( ulSetTo1ToExitFunction != 1 ) __asm volatile( "NOP" );
15 }
16
17 void vApplicationMallocFailedHook( void ) { for( ;; ); }
18 void vApplicationIdleHook( void ) { }
19 void vApplicationTickHook( void ) { }
```

Ces diverses fonctions appelées pour gérer les interruptions et nécessaires à l'ordonnanceur FreeRTOS sont reprises des exemples fournis dans l'archive de l'environnement exécutif sans modification. L'exécution de ce programme dans `qemu` se traduit par le résultat escompté, à savoir

```
$ make sim-qemu
qemu-system-riscv32 -kernel build/FreeRTOS-simple.elf  -nographic -machine sifive_e
core freq at 8624537 Hz
Usart_Init
Led_Init
Led_Hi1
Hello World
Led_Hi2
Led_Hi1
Led_Lo2
Hello World
Led_Lo1
Led_Hi2
Led_Lo2
```

après avoir légèrement modifié la règle d'origine du `Makefile` pour ajouter l'option `-nographic` et retiré les directives de connexion à `gdb` que sont `-s -S`.

De la même façon, nous reprenons intégralement le code permettant l'exploration des mutex de [3, section 4.3]. Pour rappel, nous avions proposé de protéger l'affichage de messages longs par mutex afin d'en garantir l'intégrité.

En l'absence de mutex, nous obtenons parfois un message corrompu sur l'affichage

```
1111111111111111111111111111111111111111
11111111111111111111111111222222222222222222222222222222222222222222
1111111111111111
11111111111111111111111111111111111111111
```

qui ne survient jamais si nous prenons soin de protéger les appels à `uart_puts()` par les mécanismes appropriés implémentés au niveau de l'ordonnanceur par `xSemaphoreTake()` et `xSemaphoreGive()`.

Finalement, FreeRTOS permet d'afficher la liste des tâches, leur état et leur occupation de mémoire volatile sous forme d'allocations sur la pile, ainsi que de placer une signature (*watermark*) en fin de pile pour en détecter la corruption [3, 4.1]. Ces fonctionnalités sont évidemment toujours opérationnelles sur architecture RISC-V tel que nous nous en convainquons en activant les fonctions d'affichage de la liste des tâches par

```
#define configUSE_TRACE_FACILITY        1
#define configUSE_STATS_FORMATTING_FUNCTIONS    1
```

en pensant à retirer l'option par défaut qui désactive ces fonctions (`#define configUSE_TRACE_FACILITY 0`) et une fois FreeRTOS configuré de cette façon (`make clean && make`), l'appel à la fonction `vTaskList(c);` affiche la liste des fonctions et leur statut :

```
Led_Lo2
1111111111111111111111111111111111111111
2222222222222222222222222222222222222222
1111111111111111111111111111111111111111
2222222222222222222222222222222222222222

Hello World
ps              X       1       284     5
IDLE            R       0       22      6
t1              B       1       117     3
t2              B       1       117     4
LedFlash        B       2       73      2
LedFloat        B       1       73      1
Tmr Svc         B       6       20      7
Led_Hi1
1111111111111111111111111111111111111111
2222222222222222222222222222222222222222
Led_Hi2
```

Ainsi, tous les exemples proposés dans [3] sont reproductibles sur émulateur sans avoir besoin d'acquérir le matériel correspondant.

4 GNU/Linux sur ARM, MIPS, et RISC-V

Nous ne pouvons conclure ce survol des émulateurs sans mentionner le support de GNU/Linux sur des architectures que nous ne possédons pas. Ainsi le processeur ARM inclus dans le Zynq qui équipe la Redpitaya est émulé par `qemu-system-arm` si nous exécutons `qemu-system-arm -append "console=ttyPS0,115200 root=/dev/mmcblk0 rw earlyprintk" -M xilinx-zynq-a9 -m 1024 -serial mon:stdio -dtb zynq-red_pitaya.dtb -nographic -kernel uImage -sd rootfs.ext4` lorsque nous compilons une image de GNU/Linux au moyen de `buildroot` tel que décrit à `github.com/trabucayre/redpitaya`.

Au-delà de l'aspect pédagogique d'exécuter GNU/Linux sur une plateforme exotique, cette approche permet d'exécuter des `firmware` disponibles uniquement sous forme binaire pour en sonder les vulnérabilités. C'est l'approche sélectionnée par [8] pour analyser les logiciels embarqués des caméras communiquant sur IP et détecter les failles de sécurité. Ainsi, `http://www.devttys0.com/2011/09/exploiting-embedded-systems-part-3/` décrit comment exécuter un logiciel embarqué de routeur wifi basé sur un processeur MIPS.

Finalement, RISC-V est officiellement supporté par `buildroot` : la configuration par `make qemu_riscv64_virt_defconfig` permet de générer une série de fichiers pour exécuter GNU/Linux sur `qemu`. L'ordre d'exécution est un peu verbeux avec

```
qemu-system-riscv64 -M virt -kernel output/images/bbl -append "root=/dev/vda ro console=ttyS0"
-drive file=output/images/rootfs.ext2,format=raw,id=hd0 -device virtio-blk-device,drive=hd0 -device
virtio-net-device,netdev=vmnic -netdev tap,id=vmnic,ifname=vnet0 -nographic
```

en supposant que nous lançons `qemu-system-riscv64` depuis le répertoire `buildroot` puisque tous les fichiers générés par la compilation sont supposés situés dans `output/images`. Le fichier `bbl` est le *Berkeley Boot Loader*.

Lors de la compilation d'un nouveau noyau par `buildroot`, il est judicieux d'effacer l'ancienne version du *bootloader* par `rm -rf output/build/riscv-pk` afin de forcer sa recompilation à l'issue de la compilation du nouveau noyau.

Nous pourrons rapidement nous convaincre que l'architecture cible est la bonne en consultant le contenu de `/proc/cpuinfo` qui indique `rv64imafdcsu` : nous sommes sur la bonne voie.

Alors que la programmation en espace utilisateur ne présente aucune difficulté autre que d'identifier le compilateur dans `output/host/usr/bin` de `buildroot` comme nommé `riscv64-linux-gcc`, la partie module noyau est toujours un peu plus intéressante. En effet nous constatons après avoir exécuté GNU/Linux sur `qemu` que le *devicetree* contient un certain nombre de nœuds tel que décrits dans `/sys/firmware/devicetree/base/` et en particulier le périphérique `virtio_mmio`. Bien que l'outil `devmem` permette de sonder l'état des divers registres de ce périphérique depuis l'espace utilisateur et de vérifier la cohérence de leur contenu avec la documentation technique de `docs.oasis-open.org/virtio/virtio/v1.0/csprd01/virtio-v1.0-csprd01.html` qui dit que le premier registre (adresse 0x00 lue par `devmem 0x10001000`) contient l'identifiant 0x74726976, il est toujours plus satisfaisant d'accéder aux ressources matérielles depuis un module noyau. Dans notre cas, l'exemple trivial d'accès au périphérique matériel depuis le noyau est implémenté par

```c
#include <linux/module.h>       /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */
#include <linux/init.h>         /* Needed for the macros */
#include <linux/ioport.h>       // request_mem
#include <linux/io.h>           // ioremap

#define IO_BASE1 0x10001000

static void __iomem *jmf_gpio;   //int jmf_gpio;

int hello_start(void);
void hello_end(void);

int hello_start()
{unsigned int stat;
 if (request_mem_region(IO_BASE1,0x12c,"GPIO cfg")==NULL)
     printk(KERN_ALERT "mem request failed");
 else
  {jmf_gpio=(void __iomem*)ioremap(IO_BASE1, 0x4);
   stat=readl(jmf_gpio+0x0);
   printk("stat=%x\n",stat);
   release_mem_region(IO_BASE1, 0x12c);
  }
 return 0;
}

void hello_end() {printk(KERN_INFO "Goodbye\n");}

module_init(hello_start);
module_exit(hello_end);
MODULE_LICENSE("GPL");
```

qui utilise `ioremap()` pour convertir l'adresse physique en adresse virtuelle, et se compile par

```
obj-m +=hello.o
all:
   make ARCH=riscv CROSS_COMPILE=riscv64-buildroot-linux-gnu- -C \
/home/jmfriedt/buildroot-riscv/output/build/linux-8fe28cb58bcb235034b64cbbb7550a8a43fd88be/ \
M=$(PWD) modules
```

pour donner

```
# insmod hello.ko
[  284.816000] stat=74726976
```

Nous pourrons donc sereinement tester nos développements noyau sur cette nouvelle architecture sans avoir accès au matériel.

5 Séparation de l'algorithme et accès au matériel

Nous avons mentionné l'utilisation de `qemu` pour auditer le code d'un système embarqué commercialisé par SENSeOR et qualifier la bonne initialisation des divers périphériques. Une architecture de développement qui allège considérablement le portage d'un logiciel embarqué (*firmware*) d'une architecture à une autre est une séparation rigoureuse de la partie algorithmique (portable) de la partie accédant aux périphériques matériels (fortement dépendante de chaque cible). Ce faisant, nous bénéficions par ailleurs de la capacité à fournir des signaux synthétiques aux diverses entrées simulées et ainsi tester des configurations difficiles à reproduire expérimentalement, voir rejouer des scénarios de mesure enregistrés sur du matériel afin d'observer le comportement de divers algorithmes. À titre d'exemple de la conversion analogique-numérique, alors que le code pilotant les convertisseurs sur du matériel s'écrit de la forme

```c
unsigned short interroge (unsigned short puissance, unsigned int freq,unsigned int offset,unsigned char chan)
{ unsigned int v12;
  FTW0[1] = (freq & 0xFF000000) >> 24;
  FTW0[2] = (freq & 0xFF0000) >> 16;
  FTW0[3] = (freq & 0xFF00) >> 8;
  FTW0[4] = (freq & 0xFF);
[...]
  v12 = readADC12 ();
```

```
  readerF2_CLR;                    // coupe reception
  TIM_ITConfig (TIM3, TIM_IT_Update, ENABLE);
  return (v&0x03F);
}
```

son émulation, par exemple sur PC, exploitera le même prototype de fonction mais fournira des données synthétiques par exemple par

```
unsigned short interroge(unsigned short puissance,unsigned int freq, \
  __attribute__((unused))unsigned int offset,__attribute__((unused))unsigned char chan)
{float reponse;
 reponse =exp(-(((float)freq-f01)/df)*(((float)freq-f01)/df))*3100.;
 reponse+=exp(-(((float)freq-f02)/df)*(((float)freq-f02)/df))*3100.;
 reponse+=(float)((rand()-RAND_MAX/2)/bruit); // ajout du bruit;
 if (reponse<0.) reponse=0.;
 if (reponse>4095.) reponse=4095.;
 usleep(60);
 return((unsigned short)reponse);
}
```

Nous constatons dans cet exemple que nous renvoyons la réponse bruitée attendue par le dispositif sondé lors d'une mesure, et tentons de respecter les contraintes temporelles par une temporisation de la durée de la programmation par SPI du périphérique émettant un signal (registres FTW0 dans l'implémentation matérielle) et recevant la réponse du dispositif sondé (`readADC12()`). Ce faisant, nous pouvons aussi profiter des tests unitaires sur la partie algorithmique du code puisque cette partie ainsi que l'émulation des accès aux périphériques (*mock* pour nommer l'émulation de l'accès aux périphériques matériels) seront compilables sur PC.

6 Conclusion

Même si le coût du matériel pour s'approprier les techniques de développement sur système embarqué ne cessent de chuter pour n'être plus que de quelques euros pour nombre de cartes d'évaluation, il est parfois souhaitable de tester une nouvelle architecture sans acquérir le matériel associé, ne serait-ce que pour les architectures encore difficiles à acquérir. Par ailleurs, nous avons montré comment un émulateur, en donnant accès au cœur du processeur, propose des fonctionnalités additionnelles aux *debuggers* logiciels ou matériels (JTAG) en permettant d'introduire des messages liés aux initialisations erronées de périphériques. Enfin, les émulateurs sont devenus tellement performants que même un système d'exploitation aussi complexe que GNU/Linux peut y être exécuté, par exemple pour analyser des logiciels fournis sous forme de binaire sur une plateforme exotique.

7 Remerciement

La référence bibliographique qui n'est pas librement disponible sur le web a été acquise auprès de Library Genesis à `gen.lib.rus.ec` depuis le réseau universitaire Renater, une gare SNCF, au travers de TOR ou en exploitant comme DNS 9.9.9.9. Le tribunal qui s'est abaissé à légitimer l'attaque des éditeurs commerciaux contre cette ressource indispensable à nos activités quotidiennes de recherche et développement aurait pu se poser la question de l'applicabilité de sa décision.

Références

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

[2] *ATmega16U4/ATmega32U4 8-bit Microcontroller with 16/32K bytes of ISP Flash and USB Controller*, à `ww1.microchip.com/downloads/en/devicedoc/atmel-7766-8-bit-avr-atmega16u4-32u4_datasheet.pdf` version Atmel-7766J-USB-ATmega16U4/32U4-Datasheet_04/2016

[3] Q. Macé, J.-M Friedt, *FreeRTOS : application à la réalisation d'un analyseur de réeseau numéerique sur STM32*, GNU/Linux Magazine France **207** (2017) à `http://jmfriedt.free.fr/lm_freertos.pdf`

[4] *RM0008 Reference Manual* à `www.st.com/resource/en/reference_manual/cd00171190.pdf`

[5] G. Goavec-Merou, J.-M. Friedt, *Un environnement exécutif visant la compatibilité POSIX : NuttX pour contrôler un analyseur de réseau à base de STM32*, GNU/Linux Magazine France **210** (Dec. 2017) à `connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-210/Un-environnement-executif-visant-la-compatibilite-POSIX-NuttX-pour-controler-un-analyseur-de-reseau-a-bas`

[6] D.A. Patterson & J.L. Hennessy, *Computer organization and design – the hardware/software interface, RISC-V Edition*, Elesevier-Morgan Kaufmann (2018)

[7] R. Barry, *FreeRTOS on RISC-V – Running the FreeRTOS kernel in RISC-V emulators and RISC-V hardware*, FOSDEM 2019 RISC-V devroom à `fosdem.org/2019/schedule/event/riscvfreertos/`

[8] C. Heffner, *Exploiting Network Surveillance Cameras Like a Hollywood Hacker*, Black Hat (2013) à `www.youtube.com/watch?v=B8DjTcANBx0` et le résumé disponible à `media.blackhat.com/us-13/US-13-Heffner-Exploiting-Network-Surveillance-Cameras-Like-A-Hollywood-Hacker-WP.pdf` qui mentionne "Analysis and extraction of these firmware images was accomplished using Binwalk and the Firmware-Mod-Kit, while disassembly, emulation and debugging of code was performed using IDA Pro and Qemu. [...] It is worth noting that such tools allow attackers to identify and exploit vulnerabilities in embedded systems without ever purchasing a target device for testing."