Revisiter l'example « Blink » sur Arduino Uno.

« Blink » est aux électroniciens en herbe ce qu'est « Hello, World ! » aux programmeurs débutants : le code incontournable pour se familiariser avec une nouvelle platine de prototypage. A ceci près que cette platine met en œuvre un microcontrôleur en cachant ses spécificités pour faciliter la programmation.

Ce petit article a pour but d'ouvrir de nouvelles voies dans votre recherche d'optimisation de votre code, et plus particulièrement l'adressage direct des registres du microcontrôleur pour utiliser les entrées sorties digitales.

Je me suis basé sur l'Arduino Uno car cette platine est largement répandue : même si le microcontrôleur ATMega128P est nettement plus riche en fonctionnalités (et donc compliqué à comprendre), cela reste un choix judicieux par rapport à sa vitesse de mise en œuvre (prix de la platine, aspect « plug n' play », prise en charge de l'environnement de développement d'Arduino)

Le script d'exemple initial

Lorsque vous ouvrez l'exemple « Blink » depuis l'environnement Arduino (Menu « File > Examples > 01.Basics > Blink »), vous obtenez un script dont la version épurée ressemble à ceci :

```
void setup()
{
  pinMode(13, OUTPUT);
}

void loop()
{
  digitalWrite(13, HIGH);
  delay(1000);
  digitalWrite(13, LOW);
  delay(1000);
}
```

- La compilation de ce code nous indique que le binaire résultant occupera 1.030 octets soit 3% de l'espace de stockage total.

Quelle taille possède un script vide?

La première question judicieuse à se poser est : quelle est la taille d'un programme qui ne fait rien ? D'un point de vue technique, je vous accorde qu'un programme vide n'a aucun intérêt hormis de comprendre la trame sous-jacente du programme.

Enlevons quelques lignes de code pour obtenir le résultat suivant :

```
void setup()
{
}
```

```
void loop()
{
}
```

- Le résultat de la compilation nous indique que le résultat occupera 450 octets de mémoire, soit 1% de l'espace total.

Pourquoi un programme qui ne fait rien occupe-t-il autant de place ?

Ceci est une très bonne question que vous venez de vous poser…

Pour répondre à cela, nous pouvons essayer de trouver la manière dont le compilateur a transformé notre code source en langage assembleur. Pour arriver à cela, il faut savoir très, très, très sommairement comment la compilation a lieu :

- Votre code source est compilé
- Les fonctions provenant des librairies externes (les clauses « #include » que vous avez mentionnées ou implicitement ajoutées par le système) sont extraites
- Tous les objets sont assemblés (avec édition des liens) dans un fichier portant l'extension «.elf »
- De ce fichier « .elf » sont extraits :
  - La partie programme qui sera chargé dans la Flash (fichier « .hex »)
  - Les données qui seront stockées dans l'EEPROM (fichier « .eep »)

Le désassemblage est possible au départ du fichier « .elf », et de l'utilitaire « avr-objdump »[1]

La commande « [ARDUINO]\hardware\tools\avr\bin>avr-objdump.exe -d -S Blink.ino.elf > blink.txt » nous permet de voir que beaucoup de code est généré, tel que:

- Fonction principale qui appelle la fonction setup() et loop() (en boucle)
- Initialisation du microcontrôleur
- Initialisation des vecteurs d'interruption

```
Blink.ino.elf:     file format elf32-avr


Disassembly of section .text:

00000000 <__vectors>:
 * libraries or sketches that supports cooperative threads.
 *
 * Its defined as a weak symbol and it can be redefined to implement a
 * real cooperative scheduler.
 */
static void __empty() {
   0:0c 94 34 00    jmp    0x68   ; 0x68 <__ctors_end>
   4:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>
   8:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>
   c:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>
```

1 Présent dans le répertoire « [ARDUINO]\hardware\tools\avr\bin

```
  10:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>
  14:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>
  18:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>
  1c:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>
  20:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>
  24:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>
  28:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>
  2c:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>
  30:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>
  34:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>
  38:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>
  3c:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>
  40:  0c 94 5a 00   jmp    0xb4   ; 0xb4 <__vector_16>
  44:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>
  48:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>
  4c:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>
  50:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>
  54:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>
  58:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>
  5c:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>
  60:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>
  64:  0c 94 46 00   jmp    0x8c   ; 0x8c <__bad_interrupt>

00000068 <__ctors_end>:
  68:  11 24         eor    r1, r1
  6a:  1f be         out    0x3f, r1      ; 63
  6c:  cf ef         ldi    r28, 0xFF     ; 255
  6e:  d8 e0         ldi    r29, 0x08     ; 8
  70:  de bf         out    0x3e, r29     ; 62
  72:  cd bf         out    0x3d, r28     ; 61

00000074 <__do_clear_bss>:
  74:  21 e0         ldi    r18, 0x01     ; 1
  76:  a0 e0         ldi    r26, 0x00     ; 0
  78:  b1 e0         ldi    r27, 0x01     ; 1
  7a:  01 c0         rjmp   .+2           ; 0x7e <.do_clear_bss_start>

0000007c <.do_clear_bss_loop>:
  7c:  1d 92         st     X+, r1

0000007e <.do_clear_bss_start>:
  7e:  a9 30         cpi    r26, 0x09     ; 9
  80:  b2 07         cpc    r27, r18
  82:  e1 f7         brne   .-8           ; 0x7c <.do_clear_bss_loop>
  84:  0e 94 4b 00   call   0x96   ; 0x96 <main>
  88:  0c 94 df 00   jmp    0x1be  ; 0x1be <_exit>

0000008c <__bad_interrupt>:
  8c:  0c 94 00 00   jmp    0      ; 0x0 <__vectors>

00000090 <setup>:
void setup() {
  90:  08 95         ret

00000092 <loop>:
}
void loop() {
  92:  08 95         ret
```

```
00000094 <initVariant>:
int atexit(void (* /*func*/ )()) { return 0; }

// Weak empty variant initialization function.
// May be redefined by variant files.
void initVariant() __attribute__((weak));
void initVariant() { }
  94:  08 95         ret

00000096 <main>:
void setupUSB() __attribute__((weak));
void setupUSB() { }

int main(void)
{
       init();
  96:  0e 94 a4 00  call   0x148  ; 0x148 <init>

       initVariant();
  9a:  0e 94 4a 00  call   0x94   ; 0x94 <initVariant>

#if defined(USBCON)
       USBDevice.attach();
#endif

       setup();
  9e:  0e 94 48 00  call   0x90   ; 0x90 <setup>

       for (;;) {
              loop();
              if (serialEventRun) serialEventRun();
  a2:  c0 e0         ldi    r28, 0x00     ; 0
  a4:  d0 e0         ldi    r29, 0x00     ; 0
#endif

       setup();

       for (;;) {
              loop();
  a6:  0e 94 49 00  call   0x92   ; 0x92 <loop>
              if (serialEventRun) serialEventRun();
  aa:  20 97         sbiw   r28, 0x00     ; 0
  ac:  e1 f3         breq   .-8           ; 0xa6 <main+0x10>
  ae:  0e 94 00 00  call   0      ; 0x0 <__vectors>
  b2:  f9 cf         rjmp   .-14          ; 0xa6 <main+0x10>

000000b4 <__vector_16>:
#if       defined(__AVR_ATtiny24__)       ||       defined(__AVR_ATtiny44__)       ||
defined(__AVR_ATtiny84__)
ISR(TIM0_OVF_vect)
#else
ISR(TIMER0_OVF_vect)
#endif
{
  b4:  1f 92         push   r1
  b6:  0f 92         push   r0
  b8:  0f b6         in     r0, 0x3f     ; 63
  ba:  0f 92         push   r0
```

```
  bc:  11 24         eor    r1, r1
  be:  2f 93         push   r18
  c0:  3f 93         push   r19
  c2:  8f 93         push   r24
  c4:  9f 93         push   r25
  c6:  af 93         push   r26
  c8:  bf 93         push   r27
       // copy these to local variables so they can be stored in registers
       // (volatile variables must be read from memory on every access)
       unsigned long m = timer0_millis;
  ca:  80 91 01 01   lds    r24, 0x0101
  ce:  90 91 02 01   lds    r25, 0x0102
  d2:  a0 91 03 01   lds    r26, 0x0103
  d6:  b0 91 04 01   lds    r27, 0x0104
       unsigned char f = timer0_fract;
  da:  30 91 00 01   lds    r19, 0x0100

       m += MILLIS_INC;
       f += FRACT_INC;
  de:  23 e0         ldi    r18, 0x03    ; 3
  e0:  23 0f         add    r18, r19
       if (f >= FRACT_MAX) {
  e2:  2d 37         cpi    r18, 0x7D    ; 125
  e4:  20 f4         brcc   .+8          ; 0xee <__vector_16+0x3a>
       // copy these to local variables so they can be stored in registers
       // (volatile variables must be read from memory on every access)
       unsigned long m = timer0_millis;
       unsigned char f = timer0_fract;

       m += MILLIS_INC;
  e6:  01 96         adiw   r24, 0x01    ; 1
  e8:  a1 1d         adc    r26, r1
  ea:  b1 1d         adc    r27, r1
  ec:  05 c0         rjmp   .+10         ; 0xf8 <__vector_16+0x44>
       f += FRACT_INC;
       if (f >= FRACT_MAX) {
              f -= FRACT_MAX;
  ee:  26 e8         ldi    r18, 0x86    ; 134
  f0:  23 0f         add    r18, r19
              m += 1;
  f2:  02 96         adiw   r24, 0x02    ; 2
  f4:  a1 1d         adc    r26, r1
  f6:  b1 1d         adc    r27, r1
       }

       timer0_fract = f;
  f8:  20 93 00 01   sts    0x0100, r18
       timer0_millis = m;
  fc:  80 93 01 01   sts    0x0101, r24
 100:  90 93 02 01   sts    0x0102, r25
 104:  a0 93 03 01   sts    0x0103, r26
 108:  b0 93 04 01   sts    0x0104, r27
       timer0_overflow_count++;
 10c:  80 91 05 01   lds    r24, 0x0105
 110:  90 91 06 01   lds    r25, 0x0106
 114:  a0 91 07 01   lds    r26, 0x0107
 118:  b0 91 08 01   lds    r27, 0x0108
 11c:  01 96         adiw   r24, 0x01    ; 1
 11e:  a1 1d         adc    r26, r1
```

```
 120:  b1 1d         adc    r27, r1
 122:  80 93 05 01   sts    0x0105, r24
 126:  90 93 06 01   sts    0x0106, r25
 12a:  a0 93 07 01   sts    0x0107, r26
 12e:  b0 93 08 01   sts    0x0108, r27
}
 132:  bf 91         pop    r27
 134:  af 91         pop    r26
 136:  9f 91         pop    r25
 138:  8f 91         pop    r24
 13a:  3f 91         pop    r19
 13c:  2f 91         pop    r18
 13e:  0f 90         pop    r0
 140:  0f be         out    0x3f, r0     ; 63
 142:  0f 90         pop    r0
 144:  1f 90         pop    r1
 146:  18 95         reti

00000148 <init>:

void init()
{
       // this needs to be called before setup() or some functions won't
       // work there
       sei();
 148:  78 94         sei

       // on the ATmega168, timer 0 is also used for fast hardware pwm
       // (using phase-correct PWM would mean that timer 0 overflowed half as often
       // resulting in different millis() behavior on the ATmega8 and ATmega168)
#if defined(TCCR0A) && defined(WGM01)
       sbi(TCCR0A, WGM01);
 14a:  84 b5         in     r24, 0x24    ; 36
 14c:  82 60         ori    r24, 0x02    ; 2
 14e:  84 bd         out    0x24, r24    ; 36
       sbi(TCCR0A, WGM00);
 150:  84 b5         in     r24, 0x24    ; 36
 152:  81 60         ori    r24, 0x01    ; 1
 154:  84 bd         out    0x24, r24    ; 36
       // this combination is for the standard atmega8
       sbi(TCCR0, CS01);
       sbi(TCCR0, CS00);
#elif defined(TCCR0B) && defined(CS01) && defined(CS00)
       // this combination is for the standard 168/328/1280/2560
       sbi(TCCR0B, CS01);
 156:  85 b5         in     r24, 0x25    ; 37
 158:  82 60         ori    r24, 0x02    ; 2
 15a:  85 bd         out    0x25, r24    ; 37
       sbi(TCCR0B, CS00);
 15c:  85 b5         in     r24, 0x25    ; 37
 15e:  81 60         ori    r24, 0x01    ; 1
 160:  85 bd         out    0x25, r24    ; 37

       // enable timer 0 overflow interrupt
#if defined(TIMSK) && defined(TOIE0)
       sbi(TIMSK, TOIE0);
#elif defined(TIMSK0) && defined(TOIE0)
       sbi(TIMSK0, TOIE0);
```

```
 162:  ee e6         ldi    r30, 0x6E    ; 110
 164:  f0 e0         ldi    r31, 0x00    ; 0
 166:  80 81         ld     r24, Z
 168:  81 60         ori    r24, 0x01    ; 1
 16a:  80 83         st     Z, r24
       // this is better for motors as it ensures an even waveform
       // note, however, that fast pwm mode can achieve a frequency of up
       // 8 MHz (with a 16 MHz clock) at 50% duty cycle

#if defined(TCCR1B) && defined(CS11) && defined(CS10)
       TCCR1B = 0;
 16c:  e1 e8         ldi    r30, 0x81    ; 129
 16e:  f0 e0         ldi    r31, 0x00    ; 0
 170:  10 82         st     Z, r1

       // set timer 1 prescale factor to 64
       sbi(TCCR1B, CS11);
 172:  80 81         ld     r24, Z
 174:  82 60         ori    r24, 0x02    ; 2
 176:  80 83         st     Z, r24
#if F_CPU >= 8000000L
       sbi(TCCR1B, CS10);
 178:  80 81         ld     r24, Z
 17a:  81 60         ori    r24, 0x01    ; 1
 17c:  80 83         st     Z, r24
       sbi(TCCR1, CS10);
#endif
#endif
       // put timer 1 in 8-bit phase correct pwm mode
#if defined(TCCR1A) && defined(WGM10)
       sbi(TCCR1A, WGM10);
 17e:  e0 e8         ldi    r30, 0x80    ; 128
 180:  f0 e0         ldi    r31, 0x00    ; 0
 182:  80 81         ld     r24, Z
 184:  81 60         ori    r24, 0x01    ; 1
 186:  80 83         st     Z, r24

       // set timer 2 prescale factor to 64
#if defined(TCCR2) && defined(CS22)
       sbi(TCCR2, CS22);
#elif defined(TCCR2B) && defined(CS22)
       sbi(TCCR2B, CS22);
 188:  e1 eb         ldi    r30, 0xB1    ; 177
 18a:  f0 e0         ldi    r31, 0x00    ; 0
 18c:  80 81         ld     r24, Z
 18e:  84 60         ori    r24, 0x04    ; 4
 190:  80 83         st     Z, r24

       // configure timer 2 for phase correct pwm (8-bit)
#if defined(TCCR2) && defined(WGM20)
       sbi(TCCR2, WGM20);
#elif defined(TCCR2A) && defined(WGM20)
       sbi(TCCR2A, WGM20);
 192:  e0 eb         ldi    r30, 0xB0    ; 176
 194:  f0 e0         ldi    r31, 0x00    ; 0
 196:  80 81         ld     r24, Z
 198:  81 60         ori    r24, 0x01    ; 1
 19a:  80 83         st     Z, r24
#endif
```

```
#if defined(ADCSRA)
      // set a2d prescaler so we are inside the desired 50-200 KHz range.
      #if F_CPU >= 16000000 // 16 MHz / 128 = 125 KHz
            sbi(ADCSRA, ADPS2);
 19c:  ea e7        ldi    r30, 0x7A    ; 122
 19e:  f0 e0        ldi    r31, 0x00    ; 0
 1a0:  80 81        ld     r24, Z
 1a2:  84 60        ori    r24, 0x04    ; 4
 1a4:  80 83        st     Z, r24
            sbi(ADCSRA, ADPS1);
 1a6:  80 81        ld     r24, Z
 1a8:  82 60        ori    r24, 0x02    ; 2
 1aa:  80 83        st     Z, r24
            sbi(ADCSRA, ADPS0);
 1ac:  80 81        ld     r24, Z
 1ae:  81 60        ori    r24, 0x01    ; 1
 1b0:  80 83        st     Z, r24
            cbi(ADCSRA, ADPS2);
            cbi(ADCSRA, ADPS1);
            sbi(ADCSRA, ADPS0);
      #endif
      // enable a2d conversions
      sbi(ADCSRA, ADEN);
 1b2:  80 81        ld     r24, Z
 1b4:  80 68        ori    r24, 0x80    ; 128
 1b6:  80 83        st     Z, r24
      // here so they can be used as normal digital i/o; they will be
      // reconnected in Serial.begin()
#if defined(UCSRB)
      UCSRB = 0;
#elif defined(UCSR0B)
      UCSR0B = 0;
 1b8:  10 92 c1 00  sts    0x00C1, r1
 1bc:  08 95        ret

000001be <_exit>:
 1be:  f8 94        cli

000001c0 <__stop_program>:
 1c0:  ff cf        rjmp   .-2          ; 0x1c0 <__stop_program>
```

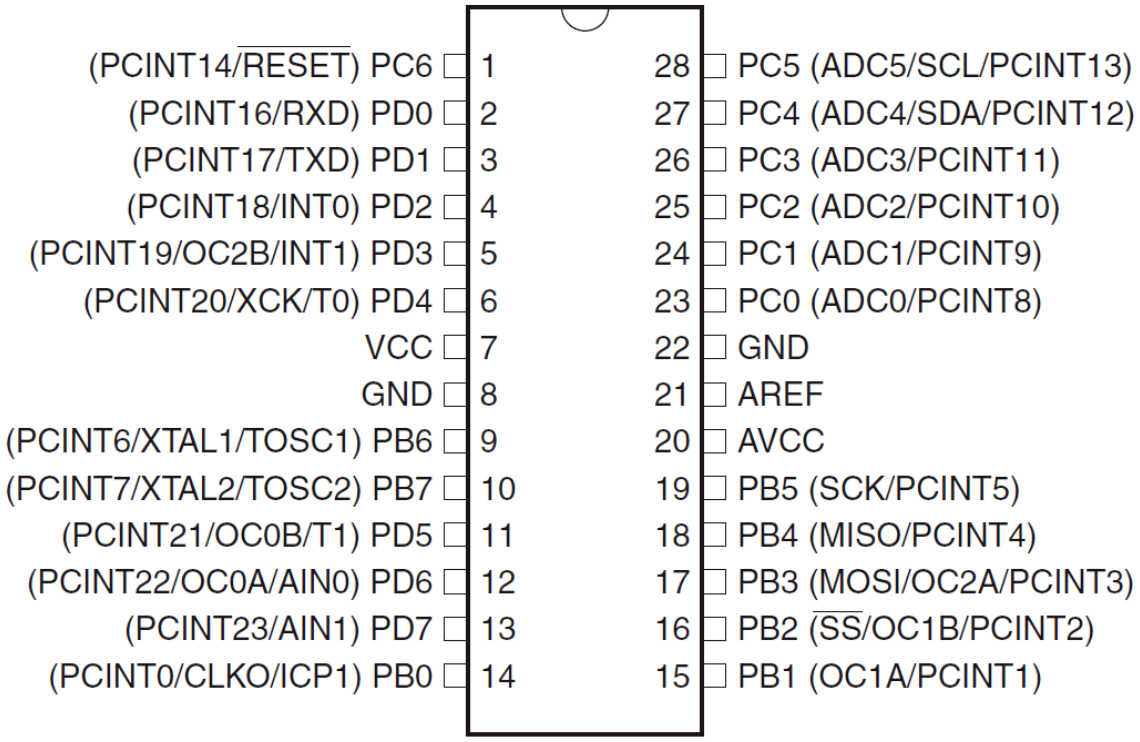
Notre programme n'occupe que 4 octets: 2 commandes « **RET**our » (lignes 0x90 et 0x92) : vu que nos fonctions sont actuellement vide, ceci est tout à fait normal.

Penchons-nous sur le microcontrôleur

Avant de pouvoir optimiser ce code d'exemple, il est nécessaire de comprendre un peu mieux quelques aspects du fonctionnement interne d'un microcontrôleur et général et de l'ATMega328P en particulier.

Pour résumer, un microcontrôleur est regroupe dans un même boitier d'un processeur, de la mémoire pour le programme et les données, ainsi que de fonctions de gestions d'entrées/sorties de toutes sortes (digitales, analogiques, série, SPI, I2C, ...)

Pour communiquer avec le monde extérieur le microcontrôleur dispose des pins de son boitier. Il est courant que plusieurs types d'entrées/sorties se partagent la même pin, comme l'illustre le schéma ci-dessous :



Pour mieux comprendre les E/S de l'ATmega328P, il faudra impérativement se plonger dans sa documentation technique. La page du produit disponible sur le site d'Atmel à l'adresse http://www.atmel.com/devices/ATMEGA328P.aspx, nous donne les liens vers la documentation complète du microcontrôleur.

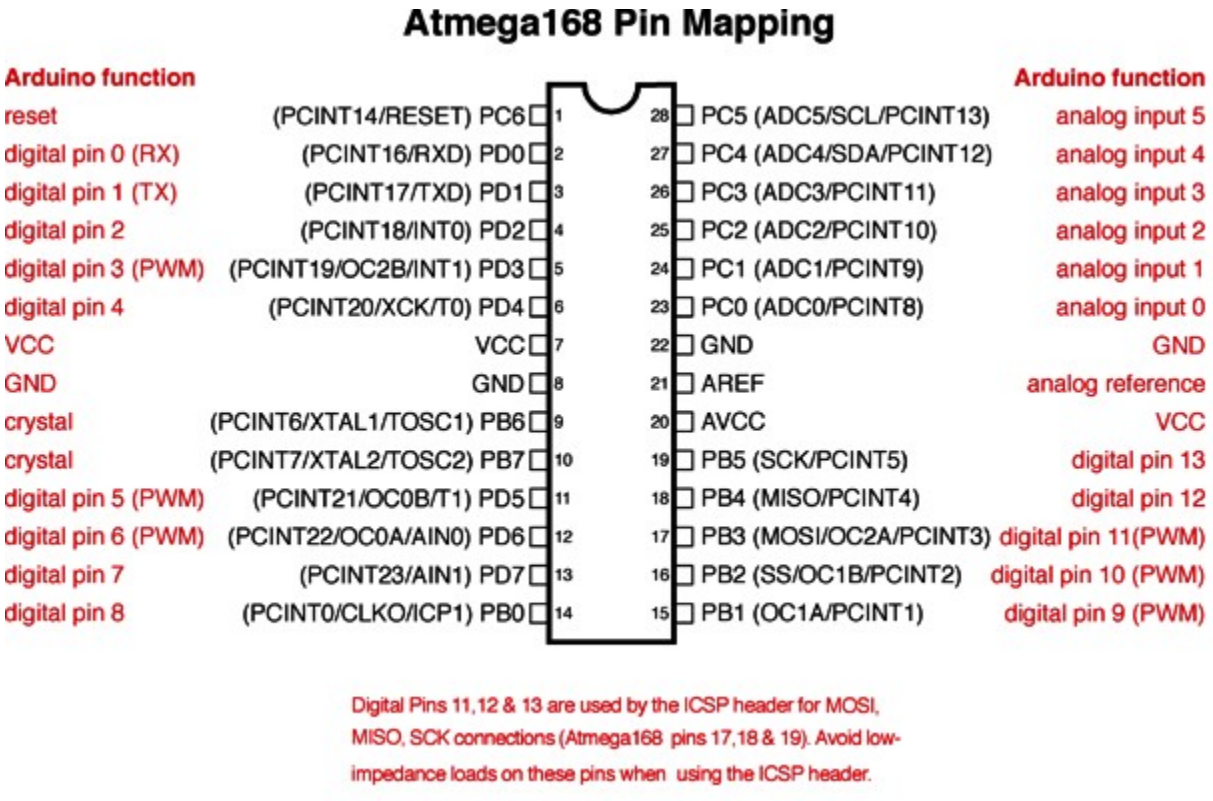
Selon cette documentation, nous apprenons que l'ATMega328P nous offre pas de 23 lignes E/S. En fait, chaque ligne correspond à un bit dans un octet appelé port : nous trouvons 3 ports (B, C, D) x 8 bits (sauf pour le port C qui n'en utilise que 7) ; ce qui nous fait 23 bits possibles.

Pour gérer chaque port, 3 registres seront nécessaires :

- Le registre DDRx (Data Direction Register) se chargeant de définir la direction (entrée ou sortie)
- Le registre PORTx se charge d'activer ou non la résistance pull-up en mode input, ou d'envoyer la valeur LOW/HIGH en mode output
- Le registre PINx permet de lire directement l'état d'une ligne sans passer par le registre PORTx

Relation entre les sorties du microcontrôleur et les ports de l'Arduino

Pour savoir quel registre il faudra manipuler pour allumer et éteindre la LED située sur le port 13 de l'Arduino, il faut cette fois-ci consulter la documentation présente sur le site http://www.arduino.cc. Dans la documentation de la platine, nous trouverons un lien (https://www.arduino.cc/en/Hacking/PinMapping168) vers un schéma de mapping entre les pins de la platine et les pins du microcontrôleur :



Ce schéma nous enseigne que le pin 13 de l'Arduino est relié à la ligne PB5 du PORT B.

En toute logique, nous devrons donc utiliser les registres DDRB et PORTB pour faire clignoter notre LED.

Logique du programme

L'algorithme que nous devrons mettre en place pourrait se décrire de la manière suivante :

setup ()	**pinMode(13, OUTPUT)** Cela se traduirait par mettre la valeur 1 dans le bit « 5 » du registre DDRB En logique booléenne, la mise à 1 d'un bit sans toucher aux autres bits se fait à l'aide d'une opération « OU » ➔ DDRB = DDRB OU 00100000
Loop( )	**digitalWrite(13, HIGH)** Cela signifie qu'il faut mettre la valeur 1 dans le bit 5 du registre PORTB

	En appliquant le principe précédant, on devra code la commande suivante : ➔ PORTB = PORTB OU 00100000 digitalWrite(13, LOW) Cela signifie qu'il faut forcer la valeur 0 dans le bit 5 du registre PORTB. En logique booléenne, il faut faire une comparaison logique ET avec un masque dont les bits à initaliser sont mis à 0 ➔ PORTB = PORTB ET 11011111

Le script revisité

Version 1 – Sans pinMode() ni digitalWrite()

A l'aide de nos observations précédentes, nous sommes maintenant en mesure de pouvoir optimiser le code d'exemple comme suit :

```
void setup()
{
  DDRB |= (1 << 5);
}

void loop()
{
  PORTB |= (1 << 5);
  delay(1000);

  PORTB &= ~(1 << 5);
  delay(1000);
}
```

Vous trouverez quelques optimisations d'écritures :

DDRB |= (1 << 5)	• « |= » est la contraction de la commande DDRB = DDRB | (1 << 5) • « 1 << 5 » permet de créer un masque avec un unique bit qui est positionné en 5ème position afin de fournir comme valeur 100000
PORTB |= (1 << 5)	• « |= » est la contraction de la commande PORTB = PORTB | (1 << 5) • « 1 << 5 » permet de créer un masque avec un unique bit qui est positionné en 5ème position afin de fournir comme valeur 100000
PORTB |= ~(1 << 5)	• « |= » est la contraction de la commande PORTB = PORTB | (1 << 5)

	• « 1 << 5 » permet de créer un masque avec un unique bit qui est positionné en 5ème position afin de fournir comme valeur 100000 • « ~ » inverse la valeur pour convertir le 1 en 0 et vice versa, de sorte que ~00100000 devienne 11011111

➢ Avec cette version nous passons à 630 octets

Version 2 – 1 seule instruction

Comme seconde optimisation, nous pouvons apporter une amélioration par rapport aux 2 instructions consistant à mettre le bit consécutivement à 1 puis à 0. En fait, il s'agit d'une opération eXclusive-OR (XOR ou ^).

Le listing résultant sera :

```
void setup()
{
  DDRB |= (1 << 5);
}

void loop()
{
  PORTB ^= (1 << 5);
  delay(1000);
}
```

➢ Avec cette version, nous passons à 622 octets

Version 3 – sans delay()

Dans cette dernière optimisation, nous allons nous passer de la fonction delay(). Pour remplacer cette fonction, il faut mettre en place un peu plus de logique

- Variable globale « dernière exécution »
- Dans la boucle loop()
  - « exécution courante » = millis() □ Récupérer la valeur actuelle de l'horloge[2]
  - Si la dernière exécution + 1000 millisecondes < exécution courante
    - PIN = PIN XOR 1
    - « Dernière exécution » = « exécution courante »

En code cela donnerait :

```
unsigned long last_run;

void setup()
```

2 Attention avec ce code, on risquera à tous les coups un plantage : la valeur de l'horloge étant codée sur 1 mot (2 octets) ; après 65536 itérations on repart à 0. Le code devrait être blindé pour tenir compte de cet aspect cyclique.

```
{
  DDRB |= (1 << 1);
}

void loop()
{
  unsigned long  this_run = millis();

  if (last_run + 1000 < this_run)
  {
    PORTB ^= (1 << 1);
    last_run = this_run;
  }
}
```

- Nous somme passés à 546 octets !

Résumé des performances

Script	Taille	Gain
Script initial	1.030 octets	-
Sans pinMode() ni digitalWrite()	630 octets	400 octets (-38%)
eXclusive OR	622 octets	408 octets (-40%)
Sans delay()	546 octets	484 octets (-47%)

En conclusion

Un environnement de développement, tel que celui fourni par Arduino, a l'avantage de faciliter la réalisation rapide de projets. Lorsqu'il s'agit de finaliser son projet, il convient d'optimiser son code en se rapprochant des commandes du microcontrôleur et en s'éloignant de fonctions de haut niveau.

L'ultime optimisation consisterait à :

- Ecrire son code en dehors du squelette imposé par Arduino (fonctions setup() et loop()) : l'équivalent d'une fonction main() contenant une boucle infinie
- Ecrire son code en assembleur : vu la complexité du programme, on pourrait travailler qu'avec les registres de travail et passer à moins de 100 octets[3]

3 Si un sketch vide fait 450 octets, on est en droit de penser qu'un programme assembleur pur couterait moins de 50 octets.