

Programmation Baremetal sur Micro:bit v2

■ ■ ■ Découverte du Micro:bit v2 et de la programmation assembleur ARM

1 – Vous allez tester le premier programme du répertoire x01-echo :

Étudiez le code du programme `echo.c` :

a. À quelle adresse se trouve le périphérique UART ?

Dans le fichier « `hardware.h` » on trouve :

```
#define UART_BASE _BASE(0x40002000)
```

b. Pouvez vous passer la communication à 115200 bauds ? Dans le fichier « `hardware.h` », on trouve :

```
#define UART_BAUDRATE_115200 0x01d60000
```

Ce qui permet de réécrire la fonction « `serial_init()` » dans le fichier « `echo.c` »

```
/* serial_init -- set up UART connection to host */
void serial_init(void)
{
    UART_ENABLE = UART_ENABLE_Disabled;
    // UART_BAUDRATE = UART_BAUDRATE_9600; /* 9600 baud */
    UART_BAUDRATE = UART_BAUDRATE_115200; /* 115200 baud */
    UART_CONFIG = FIELD(UART_CONFIG_PARITY, UART_PARITY_None);
                                /* format 8N1 */
    UART_PSELTXD = TX;          /* choose pins */
    UART_PSELRXD = RX;
    UART_ENABLE = UART_ENABLE_Enabled;
    UART_STARTTX = 1;
    UART_STARTRX = 1;
    UART_RXDRDY = 0;
    txinit = 1;
}
```

c. Comment se fait la réception d'un caractère ? Est-ce par interruption ?

La réception de caractère se fait dans la fonction « `serial_getc()` » :

```
/* serial_getc -- wait for input character and return it */
int serial_getc(void)
{
    while (! UART_RXDRDY) { }
    char ch = UART_RXD;
    UART_RXDRDY = 0;
    return ch;
}
```

On *surveille* la modification de la valeur d'un registre dont l'adresse est définie dans le fichier « `hardware.h` » :

```
#define UART_RXDRDY _REG(unsigned, 0x40002108)
```

Si ce registre est modifié, cela indique qu'un octet est disponible en lecture.

On peut alors le lire avec le registre dont l'adresse est aussi définie dans le fichier « `hardware.h` » :

```
#define UART_RXD _REG(unsigned, 0x40002518)
```

⇒ On fait du « polling » et donc on utilise pas d'interruption !

d. Modifiez le programme pour afficher le texte reçu à l'envers.

On va ajouter la fonction « `serial_puts_reverse(const char *s)` » :

```
void serial_puts_reverse(const char *s)
{
    const char *start = s;
    while (*s != '\0')
        s++;
    while (s != start)
        serial_putc(*--s);
}
```

2 – On va passer au programme `x02-instrs` dont le but est de permettre l'accès à une fonction écrite en assembleur ARM depuis un programme écrit en C.

a. Étudiez le contenu du fichier `fmain.c`:

◊ où est définie la fonction `func` ?

Dans le fichier « `fmain.c` », la fonction « `func` » est définie en `extern`:

```
extern int func(int a, int b);
```

Ce qui veut dire au « compilateur » que la fonction sera disponible dans une autre partie de l'exécutable et que ce sera au « linker » de faire le lien entre sa définition et son appel.

La fonction « `func` » est définie dans le fichier « `func.s` » en assembleur ARM:

```
@ x02-instrs/func.s
@ Copyright (c) 2018-20 J. M. Spivey

.syntax unified
.global func
.text
.thumb_func
func:
@ -----
@ Two parameters are in registers r0 and r1

    adds r0, r0, r1    @ Add r0 and r1, result in r0

@ Result is now in register r0
@ -----
    bx lr              @ Return to the caller
```

déclaration du symbole `func` en global
⇒ cette adresse est accessible à l'extérieur de ce module

étiquette `func` identifiant l'adresse des instructions à exécuter en cas d'appel

◊ comment sont passés les arguments entre le C et l'assembleur ? Est-ce conforme à l'ABI ?

Les deux premiers arguments de la fonction sont passés dans les registres `r0` et `r1`.

C'est justement l'ABI qui l'exige.

◊ comment la valeur de retour est passée à la fonction `init()` ?

Par la valeur laissée dans `r0`.

b. Comment est mesuré le temps d'exécution ?

On utilise un registre spécial fourni par le cortex M4 dont l'adresse est définie dans le fichier « `hardware.h` »:

```
/* Data watchpoint and trace */
#define DWT_CYCCNT                                _REG(unsigned, 0xe0001004)
```

Il correspond à un compteur qui est incrémenté automatiquement par un circuit dédié lors de l'exécution des instructions du processeur.

Pour « démarrer », on initialise le compteur à zéro, en écrivant dans son registre la valeur zéro, par exemple avec un macro :

```
#define clock_start()  DWT_CYCCNT = 0
#define clock_stop()   DWT_CYCCNT
```

Il suffit ensuite de lire la valeur courante de ce registre pour obtenir la valeur atteinte par ce compteur.

```
time = clock_stop();
```

On peut soustraire, les instructions nécessaires à cette lecture :

```
#define FUDGE 5
...
    time -= FUDGE;
```

Puis obtenir le temps en utilisant le fait que le processeur est cadencé à 64 MHz pour avoir le temps final :

```
printf("%d cycle%s, %s microsec\n\n",
       time*MULT, (time*MULT == 1 ? "" : "s"),
       fmt_fixed(time, FREQ, 3));
```

c. Pouvez vous modifier le programme assembleur pour augmenter la durée en réalisant plusieurs fois la somme ?

On peut dupliquer l'instruction assembleur d'incrémementation :

```
adds r0, r0, r1    @ Add r0 and r1, result in r0
adds r0, r0, r1    @ Add r0 and r1, result in r0
adds r0, r0, r1    @ Add r0 and r1, result in r0
```

- d. Est-ce que le nombre de cycle augmente linéairement ?

Si on passe d'une somme, puis deux puis 3 :

```
xterm
a = 1, b = 2
func(1, 2) = 3
func(0x1, 0x2) = 0x3
4 cycles, 0.063 microsec

Hello micro:world!

a = 1, b = 2
func(1, 2) = 5
func(0x1, 0x2) = 0x5
5 cycles, 0.078 microsec

Hello micro:world!

a = 1, b = 2
func(1, 2) = 7
func(0x1, 0x2) = 0x7
6 cycles, 0.094 microsec
```

On observe 4, 5 et 6 cycles.

Grâce au pipeline, l'allongement de l'exécution n'est que de **un cycle** à chaque fois.

Il est nécessaire de recommencer plusieurs fois la boucle pour observer une valeur stable, car le pipeline doit être dans un état correct.

- e. Utilisez l'outil de désassemblage arm-none-eabi-objdump :

```
xterm
$ arm-none-eabi-objdump -d func.o

func.o:      file format elf32-littlearm

Disassembly of section .text:

00000000 <func>:
0:      1840          adds    r0, r0, r1
2:      4770          bx       lr
```

Le résultat est-il comparable ?

Oui, on perd juste la sémantique ajoutée en commentaire dans le fichier source.

3 – Notre nouveau programme à étudier est le x03-loops.

- a. Que fait-il ?

Il réalise une multiplication entière, en réalisant une boucle.

Si on veut faire $z = x * y$ on fait y fois la somme $z = z + x$.

Pour faire y fois la somme, on fait une boucle :

- ◇ en décrémentant la valeur de y à chaque occurrence ;
- ◇ en s'arrêtant lorsque l'on atteint $y = 0$;

Ce qui donne en assembleur :

```
movs r2, #0          @ Initially z = 0
loop:
cmp r1, #0           @ Is y = 0?
beq done             @ If so, finished

subs r1, r1, #1       @ Decrease y by 1
adds r2, r2, r0        @ Increase z by x
b loop               @ Repeat
done:
movs r0, r2           @ Put z in r0
```

Ici, le registre $r1$ contient y , le registre $r2$ contient z .

- b. Vous suivrez l'exécution du code assembleur donné dans le fichier `mult.s`.

On peut le faire à la main en essayant avec des valeurs petites pour x et y .

- c. vous comparerez avec le firmware `x04-numbers`, est-ce qu'il y a des similarités ?

On utilise une instruction de comparaison et un « `bgt` », «branch if greater than» au lieu de l'instruction « `beq` », «branch if equal».

4 – Le programme x05-subrs vous propose de réaliser une factorielle en assembleur.

- a. Quels sont les rapports entre la fonction `func` et `mult` ?

La fonction « `mult` » est appelée par la fonction « `func` » pour faire la multiplication nécessaire à la factorielle.

```
1 @ x05-subrs/fact.s
2 @ Copyright (c) 2018-20 J. M. Spivey
3
4     .syntax unified
5     .global func
6
7     .text
8     .thumb_func
9 func:
10    @@@ Factorials using a loop: keep x in r4, y in r5 and maintain the
11    @@@ relationship fac(a) = fac(x) * y
12    push {r4, r5, lr}           @ Save registers and return address
13    movs r4, r0                 @ Set k to n
14    movs r5, #1                 @ Set r to 1
15
16 again:
17    cmp r4, #2                 @ Test whether k < 2
18    blo exit                   @ If so, exit the loop
19
20    @ Call mult to multiply r by k
21    movs r0, r4                 @ Prepare the call: first argument is k
22    movs r1, r5                 @ Second argument is r
23    bl mult                     @ Call the function
24    movs r5, r0                 @ Put result in r
25
26    subs r4, r4, #1             @ Decrease k by 1
27    b again                     @ Repeat the loop
28
29 exit:
30    movs r0, r5                 @ Put result in r0
31    pop {r4, r5, pc}           @ Restore regs and return
32
33    .thumb_func
34 mult:
35    @@@ Keep x and y in r0 and r1; compute the result z in r2, maintaining
36    @@@ the relationship a * b = x * y + z
37
38    movs r2, #0
39 loop:
40    cmp r0, #0
41    beq done
42
43    subs r0, r0, #1
44    adds r2, r2, r1
45    b loop
46
47 done:
48    movs r0, r2
49    bx lr
```

- b. Est-ce que le second argument passé à `func` est nécessaire ?

Non, il n'est pas nécessaire.

- c. Pourquoi dans `func`, on sauvegarde les valeurs des registres :

- ◇ `r4` et `r5` ?

Pour les préserver, car on va les modifier pour faire le travail (ligne 12).

À la fin de la fonction `func`, on pourra les restaurer (ligne 31).

- ◇ `lr` ?

Le registre `lr` mémorise l'adresse à laquelle il faudra revenir lors du retour de la fonction « `func` » (ligne 49).

Il faut donc le préserver en ligne 12 et le restaurer en ligne 31 où sa valeur est directement mise dans le registre `pc`.

C'est l'ABI qui lors de l'appel de la fonction « `func` » dans le programme C, qui mémorise la valeur du registre `pc` dans `lr` avant de sauter à l'adresse des instructions de « `func` ».

5 – Le programme x06-arrays vous propose d'utiliser les accès indexés à la mémoire :

- ▷ chargement d'une adresse dans un registre ;
- ▷ consultation de la valeur stockée à l'adresse indiquée dans ce registre ;
- ▷ utilisation d'un décalage par rapport à cette adresse pour accéder à des données à la manière d'un tableau.

a. Ajoutez dans la fonction `init()` et juste avant l'appel à la fonction `func`, une **initialisation** des valeurs du tableau `account` défini dans le code assembleur du fichier `bank.s`.

Le tableau est localisé par le symbole « `account` » défini dans le fichier assembleur « `func.s` ».

Pour le rendre accessible aux fonctions C définies dans le fichier « `fmain.c` », il est nécessaire de le rendre global :

```
@ x06-arrays/func.s
@ Copyright (c) 2018-20 J. M. Spivey

.syntax unified
.global func
.global account ----- symble global
```

Ensuite, il est possible de l'initialiser dans la fonction « `void init(void)` »

```
extern int account[10];

void init(void)
{
    unsigned time;

    serial_init();
    printf("\nHello micro:world!\n\n");

    led_init();
    clock_init();

    while (1) {
        int a, b, r;

        for(int i=0;i<10;i++) { ----- initialisation d'account défini dans le fichier « func.s »
            account[i] =i;
        }

        a = getnum("a = ");
        b = getnum("b = ");
    }
}
```

b. En vous servant de ce que vous avez appris sur les différents exercices, modifiez le firmware pour réaliser le passage des caractères d'une chaîne de **minuscule** en **majuscule**.

Vous tiendrez compte du fait que :

- ☐ une chaîne termine par un octet à zéro ;
- ☐ seul un caractère alphabétique minuscule peut être décalé en caractère alphabétique majuscule.

Soit le contenu du fichier « `lowercase_to_uppercase.s` » et du fichier « `fmain.c` » :

```
1 .syntax unified
2 .global low_to_upper
3
4 .text
5 .thumb_func
6
7 low_to_upper:
8     movs r1, #0
9 boucle:
10    ldrb r2, [r0, r1]
11    cmp r2, #0
12    beq fin
13    cmp r2, #97
14    blt continue
15    cmp r2, #122
16    bgt continue
17    subs r2, #32
18    strb r2, [r0, r1]
19 continue:
20    adds r1, #1
21    b boucle
22 fin:
23    bx lr
```

```
1 ...
2 extern char *low_to_upper(char *s);
3
4 void init(void)
5 {
6     char chaine[13]; ----- crée la chaîne dans la pile pour la rendre modifiable
7     char *ptr_src = "AaBbCcDdEeFf";
8
9     memcpy(chaine, ptr_src, 13);
10
11    serial_init();
12    printf("\nHello micro:world!\n\n");
13    printf("%s > %s\n", ptr_src, low_to_upper(chaine));
14 }
```

Le résultat :

```

xterm
Hello micro:world!

AaBbCcDdEeFf > AABCCDEEFF
```