BORDEAUX INP Enseirb-Matmeca

Guide d'utilisateur : Codesign sur carte PYNQ du pilotage d'une barre led tournante

MALLET DE CHAUNY DRUESNE Eléna

DAUNOIS Hugo

TREILLES Alexis

PROUTEAU Guillaume

Projet avancé 3A

Janvier 2025

Contents

1. Introduction

1.1. Contexte

Ce projet a pour but de montrer les compétences qui sont développées dans la filière Électronique de l'ENSEIRB-MATMECA dans le but de le présenter pendant des événements de communication et publicité de l'école et de la filière. Le résultat contient la réalisation d'un écran composé d'une barre de leds tournante pilotée par une carte PYNQ-Z2. Le principe repose sur la persistance rétienienne pour afficher des images de façon fluide. Il met en oeuvre le codesign par des ressources software et hardware tels que Jupyter et Vivado.

L'idée provient d'un projet réalisé par des étudiants en l'école Telecom ParisTech en 2012. Ce projet comporte un système d'affichage utilisant la persistance de la vision. La lame porte des LEDs et tourne à 1000 tour par minute (Chaque LED change de couleur toutes les 60μs). Le système est contrôlé par WiFi et peut lire n'importe quel fichier vidéo standard. `https://www.youtube.com/watch?v=Lr_t0wTCCFo`

2. Matériel utilisé

2.1. Leds

Le bandeau leds utilisé pour le projet est le modèle SK9822 de la marque BTF Lightning. Il mesure un mètre et comprend 144 leds, chacune adressable et programmable. Il est alimenté en 5V.

La communication avec les leds se fait par liaison SPI, un bus de données séries synchrone, qui s'organise entre un système Maître et un système Esclave. Cette communication utilise les signaux suivant :

- Un signal d'horloge pour synchroniser l'affichage sur les leds du Maître vers l'Esclave
- Un signal de données contenant les données des leds du Maître vers l'Esclave
- Un signal de données de l'Esclave vers le Maître

L'horloge utilisée dans notre cas a été fixée à 25 MHz. Cela nous permet de réaliser le rafraîchissement du bandeau leds de sorte à avoir 360 rafraîchissements par tour de la barre. La fréquence maximale possible avec ce modèle est de 30 MHz, mais l'horloge de 25 MHz était plus cohérente avec le reste du projet qui fonctionne avec 360 trames par tour.

Note : Les données sur le bandeau leds ont été trouvées sur les sites suivants :

https://www.amazon.fr/BTF-LIGHTING-Non-Imperméable-Individuellement-Adressable

`https://www.pololu.com/file/0J1234/sk9822_datasheet.pdf`

2.2. Moteur

Le moteur choisi est le modèle A2212 930KV Brushless Motor lié au 2-4S 30A Brushless ESC (Electronic Speed Controler). Cet ensemble est principalement utilisé dans la construction de drones qui nécessitent une très grande vitesse de rotation, ce qui est également recherché dans ce projet.

On décrit les caractéristiques du moteur dans le tableau suivant :

Tour par min Max	930
Courant Max	10A
Courant Min	4A
Tension	5V

L'ESC sert d'intermédiaire entre le moteur nécessite et les signaux d'alimentation et de contrôle. Il nécessite une tension 12V pour fonctionner correctement et va transformer ces 12V en 5V directement pour le moteur.

L'ESC récupère également un signal de données pour le contrôle de la vitesse du moteur. Le signal utilisé est une PWM (Pulse Width Modulation), expliquée ultérieurement, qui est transformé en signal triphasé.

Cependant la vitesse de ce moteur est trop rapide pour garder une certaine sécurité. En effet la rotation d'une barre d'un mètre peut entraîner des risques non négligeables si la structure mécanique n'est pas asez solide.

Pour palier ce problème nous avons choisi de relier le moteur à une roue plus grande sur laquelle sont fixées les leds. Cela permet de réduire la vitesse de rotation du bandeau leds. Le rapport choisi est d'environ 10 c'est-à-dire que pour un tour de la grande roue, le moteur en réalise 10. La roue reliée au moteur par un système de poulie avec un élastique de couture.

Note : Les données sur le moteur et l'ESC ont été trouvées sur le site suivant :

https://www.amazon.fr/QWinOut-Brushless-Outrunner-Controller-Aircraft/dp/B08F585

3. Contrôle des LED

3.1. Introduction et spécifications

L'objectif principal du contrôle des LED est d'afficher des informations stockées en mémoire sous la forme de plusieurs bandeaux (frame) de 144 LEDs, chaque frame correspondant à un angle de rotation. À chaque passage du capteur à effet Hall, la lecture des données doit recommencer à partir du début.

Note : Le code de communication des LED se base sur les informations disponibles sur le site suivant : `https://www.pololu.com/product/3095` car la fiche technique des LED présente des erreurs.

3.2. Description fonctionnelle

La description fonctionnelle pour le contrôle des LED est présentée ci-dessous :

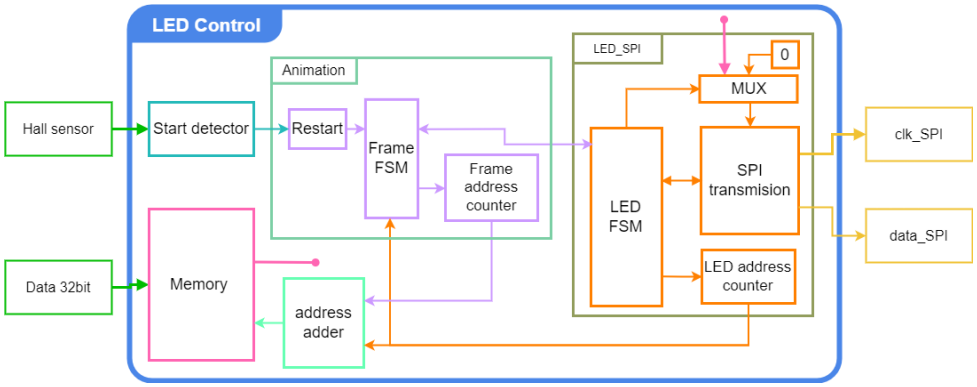


Figure 1: Description fonctionnelle du contrôle des LEDs

→ Mémoire

La mémoire joue le rôle d'intermédiaire entre les données générées par le code Python et la partie VHDL du contrôle des LED. Elle stocke des données sur 32 bits nécessaires à l'animation, suivant le schéma ci-dessous :

Memory layout		Address
FRAME 1	LED 1	0
	LED 2	1
	⋮	⋮
	LED N	N-1
FRAME 2	LED 1	N
	LED 2	N+1
	⋮	⋮
	LED N	2*(N-1)
FRAME N	⋮	⋮

Ainsi, on peut déduire la formule de l'adresse permettant d'accéder à la donnée qui nous intéresse est :

$$\text{Adresse} = N \times \text{frame} + \text{led}$$

→ **Address Adder**

Description : Ce bloc génère les adresses mémoire nécessaires pour accéder aux différentes données des LED en fonction de la frame et du numéro de la LED.

Entrée : Valeurs du *frame address counter* et du *LED address counter*.

Sortie : Adresse pour accéder au données de la *Mémoire*.

→ **Start Detector**

Nom VHDL : anti_rebond

Description : Permet la détection d'un front montant lorsque le *HALL sensor* détecte un aimant.

Entrée : Signal du *Hall Sensor*.

Sortie : Signal actif sur un coup d'horloge lorsqu'un front montant est détecté.

Interaction avec d'autres blocs : Reset de la *frame FSM*, et donc des blocs *animation* et *LED_SPI*.

→ **Restart**

Nom VHDL : latch2

Description : Retarde de deux coups d'horloge le signal présent en entrée.

Entrée : Signal Start Detector.

Sortie : Signal Start Detector retardé.

Interaction avec d'autres blocs : Il permet de générer le start de *frame FSM* après un reset.

→ **Frame FSM**

Nom VHDL : fsm_animation

Description : Machine d'état qui gère le numéro de la frame et qui active l'envoi des 144 LEDs.

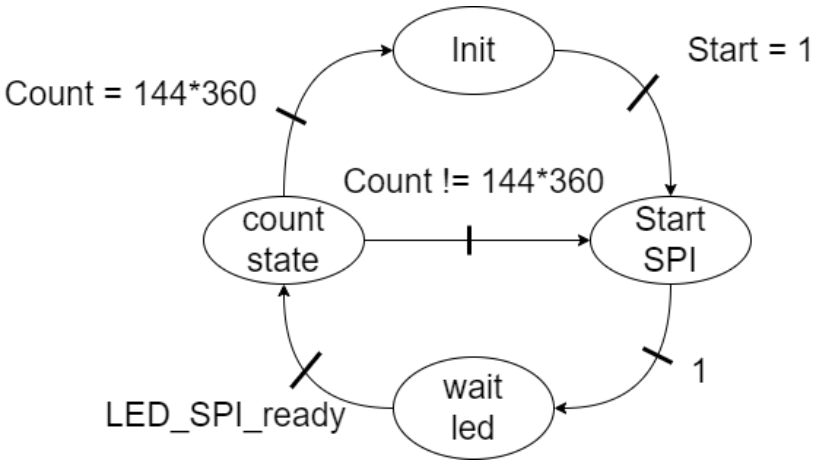
Entrée : Signal *Restart* du *Start Detector*, adresses du *Frame Address Counter* et signal de *led_fsm*.

Sortie : Commandes vers le *Frame Address Counter* et le **LED FSM**.

Interaction avec d'autres blocs :

- Reçoit le signal de redémarrage du *Start Detector*.
- Incrémente et réinitialise *Frame Address Counter*.
- Démarre la séquence d'envoi des données des LEDs pour le bandeau grâce à *LED FSM*.

Fonctionnement :



→ **Frame Address Counter**

Nom VHDL : count_addr

Description : Génère les adresses permettant de sélectionner la frame désirée. Incrémentation par 144 (le nombre de LEDs sur le bandeau).

Entrée : Signaux de contrôle du *Frame FSM*.

Sortie : Adresse pour la mémoire sur 16 bits.

Interaction avec d'autres blocs : Envoie des adresses au *Address Adder* et au *Frame FSM*, et reçoit des instructions pour incrémenter ou réinitialiser son compteur.

→ **LED FSM**

Nom VHDL : FSM_spi

Description : Machine d'état contrôlant l'affichage des LED sur un bandeau en fonction des données en mémoire.

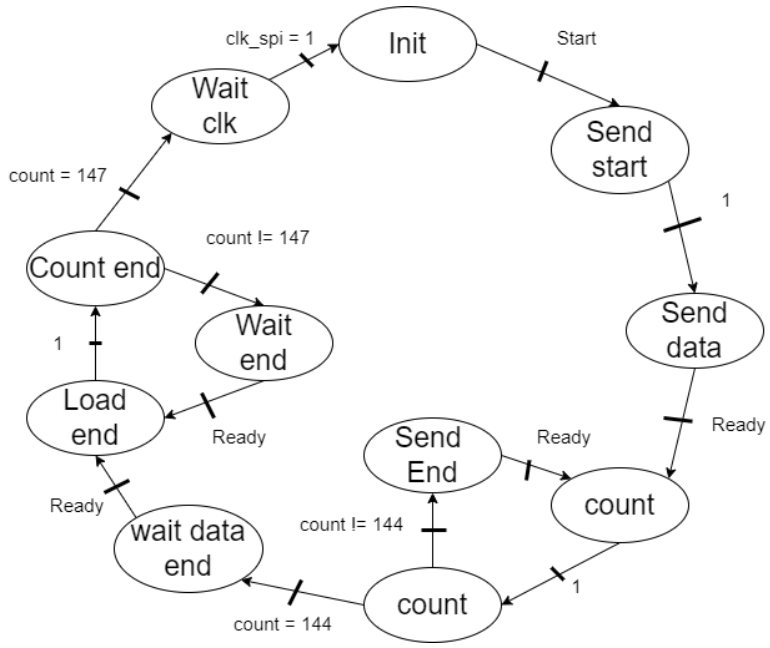
Entrée : Signal de *Frame FSM*, signal de *SPI_transmission*, signal d'horloge SPI et valeur du *LED Address Counter*.

Sortie : Commandes pour le *SPI Transmission*, *LED Address Counter*, *MUX* et *Frame FSM*.

Interaction avec d'autres blocs :

- Reçoit le start de *Frame FSM*.
- Reçoit le ready de *SPI transmission*.
- Envoie le start au *SPI Transmission* pour l'affichage des LED.
- Envoie le *led_ready* au *Frame FSM*.
- Change l'adresse du *MUX* pour sélectionner les données à envoyer.
- Incrémente et réinitialise *LED Address Counter*.

Fonctionnement :



→ **LED Address Counter**

Nom VHDL : count_addr

Description : Génère les adresses pour sélectionner la LED désirée.

Entrée : Signal de *LED FSM*.

Sortie : Adresse de la LED sur 8 bits (0-147).

Interaction avec d'autres blocs : Reçoit des instructions du *LED FSM* et envoie l'adresse au **Address Adder** et à la *FSM*.

→ **SPI Transmission**

Nom VHDL : serialisation

Description : Gère l'envoi des données via le protocole SPI pour piloter les LED.

Entrée : Données du *MUX* et signaux de commande de la *FSM*.

Sortie : Signal *ready*, *clock* et données pour le SPI.

Interaction avec d'autres blocs :

- Reçoit des données du *MUX*.
- Envoie des données SPI aux LED via le bus SPI.
- Indique qu'il peut de nouveau envoyer des données.

→ **MUX**

Nom VHDL : mux

Description : Sélectionne la source des données à envoyer via SPI.

Entrée : Données sur 32 bits (mémoire ou 0) et données de contrôle.

Sortie : Données SPI finales.

Interaction avec d'autres blocs :

- Sélectionne les données à envoyer selon *LED FSM*.
- Dirige les données finales via SPI.

Afin de simplifier la mise en place de tous ces blocs, nous avons mis en place des génériques. Ils permettent de rendre les blocs plus modulables. Ainsi, il est possible de configurer :

- le nombre de LEDs sur le bandeau
- la taille en bits du nombre de LEDs
- le nombre de frames :

$$\text{count} = \text{nb\_led} \times \text{nb\_frame}$$

- la taille en bits du nombre de frames
- la fréquence de la clock SPI :

$$\text{freq\_div} = \frac{f_{\text{processeur}}}{2 \times f_{\text{spi}}}$$

4. Traitement d'image et gestion de la mémoire

Dans cette section, nous décrivons le processus de traitement d'image permettant de convertir une image standard en un format exploitable par l'écran LED rotatif. Ce traitement inclut le redimensionnement de l'image, la transformation en coordonnées polaires et le stockage en mémoire.

4.1. Chargement et redimensionnement de l'image

La fonction `load_image_to_matrix` est utilisée pour charger une image, la redimensionner à 144×144 pixels et la convertir en une matrice NumPy pour un traitement ultérieur.

Listing 1: Chargement et redimensionnement de l'image

```
def load_image_to_matrix(image_path, size=(144, 144)):
    img = Image.open(image_path)
    img = img.resize(size, Image.LANCZOS)
    img = img.convert('RGB')
    img_matrix = np.array(img)
    return img_matrix
```

Entrées:

- `image_path`: Chemin d'accès à l'image.
- `size`: Tuple définissant la taille cible de l'image (par défaut: 144×144).

Sortie:

- Une matrice NumPy `img_matrix` de taille $(144, 144, 3)$ contenant les pixels en format RGB.

4.2. Transformation en coordonnées polaires

La fonction `get_pixel_from_polar` permet d'extraire la couleur d'un pixel dans la matrice d'image en utilisant ses coordonnées polaires (r, α), converties en coordonnées cartésiennes.

Listing 2: Transformation polaire vers cartésienne

```
def get_pixel_from_polar(image_matrix, angle_deg, radius):
    height, width, _ = image_matrix.shape
    center_x, center_y = width // 2, height // 2
    angle_rad = math.radians(angle_deg)
    x = int(center_x + radius * math.sin(angle_rad))
    y = int(center_y + radius * math.cos(angle_rad))
    if 0 <= x < width and 0 <= y < height:
        return tuple(image_matrix[y, x])
    else:
        return None
```

Entrées:

- `image_matrix`: Matrice NumPy représentant l'image.
- `angle_deg`: Angle en degrés.
- `radius`: Distance depuis le centre.

Sorties:

- Un tuple (R, G, B) représentant la couleur du pixel si les coordonnées sont valides.
- `None` si les coordonnées calculées sont hors de l'image.

4.3. Traitement des pixels et stockage en mémoire

La fonction `process_pixels_polar` parcourt l'image en utilisant une boucle imbriquée sur les angles et les rayons, extrait les pixels correspondants et les stocke en mémoire sur la carte FPGA.

Listing 3: Traitement des pixels en coordonnées polaires

```
def process_pixels_polar(image_matrix, mmio, angle_step=1, radius_step=1):
    led_index = 0  # Index des LEDs en m moire
    for angle_deg in range(1, 360, angle_step):
        for radius in range(-72, 72, radius_step):
            pixel = get_pixel_from_polar(image_matrix, angle_deg, radius)
            if angle_deg == 359:
                print("zero")
                pixel = (0,0,0)
            if pixel is not None:
                set_led_color(mmio, angle_deg, led_index, pixel)
                led_index += 1
        with open("led_output.txt", "a") as file:
            file.write("\n")
```

Entrées:

- `image_matrix`: L'image chargée sous forme de matrice NumPy.
- `mmio`: Objet de gestion de mémoire pour la communication avec le FPGA.
- `angle_step`: Pas d'incrémentation pour les angles (par défaut: 1°).
- `radius_step`: Pas d'incrémentation pour les rayons (par défaut: 1 pixel).

Sorties:

- La fonction n'a pas de valeur de retour, mais écrit les données des pixels en mémoire et dans un fichier texte.

4.4. Écriture des données en mémoire

La fonction `set_led_color` stocke les informations des pixels traités dans la mémoire du FPGA sous un format spécifique. Chaque LED est représentée sur 32 bits, comprenant les informations suivantes :

- **3 bits** : Indicateur d'activation de la LED (enable).
- **5 bits** : Luminosité (`brightness`), allant de 0 à 31.
- **8 bits** : Canal de couleur rouge (`R`).
- **8 bits** : Canal de couleur verte (`G`).
- **8 bits** : Canal de couleur bleue (`B`).

Listing 4: Écriture des données des pixels en mémoire

```
def set_led_color(mmio, frame, led_index, color):
    brightness = max(0, min(31, light))  # Luminosit  sur 5 bits (0-31)

    # Conversion des couleurs RGB sur 8 bits en valeurs compress es sur 4 bits
    r = int(color[0] // 16)  # R duction de 8 bits    4 bits
    g = int(color[1] // 16)
    b = int(color[2] // 16)

    # Cr ation du mot de 32 bits avec les champs requis
    color_hex = (0b111 << 29) | (brightness << 24) | (b << 16) | (g << 8) | r

    # Calcul de l'adresse m moire pour la LED concern e
    address_offset = (frame * NUM_LEDS + led_index) * 4  # 1 LED = 4 octets (32 bits)
```

```
    #   criture   de la valeur calcul e     l'adresse correspondante
    mmio.write(address_offset, color_hex)

    #   criture   dans un fichier texte pour suivi/debug
    with open("led_output.txt", "a") as file:
        file.write(f"{hex(color_hex)[2:].upper()} ")
```

Explication du calcul d'adresse : L'adresse est calculé de la manière suivante :

$$\text{Adresse mémoire} = (\text{Numéro de frame} \times \text{Nombre de LEDs}) + \text{Index de la LED}$$

L'offset d'adresse est ensuite multiplié par 4 pour correspondre à la taille de stockage en mémoire (**4 octets par LED**).

Explication de la conversion des couleurs : Les valeurs des couleurs R, G et B sont initialement codées sur **8 bits** (0-255). Pour être stockées dans la mémoire FPGA, elles sont utilisées sans réduction et directement encodées sur 8 bits chacun.

Le format des 32 bits stockés en mémoire est organisé comme suit :

- **3 bits** : Enable (toujours fixé à `0b111`).
- **5 bits** : Luminosité (`brightness`) entre 0 et 31.
- **8 bits** : Couleur rouge (`R`).
- **8 bits** : Couleur verte (`G`).
- **8 bits** : Couleur bleue (`B`).

Structure finale des données en mémoire (format 32 bits) :

Bits	31-29	28-24	23-16	15-8	7-0
Champ	Enable (`0b111`)	Brightness (5 bits)	Red (8 bits)	Green (8 bits)	Blue (8 bits)

Calcul du mot de 32 bits : L'encodage des valeurs est réalisé en décalant les bits à leur position respective dans le mot mémoire :

Listing 5: Encodage des couleurs en 32 bits

```
def set_led_color(mmio, frame, led_index, color):
    brightness = max(0, min(31, light))  # Luminosit  sur 5 bits (0-31)

    # R cup ration des valeurs R, G, B (8 bits chacune)
    r = int(color[0])
    g = int(color[1])
    b = int(color[2])

    # Construction du mot de 32 bits
    color_hex = (0b111 << 29) | (brightness << 24) | (r << 16) | (g << 8) | b

    # Calcul de l'adresse m moire pour la LED concern e
    address_offset = (frame * NUM_LEDS + led_index) * 4  # 1 LED = 4 octets (32 bits)

    #   criture   de la valeur calcul e en m moire
    mmio.write(address_offset, color_hex)

    # Sauvegarde dans un fichier pour suivi/debug
    with open("led_output.txt", "a") as file:
        file.write(f"{hex(color_hex)[2:].upper()} ")
```

Entrées:

- `mmio`: Objet mémoire pour écrire les données dans le FPGA.
- `frame`: Numéro de la frame actuelle.

- `led_index`: Index de la LED à mettre à jour.
- `color`: Tuple RGB représentant la couleur de la LED.

Sorties:

- La fonction écrit la couleur convertie en mémoire et enregistre les valeurs dans un fichier texte pour le suivi.

4.5. Initialisation et exécution

Le bloc suivant initialise la mémoire et lance le traitement d'image :

Listing 6: Exécution du traitement d'image

```
# Configuration
NUM_LEDS = 144  # Nombre de LEDs par frame
NUM_FRAMES = 361  # Nombre total de frames
light = 10

# Initialisation de la m moire
base_address = 0x40000000
mem_size = 1024 * NUM_FRAMES
mmio = MMIO(base_address, mem_size)

# Chargement et traitement de l'image
image_path = "4RC.png"
image_matrix = load_image_to_matrix(image_path)
process_pixels_polar(image_matrix, mmio, angle_step=1, radius_step=1)

print("Traitement termin  !")
```

5. Contrôle du moteur

5.1. Introduction et spécifications

Le moteur est un moteur de drone brushless contrôlé par un ESC (Electronic Speed Controler). Celui-ci gère les phases du moteur brushless.

Les moteur de drone avec ESC intègrent beauceaup de protocoles correspondant au types de signaux en entrée du contrôleur :

- Analogique :
  - PWM ⇒ Modulation de largeur d'impulsion
  - PPM ⇒ Modulation de la position d'impulsion
  - Oneshot 125 ⇒ Signal 4kHz qui émet une impulsion de largeur variant de 125 µs à 250 µs
  - Oneshot 42 ⇒ signal 8kHz deux fois plus rapide que le Oneshot 125
  - Multishot ⇒ même type de signal que pour Oneshot 42 mais avec taux de rafraîchissement à 32 kHz
- Numérique :
  - Dshot 150, 300, 600 ...
  - Proshot 1000

(source : https://www.electroya.com/fr/variadores-o-esc-protocolos/)

Pour ce projet il a été décidé d'utiliser une PWM en tant que signal de contrôle de l'ESC pour des questions de facilité de génération et de contrôle. On cherche donc à obtenir une PWM à 50Hz avec une largeur d'impulson réglable entre 1ms (vitesse minimale du moteur) et 2ms (vitesse maximale).

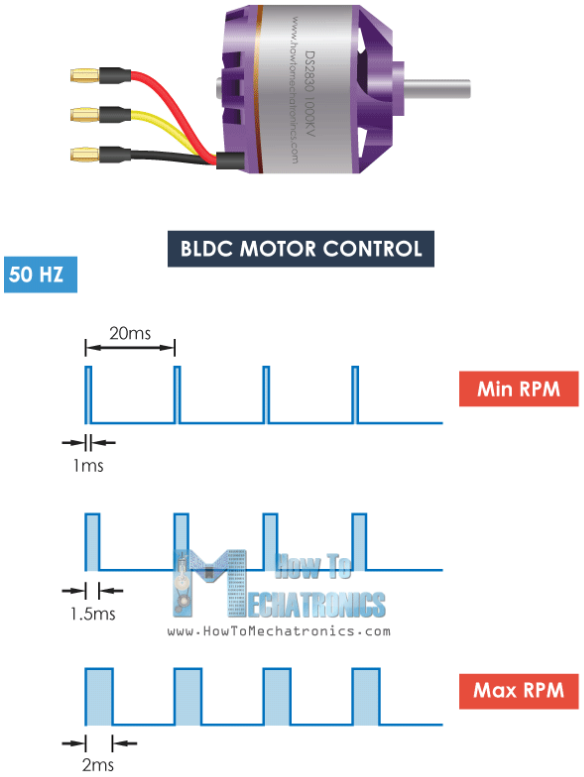


Figure 2: Protocole PWM avec ESC

(source : https://www.electroya.com/fr/variadores-o-esc-protocolos/)

Dans le cas de l'ESC utilisé (QWinOut 2-4S 30A) et de beaucoup d'autres, il est nécessaire de mettre en place une séquence de calibration des commandes de l'ESC. Pour cela les datasheet donnent les détails suivants :

- Débrancher l'ESC
- Mettre la gâchette de vitesse de la manette en position max pendant un certain temps ⇒ envoyer le signal correspondant à la vitesse maximale (largeur d'impulsion de 2 ms) pendant un certains temps
- Mettre la gâchette de vitesse en position minimale pendant un certain temps ⇒ impulsion de largeur 1 ms pendant un temps

Une dernière information importante concernant l'ESC réside dans la tension minimale des signaux de commande que celui-ci doit recevoir : **5V OU PLUS**

Pour résoudre le problème de signaux de commande à 3.3V, on utilise le montage suivant qui nécessite un rail d'alimentation 5V et 3.3v :

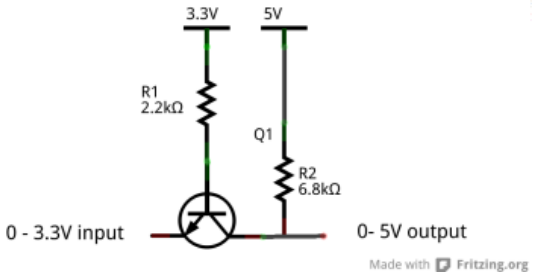


Figure 3: Montage de conversion signaux 3.3v ⇒ 5v

(source : https://electronics.stackexchange.com/questions/82104/single-transistor-level-up-shifter)

5.2. Partie VHDL

Pour cette partie, nous nous concentrons sur la génération du signal PWM de commande moteur à partir des blocs IP décrits en VHDL. Ces blocs sont représentés dans le diagramme suivant :

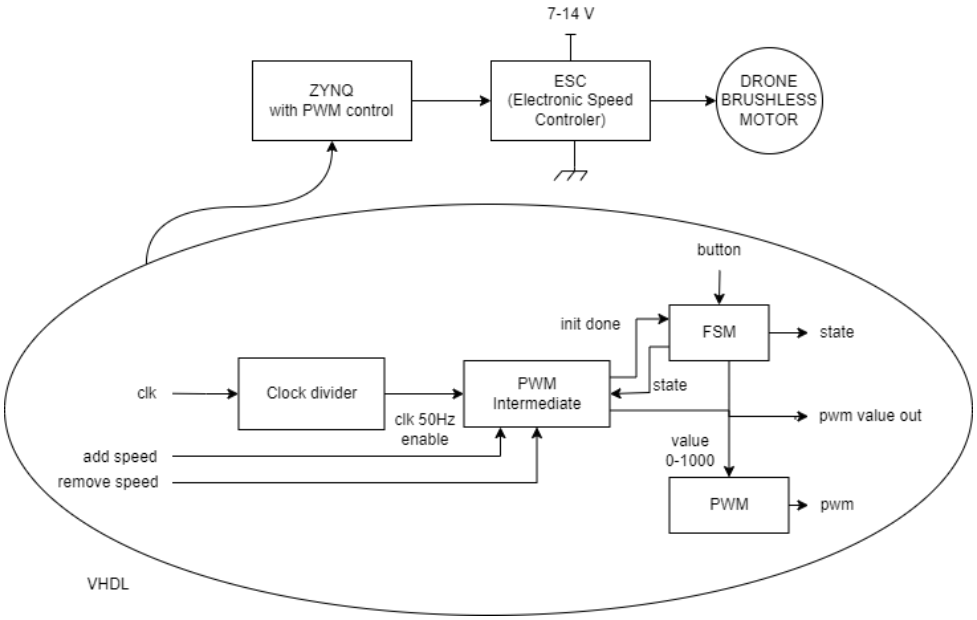


Figure 4: Blocs VHDL du contrôle moteur

Les timings sont basés sur le signal d'horloge de la carte Pynq fourni par le processeur Zynq (Cortex A9), réglé ici à 100 MHz (valeur par défaut).

Étapes principales

Création du signal *clk_enable* La première étape consiste en la génération d'un signal *clk_enable* à une fréquence de 50 Hz pour synchroniser les opérations. Un compteur est utilisé pour diviser l'horloge principale à 100 MHz. Le calcul est réalisé comme suit :

$$100\ \text{MHz} \Rightarrow \text{T} = 10\ \text{ns}$$
$$50\ \text{Hz} \Rightarrow \text{T} = 20\ \text{ms}$$
Donc :

$$\text{max\_count} = \frac{20 \times 10^{-3}}{10 \times 10^{-9}} = 2 \times 10^{6} = 2\,000\,000$$

Ainsi, le compteur doit avoir une valeur maximale de 2 000 000 pour générer un signal *clk_enable* à 50 Hz.

Machine d'état pour la calibration et la commande moteur Une machine d'état (FSM) est utilisée pour gérer les différentes phases du démarrage et de la commande moteur. Les états suivants sont définis :

- **WAITING** : Attente de l'appui de l'utilisateur sur un bouton pour démarrer.
- **INIT** : Phase d'initialisation avec génération d'un signal PWM pour la vitesse maximale.
- **UP** : Montée progressive de la vitesse du moteur.
- **STABLE** : État stable avec maintien de la vitesse configurée.
- L'état `WAITING` surveille l'appui de l'utilisateur sur un bouton pour enclencher la séquence d'initialisation.
- Dans l'état `INIT`, le signal PWM est configuré pour la vitesse maximale, en lien avec le bloc `pwm_intermediate`.
- L'état `UP` implémente une rampe de vitesse, augmentant progressivement la commande PWM jusqu'à atteindre la vitesse cible.
- Une fois la vitesse souhaitée atteinte, l'état `STABLE` est activé pour maintenir cette vitesse. Des boutons permettent de l'ajuster dynamiquement.

Un signal PWM progressif et contrôlé est essentiel pour éviter les oscillations dues au régulateur.

Description des blocs VHDL

`clock_divider`

- **Rôle** : Divise l'horloge principale (100 MHz) pour générer un signal `clk_50hz_enable` à 50 Hz.
- **Entrées** :
  - `clk` : Horloge principale.
  - `reset` : Signal de réinitialisation.
- **Sortie** :
  - `clk_50hz_enable` : Signal enable activé à 50 Hz.
- **Paramètres fixes** :
  - `DIVISOR` : Constante fixée à 1 000 000.

`fsm_esc`

- **Rôle** : Gère les états du moteur (attente, initialisation, montée de vitesse, stable).
- **Entrées** :
  - `clk` : Horloge principale.
  - `reset` : Signal de réinitialisation.
  - `button` : Bouton pour démarrer.
  - `value` : Valeur PWM.
  - `init_done` : Indicateur de fin d'initialisation.
- **Sorties** :
  - `state` : État courant de la FSM (2 bits).
  - `ready` : Indicateur d'état stable.

`pwm_intermediate`

- **Rôle** : Génère les valeurs PWM en fonction des états de la FSM.
- **Entrées** :
  - `clk` : Horloge principale.
  - `clk_50hz_enable` : Signal enable à 50 Hz.
  - `reset` : Signal de réinitialisation.
  - `state` : État courant de la FSM.
  - `add_speed`, `remove_speed` : Signaux pour ajuster la vitesse.
- **Sorties** :
  - `pwm_value` : Valeur PWM calculée.
  - `init_done` : Signal indiquant la fin de l'initialisation.
- **Paramètres fixes** :
  - `INIT_PWM` : Valeur initiale pour 1 ms.
  - `max_pwm` : Valeur maximale (280 par défaut).

`pwm_generator`

- **Rôle** : Génère le signal PWM en fonction des valeurs calculées.
- **Entrées** :
  - `clk` : Horloge principale.
  - `reset` : Signal de réinitialisation.
  - `duty_cycle` : Valeur du rapport cyclique.
- **Sortie** :
  - `pwm_out` : Signal PWM généré.

- **Paramètres fixes** :
  - `CLOCK_FREQ` : Fréquence d'horloge (100 MHz).
  - `PWM_FREQ` : Fréquence du signal PWM (50 Hz).
  - `MIN_PWM` : Durée minimale (1 ms).
  - `MAX_PWM` : Durée maximale (2 ms).

`top_level`

- **Rôle** : Intègre tous les modules pour générer un bloc IP.
- **Entrées** :
  - `clk`, `rst`, `button` : Horloge, réinitialisation, bouton.
  - `add_speed`, `remove_speed` : Commandes d'ajustement de vitesse.
- **Sorties** :
  - `pwm` : Signal PWM final.
  - `state` : État courant de la FSM.
  - `pwm_value_out` : Valeur PWM actuelle.

6. Améliorations possibles

Dans cette section, nous proposons plusieurs améliorations qui pourraient être apportées au projet pour améliorer la qualité d'affichage, la stabilité mécanique et la gestion du moteur.

6.1. Amélioration du système de synchronisation

Actuellement, la synchronisation de l'affichage repose sur un capteur à effet Hall détectant un aimant fixé sur l'axe de rotation. Cette méthode présente plusieurs limites :

- L'affichage est mis à jour **deux fois par tour**, ce qui entraîne un léger décalage entre les images successives.
- La vitesse de rafraîchissement du bandeau LED n'est pas parfaitement synchronisée avec la vitesse de rotation du moteur.
- De légers écarts dans la vitesse de rotation du moteur causent des imprécisions dans l'affichage.

L'un des principaux problèmes vient du fait que la position du bandeau LED est connue uniquement une fois par tour grâce au capteur à effet Hall. Entre ces points de référence, la position angulaire est estimée en supposant une vitesse constante. Cela entraîne plusieurs effets indésirables :

- Comme l'affichage est rafraîchi **360 fois par tour**, mais que la synchronisation avec la vitesse réelle du moteur n'est pas parfaite, il arrive que le moteur termine un tour avant que les 360 mises à jour aient été effectuées, ou inversement, que toutes les mises à jour soient effectuées avant la fin du tour.

- Les **deux images affichées par tour ne sont pas parfaitement alignées**, car l'erreur d'estimation de la vitesse entraîne un léger décalage entre les deux rafraîchissements successifs.

Une solution efficace pour corriger ce problème serait d'utiliser une **roue codeuse** offrant une résolution d'1° par impulsion. Cela permettrait :

- D'obtenir une **mesure précise et continue** de la position angulaire du bandeau LED.
- De supprimer l'approximation de la position entre deux tours et ainsi garantir un affichage toujours parfaitement aligné.
- D'assurer une synchronisation exacte entre la vitesse de rafraîchissement du bandeau LED et la vitesse de rotation du moteur.

En l'absence d'une roue codeuse, une solution alternative serait d'introduire un **asservissement en fréquence** du moteur. En comptant le nombre de fronts d'horloge dans un tour et en comparant ce nombre à la fréquence des mises à jour de l'affichage, il serait possible d'ajuster dynamiquement la commande du moteur pour maintenir une synchronisation précise.

6.2. Amélioration du démarrage du moteur

Lors du démarrage du moteur, la vitesse augmente trop rapidement. Actuellement, la séquence **"UP"** de la PWM incrémente la vitesse de 1 toutes les 20 ms. Cette montée en puissance brutale provoque un patinage de la courroie, ce qui nuit à la stabilité mécanique du système.

Pour résoudre ce problème, il faudrait :

- **Augmenter la durée de montée en puissance** en modifiant l'incrémentation : au lieu d'ajouter 1 toutes les 20 ms, on pourrait ajouter 1 toutes les 200 ms, ce qui ralentirait progressivement la montée en régime du moteur.

6.3. Optimisation de la partie mécanique

L'un des principaux problèmes mécaniques du projet est la **fragilité du collecteur rotatif**. Ce dernier présente plusieurs limitations qui affectent à la fois la stabilité du système et la qualité de l'affichage.

Problèmes identifiés

Actuellement, le collecteur rotatif utilisé n'est pas adapté aux contraintes mécaniques et électriques du système. Les problèmes observés sont les suivants :

- **Problème évident de sécurité**
- **Fixation insuffisante de la structure rotative** : La partie imprimée en 3D est faiblement maintenue, en grande partie à cause de l'inadaptation du collecteur rotatif utilisé. Cette mauvaise fixation entraîne des vibrations et des désalignements.
- **Jeu mécanique sur l'axe de rotation** : Le poids du système combiné à la méthode d'accroche actuelle génère un jeu perceptible au niveau de l'axe de rotation. Ce jeu perturbe la régularité de la rotation et introduit des imprécisions dans l'affichage.

- **Glitchs dans l'affichage** : Des anomalies d'affichage, telles que l'activation de toutes les LEDs en rouge, ont été observées de manière intermittente. Ces glitchs apparaissent notamment lorsque l'alimentation est coupée. Cela suggère un problème de transmission du courant à travers le collecteur rotatif, probablement dû à des faux contacts ou à une chute de tension excessive.
- **Limite de la vitesse de rotation** : Pour obtenir un affichage stable reposant sur la **persistance rétinienne**, la vitesse de rotation doit être significativement augmentée. Cependant, le collecteur rotatif actuel ne supporte pas de telles vitesses, ce qui limite fortement l'efficacité du rendu visuel.

Solutions envisagées

Le collecteur rotatif actuel est une source majeure de problèmes mécaniques et électriques. La solution prioritaire est son remplacement par un modèle **industriel de haute qualité**, même si cela représente un coût plus élevé.

Un collecteur **en métal avec des roulements de précision**, offrant des contacts fiables et une faible résistance, permettrait d'éliminer les glitchs d'affichage et les pertes de tension observées.

De plus, la fixation actuelle ne supporte l'axe que d'un seul côté, ce qui engendre du jeu et des vibrations. Une **fixation des deux côtés**, associée à un support rigide, garantirait une meilleure stabilité et permettrait d'augmenter la vitesse de rotation sans risque.

Par ailleurs, les données des LED sont actuellement envoyées en 3.3V, alors qu'il est préférable de les transmettre en 5V pour assurer un fonctionnement optimal. Il serait donc nécessaire d'ajouter un convertisseur de tension précis afin d'adapter correctement le signal.

Conclusion

L'amélioration du système passe principalement par trois axes : **une meilleure synchronisation de l'affichage**, **un démarrage moteur plus progressif** et **une optimisation mécanique du collecteur rotatif**.

Le remplacement du collecteur rotatif par un modèle **industriel en métal avec fixation des deux côtés** est une priorité. Il permettrait d'éliminer les glitchs d'affichage, de stabiliser la rotation et d'augmenter la vitesse pour atteindre un affichage fluide basé sur la persistance rétinienne.

L'intégration d'une **roue codeuse à 1° de résolution** garantirait une parfaite synchronisation entre l'affichage et la position du bandeau LED, supprimant les erreurs d'alignement entre les images.

Enfin, une **réduction de la rampe d'accélération** du moteur permettrait d'éviter le patinage et d'améliorer la durabilité du système mécanique.

Problème	**Solution proposée**
Synchronisation imparfaite de l'affichage	Utiliser une roue codeuse avec une résolution de 1°
Vitesse de rotation non totalement contrôlée	Assurer un asservissement en fréquence du moteur en fonction des fronts d'horloge
Démarrage trop rapide du moteur	Réduire l'incrémentation de la PWM pour éviter le patinage
Collecteur rotatif fragile	Améliorer le modèle ou utiliser une transmission d'énergie sans contact
Tension des données pour les LED	Envoyer les données en 5V pour eviter les glitchs
Problèmes mécaniques (vibrations)	Améliorer l'équilibrage et la fixation des composants

Table 1: Résumé des améliorations possibles

L'intégration de ces améliorations permettrait d'obtenir un affichage plus stable, plus précis et mieux synchronisé, tout en améliorant la fiabilité mécanique du dispositif.