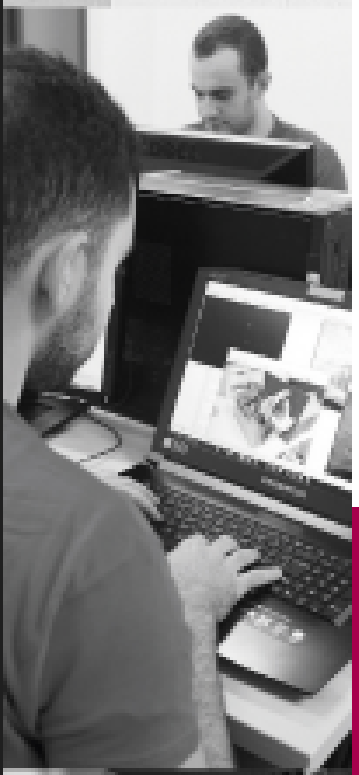


ENSEA

Beyond Engineering

Compte rendu TP4

Des usages multiples des Timers

derridj mellyna weber loic

14 novembre 2025

4.1 Préparation

4.1.2 Questions de préparation relatives au moteur pas à pas.

On relève :
$f_{timer3} = 80Mhz \quad PSC = 0 \quad ARR = 65535$. Alors :

$$f = \frac{f_{timer3}}{(ARR+1)(PSC+1)} = 1220Hz$$

On considère toujours PSC=0 et on souhaite une fréquence de 2 tours par seconde alors :

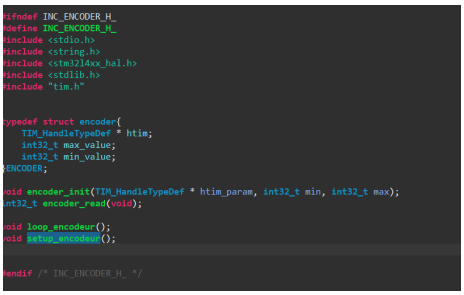
$$ARR = \frac{f_{timer3}}{f_{souhaité}} - 1 = 99999Hz$$

Vitesse	**tr/min**	**tr/s**	**Valeur du registre ARR pour PSC = 79**
1	1	0.0167	14969
2	2	0.033	7574
3	5	0.083	3011
4	10	0.167	1496
5	30	0.5	499
6	60	1	249
7	90	1.5	165
8	120	2	124

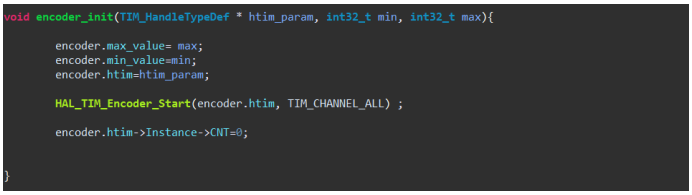
TABLE 1 – Valeur du registre ARR pour différentes vitesses et PSC = 79

4.2. - partie I : le driver du codeur incrémental

On place la structure dans le fichier.h et la variable static dans le .c pour l'utiliser par la suite :



Les entrées channel3 et channel4 sont utilisées pour recevoir deux signaux d'un codeur incrémental. En mode encodeur, le timer interprète les fronts montants et descendants des signaux pour déterminer la direction de rotation et incrémenter ou décrémenter le compteur. On peut également (en fonction du déphasage des deux signaux) déterminer le sens de la rotation.
fonction encoder init :



La fonction HAL TIM Encoder Start est une fonction d'état qui permet de lire l'état de l'encodeur (Hal Error, Hal Okay etc) et elle permet de lancer l'encodeur. Elle prend en entrée un pointeur sur TIM HandleTypeDef et le channel du timer (puisque aucun n'est précisé en particulier on choisit TIM CHANNEL ALL donc tous disponibles).

fonction encoder read :

```
int32_t encoder_read(void){

        int32_t valeur = encoder.htim->Instance->CNT; // valeur du CNT

        if(valeur > encoder.max_value) { // condition de saturation haute
            encoder.htim->Instance->CNT=encoder.max_value;

            valeur = encoder.max_value; // valeur fixé au max
        }

        if(valeur < encoder.min_value) { // condition de saturation basse
            encoder.htim->Instance->CNT=encoder.min_value;
            valeur = encoder.min_value; // valeur aussi fixé au min
        }

        ecran(valeur); // affichage sur l'oled
        return valeur;


}

void setup_encodeur(){
    encoder_init(&htim2, -8,8);

}

void loop_encodeur(void){
    int velue = encoder_read();
    printf(" %d \r\n", velue );
}
```

Dans l'appel de la fonction encoder init (dans le setup) on passe en argument htim2 car on travaille sur le timer 2. Cette variable est issue des fichiers dans STM32.

On rajoute la fonction écran qui permet d'afficher sur l'OLED la valeur du codeur.

La fonction écran se base sur les codes fournies en annexe (sur Moodle fichier ssd1315.h fonts.h) utilisé lors des TP2 et 3, on la rappelle ci-dessous :

```
void int_to_string(int value, char *buffer) {
    sprintf(buffer, "%d", value);
}

void ecran(int32_t x){
    char c[20];     // assez grand pour contenir le résultat
    int_to_string(x, c);
    ssd1315_Init();
    ssd1315_Clear(SSD1315_COLOR_BLACK);
    ssd1315_Draw_String(0,0,c ,&Font_16x26);
    ssd1315_Refresh();
    HAL_GPIO_WritePin(GPS_ENN_GPIO_Port,GPS_ENN_Pin,1);
}
```

4.3. - partie II : le driver du moteur pas à pas

De la même façon que la partie précédente, on initialise une variable static (peut donc être utilisée par toutes les fonctions du fichier) nommée motor .

```c
#include "stepperMotor.h"
#include "main.h"

static STEPPERMOTOR motor ;

void init_stepper(TIM_HandleTypeDef * htim_pwm_p, uint32_t pwm_channel_number_p,
        GPIO_TypeDef * gpio_direction_p, uint16_t gpio_direction_pin_p,
        GPIO_TypeDef * gpio_ms1_p, uint16_t gpio_ms1_pin_p,
        GPIO_TypeDef * gpio_ms2_p, uint16_t gpio_ms2_pin_p,
        GPIO_TypeDef * gpio_enable_p, uint16_t gpio_enable_pin_p){

    // Copie des parametres dans la structure locale
    motor.htim_pwm= htim_pwm_p;
    motor.pwm_channel_number= pwm_channel_number_p;
    motor.gpio_direction= gpio_direction_p;
    motor.gpio_direction_pin= gpio_direction_pin_p;
    motor.gpio_ms1= gpio_ms1_p;
    motor.gpio_ms1_pin= gpio_ms1_pin_p;
    motor.gpio_ms2= gpio_ms2_p;
    motor.gpio_ms2_pin= gpio_ms2_pin_p;
    motor.gpio_enable= gpio_enable_p;
    motor.gpio_enable_pin= gpio_enable_pin_p;

    // Configuration initiale des GPIO du driver
    HAL_GPIO_WritePin(motor.gpio_direction, motor.gpio_direction_pin,0); // sens initial horaire
    HAL_GPIO_WritePin(motor.gpio_enable, motor.gpio_enable_pin,1);       // disable au début, ça tourne pas
    HAL_GPIO_WritePin(motor.gpio_ms1, motor.gpio_ms1_pin,0);             // microstep MS1 = 0
    HAL_GPIO_WritePin(motor.gpio_ms2, motor.gpio_ms2_pin,0);             // microstep MS2 = 0

    // Demarrage du PWM
    HAL_TIM_PWM_Start(motor.htim_pwm,motor.pwm_channel_number);
}
```

La fonction Hal Tim Pwm STart permet de générer un signal PWM sur un canal donné.

fonction launch stepper :

```c
void launch_stepper(int speed){

    // Valeur absolue de la vitesse pour indexer le tableau
    int abs_speed_value = abs(speed);

    // Limite la vitesse maximale si > 8
    if ( abs_speed_value >8){
        abs_speed_value= 8;
    }


    motor.speed=speed;

    // Tableau des valeurs d'ARR correspondant aux vitesses 1 à 8
    uint16_t period_counter_value[8]={14969,7574,3011,1496,499,249,165,124}; // valeurs de la préparation

    // Active le driver (enable = 0 pour certains drivers type A4988)
    HAL_GPIO_WritePin(motor.gpio_enable, motor.gpio_enable_pin,0);

    // Détermination du sens selon le signe de speed
    if ( speed >0){
            HAL_GPIO_WritePin(motor.gpio_direction, motor.gpio_direction_pin,0); // sens horaire
        }
    else {
            HAL_GPIO_WritePin(motor.gpio_direction, motor.gpio_direction_pin,1); // sens trigo
    }

    // Mise à jour de l'ARR selon la vitesse choisie
    motor.htim_pwm->Instance->ARR=period_counter_value[abs_speed_value-1];

    // PWM à 50% du cycle
    motor.htim_pwm->Instance->CCR1= motor.htim_pwm->Instance->ARR/2;
}
```