Documentation technique

Programme DriverMega.ino

Voici le code étudié :

```
13  #include <avr/sleep.h>
14  #include <avr/power.h>
15
16  #define N_PATTERNS 32
17
18  #define N_PORTS 10
19  #define N_DIVS 10
20
21  //ports A C L B K F H D G J
22
23  #define COMMAND_SWITCH 0b00000000
24  #define COMMAND_DURATION 0b00110000
25  #define MASK_DURATION 0b00111111
26  #define COMMAND_COMMITDURATIONS 0b00010000
27
28  #define WAIT(a) __asm__ __volatile__ ("nop")
29  #define OUTPUT_WAVE(pointer, d)  PORTA = pointer[d*N_PORTS + 0];
29  PORTC = pointer[d*N_PORTS + 1]; PORTL = pointer[d*N_PORTS + 2]; PORTB
29  = pointer[d*N_PORTS + 3]; PORTK = pointer[d*N_PORTS + 4]; PORTF =
29  pointer[d*N_PORTS + 5]; PORTH = pointer[d*N_PORTS + 6];  PORTD =
29  pointer[d*N_PORTS + 7]; PORTG = pointer[d*N_PORTS + 8]; PORTJ =
29  pointer[d*N_PORTS + 9]
30
31  static byte bufferA[N_PATTERNS * N_DIVS * N_PORTS];
32  static byte bufferB[N_PATTERNS * N_DIVS * N_PORTS];
33
34  void setup()
35  {
36   //set as output ports A C L B K F H D G J
37   DDRA = DDRC = DDRL = DDRB = DDRK = DDRF = DDRH = DDRD = DDRG = DDRJ
37  = 0xFF;
38  //low signal on all of them
39   PORTA = PORTC = PORTL = PORTB = PORTK = PORTF = PORTH = PORTD =
39  PORTG = PORTJ = 0x00;
40
41  //clear the buffers
42   for (int i = 0; i < (N_PATTERNS * N_DIVS * N_PORTS); ++i) {
43     bufferA[i] = bufferB[i] = 0;
44   }
45
46  //initial pattern
47   for (int i = 0; i < (N_PORTS*N_DIVS/2); ++i){
48      bufferA[i] = 0xFF;
49   }
50
51  //a patterns of 01s at pin 22, for debuging and adjusting times
52  for(int i = 0; i < N_DIVS; ++i){
53   if (i % 2 == 0){
```

```
    bufferA[i * N_PORTS] |= 0b00001000; // HIGH
    bufferA[(N_PORTS * N_DIVS / 2) + i * N_PORTS] |= 0b00001000; // HIGH in lower half
   } else{
    bufferA[i * N_PORTS] &= 0b11110111; // LOW
    bufferA[(N_PORTS * N_DIVS / 2) + i * N_PORTS] &= 0b11110111; // LOW in lower half
   }
}

// generate a sync signal of 40khz in pin 2
 pinMode (2, OUTPUT);
 noInterrupts();           // disable all interrupts
 TCCR3A = bit (WGM10) | bit (WGM11) | bit (COM1B1); // fast PWM, clear OC1B on compare
 TCCR3B = bit (WGM12) | bit (WGM13) | bit (CS10);   // fast PWM, no prescaler
 OCR3A =  (F_CPU / 40000L) - 5; //should only be -1 but fine tunning with the scope determined that -5 gave 40kHz almost exactly
 OCR3B = (F_CPU / 40000L) / 2;
 interrupts();             // enable all interrupts

 //sync in signal at pin 3
 pinMode(3, INPUT_PULLUP); //please connect pin3 to pin 2

// disable everything that we do not need
 ADCSRA = 0;  // ADC
 power_adc_disable ();
 power_spi_disable();
 power_twi_disable();
 power_timer0_disable();
 power_usart1_disable();
 power_usart2_disable();
 power_usart3_disable();
 //power_usart0_disable();

Serial.begin(115200);

byte bReceived = 0; //octet reçu via Serial
 bool byteReady = false; //drapeau : un octet est disponible ?
 bool isSwitch = false; //drapeau : commande SWITCH reçue ?
 bool isPatternForMe = false; //drapeau : cet octet fait partie du pattern à charger ?
 bool isDuration = false; //drapeau : cet octet contient un fragment de durée ?
 bool isCommitDurations = false; //drapeau : commande de validation des durées reçue ?
 byte nextMsg = 0; //valeur à renvoyer par défaut si commande inconnue
 int writtingIndex = 0; //index pour l'écriture progressive d'un pattern

bool emittingA = true; //indique quel buffer (A ou B) est actuellement émis
//Pointeurs vers le début (H) et le milieu (L) du buffer A
 byte* emittingPointerH = & bufferA[0];
 byte* emittingPointerL = & bufferA[N_PORTS * N_DIVS / 2];
//Sauvegarde des pointeurs "à zéro" pour pouvoir réinitialiser après un switch
 byte* emittingPointerZeroH = & bufferA[0];
```

```
 byte* emittingPointerZeroL = & bufferA[N_PORTS * N_DIVS / 2];
//Pointeurs pour écrire dans le buffer de réception (l'autre buffer)
 byte* readingPointerH = & bufferB[0];
 byte* readingPointerL = & bufferB[N_PORTS * N_DIVS / 2];

byte durations[N_PATTERNS];
 byte durationsBuffer[N_PATTERNS];
// On met tout à zéro
 for(int i = 0; i < N_PATTERNS; ++i){
   durations[i] = durationsBuffer[i] = 0;
 }
// On force la durée du pattern 0 à 1 division temporelle par défaut
 durations[0] = durationsBuffer[0] = 1;

byte currentPattern = 0; //motif courant dans la séquence
byte currentPeriods = 0;//compte les divisions émises dans ce motif
byte durationsPointer = 0;//index bit-à-bit lors de la réception des
durées
byte currentDuration = 0;//durée lue pour le motif en cours
bool patternComplete = false;//drapeau : motif complètement émis ?
bool lastPattern = false;//drapeau : on est sur le dernier motif (31)
?
byte nextPattern = 0; //motif suivant (courant+1)
byte nextDuration = 0;//durée du motif suivant
bool returnToFirstPattern = false; //drapeau : la séquence doit
revenir au motif 0 ?

LOOP:
 // Attendre front descendant
 while (!(PINE & 0b00100000));     // puis attendre que ça redescende
(0)

OUTPUT_WAVE(emittingPointerH, 0); //émission des 8 bits de chaque
port à la division 0
 byteReady = Serial._dataAvailable(); //on vérifie si un octet est
déjà arrivé dans le buffer matériel du Serial
 OUTPUT_WAVE(emittingPointerH, 1); //émission de la division 1
 bReceived = Serial._peekData(); //on lit l'octet en tête si présent,
mais on ne le retire pas du buffer.
 OUTPUT_WAVE(emittingPointerH, 2); //émission de la division 2
 isSwitch = bReceived == COMMAND_SWITCH; //si l'octet cumulé vaut
exactement 0x00, on prépare le flag "switch buffer".
 isCommitDurations = bReceived == COMMAND_COMMITDURATIONS; //même
principe pour la validation des durées
 OUTPUT_WAVE(emittingPointerH, 3); //émission de la division 3
 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
émises pour le motif courant.
 OUTPUT_WAVE(emittingPointerH, 4); //émission de la division 4
 nextMsg =  bReceived - 1; //pré-calcule la réponse générique (ack ou
message d'erreur minimaliste.
 OUTPUT_WAVE(emittingPointerL, 0); //émission de la division 0
 isDuration = (bReceived & MASK_DURATION) == COMMAND_DURATION; //on
masque les 6 LSB et compare à 0x30 : détecte un byte "durée".
 nextPattern = currentPattern + 1; //on anticipe l'index du motif
suivant
 OUTPUT_WAVE(emittingPointerL, 1); //émission de la division 1
 nextDuration = durations[nextPattern]; //lecture de la durée stockée
pour le motif suivant
```

```
134   OUTPUT_WAVE(emittingPointerL, 2); //émission de la division 2
134   patternComplete = (currentPeriods ==
134  durations[currentPattern]);//vrai si on a déjà émis autant de
134  divisions que la durée allouée au motif en cours
135   OUTPUT_WAVE(emittingPointerL, 3); //émission de la division 3
135   lastPattern = (currentPattern+1 == N_PATTERNS); //vrai si on était
135  sur le dernier motif (31)
135  returnToFirstPattern = nextDuration == 0; //vrai si le motif suivant
135  est configuré pour durer 0 divisions.
136   OUTPUT_WAVE(emittingPointerL, 4); //émission de la division 4
136
137  if (patternComplete){
138   currentPeriods = 0;
138   ++currentPattern;
139   if (lastPattern  || returnToFirstPattern){
140     currentPattern = 0;
140     emittingPointerH = emittingPointerZeroH;
141     emittingPointerL = emittingPointerZeroL;
141   }else{
142     emittingPointerH += (N_DIVS * N_PORTS);
143     emittingPointerL += (N_DIVS * N_PORTS);
143   }
144  }
145
146  if (byteReady) {
146   if (isSwitch) {
147     // --- SWITCH BUFFER ---
148     Serial.write(COMMAND_SWITCH);
148     emittingA = !emittingA;
149     // On swap A↔B pour émission vs. réception
149     if (emittingA) {
150       emittingPointerH = &bufferA[0];
151       emittingPointerL = &bufferA[N_PORTS*N_DIVS/2];
152       readingPointerH  = &bufferB[0];
153       readingPointerL  = &bufferB[N_PORTS*N_DIVS/2];
154     } else {
155       emittingPointerH = &bufferB[0];
156       emittingPointerL = &bufferB[N_PORTS*N_DIVS/2];
157       readingPointerH  = &bufferA[0];
158       readingPointerL  = &bufferA[N_PORTS*N_DIVS/2];
159     }
160     emittingPointerZeroH = emittingPointerH;
161     emittingPointerZeroL = emittingPointerL;
162
163     writtingIndex = 0;
164     durationsPointer = 0;
165   } else if (isPatternForMe) {
166     // --- RÉCEPTION D'UN OCTET DE PATTERN ---
167     // On compose 4 bits hauts et 4 bits bas dans readingPointerH[]
168     if (writtingIndex % 2 == 0)
169       readingPointerH[writtingIndex/2] = bReceived & 0xF0;
170    } else {
171       readingPointerH[writtingIndex/2] |= (bReceived >> 4);
172    }
173     ++writtingIndex;
174   } else if (isDuration) {
175     // --- RÉCEPTION D'UN OCTET DE DURÉE —
176     Serial.write( bReceived );
177     if (durationsPointer % 4 == 0) {
178       durationsBuffer[durationsPointer / 4] = bReceived & 0b11000000;
```

```
 } else {
     durationsBuffer[durationsPointer / 4] |= (bReceived &
0b11000000) >> (durationsPointer % 4 * 2);
 }
   ++durationsPointer;
} else if (isCommitDurations) {
   // --- COMMIT DES DURÉES —
   Serial.write(bReceived);
   for (int i = 0; i < N_PATTERNS; ++i){
     durations[i] = durationsBuffer[i];
 }
   durationsPointer = 0;
} else {
   // --- COMMANDE INCONNUE —
   Serial.write(nextMsg);
 }
 // On consomme enfin l'octet
 Serial._discardByte();
}

void loop() {}
```

1. Commentaires d'en-tête et instructions de compilation

Les premiers commentaires (lignes 1–11) indiquent :

- Qu'il faut ajouter trois fonctions inline dans `HardwareSerial.h` pour accéder aux octets reçus (`_dataAvailable()`, `_peekData()`, `_discardByte()`).
- Que, pour la version Arduino IDE > 1.6, il faut compiler avec l'option `-O3` (optimisation vitesse).
- De commenter la ligne `power_usart3_disable();` si elle génère une erreur.

Ces ajustements sont purement pour contourner des limitations de la bibliothèque `HardwareSerial` et d'optimiser le code.

2. Inclusion de bibliothèques et définitions de constantes

```
13  #include <avr/sleep.h>
14  #include <avr/power.h>
15
16  #define N_PATTERNS 32
17
18  #define N_PORTS 10
19  #define N_DIVS 10
20
21  //ports A C L B K F H D G J
22
23  #define COMMAND_SWITCH 0b00000000
24  #define COMMAND_DURATION 0b00110000
25  #define MASK_DURATION 0b00111111
26  #define COMMAND_COMMITDURATIONS 0b00010000
27
28  #define WAIT(a) __asm__ __volatile__ ("nop")
29  #define OUTPUT_WAVE(pointer, d)  PORTA = pointer[d*N_PORTS + 0]; PORTC
29  = pointer[d*N_PORTS + 1]; PORTL = pointer[d*N_PORTS + 2]; PORTB =
29  pointer[d*N_PORTS + 3]; PORTK = pointer[d*N_PORTS + 4]; PORTF =
29  pointer[d*N_PORTS + 5]; PORTH = pointer[d*N_PORTS + 6];  PORTD =
29  pointer[d*N_PORTS + 7]; PORTG = pointer[d*N_PORTS + 8]; PORTJ =
29  pointer[d*N_PORTS + 9]
```

- `COMMAND_SWITCH` : Indique "change de buffer" : dès qu'un octet reçu vaut exactement 0, on bascule l'émission de `bufferA` vers `bufferB` (ou inverse). `Switch` permet donc de passer de l'écriture dans un buffer à l'émission de l'autre, garantissant une mise à jour sans artefacts.

- `COMMAND_DURATION` : Motif de bits en position haute (`b5` et `b4` à 1) signalant le transfert de données de durée (2 bits × 4). `Duration` découpe donc chaque durée sur 2 bits, stockés à raison de 4 octets par pattern (2 bits × 4 = 8 bits), grâce au masque.

- `MASK_DURATION` : Masque pour extraire les 6 bits de poids faible du byte reçu : on l'applique par `bReceived & MASK_DURATION` pour isoler la partie "commande + payload" sans les bits supérieurs réservés.

- `COMMAND_COMMITDURATIONS` : Valeur précise (`b4` à 1, `b5` à 0) qui, une fois reconnue, commande la validation de toutes les durées accumulées dans `durationsBuffer[]` vers `durations[]`. `CommitDurations` confirme donc à l'Arduino qu'il peut désormais utiliser les nouvelles durées pour cadencer la lecture des patterns.

Utilisation dans le programme :

1. *Détection d'une commande "switch"*

```
131  isSwitch = (bReceived == COMMAND_SWITCH);
```

Si l'octet reçu vaut `0b00000000`, on sait que l'hôte veut que l'Arduino bascule le buffer actif d'émission.

2. *Détection d'un octet de "durée"*

```
134  isDuration = ((bReceived & MASK_DURATION) == COMMAND_DURATION);
```

- On applique `bReceived & 0b00111111` pour garder les 6 bits de poids faible.
- Si le résultat vaut `0b00110000`, l'octet transporte un fragment de durée à stocker dans `durationsBuffer`.

3. *Détection de la commande de "commit" des durées*

```
131  isCommitDurations = (bReceived == COMMAND_COMMITDURATIONS);
```

Quand l'octet reçu vaut `0b00010000`, on sait que l'envoi de durées est terminé ; on copie alors `durationsBuffer` dans `durations`.

- `N_PATTERNS` (32) : nombre de motifs/patterns stockés.
- `N_PORTS` (10) et `N_DIVS` (10) déterminent la taille des buffers: chaque pattern comporte 10 ports et 10 divisions temporelles.

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

- `nop` (No OPeration) est une instruction assembleur qui ne fait rien, elle coûte un cycle d'horloge.
- **Rôle** : insérer un tout petit délai (exactement 1 cycle CPU) à l'endroit où la macro est appelée.
- **Paramètre** `a` : n'est pas utilisé dans la macro ; c'est souvent un artifice pour conserver la même syntaxe (`WAIT(5)` par exemple) si on veut lier la macro à un argument, ou pour documenter l'intention.
- **Usage typique** : ajuster le placement des instructions en mémoire ou entre deux accès à un port pour garantir un échantillonnage stable ou éviter des métastabilités.

```
29  #define OUTPUT_WAVE(pointer, d)  PORTA = pointer[d*N_PORTS + 0]; PORTC
29  = pointer[d*N_PORTS + 1]; PORTL = pointer[d*N_PORTS + 2]; PORTB =
29  pointer[d*N_PORTS + 3]; PORTK = pointer[d*N_PORTS + 4]; PORTF =
29  pointer[d*N_PORTS + 5]; PORTH = pointer[d*N_PORTS + 6];  PORTD =
29  pointer[d*N_PORTS + 7]; PORTG = pointer[d*N_PORTS + 8]; PORTJ =
29  pointer[d*N_PORTS + 9]
```

Cette macro écrit en un seul appel sur les **10 registres de port** (A, C, L, B, K, F, H, D, G, J) la valeur contenue dans un buffer.

- `pointer` : pointeur sur un tableau `byte` qui contient, pour chaque "division temporelle" et chaque port, la valeur à appliquer (0 ou 1 dans chaque bit du port).
- `d` : indice de la "division temporelle" courante (allant de 0 à `N_DIVS-1`).
- `d*N_PORTS + i` : calcule l'offset dans le buffer pour le i-ème port (i=0...9).
- Chaque instruction `PORTx = …;` écrit **8 bits** d'un coup sur le port .

3. Déclaration des buffers globaux

Deux buffers (A et B) permettent de stocker en double buffer des trames de sortie : pendant qu'on émet l'un, on remplit l'autre via la liaison série.

```
31  static byte bufferA[N_PATTERNS * N_DIVS * N_PORTS];
32  static byte bufferB[N_PATTERNS * N_DIVS * N_PORTS];
```

4. Configuration de la fonction `setup()`

4.1. Configuration des broches

On configure d'abord les registres de direction :

```
34  void setup()
35  {
36   //set as output ports A C L B K F H D G J
37   DDRA = DDRC = DDRL = DDRB = DDRK = DDRF = DDRH = DDRD = DDRG = DDRJ =
37  0xFF;
```

- Sur le microcontrôleur ATmega2560 de l'Arduino Mega, chaque port (A, C, L, B, K, F, H, D, G, J) possède un registre `DDR` (Data Direction Register).

- Mettre un bit à `1` dans `DDRX` configure la broche correspondante du port en **sortie**.

- `0xFF` en hexadécimal, soit binaire `11111111`, fixe tous les 8 bits à 1 : **toutes** les broches des ports A, C, L, B, K, F, H, D, G et J deviennent des sorties.

On met ensuite à 0 tous les ports :

```
38  //low signal on all of them
39   PORTA = PORTC = PORTL = PORTB = PORTK = PORTF = PORTH = PORTD =
39  PORTG = PORTJ = 0x00;
```

Le registre `PORTX` contrôle le niveau logique appliqué aux broches configurées en sortie. En écrivant `0x00` (binaire `00000000`), on place **tout le port à LOW**, c'est-à-dire 0 V sur chaque broche. Ceci garantit qu'au démarrage, aucun signal indésirable n'est émis.

Enfin, on initialise les buffers de motif :

```
41   //clear the buffers
42   for (int i = 0; i < (N_PATTERNS * N_DIVS * N_PORTS); ++i) {
43     bufferA[i] = bufferB[i] = 0;
44   }
```

- Les deux tableaux `bufferA` et `bufferB` contiennent les valeurs à envoyer, à chaque pas de temps, sur les 10 ports.

- Leur taille totale est `N_PATTERNS * N_DIVS * N_PORTS` octets.
    - `N_PATTERNS` : nombre de motifs différents stockables (32).
    - `N_DIVS` : nombre de "divisions temporelles" par motif (10).
    - `N_PORTS` : nombre de ports émis en parallèle (10).
- Cette boucle for met **tous** les éléments de ces deux buffers à zéro, assurant un état connu (aucun bit à 1) avant de charger un motif ou une séquence.

4.2. Initialisation et "pattern" de base

On commence par remplir la moitié haute du bufferA à 0xFF :

```
46  //initial pattern
47   for (int i = 0; i < (N_PORTS*N_DIVS/2); ++i){
48      bufferA[i] = 0xFF;
49   }
```

- `N_PORTS * N_DIVS` est le nombre d'octets par pattern.
- Divisé par 2, cela correspond à la première moitié du pattern (on émettra deux "moitiés" successives de `N_DIVS` divisions).
- En mettant chaque `bufferA[i] = 0xFF`, on fait en sorte que **tous les bits** (toutes les broches des 10 ports) soient à **1** (HIGH) pour cette première moitié temporelle.
- La deuxième moitié du buffer (indices ≥ `N_PORTS * N_DIVS / 2`) reste à 0, donc toutes sorties à LOW.

Cette organisation (division en 2 du buffer) permet de simplifier l'organisation de la mémoire et celle du double buffering ainsi que de gagner en souplesse sur l'émission des patterns.

4.3. Debug

On y superpose un signal de 0 et de 1 sur la broche 22 (pour debug):

```
52  //a patterns of 01s at pin 22, for debuging and adjusting times
53  for(int i = 0; i < N_DIVS; ++i){
54   if (i % 2 == 0){
55     bufferA[i * N_PORTS] |= 0b00001000; // HIGH
56     bufferA[(N_PORTS * N_DIVS / 2) + i * N_PORTS] |= 0b00001000; //
56  HIGH in lower half
57   } else{
58     bufferA[i * N_PORTS] &= 0b11110111; // LOW
59     bufferA[(N_PORTS * N_DIVS / 2) + i * N_PORTS] &= 0b11110111; // LOW
60  in lower half
61   }
62  }
```

- **Index de l'octet** correspondant à la broche 22 (PA3) pour la division temporelle `i` :
  - Première moitié → `i * N_PORTS + 0` (le port A est à l'emplacement +0)
  - Seconde moitié → `(N_PORTS * N_DIVS / 2) + i * N_PORTS + 0`
- **Masque** `0b00001000` : active ou désactive le bit 3 de l'octet, c'est-à-dire PA3.
- Pour `i` **pair** on fait un **OR** (`|= 0b00001000`) → PA3 = HIGH.
- Pour `i` **impair** on fait un **AND** avec l'inverse (`&= 0b11110111`) → PA3 = LOW.

Ainsi, au cours des 10 divisions temporelles, PA3 passera alternativement à 1 puis 0 (motif "0101..."), et cela sur **les deux moitiés** de **bufferA**.

En gros cela allume la pin22 alternativement (c'est 5 mini créneaux) et cela permet de vérifier que tout fonctionne bien.

4.3. Génération d'un signal de synchronisation à 40 kHz (broche 2)

Cette section configure le **Timer 3** de l'ATmega pour générer, **matériellement**, un signal carré synchronisé à **40 kHz** sur la broche associée à **OC3B** (reliée à D2).

```
66  // generate a sync signal of 40khz in pin 2
67   pinMode (2, OUTPUT);
68   noInterrupts();           // disable all interrupts
69   TCCR3A = bit (WGM10) | bit (WGM11) | bit (COM1B1); // fast PWM, clear
69  OC1B on compare
70   TCCR3B = bit (WGM12) | bit (WGM13) | bit (CS10);   // fast PWM, no
70  prescaler
71   OCR3A =  (F_CPU / 40000L) - 5; //should only be -1 but fine tunning
71  with the scope determined that -5 gave 40kHz almost exactly
72   OCR3B = (F_CPU / 40000L) / 2;
73   interrupts();             // enable all interrupts
```

Elle :

- Met la broche 2 en **sortie** (c'est là que sortira le signal PWM).

PWM = Pulse Width Modulation → modulation de largeur d'impulsion :

1. *C'est un signal numérique (0 ou 1, HIGH ou LOW) dont la fréquence est fixe, mais la durée pendant laquelle il reste à 1 varie.*
2. *En d'autres termes, on "allume" et "éteint" rapidement une broche pour créer un signal dont la moyenne sur le temps peut représenter une valeur analogique.*

- Désactive temporairement toutes les interruptions pour éviter qu'un appel en ISR ne perturbe la configuration fine des registres du timer.
- **Mode Fast PWM, TOP = OCR3A**
  - Les quatre bits WGM13:WGM10 à `1` sélectionnent le mode 15 (Fast PWM, TOP variable).
  - En mode non-inverting (`COM3B1=1`), la sortie OC3B passe à HIGH au début de la période (compteur = 0) et se remet à LOW quand le compteur atteint OCR3B.
- **CS10 = 1** → on utilise directement la fréquence CPU (`F_CPU`) sans division.
- **CR3A** fixe la valeur de comparaison qui détermine la **période** du PWM :

$$f = \frac{F_{CPU}}{\text{OCR3A} + 1} \approx 40kHz$$

  On soustrait "–5" au lieu de "–1" pour un ajustement au scope, afin d'obtenir précisément 40 kHz à cause de tolérances matérielles.
- **OCR3B** à la moitié de OCR3A → génère un **rapport cyclique 50 %**.
- Restaure les interruptions maintenant que le timer est configuré.

A quoi sert une horloge ?

4 Analogie

Imagine un orchestre :

- Chaque musicien joue sa note quand il voit le **chef d'orchestre lever la main**.
- Ici :
  - Le **signal PWM 40 kHz** = chef d'orchestre
  - Les transducteurs = musiciens
  - Chaque impulsion du signal indique quand **mettre à jour le pattern sur les sorties**.

Sans ce signal, chacun jouerait à sa propre vitesse → le motif acoustique serait détruit.

5 Résumé

- Le signal de 40 kHz est **l'horloge de référence** du système.
- Il garantit que **tous les transducteurs changent d'état exactement au même moment**.
- Sans lui, la lévitation acoustique **ne serait pas stable**.

4.4. Entrée de synchronisation (broche 3)

```
75    //sync in signal at pin 3
76    pinMode(3, INPUT_PULLUP); //please connect pin3 to pin 2
```

On attend un front descendant sur pin 3 connecté à pin 2 pour cadencer la boucle d'émission.

4.5. Désactivation des périphériques non utilisés pour économiser l'énergie

```
79    // disable everything that we do not need
80    ADCSRA = 0;  // ADC
81    power_adc_disable ();
82    power_spi_disable();
83    power_twi_disable();
84    power_timer0_disable();
85    power_usart1_disable();
86    power_usart2_disable();
87    power_usart3_disable();
88    //power_usart0_disable();
```

On garde tout de même le port `USART0` activé afin de pouvoir utiliser la communication série avec l'application Java.

3.6. Initialisation du port série et variables de protocole

On initialise tout d'abord le port série en configurant l'USART0 à 115 200 bauds pour l'échange de commandes et de données avec l'hôte (PC ou microcontrôleur maître). :

```
90    Serial.begin(115200);
```

Ensuite, on déclare les variables de réception :

```
92    byte bReceived = 0; //octet reçu via Serial
93    bool byteReady = false; //drapeau : un octet est disponible ?
94    bool isSwitch = false; //drapeau : commande SWITCH reçue ?
95    bool isPatternForMe = false; //drapeau : cet octet fait partie du
95  pattern à charger ?
96    bool isDuration = false; //drapeau : cet octet contient un fragment
96  de durée ?
```

```
97   bool isCommitDurations = false; //drapeau : commande de validation
97  des durées reçue ?
98   byte nextMsg = 0; //valeur à renvoyer par défaut si commande inconnue
98   int writtingIndex = 0; //index pour l'écriture progressive d'un
99  pattern
99
```

Ces drapeaux sont recalculés à chaque itération de la boucle principale pour parser l'octet reçu (bReceived).

Puis, il faut gérer la double bufférisation :

```
101   bool emittingA = true; //indique quel buffer (A ou B) est
101  actuellement émis
     //Pointeurs vers le début (H) et le milieu (L) du buffer A
102   byte* emittingPointerH = & bufferA[0];
103   byte* emittingPointerL = & bufferA[N_PORTS * N_DIVS / 2];
     //Sauvegarde des pointeurs "à zéro" pour pouvoir réinitialiser après
     un switch
104   byte* emittingPointerZeroH = & bufferA[0];
105   byte* emittingPointerZeroL = & bufferA[N_PORTS * N_DIVS / 2];
     //Pointeurs pour écrire dans le buffer de réception (l'autre buffer)
      byte* readingPointerH = & bufferB[0];
106   byte* readingPointerL = & bufferB[N_PORTS * N_DIVS / 2];
107
```

- `emittingPointerH/L` pointent sur la moitié active du buffer à émettre (la première moitié des divisions temporelles puis la seconde).
- `readingPointerH/L` pointent sur la moitié correspondante de l'autre buffer, où les nouveaux motifs seront chargés.
- Quand on reçoit la commande SWITCH, on inverse `emittingA` et on échange simplement ces pointeurs entre A et B.

Vient alors l'initialisation des durées de pattern :

```
109   byte durations[N_PATTERNS];
110   byte durationsBuffer[N_PATTERNS];
     // On met tout à zéro
111   for(int i = 0; i < N_PATTERNS; ++i){
112     durations[i] = durationsBuffer[i] = 0;
113   }
     // On force la durée du pattern 0 à 1 division temporelle par défaut
      durations[0] = durationsBuffer[0] = 1;
114
```

- `durations[]` stocke pour chaque motif (0 à 31) le nombre de divisions temporelles pendant lesquelles il reste actif.
- `durationsBuffer[]` sert à accumuler les nouvelles durées reçues avant le commit.
- On met par défaut le pattern 0 à durer **1** division, pour éviter une division par zéro lors du premier cycle.

Finalement, on met à jour les variables qui contrôlent le cycle d'émission :

```
116  byte currentPattern = 0; //motif courant dans la séquence
117  byte currentPeriods = 0;//compte les divisions émises dans ce motif
118  byte durationsPointer = 0;//index bit-à-bit lors de la réception des
     durées
119  byte currentDuration = 0;//durée lue pour le motif en cours
120  bool patternComplete = false;//drapeau : motif complètement émis ?
121  bool lastPattern = false;//drapeau : on est sur le dernier motif (31)
     ?
122  byte nextPattern = 0; //motif suivant (courant+1)
123  byte nextDuration = 0;//durée du motif suivant
124  bool returnToFirstPattern = false; //drapeau : la séquence doit
     revenir au motif 0 ?
```

- `currentPattern` et `currentPeriods` gèrent la progression au sein des divisions temporelles du pattern actif.
- À chaque front de synchronisation, on incrémente `currentPeriods` ; quand il atteint `durations[currentPattern]`, `patternComplete` passe à true et on avance `currentPattern`.
- Si on est arrivé au pattern final (31) ou si la durée du suivant est nulle, on revient au motif 0.

3.5. Boucle principale avec `goto LOOP`

Contrairement au style habituel `void loop()`, ici on utilise un label `LOOP:` et un `goto` pour réduire le code généré par l'IDE.

3.5.1. Synchronisation sur front descendant (cycle PWM 40 kHz)

```
125  LOOP:
126   // Attendre front descendant
127   while (!(PINE & 0b00100000));     // puis attendre que ça redescende
     (0)
```

Comme cette broche (PE5) est raccordée au signal PWM de pin 2 via pin 3, on se positionne sur le front montant, puis le code suivant attendra la descente effective du signal avant la prochaine itération.

- **PE5** correspond à la broche **3** ; ce signal est le reflet du PWM 40 kHz généré sur la broche 2.
- On synchronise ainsi chaque itération de la boucle sur une période complète du signal de référence, garantissant un timing constant.

3.5.2. Emission du motif et évaluation du byte série

Chaque itération, on exécute successivement cinq appels à la macro `OUTPUT_WAVE(...)`, envoyant les octets préparés dans `emittingPointerH[]` pour les divisions temporelles 0→4, puis cinq pour `emittingPointerL[]`, divisions 0→4 de la seconde moitié du pattern :

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

Pour chaque division temporelle `d` :

1. `OUTPUT_WAVE` écrit en parallèle sur les 10 ports la valeur `pointer[d*N_PORTS + i]`.
2. Au **deuxième** appel (d=1), on lit si un octet est prêt :
   - `byteReady = Serial._dataAvailable();`
   - `bReceived = Serial._peekData();`

     On ne consomme pas encore l'octet, juste on le lit pour décider de la suite.
3. Entre les écritures, on calcule divers drapeaux et compteurs en fonction de `bReceived` :
   - `isSwitch`, `isCommitDurations`
   - `isPatternForMe` (on regarde les 4 LSB (= bit de point faible) pour voir si ce pattern est destiné à cet esclave)
   - `nextMsg = bReceived - 1` : valeur par défaut à renvoyer si commande inconnue
   - `isDuration` pour détecter l'octet de durée
   - `nextPattern`, `nextDuration` : préparation pour tester si on doit repasser au motif 0
   - `patternComplete` dès que le nombre de divisions émises atteint `durations[currentPattern]`
   - `lastPattern` si on est sur le dernier motif (31)
   - `returnToFirstPattern` si la durée du suivant vaut 0

Par cette alternance d'`OUTPUT_WAVE` et de calculs, on s'assure que l'émission reste **cycle-exacte** à 40 kHz, tout en "streamant" le protocole série en arrière-plan.

3.5.3 Avancement du motif

Après les 10 divisions temporelles, on regarde si `patternComplete` est vrai :

```
140  if (patternComplete){
141   currentPeriods = 0;
142   ++currentPattern;
143   if (lastPattern  || returnToFirstPattern){
144     currentPattern = 0;
145     emittingPointerH = emittingPointerZeroH;
146     emittingPointerL = emittingPointerZeroL;
147   }else{
148     emittingPointerH += (N_DIVS * N_PORTS);
149     emittingPointerL += (N_DIVS * N_PORTS);
150   }
151  }
```

- `currentPeriods` remis à zéro pour compter de nouvelles divisions.
- `currentPattern` incrémenté, ou remis à 0 si fin de séquence ou durée nulle.
- `emittingPointerH/L` avancent de `N_DIVS*N_PORTS` octets pour pointer vers la prochaine moitié de buffer à émettre.

3.5.4 Gestion du protocole série

```
153  if (byteReady) {
154   if (isSwitch) {
        // --- SWITCH BUFFER ---
155     Serial.write(COMMAND_SWITCH);
156     emittingA = !emittingA;
        // On swap A↔B pour émission vs. réception
157     if (emittingA) {
158       emittingPointerH = &bufferA[0];
159       emittingPointerL = &bufferA[N_PORTS*N_DIVS/2];
160       readingPointerH  = &bufferB[0];
161       readingPointerL  = &bufferB[N_PORTS*N_DIVS/2];
162     } else {
163       emittingPointerH = &bufferB[0];
164       emittingPointerL = &bufferB[N_PORTS*N_DIVS/2];
165       readingPointerH  = &bufferA[0];
166       readingPointerL  = &bufferA[N_PORTS*N_DIVS/2];
167     }
168     emittingPointerZeroH = emittingPointerH;
169     emittingPointerZeroL = emittingPointerL;
170
171     writtingIndex = 0;
172     durationsPointer = 0;
173   } else if (isPatternForMe) {
        // --- RÉCEPTION D'UN OCTET DE PATTERN ---
        // On compose 4 bits hauts et 4 bits bas dans readingPointerH[]
174     if (writtingIndex % 2 == 0)
175       readingPointerH[writtingIndex/2] = bReceived & 0xF0;
176    } else {
177       readingPointerH[writtingIndex/2] |= (bReceived >> 4);
178    }
179     ++writtingIndex;
180   } else if (isDuration) {
        // --- RÉCEPTION D'UN OCTET DE DURÉE —
181     Serial.write( bReceived );
182     if (durationsPointer % 4 == 0) {
183       durationsBuffer[durationsPointer / 4] = bReceived & 0b11000000;
183   } else {
184       durationsBuffer[durationsPointer / 4] |= (bReceived &
185  0b11000000) >> (durationsPointer % 4 * 2);
185   }
186     ++durationsPointer;
187  } else if (isCommitDurations) {
188     // --- COMMIT DES DURÉES —
        Serial.write(bReceived);
189     for (int i = 0; i < N_PATTERNS; ++i){
190       durations[i] = durationsBuffer[i];
191   }
192     durationsPointer = 0;
193  } else {
194     // --- COMMANDE INCONNUE —
        Serial.write(nextMsg);
195   }
196   // On consomme enfin l'octet
      Serial._discardByte();
197  }
198
```

- `Switch` : confirme au maître, échange les rôles des buffers A/B.
- `PatternForMe` : assemble un motif 10 bits (2×4 bits) dans `readingPointerH`.

- `Duration` **:** stocke progressivement 2 bits×4 octets par motif dans `durationsBuffer`**.**
- `CommitDurations` **:** valide toutes les nouvelles durées en copiant le tampon tampon.
- `Inconnu` **:** renvoie `nextMsg = bReceived-1` pour signaler une erreur ou un ack générique.

5. Fonction `loop()` vide

```
1    }
2
3   void loop() {}
```

La boucle loop() reste vide puisqu'on utilise goto LOOP dès setup() pour la boucle principale.

Conclusion :
Le programme gère une double-bufferisation de motifs (bufferA, bufferB) envoyés simultanément sur 10 ports à une fréquence de division temporelle fixée (40 kHz). Les motifs et leurs durées sont reçus par le port série selon un protocole léger :

- *Switch* pour permuter buffer émis/réception,
- *PatternForMe* pour charger les octets de pattern,
- *Duration* pour charger les durées de chaque pattern,
- *CommitDurations* pour valider les durées reçues.

Le tout est synchronisé par un signal PWM 40 kHz (pin 2→pin 3) et optimisé pour la basse consommation en désactivant les périphériques inutiles.