# Documentation technique

*Programme NewDriverMega.ino*

Voici le code étudié :

```
1   #include <avr/sleep.h>
2   #include <avr/power.h>
3
4   #define N_PATTERNS 2
5   #define N_PORTS 10
6   #define N_DIVS 10
7   #define PATTERN_PERIODS 400000  // nombre de périodes 40 kHz avant de
7   passer au pattern suivant donc là on fait 500000*1/40000 donc 12.5s
7
8   #define WAIT() __asm__
9   __volatile__("nop\n\t""nop\n\t""nop\n\t""nop\n\t""nop\n\t""nop\n\t""n
9   op\n\t""nop\n\t""nop\n\t""nop\n\t""nop\n\t""nop\n\t""nop\n\t""nop\n\t
9   ")
9
10  #define OUTPUT_WAVE(pointer, d)  \
11    PORTA = pointer[d*N_PORTS + 0]; \
11    PORTC = pointer[d*N_PORTS + 1]; \
11    PORTL = pointer[d*N_PORTS + 2]; \
11    PORTB = pointer[d*N_PORTS + 3]; \
11    PORTK = pointer[d*N_PORTS + 4]; \
11    PORTF = pointer[d*N_PORTS + 5]; \
11    PORTH = pointer[d*N_PORTS + 6]; \
11    PORTD = pointer[d*N_PORTS + 7]; \
11    PORTG = pointer[d*N_PORTS + 8]; \
11    PORTJ = pointer[d*N_PORTS + 9];
11
12
13  static const byte patterns[N_PATTERNS * N_DIVS * N_PORTS] = {
14
15
16     0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
17    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
18    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
19    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
20    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
21    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
22    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
23    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
24    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
25    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
26
27
28     0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
29    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
30    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
31    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
32    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
33    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

```
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

      };

static byte bufferA[N_DIVS * N_PORTS];
static byte bufferB[N_DIVS * N_PORTS];

void setup() {
  // --- Configuration des ports en sortie ---
  DDRA = DDRC = DDRL = DDRB = DDRK = DDRF = DDRH = DDRD = DDRG = DDRJ
= 0xFF;
  PORTA = PORTC = PORTL = PORTB = PORTK = PORTF = PORTH = PORTD =
PORTG = PORTJ = 0x00;

  //clear the buffers
  for (int i = 0; i < (N_PATTERNS * N_DIVS * N_PORTS); ++i) {
    bufferA[i] = 0;
    bufferB[i] = 0;
  }

  //initial pattern ==> sinus avec toute les sorties qui sont en
phases
  for (int i = 0; i < (N_PORTS*N_DIVS/2); ++i){
     bufferA[i] = 0xFF;
  }

  // --- PWM 40 kHz sur pin 2 ---
  pinMode(2, OUTPUT);
  noInterrupts();
  TCCR3A = bit(WGM10) | bit(WGM11) | bit(COM1B1);
  TCCR3B = bit(WGM12) | bit(WGM13) | bit(CS10);
  OCR3A = (F_CPU / 40000L) - 3;
  OCR3B = (F_CPU / 40000L) / 2;
  interrupts();

  // --- Sync in pin ---
  pinMode(3, INPUT_PULLUP); // connectée à pin 2 pour synchro

  // --- Désactivation des périphériques inutiles ---
  ADCSRA = 0;
  power_adc_disable();
  power_spi_disable();
  power_twi_disable();
  power_timer0_disable();
  power_usart1_disable();
  power_usart2_disable();
  power_usart3_disable();

  // L'idée est la suivante : on parcourt "patterns" par pas de N_Div
* N_ports avec deux buffers comme avant, un lecture et un écriture
  // La variable de ce code est donc le timing que l'on souhaite
entre chaque chgmt de pattern = "pattern_periods"

  bool useA = true;
  unsigned long currentPeriods = 0;
  byte currentPatternIndex = 0;  // index du pattern courant dans
patterns
```

```
89
89    byte* activeBuffer = bufferA;
91    byte* nextBuffer   = bufferB;
92
93    // copie initiale du premier pattern dans bufferA
94    for(int i=0; i<N_PORTS*N_DIVS; i++)
95        bufferA[i] = patterns[i];
96
97    // pré-remplissage du bufferB avec le pattern suivant
98    for(int i=0; i<N_PORTS*N_DIVS; i++)
99        bufferB[i] = patterns[(currentPatternIndex + 1) * N_PORTS *
100 N_DIVS + i];
101
101   byte* emittingPointerH = &activeBuffer[0];
102   byte* emittingPointerL = &activeBuffer[N_PORTS * N_DIVS / 2];
103
104   LOOP:
105     while (PINE & 0b00100000); // attente synchro
106
107     OUTPUT_WAVE(emittingPointerH, 0); WAIT();
108     OUTPUT_WAVE(emittingPointerH, 1); WAIT();
109     OUTPUT_WAVE(emittingPointerH, 2); WAIT();
110     OUTPUT_WAVE(emittingPointerH, 3); WAIT();
111     OUTPUT_WAVE(emittingPointerL, 0); WAIT();
112     OUTPUT_WAVE(emittingPointerL, 1); WAIT();
113     OUTPUT_WAVE(emittingPointerL, 2); WAIT();
114     OUTPUT_WAVE(emittingPointerL, 3); WAIT();
115
116     // incrémente compteur de périodes
117     currentPeriods++;
118     if(currentPeriods >= PATTERN_PERIODS) {
119         currentPeriods = 0;
120
121         // switch des buffers
122         useA = !useA;
123         activeBuffer = useA ? bufferA : bufferB;
124         nextBuffer   = useA ? bufferB : bufferA;
125
125         // avance pattern
126         currentPatternIndex++;
127         if(currentPatternIndex >= N_PATTERNS) currentPatternIndex =
128 0;
129
130         // met à jour pointeurs pour le buffer actif
131         emittingPointerH = &activeBuffer[0];
132         emittingPointerL = &activeBuffer[N_PORTS * N_DIVS / 2];
133
134         // prépare le prochain pattern dans le buffer inactif
134         int nextPatternIndex = currentPatternIndex + 1;
135         if(nextPatternIndex >= N_PATTERNS) nextPatternIndex = 0;
136
137         for(int i=0; i<N_PORTS*N_DIVS; i++)
138             nextBuffer[i] = patterns[nextPatternIndex * N_PORTS *
138 N_DIVS + i];
139     }
139
140     goto LOOP;
141 }
142
143 // Boucle Arduino vide (jamais utilisée)
```

```
144  void loop() {}
145
146
```

## 1. Suppression des fonctions superflues

Notre volonté première en modifiant ce code est de simplifier au maximum le code de départ.

Cela passe par la suppression des commandes superflues pour notre application :

```
13  #define COMMAND_SWITCH 0b00000000
14  #define COMMAND_DURATION 0b00110000
15  #define MASK_DURATION 0b00111111
16  #define COMMAND_COMMITDURATIONS 0b00010000
```

Ces variables étaient précédemment utilisées dans le but de permettre un envoi en temps réel des phases alors calculées avec le logiciel Java.
Or nous souhaitons nous détacher de ce logiciel et n'avons pas besoin de faire du temps réel. Nous pouvons donc nous permettre de supprimer ces commandes.

Il en va de même pour ces variables-ci :

```
87  byte bReceived = 0; //octet reçu via Serial
88   bool byteReady = false; //drapeau : un octet est disponible ?
89   bool isSwitch = false; //drapeau : commande SWITCH reçue ?
90   bool isPatternForMe = false; //drapeau : cet octet fait partie du
90  pattern à charger ?
91   bool isDuration = false; //drapeau : cet octet contient un fragment
91  de durée ?
92   bool isCommitDurations = false; //drapeau : commande de validation
92  des durées reçue ?
93   byte nextMsg = 0; //valeur à renvoyer par défaut si commande inconnue
93   int writtingIndex = 0; //index pour l'écriture progressive d'un
94  pattern
94
    //Sauvegarde des pointeurs "à zéro" pour pouvoir réinitialiser après
100 un switch
100  byte* emittingPointerZeroH = & bufferA[0];
101  byte* emittingPointerZeroL = & bufferA[N_PORTS * N_DIVS / 2];
102 //Pointeurs pour écrire dans le buffer de réception (l'autre buffer)
102  byte* readingPointerH = & bufferB[0];
103  byte* readingPointerL = & bufferB[N_PORTS * N_DIVS / 2];
104
105  byte durations[N_PATTERNS];
106  byte durationsBuffer[N_PATTERNS];
107


     byte currentPeriods = 0;//compte les divisions émises dans ce motif
114  byte durationsPointer = 0;//index bit-à-bit lors de la réception  des
115 durées
115  byte currentDuration = 0;//durée lue pour le motif en cours
116  bool patternComplete = false;//drapeau : motif complètement émis ?
117  bool lastPattern = false;//drapeau : on est sur le dernier motif (31)
118 ?
118
    byte nextDuration = 0;//durée du motif suivant
```

```
120   bool returnToFirstPattern = false; //drapeau : la séquence doit
121  revenir au motif 0 ?
121
```

Certaines variables sont cependant similaires au premier code :

**I- Le booléen permettant de savoir quel buffer est actuellement utilisé**

Avant on utilisait :

```
96    bool emittingA = true;
```

Maintenant on utilise :

```
87    bool useA = true;
```

**II- La variable permettant d'indiquer le pattern actuellement émis**

Avant on utilisait :

```
113    byte currentPattern = 0;
```

Maintenant on utilise :

```
89    byte currentPatternIndex = 0;  // index du pattern courant dans
89   patterns
```

**III- Le buffer suivant à traiter**

Avant on utilisait :

```
119    byte nextPattern = 0;
```

Maintenant on utilise :

```
92    byte* activeBuffer = bufferA;
93    byte* nextBuffer   = bufferB;
```

## 2. Choix de la durée du wait en fonction du processeur Arduino

Le code DriverMega utilise une fonction permettant d'attendre à chaque envoi de pattern afin de combler les latences intrinsèques du processeur Arduino, permettant ainsi d'assurer l'envoi synchrone des informations.

La fonction utilisée est la suivante :

```
13  #define WAIT(a) __asm__ __volatile__ ("nop")
```

Pourquoi utiliser cette fonction ? Afin de remplacer les fonctions/variables du programme initial qui entraînait des tours de processeur dans le programme initial (en gras) :

```
129   OUTPUT_WAVE(emittingPointerH, 0); //émission des 8 bits de chaque
      port à la division 0
129   byteReady = Serial._dataAvailable(); //on vérifie si un octet est
      déjà arrivé dans le buffer matériel du Serial
130   OUTPUT_WAVE(emittingPointerH, 1); //émission de la division 1
130   bReceived = Serial._peekData(); //on lit l'octet en tête si
      présent, mais on ne le retire pas du buffer.
131   OUTPUT_WAVE(emittingPointerH, 2); //émission de la division 2
131   isSwitch = bReceived == COMMAND_SWITCH; //si l'octet cumulé vaut
      exactement 0x00, on prépare le flag "switch buffer".
131   isCommitDurations = bReceived == COMMAND_COMMITDURATIONS; //même
      principe pour la validation des durées
132   OUTPUT_WAVE(emittingPointerH, 3); //émission de la division 3
132   isPatternForMe = (bReceived & 0b00001111) == 1; //on regarde les 4
      LSB pour voir si ce byte contient un fragment du pattern
      ++currentPeriods; //on incrémente le compteur de périodes déjà
132   émises pour le motif courant.
      OUTPUT_WAVE(emittingPointerH, 4); //émission de la division 4
133   nextMsg =  bReceived - 1; //pré-calcule la réponse générique (ack
133   ou message d'erreur minimaliste.
      OUTPUT_WAVE(emittingPointerL, 0); //émission de la division 0
134   isDuration = (bReceived & MASK_DURATION) == COMMAND_DURATION; //on
134   masque les 6 LSB et compare à 0x30 : détecte un byte "durée".
      nextPattern = currentPattern + 1; //on anticipe l'index du motif
134   suivant
      OUTPUT_WAVE(emittingPointerL, 1); //émission de la division 1
135   nextDuration = durations[nextPattern]; //lecture de la durée
135   stockée pour le motif suivant
      OUTPUT_WAVE(emittingPointerL, 2); //émission de la division 2
136   patternComplete = (currentPeriods ==
136   durations[currentPattern]);//vrai si on a déjà émis autant de
      divisions que la durée allouée au motif en cours
      OUTPUT_WAVE(emittingPointerL, 3); //émission de la division 3
137   lastPattern = (currentPattern+1 == N_PATTERNS); //vrai si on était
137   sur le dernier motif (31)
137
```

```
138    returnToFirstPattern = nextDuration == 0; //vrai si le motif
      suivant est configuré pour durer 0 divisions.
       OUTPUT_WAVE(emittingPointerL, 4); //émission de la division 4
```

En pratique, l'appel aux différentes variables ainsi qu'aux fonctions consomme des cycles de processeur : isDuration, nextPattern, patternComplete, ...
Comme ces fonctions / variables ne sont plus présentes dans le nouveau programme, il fallait donc les remplacer de sorte à combler les cycles de processeur. C'est ici qu'intervient la fonction wait().

Le code précédent est ainsi remplacé par cette partie du nouveau code :

```
106  LOOP:
107    while (PINE & 0b00100000); // attente synchro
108
109    OUTPUT_WAVE(emittingPointerH, 0); WAIT();
110    OUTPUT_WAVE(emittingPointerH, 1); WAIT();
111    OUTPUT_WAVE(emittingPointerH, 2); WAIT();
112    OUTPUT_WAVE(emittingPointerH, 3); WAIT();
113    OUTPUT_WAVE(emittingPointerL, 0); WAIT();
114    OUTPUT_WAVE(emittingPointerL, 1); WAIT();
115    OUTPUT_WAVE(emittingPointerL, 2); WAIT();
116    OUTPUT_WAVE(emittingPointerL, 3); WAIT();
```

## 3. Choix de la durée du wait en fonction du processeur Arduino

En fonction du code et des fonctions/variables utilisées, le nombre de "nop" en entrée de la fonction WAIT doit être modifié. En effet, deux fonctions différentes correspondent à deux nombres de tours de processeurs différents (et de même pour le remplissage de variable en mémoire).

Donc de la même façon que le code doit être adapté si l'on change de fonction, il faut adapté la "longueur" de la fonction WAIT() en fonction de la forme de notre code.

De ce fait, lorsque l'on change le nombre de patterns que l'on souhaite envoyés, il faut modifié le nombre de "nop" en entrée de la fonction WAIT().

Prenons deux exemples :

**I- Le cas où l'on souhaite envoyer deux patterns**

La fonction WAIT() est alors définie comme :

```
12  #define WAIT() __asm__
    __volatile__("nop\n\t""nop\n\t""nop\n\t""nop\n\t""nop\n\t""nop\n
    \t""nop\n\t""nop\n\t""nop\n\t""nop\n\t""nop\n\t""nop\n\t""nop\n\
    t""nop\n\t")
```

**II- Le cas où l'on souhaite envoyer deux patterns**

La fonction WAIT() est alors définie comme :

```
12  #define WAIT() __asm__
    __volatile__("nop\n\t""nop\n\t""nop\n\t""nop\n\t""nop\n\t")
```

Notons pour finir que le nombre de "nop" a appliqué est entièrement empirique est doit être déterminé expérimentalement.

La méthode est la suivante :

- Brancher l'arduino à l'ordinateur (sans le proto) ;
- Ne pas oubliez de brancher les pins 2 et 3 entre elles pour assurer la synchronisation ;
- Récupérez le signal d'un pin à l'oscilloscope (en n'oubliant pas de connecter la masse ) ;
- Envoyer le code ;
- Vérifier que le créneau est stable à 40kHz et symétrique.

Vous vous retrouverez souvent dans le cas où plusieurs signaux seront superposés. il faudra alors changer le nombre de "nop" car cette configuration n'est pas la bonne.

## 4. Liste des variables essentielles qui ont été ajoutées

- `bool useA : variable permettant de savoir si on doit remplir le Buffer A ou le Buffer B ;`
- `#define PATTERN_PERIODS 400000 : variable désignant la durée d'un pattern. Une fois la durée`
- `unsigned long currentPeriods : nous souhaitons pouvoir modifier la période de changement de pattern aisément. CurrentPeriods est le compteur qui s'incrémente en continu pour savoir si la constante PATTERN_PERIODS est atteinte.`
- `byte currentPatternIndex = 0;`