Informatique embarquée 2/16

J.-M Friedt

FEMTO-ST/département temps-fréquence

jmfriedt@femto-st.fr

transparents à `jmfriedt.free.fr`

12 janvier 2017

Introduction

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. instruction pour contrôler les optimisations
5. variables globales et `goto`

Passage du C au langage machine

1. étapes du compilateur : préprocesseur, assembleur, compilateur, 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
```

Les opcodes

Mnémonique → opcode lors de l'assemblage (bijection)

	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
0x00	NOP	AJMP	LJMP	RR	INC	INC	INC	INC	INC	INC	INC	INC	INC	INC	INC	INC
0x10	JBC	ACALL	LCALL	RRC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC
0x20	JB	AJMP	RET	RL	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD
0x30	JNB	ACALL	RETI	RLC	ADDC	ADDC	ADDC	ADDC	ADDC	ADDC	ADDC	ADDC	ADDC	ADDC	ADDC	ADDC
0x40	JC	AJMP	ORL	ORL	ORL	ORL	ORL	ORL	ORL	ORL	ORL	ORL	ORL	ORL	ORL	ORL
0x50	JNC	ACALL	ANL	ANL	ANL	ANL	ANL	ANL	ANL	ANL	ANL	ANL	ANL	ANL	ANL	ANL
0x60	JZ	AJMP	XRL	XRL	XRL	XRL	XRL	XRL	XRL	XRL	XRL	XRL	XRL	XRL	XRL	XRL
0x70	JNZ	ACALL	ORL	JMP	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV
0x80	SJMP	AJMP	ANL	MOVC	DIV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV
0x90	MOV	ACALL	MOV	MOVC	SUBB	SUBB	SUBB	SUBB	SUBB	SUBB	SUBB	SUBB	SUBB	SUBB	SUBB	SUBB
0xa0	ORL	AJMP	MOV	INC	MUL	?	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV
0xb0	ANL	ACALL	CPL	CPL	CJNE	CJNE	CJNE	CJNE	CJNE	CJNE	CJNE	CJNE	CJNE	CJNE	CJNE	CJNE
0xc0	PUSH	AJMP	CLR	CLR	SWAP	XCH	XCH	XCH	XCH	XCH	XCH	XCH	XCH	XCH	XCH	XCH
0xd0	POP	ACALL	SETB	SETB	DA	DJNZ	XCHD	XCHD	DJNZ	DJNZ	DJNZ	DJNZ	DJNZ	DJNZ	DJNZ	DJNZ
0xe0	MOVX	AJMP	MOVX	MOVX	CLR	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV
0xf0	MOVX	ACALL	MOVX	MOVX	CPL	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV

Architecture CISC (*Complex instruction set computer*) [1]

ALU : un démultiplexeur qui convertit une valeur d'opcode en action sur les registres en entrée.

1. http://www.win.tue.nl/~aeb/comp/8051/set8051.html

Les instructions (MSP430)

- `RRC` Rotate right through carry
- `SWPB` Swap bytes
- `RRA` Rotate right arithmetic
- `SXT` Sign extend byte to word
- `PUSH` Push value onto stack
- `CALL` Subroutine call ; push PC and move source to PC
- `RETI` Return from interrupt ; pop SR then pop PC
- `JNE/JNZ` Jump if not equal/zero
- `JEQ/JZ` Jump if equal/zero
- `JNC/JLO` Jump if no carry/lower
- `JC/JHS` Jump if carry/higher or same
- `JN` Jump if negative
- `JGE` Jump if greater or equal
- `JL` Jump if less
- `JMP` Jump (unconditionally)
- `MOV` Move source to destination
- `ADD` Add source to destination
- `ADDC` Add source and carry to destination
- `SUBC` Subtract source from destination (with carry)
- `SUB` Subtract source from destination
- `CMP` Compare (pretend to subtract) source from destination
- `DADD` Decimal add source to destination (with carry)
- `BIT` Test bits of source AND destination
- `BIC` Bit clear (dest &= ∼src)
- `BIS` Bit set (logical OR)
- `XOR` Exclusive or source with destination
- `AND` Logical AND source with destination (dest &= src)

Architecture RISC (*Reduced instruction set computer*) :

instruction + argument de taille fixe ⇒ rapide à décoder mais parfois complexe à lire

Pour AVR :

`http://www.atmel.com/images/Atmel-0856-AVR-Instruction-Set-Manual.pdf`

objdump -dSt

```
08048274 <_init>:
 8048274:       53                      push   %ebx
 8048275:       83 ec 08                sub    $0x8,%esp
 8048278:       e8 83 00 00 00          call   8048300 <__x86.get_pc_thunk.bx>
 804827d:       81 c3 d7 13 00 00       add    $0x13d7,%ebx
 8048283:       8b 83 fc ff ff ff       mov    -0x4(%ebx),%eax
 8048289:       85 c0                   test   %eax,%eax
 804828b:       74 05                   je     8048292 <_init+0x1e>
 804828d:       e8 1e 00 00 00          call   80482b0 <__gmon_start__@plt>
 8048292:       83 c4 08                add    $0x8,%esp
 8048295:       5b                      pop    %ebx
 8048296:       c3                      ret
...
int main()
{volatile int i=i_init;
 80483cb:       55                      push   %ebp
 80483cc:       89 e5                   mov    %esp,%ebp
 80483ce:       83 ec 10                sub    $0x10,%esp
 80483d1:       c7 45 fc 03 00 00 00    movl   $0x3,-0x4(%ebp)
 i=i+1;
 80483d8:       8b 45 fc                mov    -0x4(%ebp),%eax
 80483db:       83 c0 01                add    $0x1,%eax
 80483de:       89 45 fc                mov    %eax,-0x4(%ebp)
 80483e1:       c9                      leave
 80483e2:       c3                      ret
}
```

- Obtention de la séquence d'opcodes et d'arguments en mémoire du microcontrôleur
- Noter que cette dernière étape est possible même si nous ne possédons que le binaire exécutable

Optimisations du code C

Taille des variables

- `(unsigned) char` : 8 bits
- `(unsigned) short` : 16 bits
- `(unsigned) long` : 32 bits *The size of int should be the machine word size (the optimal size that the system can handle at one go).* [2]
- `(unsigned long long)` : 64 bits
- les nombres à virgule flottante simple précision `float` et double précision `double`

Utiliser le type présentant le meilleur compromis espace/précision des données (analyse rationnelle)

2. sur Atmega32U4, `sizeof(int)` renvoie 2 alors que sur PC, cette même commande renvoie 4.

Optimisations de gcc

http://www.redhat.com/magazine/011sep05/features/gcc/

-O0 Barely any transformations are done to the code, just code generation. At this level, the target code can be debugged with no loss of information.

-O1 Some transformations that preserve execution ordering. Debuggability of the generated code is hardly affected. User variables should not disappear and function inlining is not done.

-O2 More aggressive transformations that may affect execution ordering and usually provide faster code. Debuggability may be somewhat compromised by disappearing user variables and function bodies.

-O3 Very aggressive transformations that may or may not provide better code. Some semantics may be modified (particularly floating point). Order of execution completely distorted. Debuggability is seriously compromised.

-Os Optimize for size. This option enables transformations that reduce the size of the generated code. Ironically, this may sometimes improve application performance because there simply is less code to execute. This may lead to reduced memory footprints which may produce fewer page faults.

Allocation de mémoire

1. on connait l'espace mémoire occupé
2. pas de multi-processus $\Rightarrow$ pas de ressources distribuées
3. passage de paramètres par pointeur et non par valeur
4. Pour le C, la mémoire est un ensemble d'octets qui se manipule par les pointeurs
5. `*(type *)(adresse)=valeur;` (*interdit sur les architectures à MMU*)

```
char c[4]={0x12,0x34,0x45,0x56};
short *s;
int *i;
s=(short*)c; printf("%x",s[1]);  // 5645
i=(int*)c;   printf("%x",i[0]);  // 56453412
```

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     #1lo(3), @r1
        mov     #1lo(0), 2(r1)
        cmp     #1lo(15), 2(r1)
        jhs     .L8
.L6:
        add     2(r1), @r1
        add     #1lo(1), 2(r1)
        cmp     #1lo(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__
```

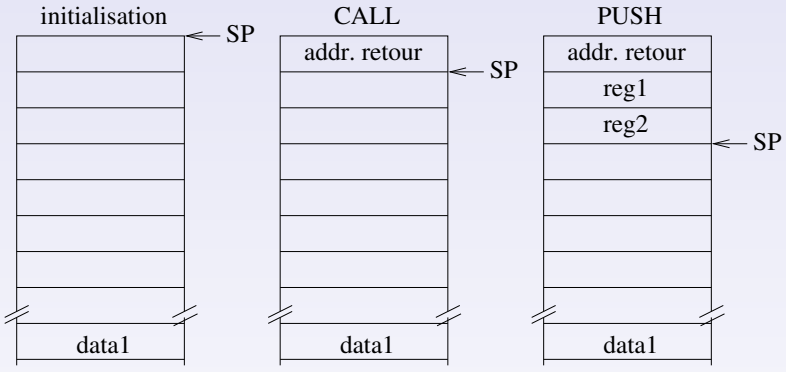
Cas des interruptions

1. Une interruption est une méthode pour interrompre l'exécution séquentielle d'un programme lorsqu'un évènement qui ne peut pas attendre survient
2. Plusieurs sources d'interruptions disponibles sur microcontrôleur (GPIO, timer, communication) ...
3. ... selon les architectures, un gestionnaire d'interruption teste la source, ou plusieurs gestionnaires.
4. Un gestionnaire d'interruption doit être le plus bref possible : définir un drapeau et quitter
5. Passage de paramètres : variable globale

Le program counter

Où en est l'exécution du code.
Le PC en soi est caché, mais comprendre son utilisation évite bien des déboires :

- un saut (`jmp`) correspond à stocker une nouvelle valeur absolue dans le PC (PC=xxxx)
- un saut relatif (`bra`) correspond à une opération sur PC (PC+=xx)
- un appel de procédure (`call`) correspond à *empiler* PC, effectuer les opérations de la procédure, et revenir (`ret`) en dépilant le PC.
- PC est initialisé au Reset
- la pile sert à conserver temporairement des variables (`push`, `pop`) et lors de l'appel d'interruptions.



Manipuler la pile sans la remettre en état dans une procédure est une garantie de plantage

Variables globales

```
volatile int global_index , tim0 , heure , minute , seconde ;

void IRQ_Handler (void)          // unique gestionnaire pour ADuC7026
{ if (IRQSIG & UART_BIT)
    {                            // UART Interrupt
      global_tab[global_index] = my_getchar ();
      if (global_index < (NB_CHARS - 1))
        global_index++;
    }
  if (IRQSIG & GP_TIMER_BIT)
    {                            // Timer1 Interrupt
      GLOBAL_TIMER1++;
      T1CLRI = 1;                // Clear Timer 1 interrupt
    }
  if (IRQSIG & RTOS_TIMER_BIT)
    { tim0++;                    // Timer0 Interrupt
      seconde++;
      if (seconde > 600)
        {seconde = 0;minute++;} // seconde en 1/10 seconde
      if (minute > 60) {minute = 0;heure++;}
      if (heure > 24)  {heure = 0;}
      T0CLRI = 0;
    }
}
```

goto

Instruction prohibée car rendant le code difficile à lire [3]

```
                Chapter 7: Centralized exiting of functions

Albeit deprecated by some people, the equivalent of the goto statement is
used frequently by compilers in form of the unconditional jump instruction.

The goto statement comes in handy when a function exits from multiple
locations and some common work such as cleanup has to be done.  If there is
cleanup needed then just return directly.

...
The rationale for using gotos is:

- unconditional statements are easier to understand and follow
- nesting is reduced
- errors by not updating individual exit points when making
    modifications are prevented
- saves the compiler work to optimize redundant code away ;)
```

Version C des commandes de saut (jmp ou bra)

3. https://www.kernel.org/doc/Documentation/CodingStyle

Exemple

Code source du noyau Linux : drivers/gpio/gpio-timberdale.c

```
     if ((trigger & IRQ_TYPE_EDGE_BOTH) == IRQ_TYPE_EDGE_BOTH) {
             if (ver < 3) {
                     ret = -EINVAL;
                     goto out;
             }
             else {
...
out:
        spin_unlock_irqrestore(&tgpio->lock, flags);
        return ret;
```

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
```

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'éxécution différents

Déverminage de compilateur (AVR, gcc-avr 4.8.1)

sans volatile

```
[...]
char cmpt=5;

ISR(TIMER1_OVF_vect)
{ cmpt++;
  if(cmpt>99) cmpt=0;
  if (cmpt == 5) PORTC^=0X02;
}

int main(void)
{ TCCR1A=0x40;TCCR1B=0x4A;TCCR1C=0x80;
  TIMSK1=0x23;OCR1A=0;
  sei();
  DDRC = 0XFF ;
  while (1)
    if (cmpt == 5) PORTC^=0X01;
}
```

```
[...]
  22:coin.c       ****   TIMSK1=0X23;OCR1A=0;
 197 0012 83E2                ldi r24,lo8(35)
 198 0014 8093 6F00           sts 111,r24
 199 0018 1092 8900           sts 136+1,__zero_reg__
 200 001c 1092 8800           sts 136,__zero_reg__
  24:coin.c       ****   sei();
 205 0020 7894                sei
  26:coin.c       **** DDRD = 0X0F ;
 210 0022 8FE0                ldi r24,lo8(15)
 211 0024 8AB9                out 0xa,r24
  27:coin.c       **** DDRC = 0XFF ;
 214 0026 8FEF                ldi r24,lo8(-1)
 215 0028 87B9                out 0x7,r24
  30:coin.c       **** while (1)
  31:coin.c       ****         {
  32:coin.c       ****         if (cmpt == 5)
  33:coin.c       ****                 PORTC^=0X01;
 218 002a 91E0                ldi r25,lo8(1)
 219                     .L9:
  32:coin.c       ****         if (cmpt == 5)
 222 002c 8091 0000           lds r24,cmpt
 223 0030 8530                cpi r24,lo8(5)
 224 0032 01F0                breq .L7
 225                     .L10:
 226 0034 00C0                rjmp .L10
 227                     .L7:
 230 0036 88B1                in r24,0x8
 231 0038 8927                eor r24,r25
 232 003a 88B9                out 0x8,r24
 233 003c 00C0                rjmp .L9
```

Déverminage de compilateur (AVR, gcc-avr 4.8.1)

avec volatile

```
[...]
volatile char cmpt=5;

ISR(TIMER1_OVF_vect)
{ cmpt++;
  if(cmpt>99) cmpt=0;
  if (cmpt == 5) PORTC^=0X02;
}

int main(void)
{ TCCR1A=0x40;TCCR1B=0x4A;TCCR1C=0x80;
  TIMSK1=0x23;OCR1A=0;
  sei();
  DDRC = 0XFF ;
  while (1)
    if (cmpt == 5) PORTC^=0X01;
}
```

```
[...]
 22:coin.c        ****   TIMSK1=0X23;OCR1A=0;
194 0012 83E2                ldi r24,lo8(35)
195 0014 8093 6F00           sts 111,r24
196 0018 1092 8900           sts 136+1,__zero_reg__
197 001c 1092 8800           sts 136,__zero_reg__
 23:coin.c        ****
 24:coin.c        ****   sei();
202 0020 7894                sei
 26:coin.c        **** DDRD = 0X0F ;
207 0022 8FE0                ldi r24,lo8(15)
208 0024 8AB9                out 0xa,r24
 27:coin.c        **** DDRC = 0XFF ;
211 0026 8FEF                ldi r24,lo8(-1)
212 0028 87B9                out 0x7,r24
 30:coin.c        **** while (1)
 31:coin.c        ****         {
 32:coin.c        ****         if (cmpt == 5)
 33:coin.c        ****                 PORTC^=0X01;
215 002a 91E0                ldi r25,lo8(1)
216                      .L7:
 32:coin.c        ****         if (cmpt == 5)
219 002c 8091 0000           lds r24,cmpt
220 0030 8530                cpi r24,lo8(5)
221 0032 01F4                brne .L7
224 0034 88B1                in r24,0x8
225 0036 8927                eor r24,r25
226 0038 88B9                out 0x8,r24
227 003a 00C0                rjmp .L7
```

Mise en pratique

Communication : mise en forme des informations échangées avec un utilisateur (absence de `stdio`)

Exploitation des interruptions du AVR :

- la majorité des périphériques matériels ont une interruption qui leur est associée
- au lieu d'attendre la fin d'une tâche (transmission de données, conversion), une interruption est déclenchée tandis que le processeur a continué à avancer dans le programme,
- souvent difficiles à déverminer $\Rightarrow$ garder les ISR aussi simples que possibles.
- Parfois une ISR par interruption, parfois une ISR globale qui nécessite de tester quelle interruption a été déclenchée.