Pointeurs = adresse mémoire

```
#include <avr/io.h>
int main(){PORTB=0x42;}
```

se compile avec `avr-gcc -mmcu=atmega32u4 ...`

1. `locate avr/io.h` : `/usr/lib/avr/include/avr/io.h`
2. dans `/usr/lib/avr/include/avr/io.h` :
```
#elif defined (__AVR_ATmega32U4__)
#  include <avr/iom32u4.h>
```
3. dans `/usr/lib/avr/include/avr/iom32u4.h` :
```
#define PORTB _SFR_IO8(0x05)
```
4. dans `/usr/lib/avr/include/avr/sfr_defs.h`
```
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(→
    ↪mem_addr))
#define _SFR_IO8(io_addr) _MMIO_BYTE((io_addr) + →
    ↪__SFR_OFFSET)
```

Conclusion : `PORTB=0x42` devient [1]

```
(*(volatile uint8_t *)((0x05) + 0x20))=0x42;
```

1. `avr-gcc -E`

Passage de paramètres

Par valeur et par référence (pointeur)

```
#include <stdio.h>
int fp(int* i) {(*i)++;return(*i);}
int fv(int i) {i++;return(i);}

int main()
{int i=3;
 printf("%d %d %d\n",i,fv(i),i);
 printf("%d %d %d\n",i,fp(&i),i);
}
```

Passage de paramètres

Par valeur et par référence (pointeur)

```c
#include <stdio.h>
int fp(int* i) {(*i)++;return(*i);}
int fv(int i) {i++;return(i);}

int main()
{int i=3;
 printf("%d %d %d\n",i,fv(i),i);
 printf("%d %d %d\n",i,fp(&i),i);
}
```

```
$ gcc -Wall -o t t.c
$ ./t
3 4 3
4 4 3
```

Les arguments de `printf()` sont évalués de droite à gauche sur x86 !
(mais ce comportement n'est **pas standardisé**)
Exemple du compilateur LLVM (compilateur `clang`) :

```
$ clang -Wall -o tt t.c
$ ./tt
3 4 3
3 4 4
```

Passage de paramètres

Économiser ressources (& temps) par les pointeurs sur les structures

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>                    // ulimit -a : stack size
struct m {int re[1024*924];}; // (kbytes, -s) 8192
                // 1024x1024 segfaulte car *2* copies de m
void f1(struct m *a) {a->re[1]=a->re[1]+1;} // 2..1001
void f2(struct m a)  {a.re[1]=a.re[1]+1;}    // tjs 2

int main()
{time_t t1,t2,t3; int k;
 struct m mat;
 mat.re[1]=1;
 time(&t1);
 for (k=0;k<1000;k++) f1(&mat);
 time(&t2);
 for (k=0;k<1000;k++) f2(mat);
 time(&t3);
 printf("t1=%d t2=%d\r\n",t2-t1,t3-t2);    // t1=0 t2=8
}
```

```
TIME(2)                    Linux Programmer's Manual                    TIME(2)
NAME
       time - get time in seconds
SYNOPSIS
       #include <time.h>
       time_t time(time_t *t);
```

Passage de paramètres

```c
#include <stdio.h>
int main()
{int a=8,b=5;
 printf("%d",a*b);
}
```

```
gcc -S fichier.c
        movl    $8, -4(%rbp)
        movl    $5, -8(%rbp)
        movl    -4(%rbp), %eax
        imull   -8(%rbp), %eax
        movl    %eax, %esi
```

tandis que

```
gcc -O2 -S fichier.c
        movl    $40, %esi
```

gcc a **précalculé** le résultat

Types et endianness

```c
#include <stdio.h>
#include <arpa/inet.h>  //  /usr/include/netinet/in.h

long main()
{long i=0x12345678;
 long *ii;
 char cc[8]={1,2,3,4,5,6,7,8};
 char *c;
 printf("%d\n",sizeof(i));
 c=(char*)&i;
 printf("%x %x %x %x\n",c[0],c[1],c[2],c[3]);
// internet is big endian: MSB on lowest address
// x86 is little endian: LSB on lowest address
 ii=(long*)cc;
 printf("%x\n",*ii);
 printf("%x\n",htonl(*ii));
}
```

Types et endianness

```c
#include <stdio.h>
#include <arpa/inet.h>  //  /usr/include/netinet/in.h

long main()
{int i=0x12345678;
 int *ii;
 char cc[8]={1,2,3,4,5,6,7,8};
 char *c;
 printf("%d\n",sizeof(i));
 c=(char*)&i;
 printf("%x %x %x %x\n",c[0],c[1],c[2],c[3]);
// internet is big endian: MSB on lowest address
// x86 is little endian: LSB on lowest address
 ii=(int*)cc;
 printf("%x\n",*ii);
 printf("%x\n",htonl(*ii));
}
```

```
8
78 56 34 12
4030201
1020304
```

```
sur model name : Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
```

Passage de paramètres de type arbitraire

Passage de paramètre lors de la création de tâche dans FreeRTOS [2]

```
BaseType_t xTaskCreate(TaskFunction_t pvTaskCode,
   const char * const pcName, unsigned short usStackDepth, void *pvParams,
   UBaseType_t uxPriority, TaskHandle_t *pxCreatedTask
  );

void vTaskCode( void * pvParameters )
{ for( ;; ) { /* Task code goes here. */ } }
```

Gestion des interruptions dans le noyau Linux [3]

```
int request_irq(unsigned int irq,
    irqreturn_t (*handler)(int, void *, struct pt_regs *),
    unsigned long flags,
    const char *dev_name,
    void *dev_id);

void free_irq(unsigned int irq, void *dev_id);
```

Passage de paramètre par `void*` : on fournit l'emplacement mémoire contenant le paramètre sans en préciser l'organisation.

2. https://www.freertos.org/a00125.html
3. https://www.oreilly.com/library/view/linux-device-drivers/

Passage de paramètres de type arbitraire

```c
#include <stdio.h>
void ma_fonction(void* param)
{long *p=(long*)param;
 printf("%lx",*p);
}

int main()
{long l=0x42;
 ma_fonction(&l);
}
```

```
$ gcc -Wall -o t t.c
$ ./t
42
```

Comment passer 4 entiers en argument du même prototype de fonction ?

Passage de paramètres de type arbitraire

```
#include <stdio.h>
struct mon_param {long p1;long  p2;long p3;long p4;};

void ma_fonction(void* param)
{struct mon_param *p=(struct mon_param*)param;
 printf("%lx %lx %lx %lx\n",p->p1,p->p2,p->p3,p->p4);
}

int main()
{struct mon_param p={.p1=0x42,.p2=0x43,.p3=0x44,.p4=0x45};
 ma_fonction(&p);
}
```

```
$ gcc -Wall -o t t.c
$ ./t
42 43 44 45
```

La notation (utilisée par exemple dans la déclaration des `fops` du noyau Linux[4]) est identique à

```
struct mon_parametre p;
p.p1=0x42; p.p2=0x43; p.p3=0x44; p.p4=0x45;
```

4. `https://www.tldp.org/LDP/lkmpg/2.4/html/c577.htm` : "you should be aware that any member of the structure which you don't explicitly assign will be initialized to NULL by gcc"

Pointeurs de fonctions

Une fonction n'est qu'une adresse mémoire à laquelle saute le processeur (assignation du PC) lors de son appel :

```c
#include <stdio.h>
int f1(void) {return(1);}
int f2(void) {return(2);}
int f3(int i,int j) {return(i+j);}

int main()
{int (*ma_fonction)(void);
 int (*fonction)(int,int);
 ma_fonction=&f1; printf("%d ",(ma_fonction)());
 ma_fonction=&f2; printf("%d ",(ma_fonction)());
 fonction=&f3; printf("%d\n",(fonction)(1,2));
}
```

```
$ gcc -Wall -o function function.c
$ ./function
1 2 3
```

⇒ possibilité de dynamiquement modifier le comportement d'un programme au cours de son exécution (e.g. pour s'adapter à un nouvel environnement)

Pointeurs de fonctions

Exemple d'utilisation de pointeurs de fonctions : [5]

```
/* Functions pointer which are initialized in eMBInit( ).
   Depending on the mode (RTU or ASCII) the are set to the
   correct implementations. */
static pvMBFrameStart pvMBFrameStartCur;
static pvMBFrameStop  pvMBFrameStopCur;

case MB_RTU:
     pvMBFrameStartCur = eMBRTUStart;
     pvMBFrameStopCur  = eMBRTUStop;
...
case MB_ASCII:
     pvMBFrameStartCur = eMBASCIIStart;
     pvMBFrameStopCur  = eMBASCIIStop;
peMBFrameSendCur = eMBASCIISend;
```

avec [6]

```
typedef void ( *pvMBFrameStart) ( void );
typedef void ( *pvMBFrameStop ) ( void );
```

fournit la capacité de fonctions polymorphiques (C++) au C.

---

5. https://github.com/cwalter-at/freemodbus/modbus/mb.c
6. https://github.com/cwalter-at/freemodbus/modbus/include/mbframe.h

Exemple du noyau Linux

Une structure `file_operations` contient les pointeurs de fonctions vers les implémentations des divers appels système

Exemple : `drivers/acpi/battery.c`

```
static struct acpi_driver acpi_battery_driver = {
        .name = "battery",
        .class = ACPI_BATTERY_CLASS,
        .ids = battery_device_ids,
        .flags = ACPI_DRIVER_ALL_NOTIFY_EVENTS,
        .ops = {.add = acpi_battery_add,
                .remove = acpi_battery_remove,
                .notify = acpi_battery_notify,
                },
        .drv.pm = &acpi_battery_pm,
};
```

ou `drivers/char/mem.c`

```
static const struct file_operations port_fops = {
        .llseek         = memory_lseek,
        .read           = read_port,
        .write          = write_port,
        .open           = open_port,
};
static int open_port(struct inode *inode, struct file *→
    ↪filp)
{return capable(CAP_SYS_RAWIO) ? 0 : -EPERM;}
```

Allocation de pointeurs ? !

NE JAMAIS FAIRE ÇA !

```
#ifndef __AVR__
#include <stdio.h>
#endif

char * ma_fonction(int i)
{char *c; if (i==0) c="87654321"; else c="12345678";}

int main()
{int k; volatile char *c;
 c=ma_fonction(0);
#ifndef __AVR__
 for (k=0;k<8;k++) printf("%c",c[k]);printf("\n");
#endif
}
```

```
$ gcc -o pointer pointer.c
$ ./pointer
87654321
```

Analyse du code assembleur

```
$ avr-gcc -o pointer pointer.c
$ file pointer
pointer: ELF 32-bit LSB executable, Atmel AVR 8-bit, version
$ avr-objdump -dStD pointer  # D pour data section
```

```
00000018 <ma_fonction>:
 18: cf 93  push    r28
 1a: df 93  push    r29
 1c: 00 d0  rcall   .+0        ; alloc 2 bytes on stack
 1e: 00 d0  rcall   .+0        ; re 2 bytes on stack (c*)
 20: cd b7  in      r28, 0x3d  ; read from stack
 22: de b7  in      r29, 0x3e  ; read from stack
 24: 9c 83  std     Y+4, r25   ; 0x04
 26: 8b 83  std     Y+3, r24   ; 0x03
 28: 8b 81  ldd     r24, Y+3   ; 0x03
 2a: 9c 81  ldd     r25, Y+4   ; 0x04
 2c: 89 2b  or      r24, r25
 2e: 29 f4  brne    .+10       ; 0x3a <ma_fonction+0x22>
 30: 80 e6  ldi     r24, 0x60  ; contenu de x60 en RAM
 32: 90 e0  ldi     r25, 0x00  ; 0
 34: 9a 83  std     Y+2, r25   ; 0x02
 36: 89 83  std     Y+1, r24   ; 0x01
 38: 04 c0  rjmp    .+8        ; 0x42 <__SREG__+0x3>
 3a: 89 e6  ldi     r24, 0x69  ; contenu de x69 en RAM
 3c: 90 e0  ldi     r25, 0x00  ; 0
 3e: 9a 83  std     Y+2, r25   ; renvoi du pointeur
 40: 89 83  std     Y+1, r24   ; contenu dans {r25,r24}
```

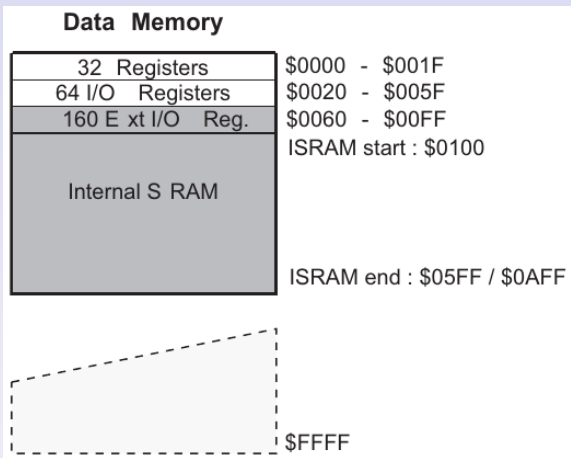
```
 42: 00 00  nop
 44: 0f 90  pop     r0         ; pop 4 bytes du stack
 46: 0f 90  pop     r0
 48: 0f 90  pop     r0
 4a: 0f 90  pop     r0
 4c: df 91  pop     r29
 4e: cf 91  pop     r28
 50: 08 95  ret

Disassembly of section .data:

00800060 <__data_start>:
 800060: 38 37  cpi     r19, 0x78       ; 120
 800062: 36 35  cpi     r19, 0x56       ; 86
 800064: 34 33  cpi     r19, 0x34       ; 52
 800066: 32 31  cpi     r19, 0x12       ; 18
 800068: 00 31  cpi     r16, 0x10       ; 16
 80006a: 32 33  cpi     r19, 0x32       ; 50
 80006c: 34 35  cpi     r19, 0x54       ; 84
 80006e: 36 37  cpi     r19, 0x76       ; 118
 800070: 38 00  .word   0x0038  ; ????
```

avec Y (R29 :R28) : stocke l'adresse du tableau dans {Y+1,Y+2}

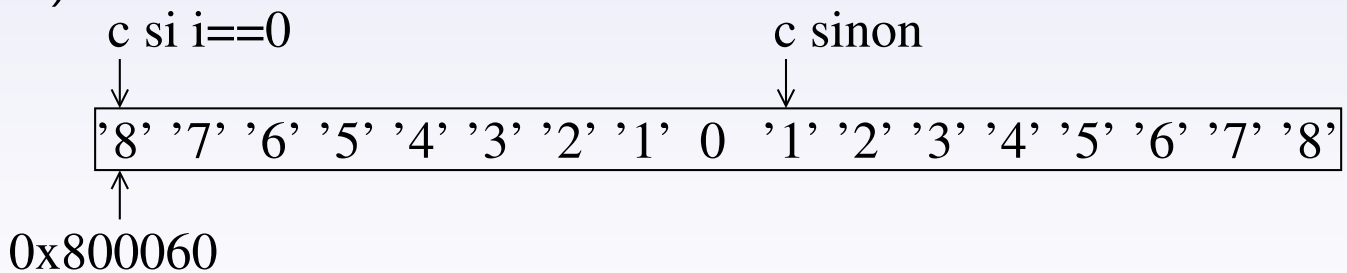
Analyse du code assembleur

```
$ avr-gcc -o pointer pointer.c
$ file pointer
pointer: ELF 32-bit LSB executable, Atmel AVR 8-bit, version
$ avr-objdump -dStD pointer  # D pour data section
```

```
00000052 <main>:
  52:   cf 93           push    r28
  54:   df 93           push    r29
  56:   00 d0           rcall   .+0             ; 0x58 <main+0x6>
  58:   cd b7           in      r28, 0x3d       ; 61
  5a:   de b7           in      r29, 0x3e       ; 62
  5c:   80 e0           ldi     r24, 0x00       ; 0
  5e:   90 e0           ldi     r25, 0x00       ; 0
  60:   db df           rcall   .-74            ; 0x18 <ma_fonction>
  62:   9a 83           std     Y+2, r25        ; 0x02
  64:   89 83           std     Y+1, r24        ; 0x01
```



`main` relit {Y+1,Y+2} qui contient l'adresse du tableau en mémoire (pointeur)



https://www.nongnu.org/avr-libc/user-manual/mem_sections.html Note that addr must be offset by adding 0x800000 to the real SRAM address so that the linker knows that the address is in the SRAM memory space. Thus, if you want the .data section to start at 0x1100, pass 0x801100 at the address to the linker.