Correction
-
TP 3
Interface Logiciel Matériel

Objectif

Ce TP portait sur trois aspects de l'interfaçage entre logiciel et matériel : la manipulation des registres, l'utilisation des interruptions et l'utilisation des timer.

Il est également l'occasion d'introduire dans la chaîne de compilation une bibliothèque tierce qui simule l'acquisition d'imagettes à la manière ce qui a été vu dans le TP2.

Modifier un registre

`registerAccess.h` est le fichier d'en-tête des fonctions développées pour modifier un bit dans un registre. Il contient aussi la fonction de test.

```
#ifndef REGISTERACCESS_H_
#define REGISTERACCESS_H_

#include <stdint.h>

/**
 * Met à 1 un bit d'un mot (4 octets). Si le bit visé est déjà à 1, rien n'est fait.
 * @param p_register pointeur sur le mot à modifier
 * @param bit_index numéro du bit à passer à 1. Une valeur supérieur à 31 ne modifira
rien.
 */
void setBit(volatile uint32_t* p_register, uint8_t bit_index);

/**
 * Met à 0 un bit d'un mot (4 octets). Si le bit visé est déjà à 1, rien n'est fait.
 * @param p_register pointeur sur le mot à modifier
 * @param bit_index numéro du bit à passer à 1. Une valeur supérieur à 31 ne modifira
rien.
 */
void clearBit(volatile uint32_t* p_register, uint8_t bit_index);

/**
 * Teste différentes valeurs de paramètres pour setbit() et clearBit().
 * @return Retourne 0 si les fonctions ont toujours eu l'effet prévu. Retourne le numéro
du dernier test en échec sinon.
 */
int test_registerAccess();

#endif
```

`registerAccess.c` est le fichier source des fonctions développées pour modifier un bit dans un registre ainsi que de la fonction de test.

```c
#include "registerAccess.h"


int test_registerAccess(){
	uint32_t reg = 0;
	unsigned int error = 0;

	// -- set low bits

	setBit(&reg, 0);
	if(reg!=1)
		error = 1;

	setBit(&reg, 1);
	if(reg!=2+1)
		error = 2;

	setBit(&reg, 7);
	if(reg!=128+2+1)
		error = 3;

	// rewrite the same bit
	setBit(&reg, 7);
	if(reg!=128+2+1)
		error = 4;

	// -- clear low bits

	clearBit(&reg, 0);
	if(reg!=128+2)
		error = 5;

	clearBit(&reg, 1);
	if(reg!=128)
		error = 6;

	// clear already cleared bit
	clearBit(&reg, 2);
	if(reg!=128)
		error = 7;

	clearBit(&reg, 7);
	if(reg!=0)
		error = 8;

	// -- set high bits

	setBit(&reg, 30);
	if(reg!=0x40000000)
		error = 10;

	setBit(&reg, 31);
	if(reg!=0xC0000000)
		error = 11;

	// set out of bound bit
	setBit(&reg, 32);
	if(reg!=0xC0000000)
		error = 12;

	// -- clear high bits

	clearBit(&reg, 30);
	if(reg!=0x80000000)
		error = 13;

	clearBit(&reg, 31);
	if(reg!=0x00000000)
		error = 14;

	reg = 0xffffffff;

	clearBit(&reg, 32);
	if(reg!=0xffffffff)
		error = 15;

	clearBit(&reg, 60);
	if(reg!=0xffffffff)
		error = 16;

	return error;
}

void setBit(volatile uint32_t *p_register, uint8_t bit_index){
	if(bit_index < sizeof(uint32_t)*8)
		*p_register = *p_register | (1 << bit_index);
}

void clearBit(volatile uint32_t *p_register, uint8_t bit_index){
	if(bit_index < sizeof(uint32_t)*8)
		*p_register = *p_register & ~(1 << bit_index);
}
```

La fonction de test travaille sur une variable locale et non un registre car il n’est pas toujours possible de voir les écritures faites sur un registre en lisant celui-ci. De par la fonction de certains bits de registres, leur valeurs peuvent changer sans écriture du programme, ou même ne pas être accessible en écriture.

Gérer les interruptions

Voici les fonctions développées pour manipuler les interruptions.

```
**
* Enable an interrupt and set the handler.
* Used the value of the macro INTERRUPT_MASK_REGISTER.
* @param irq number of the interrupt to activate
* @param handler function to call when the interrupt \a irq is triggered
*/
void activate_interrupt(uint32_t irq, void* handler) {
	volatile uint32_t* interrupt_mask_register = (uint32_t*) 0x80000240;
	last_isr_ctx = bcc_isr_register(irq, handler, (void*)0);
	setBit(interrupt_mask_register, irq);
}
/**
* Disable an interrupt.
* Used the value of the macro INTERRUPT_MASK_REGISTER
* @param irq number of interrupt to disable
*/
void disable_interrupt(uint32_t irq) {
	volatile uint32_t* interrupt_mask_register = (uint32_t*) 0x80000240;
	clearBit(interrupt_mask_register, irq);
}
/**
* Trigger the interrupt in parameter.
* Used the value of the macro INTERRUPT_FORCE_REGISTER
* @param irq number of interruption to trigger
*/
void force_interrupt(uint32_t irq) {
	//*interrupt_force_register = *interrupt_force_register | (1 << irq);
	volatile uint32_t* interrupt_force_register = (uint32_t*) 0x80000208;
	setBit(interrupt_force_register, irq);
}
```

Comme la lecture du code de test suivant le laissait présager, les variables `compteur_test` et `g_nb_interruptions` valent 1 à la fin.

```
uint32_t compteur_test = 0;
activate_interrupt(9, votre_handler);
while ((g_nb_interruptions == 0) && (compteur_test < 100)) {
	compteur_test++;
	force_interrupt(9);
}
disable_interrupt(9) ;
force_interrupt(9) ;
```

Le compteur `g_nb_interruptions` n'est incrémenté qu'un fois, car le dès que le handler qui l'incrémente est appelé par `force_interrupt(9)`, le programme sort de la boucle. On ne passe donc qu'une fois dans la boucle et `compteur_test` n'est alors incrémenté qu'une fois. Hors de la boucle, l'interruption est désactivé, donc le handler n'est pas rappelé : `g_nb_interruptions` n'est pas incrémenté une seconde fois.

État des registres d'interruption avant le test précédent :

```
0x80000204  Interrupt pending register              0x00000000
0x80000240  Interrupt mask register  0              0x00000000
0x80000280  Interrupt force register  0             0x00000000
```

État des registres d'interruption après :

```
0x80000204  Interrupt pending register              0x00000000
0x80000240  Interrupt mask register  0              0x00000000
0x80000280  Interrupt force register  0             0x00000200
```

Le registre `pending` est toujours vu à 0 car les interruptions ont été traité au moment où la lecture des registres est faite : aucune n'est en attente.

Le bit 10 du registre `mask` a été mis à 1 par `activate_interrupt(9,…)`, mais remit ensuite à 0 par `disable_interrupt(9)`.

Le seul changement observable est donc dans le registre `force` puisque le dernier appel à `force_interrupt(9)` n'a pas levé d'exception, celle-ci ayant été désactivé préalablement.

Gérer un timer

Après l'exécution de `manage_timer()`, le contenu de `measured_times` montre que les appels au handler sont parfaitement régulier (5001) à deux exceptions près : les deux premiers appels.
Le premier s'explique très simplement puisqu'il n'indique pas la durée après un précédent appel, mais la durée depuis le démarrage du programme. Il faut donc ajouter à 5000 tout le code exécuté avant le démarrage du timer.

```
void monitored_interrupt_handler(int irq) {
	if (g_nb_interruptions < MAX_NB_MEASURES)
		measured_times[g_nb_interruptions] = get_elapsed_time();
	g_nb_interruptions++;
}

void start_timer(uint32_t* timer_counter_register, uint32_t reload_value) {
	*timer_counter_register = reload_value;			// Timer x Value register
	*(timer_counter_register + 1) = reload_value;		// Timer x Reload value register
	*(timer_counter_register + 2) = 0xF; //0b1111;		// Timer x Control register
}

void manage_timer(){
	uint32_t* timer2_counter_register = (uint32_t*) 0x80000320;
	activate_interrupt(9, monitored_interrupt_handler);
	start_timer(timer2_counter_register, 5000);

	while (g_nb_interruptions < 100) {
	}
}
```

Le second à 4996, s'explique moins simplement, mais l'écart à la valeur attendu très faible (5 de moins). Il semble s'agir d'un effet de bord lié au cache, qui disparaît si celui-ci est préalableent désactivé :

```
__asm__ __volatile__ ( " sta %%g0, [%%g0] 2\n\t" :::);
```

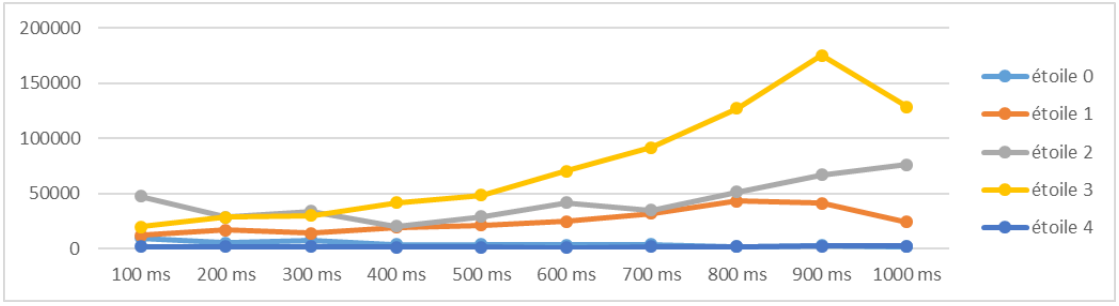
S'interfacer avec une bibliothèque

Le code suivant permet, pendant 1 seconde, d'acquérir 5 images chaque 100ms, puis de calculer le flux pondéré (cf. TP2) pour chacune d'elles.

```
#define MAX_STEP 10
#define NB_IMG 5

float img[NB_IMG][36];
float w[NB_IMG][MAX_STEP];
Windows_producer wp;
int g_nb_interruptions = 0;

volatile uint32_t* timer2_counter_register = (uint32_t*) 0x80000320;

void images_reveived_handler(int irq) {
    g_nb_interruptions++;
}

void produce_images_handler(int irq){
    produce_images(&wp);
}

int main(void){
    init(&wp, img, NB_IMG);
    enable_irq(&wp,10);

    start_timer(timer2_counter_register, 100000); // temps en µs = 100ms

    activate_interrupt(10, images_reveived_handler);
    activate_interrupt(9, produce_images_handler);

    unsigned int nb_interruptions_generees = 0;
    while (nb_interruptions_generees < MAX_STEP) {
        if (g_nb_interruptions > nb_interruptions_generees) {
            for (unsigned int i = 0; i < NB_IMG; i++) {
                w[i][nb_interruptions_generees] = computeFluxPondere(img[i],get_mask(&wp, i));
            }
            nb_interruptions_generees++;
        }
    }
    return EXIT_SUCCESS;
}
```

Les données de flux pondérés, enregistrées dans w et récupérées par GDB en fin de programme :

	100 ms	200 ms	300 ms	400 ms	500 ms	600 ms	700 ms	800 ms	900 ms	1000 ms
étoile 0	9752,15137	5705,05566	7828,98633	4164,97266	3882,55664	3643,65918	3995,21021	2263,17993	3087,81885	2117,39819
étoile 1	12249,8369	17243,2246	14368,084	19519,2031	21593,4688	24878,377	31920,8555	43319,3711	41490,1055	24324,1992
étoile 2	47512,3828	28626,6602	34134,7031	20446,1582	29193,7949	41713,9492	34957,5586	51639,9492	67143,2578	76419,5
étoile 3	20128,2441	28621,9141	30251,8223	41658,8047	48502,75	70287,4375	91742,875	127490,141	175133,922	128518,172
étoile 4	2166,51294	2182,8728	1881,45129	1785,98438	1599,10864	1291,01758	1861,901	1946,19995	2572,52295	2476,62378



Si les valeurs peuvent varier d'un compilateur à l'autre (basées sur rand()), l'évolution de la luminosité de chaque étoile est indépendante.