Électronique numérique

J.-M Friedt

FEMTO-ST/département temps-fréquence

jmfriedt@femto-st.fr

transparents à `jmfriedt.free.fr`

22 novembre 2020

Plan des interventions :

7 cours/TP d'introduction au STM32 en *C baremetal* :

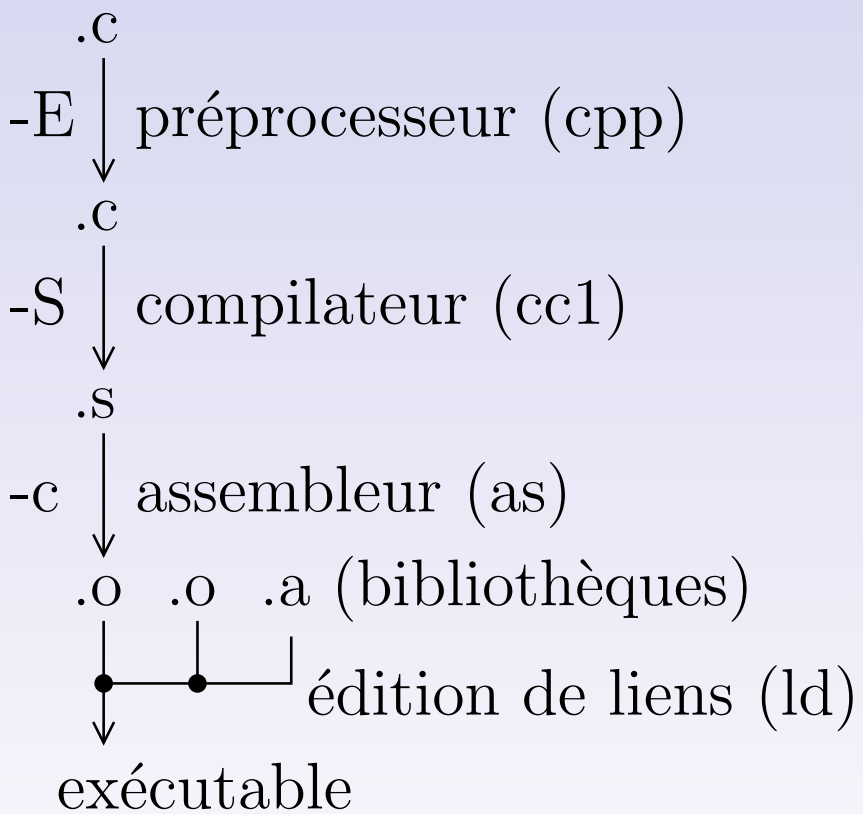
1. Électronique numérique et conception du circuit
   L3 : aspects analogiques, consommation électrique, lecture de datasheet
   Survol des divers périphériques qui seront abordés (RS232, SPI, timer, ADC)
   représentation des données (tailles/encodage), masques, architecture
   Rappels sur Atmega32U4 (Makefile, compilation, masques ...)
2. Premiers pas sur le STM32, adresses des périphériques, architecture
3. Fonctionnement de gcc et optimisations :
   préprocess–compilateur–assembleur–linker, passage C à assembleur, pointeurs
4. bibliothèques et séparation algorithme/matériel, simulateurs
   libopencm3, newlib & stubs, ressources requises par les bibliothèques
5. Bus de communication série, synchrone/asynchrone
6. arithmétique sur systèmes embarqués
   entiers, flottants, convertir un algorithme exprimé en nombres à virgules vers des entiers, timers
7. interruptions et acquisition de données analogiques, fréquence d'échantillonnage
   vecteurs d'interruption, gestion des horloges du STM32, ADC

Compilateur : pourquoi le maîtriser ?

1. passage du C au langage machine
2. conséquences du compilateur : les options d'optimisation font varier le temps d'exécution
3. spécificités du développement sur système à ressources réduites : pas d'allocation dynamique de mémoire, processus monolithique unique
4. instructions pour contrôler les optimisations
5. variables globales et `goto`, interruptions

Passage du C au langage machine

1. étapes du compilateur : préprocesseur, compilateur, assembleur, éditeur de liens (*linker*)
2. `gcc` est un frontend pour appeler le préprocesseur (`cpp`, `gcc -E`), le compilateur (`cc1`, `gcc -S`), l'assembleur (`as`, `gcc -c`) et le linker (`ld`)
3. séquence de compilation se découvre par `gcc -v`



Séquence de compilation

Comment passer de ce programme en C à une suite d'opcodes compréhensibles par un processeur ?

```
#include <math.h>

#define i_init 3

int main()
{volatile int i=i_init;
 i=i+1;
}
```

gcc -E

```
...
extern double asin (double __x) __attribute__ ((→
    ↪__nothrow__ , __leaf__)); extern double __asin (→
    ↪double __x) __attribute__ ((__nothrow__ , __leaf__))→
    ↪;

extern double atan (double __x) __attribute__ ((→
    ↪__nothrow__ , __leaf__)); extern double __atan (→
    ↪double __x) __attribute__ ((__nothrow__ , __leaf__))→
    ↪;
...
struct exception

  {
    int type;
    char *name;
    double arg1;
    double arg2;
    double retval;
  };
...
int main()
{volatile int i=3;
 i=i+1;
}
```

gcc -S

```
    .file   "t.c"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushl   %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl    %esp, %ebp
    .cfi_def_cfa_register 5
    subl    $16, %esp
    movl    $3, -4(%ebp)
    movl    -4(%ebp), %eax
    addl    $1, %eax
    movl    %eax, -4(%ebp)
    leave
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Debian →
        ↪4.9.1-4) 4.9.1"
    .section        .note.→
        ↪GNU-stack,"",→
        ↪@progbits
```

De l'utilité de connaître l'assembleur

```
int main()
{volatile unsigned short j=3,k;
 for (k=1;k<15;k++) {j+=k;}
}
```

```
.L6:
        add     2(r1), @r1
        add     #llo(1), 2(r1)
        cmp     #llo(15), 2(r1)
        jlo     .L6
```

```
int main()
{volatile unsigned short j=3,k;
 for (k=15;k>0;k--) {j+=k;}
}
```

```
.L6:
        add     2(r1), @r1
        add     #llo(-1), 2(r1)
        jne     .L6
```

Mais un compilateur sait analyser le code : exemple de avr-gcc

```
88 e0           ldi     r24, 0x08       ; 8
for(i=0;i<8;++i) asm("nop"); // Back porch (wait)
00 00           nop
81 50           subi    r24, 0x01       ; 1
e9 f7           brne    .-6             ; 0x198 <__vector_17+0x24>
```

Comparaison de compilateurs

Dans tous les cas :

```
void delay (int length) {while (length >=0) length--;}
  // delay(5000)=1 ms
```

arm-thumb-elf-gcc (GCC) 4.0.0

```
196             delay:
197                     @ lr needed for prologue
198 0000 00E0           b       .L2
199             .L3:
200 0002 0138           sub     r0, r0, #1
201             .L2:
202 0004 0028           cmp     r0, #0
203 0006 FCDA           bge     .L3
204                     @ sp needed for prologue
205 0008 7047           bx      lr
207 000a 0000           .align  2
208                     .global jmf_putchar
209                     .code 16
210                     .thumb_func
```

Keil ARM Compiler V2.42

```
  47: void delay (int length) {       // delay(5000)=1 ms
00000000  B401  PUSH    {R0}
  48:     while (length >=0) length--;
00000002  E003  B       L_1  ; T=0x0000000C
00000004      L_3:
00000004  A800  ADD     R0,R13,#0x0
00000006  6801  LDR     R1,[R0,#0x0] ; length
00000008  3901  SUB     R1,#0x1
0000000A  6001  STR     R1,[R0,#0x0] ; length
0000000C      L_1:
0000000C  A800  ADD     R0,R13,#0x0
0000000E  6800  LDR     R0,[R0,#0x0] ; length
00000010  2800  CMP     R0,#0x0
00000012  DAF7  BGE     L_3  ; T=0x00000004
  49: }
00000014  B001  ADD     R13,#0x4
00000016  4770  BX      R14
00000018      ENDP ; 'delay?T'
```

Fonctionnalités égales mais temps d'exécution différents

Multiplication

- AVR a une multiplication (signée ou non) en 2 cycles d'horloge
- décaler revient à multiplier/diviser par 2

```
int main()
{volatile char k=
    ↪=5;
 k=k*10;
}
```

se compile en

```
ldi r24,lo8(5)
mov r25,r24
lsl r25
lsl r25
add r24,r25
lsl r24
```

lsl puis lsl est $\times 4$, add est $\times 5$, et lsl est $\times 10$.

```
int main()
{volatile char c1=
    ↪=5,c2=10;
 c2=c1*10;
}
```

se compile en

```
ldi r24,lo8(5)
lsl r24
mov r25,r24
lsl r25
lsl r25
add r24,r25
```

lsl est $\times 2$, puis lsl lsl est $\times 8$, add est $8+2=10$.

```
int main()
{volatile char c1=
    ↪=5,c2=10;
 c2=c1*c2;
}
```

se compile en

```
ldi r24,lo8(5)
ldi r25,lo8(10)
mul r24,r25
```

mul effectue la multiplication en 2 cycles d'horloge !

Exercice : comment multiplier par 15 ?

Compilation séparée et édition de liens

- Réutilisation du code : un ensemble de fonctions par fichier

```
int mon_carre(int i)
{return(i*i);}
```

- Compiler chaque source C en un objet (`gcc -c source.c` → `source.o`)
- Lier ces objets (et bibliothèques) en un exécutable
- Déclaration de variables dans un autre fichier : `extern`
- Déclaration des prototypes des fonctions (sans leur implémentation) : fichier d'entête

```
int mon_carre(int);
```

pas de programme dans le fichier d'entête

- bibliothèque statique (archive) : `ar rcs archive.a objets.o` ⇒ se lier à un unique binaire permettant de propager un correctif et réutiliser ses fonctions.

Carte mémoire (édition de liens – ld)

```
/*
 * This file is part of the libopencm3 project.
 *
 * Copyright (C) 2009 Uwe Hermann <uwe@hermann-uwe.de>
[...]

MEMORY
{
rom (rx) : ORIGIN = 0x08000000, LENGTH = 128K
ram (rwx) : ORIGIN = 0x20000000, LENGTH = 32K
}

INCLUDE cortex-m-generic.ld
```

Table 4. Memory mapping vs. Boot mode/physical remap in STM32F410

Addresses	Boot/Remap in main Flash memory	Boot/Remap in embedded SRAM	Boot/Remap in System memory
0x2000 0000 - 0x2002 7FFF	SRAM (32 KB)	SRAM (32KB)	SRAM (32KB)
0x1FFF 0000 - 0x1FFF 77FF	System memory	System memory	System memory
0x0802 0000 - 0x1FFE FFFF	Reserved	Reserved	Reserved
0x0800 0000 - 0x0801 FFFF	Flash memory	Flash memory	Flash memory
0x0400 000 - 0x07FF FFFF	Reserved	Reserved	Reserved
0x0000 0000 - 0x0001 FFFF(1)	Flash (128 KB) Aliased	SRAM1 (32 KB) Aliased	System memory (30 KB) Aliased

1. Even when aliased in the boot memory space, the related memory is still accessible at its original memory space.

avec `cortex-m-generic.ld` : ... RM0401 Reference manual

Carte mémoire (édition de liens – ld)

```
EXTERN (vector_table)

/* entry point of the output file. */
ENTRY(reset_handler)
[...]
/* Define sections. */
SECTIONS
{
.text : {
*(.vectors) /* Vector table */
*(.text*) /* Program code */
. = ALIGN(4);
*(.rodata*) /* Read-only data */
. = ALIGN(4);
} >rom

[...]
.data : {
_data = .;
*(.data*) /* RW initialized data */
. = ALIGN(4);
_edata = .;
} >ram AT >rom

.bss : {
*(.bss*) /* ReW zero initialized data
*(COMMON)
. = ALIGN(4);
_ebss = .;
} >ram
[...]
. = ALIGN(4);
end = .;
}

PROVIDE(_stack=ORIGIN(ram)+LENGTH(ram)
```

Attribution de la mémoire

`size` nous informe des ressources (statiques) requises par un programme :

- `police[95*8]={0};`

```
$ avr-size taille.avr
   text    data     bss     dec     hex filename
     30       0    1520    1550     60e taille.avr
$ msp430-size taille.msp
   text    data     bss     dec     hex filename
    110       0    1522    1632     660 taille.msp
```

```
const police[95*8]={0};
$ avr-size taille.avr
   text    data     bss     dec     hex filename
     38    1520       0    1558     616 taille.avr
$ msp430-size taille.msp
   text    data     bss     dec     hex filename
   1630       0       2    1632     660 taille.msp
```

Initialisation du processeur

Un certain nombre de périphériques doivent être initialisés au démarrage du processeur : séquence d'initialisation par défaut fournie par gcc

```
$ msp430-gcc -mmcu=msp430f149 -o executable fichier.c
$ msp430-objdump -dSt executable
[...]
Disassembly of section .text:

00001100 <__watchdog_support>:
    1100:       55 42 20 01     mov.b   &0x0120,r5
    1104:       35 d0 08 5a     bis     #23048, r5      ;#0x5a08
    1108:       82 45 00 02     mov     r5,     &0x0200

0000110c <__init_stack>:
    110c:       31 40 00 0a     mov     #2560,  r1      ;#0x0a00
00001110 <__do_copy_data>:
    1110:       3f 40 00 00     mov     #0,     r15     ;#0x0000
[...]
00001128 <__do_clear_bss>:
    1128:       3f 40 00 00     mov     #0,     r15     ;#0x0000
[...]
0000113e <main>:
    113e:       04 41           mov     r1, r4
    1140:       24 53           incd    r4
[...]
```

Initialisation du processeur

Un certain nombre de périphériques doivent être initialisés au démarrage du processeur : séquence d'initialisation par défaut fournie par gcc

Option `-nostartfiles` pour interdire l'initialisation par défaut

```
$ msp430-gcc -mmcu=msp430f149 -nostartfiles -o executable fichier.c
$ msp430-objdump -dSt executable
[...]
Disassembly of section .text:

00001100 <main>:
    1100:       04 41           mov     r1, r4
    1102:       24 53           incd    r4
    1104:       21 83           decd    r1
    1106:       94 43 fc ff     mov     #1, -4(r4) ;r3 As==01, 0xfffc(r4)
    110a:       21 53           incd    r1
```

Conséquence du passage de paramètres

Gain de place **et de temps**

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
struct matrice {int reel[1024*1024];} toto1;

void fonction(struct matrice *a) {a->reel[1]=a->reel[1]+1;} // ici 2..1001
void fonction2(struct matrice a) {a.reel[1]=a.reel[1]+1;}   // ici 2

int main()
{time_t t1,t2,t3;
 int k;
 toto1.reel[1]=1;                                          // ici 1
 time(&t1);
 for (k=0;k<1000;k++) fonction(&toto1);
 time(&t2);
 for (k=0;k<1000;k++) fonction2(toto1);
 time(&t3);
 printf("t1=%d t2=%d\r\n",t2-t1,t3-t2);                    // t1=0 t2=16
}
```

volatile

1. instruction pour interdire au compilateur de faire des hypothèses sur une variable
2. sans cette instruction, le compilateur se permet d'éliminer le code inutile

Exemple : dans /usr/lib/avr/include/avr/sfr_defs.h

```
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
#define _MMIO_WORD(mem_addr) (*(volatile uint16_t *)(mem_addr))
#define _MMIO_DWORD(mem_addr) (*(volatile uint32_t *)(mem_addr))
...
#define _SFR_IO8(io_addr) _MMIO_BYTE((io_addr) + __SFR_OFFSET)
```

et dans /usr/lib/avr/include/avr/iom32u4.h (définition du Atmega 32U4)

```
#define PINB _SFR_IO8(0x03)
#define PINB0 0
...
#define DDRB _SFR_IO8(0x04)
#define DDB0 0
#define DDB1 1
```

volatile

Si on optimise sans volatile ... il ne reste rien !

Dans les deux cas : `msp430-gcc -O2 -mmcu=msp430x149 -D_GNU_ASSEMBLER_ -S`

```
int main()
{volatile unsigned short j=3,k;
 for (k=0;k<15;k++) {j+=k;}
}
```

```
        mov     #(__stack-4), r1
/* prologue end (size=2) */
        mov     #llo(3), @r1
        mov     #llo(0), 2(r1)
        cmp     #llo(15), 2(r1)
        jhs     .L8
.L6:
        add     2(r1), @r1
        add     #llo(1), 2(r1)
        cmp     #llo(15), 2(r1)
        jlo     .L6
.L8:
```

```
int main()
{unsigned short j=3,k;
 for (k=0;k<15;k++) {j+=k;}
}
```

```
main:
        mov     #(__stack-0), r1
/* prologue end (size=2) */
/* epilogue: frame size=0 */
        br      #__stop_progExec__
```

Exercices

Séparation de la partie algorithmique (portable) des accès bas-niveau

1. Proposer un programme dont la fonction principale fait clignoter une LED et affiche un message sur le terminal : le clignotement de la LED est simulé sur PC par un message. Pour ce faire, on proposera **deux** codes sources (`.c`), l'un portable et indépendant de la plateforme matérielle faisant appel à du C portable, et l'autre spécifique au matériel (port série ou `stdout` sous unix)
2. En gardant *le même programme principal*, proposer une architecture de programme qui permette d'exécuter ce même programme sur STM32
3. Proposer deux `Makefile` qui automatisent la compilation sur chaque cible.