



UNIVERSITÉ
CAEN
NORMANDIE



ESIX NORMANDIE CAEN
MÉCATRONIQUE ET SYSTÈMES NOMADES

COMPTE-RENDU

TP de Microcontrôleur 32bits ARM Cortex-M

BEHILIL Mohamed Yassine

2023-2024

COMPTE-RENDU	1
TP 1 : Gestion du temps	4
1 Travail préparatoire :	4
2 Développement de la bibliothèque MeSN_TimeBase	4
2.1 Manipulation de registres	4
2.2 Bibliothèque CMSIS	4
2.3 Interruptions	5
2.4 Analyse et conclusion	6
3 Test de la bibliothèque MeSN_TimeBase	7
TP 2 : Périphériques externes basiques	9
1 Travail préparatoire	9
2 Driver du clavier matriciel	9
2.1 Portage du driver	9
2.2 Test du driver	10
3 Driver de l'afficheur LCD	10
3.1 Portage du driver	10
3.2 Test du driver	11
TP 3. Génération d'un signal PWM	12
1 Travail préparatoire	12
1.1 Modulation à largeur d'impulsion	12
1.2 Timer et PWM	12
1.3 Assignment d'une broche physique	12
2 Développement via l'IDE et Utilisation	13
2.1 Utilisation	13
2.2 Conclusion	14
TP 4. Pilotage d'un Moteur à Courant Continu	15
1 Travail préparatoire	15
1.1 Modulation du rapport cyclique	15
1.2 Étage de puissance	15
2 Développement de la bibliothèque MotorDriver	15
2.1 Couche pilote (MotorDriver.c et MotorDriver.h)	15
2.2 Couche portabilité (MotorDriver_port.c et MotorDriver_port.h)	16
2.3 Test de la bibliothèque MotorDriver	17
2.4 Conclusion	18
TP 5. Communication SPI et carte SD	20
1 Travail préparatoire	20

1.1	Liaison SPI -----	20
1.2	Carte mémoire Secure Digital (SD) -----	21
2	<i>Liaison SPI :</i> -----	21
2.1	Vérification du fonctionnement-----	22
3	<i>Carte SD</i> -----	23
3.1	Adaptation de la couche basse SD_IO -----	23
4	<i>Test du fonctionnement</i> -----	24

TP 1 : Gestion du temps

1 Travail préparatoire :

- 📖 Le **bit 0**, nommé "CEN" (Counter Enable), dans le registre **TIMx_CR1**, contrôle la fonction d'activation/désactivation du compteur pour le périphérique TIM6.
- 📖 Le **bit UIF** du registre **TIMx_SR** agit comme un indicateur d'interruption. Si ce bit est à l'état logique 1 et que l'interruption est activée, il signale que la routine d'interruption doit être exécutée.
- 📖 Nous disposons de plusieurs timers, et il est important de noter que les registres de configuration sont identiques pour tous. Dans la documentation, chaque registre de configuration est associé à une adresse d'offset spécifique.
Dans notre cas, nous recherchons l'adresse du registre TIM6_PSC. L'offset de **TIMx_PSC** est donné comme **0x28** en notation hexadécimale. Tous les registres de configuration de TIM6 commencent à l'**adresse de base 0x40001000**. Par conséquent, l'adresse de TIM6_PSC est calculée en ajoutant l'offset à l'adresse de base, ce qui donne **0x40001028**. De la même manière, l'adresse de **TIM6_ARR est 0x4000102C**.
Un point essentiel à noter est que la connaissance de la taille des registres est essentielle pour garantir que vous manipulez le bon nombre de bits lors de la configuration des timers. La taille des registres **TIM6_PSC et TIM6_ARR est de 16 bits** chacun.
- 📖 L'**UIE** est le bit de contrôle utilisé pour activer ou désactiver les interruptions associées à la mise à jour du compteur du Timer 6, pas nécessairement au "Overflow" du timer, **(une interruption peut être déclenchée pour gérer ce changement de fréquence aussi)**.
- 📖 En consultant la documentation concernant la distribution des sources d'horloge dans le microcontrôleur, il est clair que le périphérique Timer 6 est connecté au bus APB1.
- 📖 Les sources d'horloge possibles pour générer la SYSCLOCK sont HSI, HSE.
- 📖 L'horloge du timer TIM6 est obtenue en divisant la fréquence de la SYSCLOCK par **un prescaler dans la buse AHB** puis un prescaler dans **la buse APB1**.

2 Développement de la bibliothèque MeSN_TimeBase

2.1 Manipulation de registres

```
1 | *((volatile uint32_t *) (0x40001028)) = (uint16_t)prescalVal;
```

Pour clarifier davantage cette ligne, il est essentiel de comprendre deux concepts clés. Lorsque vous déclarez une variable de la forme `type *var`, vous définissez `var` en tant qu'adresse mémoire, tandis que `*var` représente le contenu stocké à cette adresse.

Donc `(volatile uint32_t *) (0x40001028)` spécifie bien que 0x40001028 est une adresse de 32 bit (ce qui est conforme à notre connaissance selon laquelle cette adresse correspond au registre TIM6_PSC.). Cependant, étant donné que TIM6_PSC est un registre de 16 bits (comme indiqué dans notre travail préparatoire), nous utilisons la conversion `(uint16_t)`. Ainsi nous affectons la valeur du prescaler au registre TIM6_PSC.

2.2 Bibliothèque CMSIS

Fichier stm32l152xe.h :

```

#define PERIPH_BASE          (0x40000000UL)
...
#define APB1PERIPH_BASE      PERIPH_BASE
#define APB2PERIPH_BASE      (PERIPH_BASE + 0x00010000UL)
#define AHBPERIPH_BASE       (PERIPH_BASE + 0x00020000UL)
...
#define TIM6_BASE            (APB1PERIPH_BASE + 0x00001000UL)
...
#define TIM6                  ((TIM_TypeDef *) TIM6_BASE)

```

L'adresse de base de TIM6 (**TIM6_Base**) est située à **0x40001000UL**, ce qui équivaut à la somme de 0x40000000UL et de 0x00001000UL (UL signifiant unsigned long). À cette adresse, vous trouverez la structure qui contient tous les registres de configuration du TIM6, notamment :

```

typedef struct
{
    __IO uint32_t CR1;          /*!< TIM control register 1,           Address offset: 0x00 */
    __IO uint32_t CR2;          /*!< TIM control register 2,           Address offset: 0x04 */
    __IO uint32_t SMCR;         /*!< TIM slave Mode Control register,   Address offset: 0x08 */
    __IO uint32_t DIER;         /*!< TIM DMA/interrupt enable register, Address offset: 0x0C */
    __IO uint32_t SR;           /*!< TIM status register,              Address offset: 0x10 */
    __IO uint32_t EGR;          /*!< TIM event generation register,     Address offset: 0x14 */
    __IO uint32_t CCMR1;        /*!< TIM capture/compare mode register 1, Address offset: 0x18 */
    __IO uint32_t CCMR2;        /*!< TIM capture/compare mode register 2, Address offset: 0x1C */
    __IO uint32_t CCER;         /*!< TIM capture/compare enable register, Address offset: 0x20 */
    __IO uint32_t CNT;          /*!< TIM counter register,             Address offset: 0x24 */
    __IO uint32_t PSC;          /*!< TIM prescaler register,           Address offset: 0x28 */
    __IO uint32_t ARR;          /*!< TIM auto-reload register,         Address offset: 0x2C */
    uint32_t RESERVED12;        /*!< Reserved, 0x30                    */
    __IO uint32_t CCR1;         /*!< TIM capture/compare register 1,     Address offset: 0x34 */
    __IO uint32_t CCR2;         /*!< TIM capture/compare register 2,     Address offset: 0x38 */
    __IO uint32_t CCR3;         /*!< TIM capture/compare register 3,     Address offset: 0x3C */
    __IO uint32_t CCR4;         /*!< TIM capture/compare register 4,     Address offset: 0x40 */
    uint32_t RESERVED17;        /*!< Reserved, 0x44                    */
    __IO uint32_t DCR;          /*!< TIM DMA control register,          Address offset: 0x48 */
    __IO uint32_t DMAR;         /*!< TIM DMA address for full transfer,  Address offset: 0x4C */
    __IO uint32_t OR;           /*!< TIM option register,              Address offset: 0x50 */
} TIM_TypeDef;

```

Les deux lignes de code pour réinitialiser le registre de comptage et activer le compteur sont les suivantes :

```

/* Réinitialisation de la valeur du compteur */
TIM6->CNT = 0x0000 ;
/* Activation du compteur et de l'événement de mise à jour */
TIM6->CR1 |= 0x0001;

```

2.3 Interruptions

Extraction du fichier stm32l152xe.h :

```

TIM6_IRQn = 43,          /*!< TIM6 global Interrupt >

```

Cette valeur à l'aide de la datasheet :

Position	Priority	Type of priority	Acronym	Description	Address
43	50	settable	TIM6	TIM6 global interrupt	0x0000_00EC

- ↪ La mise à jour de la fonction `MeSN_InitTimeBase()` pour activer les interruptions en provenance du timer TIM6 se présente ainsi :

```
NVIC_EnableIRQ(TIM6_IRQn);
```

D'après la fonction `SystemInit()`, après un reset du MCU, le vecteur d'interruption est configuré pour être dans la mémoire flash (Flash) à 0x08000000, notamment:

System_stm32l1xx.c :

```
#if defined(USER_VECT_TAB_ADDRESS)
/* #define VECT_TAB_SRAM */
#if defined(VECT_TAB_SRAM)
#define VECT_TAB_BASE_ADDRESS    SRAM_BASE
#define VECT_TAB_OFFSET          0x00000000U
#else
#define VECT_TAB_BASE_ADDRESS    FLASH_BASE
#define VECT_TAB_OFFSET          0x00000000U
#endif /* VECT_TAB_SRAM */
#endif /* USER_VECT_TAB_ADDRESS */

void SystemInit (void)
{
#ifdef DATA_IN_ExtSRAM
    SystemInit_ExtMemCtl();
#endif /* DATA_IN_ExtSRAM */

    /* Configure the Vector Table location -----*/
    #if defined(USER_VECT_TAB_ADDRESS)
        SCB->VTOR = VECT_TAB_BASE_ADDRESS | VECT_TAB_OFFSET; //Vector Table Relocation
    #endif /* USER_VECT_TAB_ADDRESS */
}
```

- ↪ La directive **.word label** alloue un espace mémoire et initialise son contenu avec la valeur de "label". Cette technique est particulièrement utile lors de la gestion d'interruptions.
- ↪ Le nom attendu par le linker pour l'ISR du timer TIM6 est généralement **TIM6_IRQHandler()**. Ce nom se trouve dans le fichier source assembleur du startup `startup_stm32l152retx.s`, fourni par la chaîne de compilation.
- ↪ L'adresse de la case mémoire où sera rangé le label `TIM6_IRQHandler` correspond à l'adresse de base du vecteur d'interruption (0x08000000) + offset de l'interruption du timer 6 (0xEC).
- ↪ La variable **timeSinceLastReset** est déclarée en tant que variable globale car elle est destinée à être utilisée par plusieurs fonctions à travers tout le programme.

2.4 Analyse et conclusion

Dans ce TP, nous avons exploité le débordement du Timer 6 sous forme d'interruption pour l'utiliser dans la fonction **MESN_Delay(volatile uint32_t waitingTime)**. Le débordement se produit à chaque période de 100 microsecondes, et cette fonction est appelée pour créer une attente d'une durée spécifiée, '**waitingTime**'. Lorsque le microcontrôleur est alimenté, il active la routine d'interruption suivante :

```
void TIM6_IRQHandler(void) {
    /* Clear the interrupt pending bit ( UIF ) */
    TIM6 -> SR = ~0x0001 ;
    /* Process ISR */
    IncTime () ;
}
```

La variable `timeSinceLastReset` commence à s'incrémenter suite à ces interruptions. Lorsque nous appelons la fonction **MESN_Delay(uint32_t Var)**, nous examinons la valeur de la variable `timeSinceLastReset`. Étant donné qu'elle s'incrémente toutes les **100 microsecondes**, nous souhaitons rester dans cette fonction d'attente sans effectuer aucune action jusqu'à ce que la variable **timeSinceLastReset** atteigne une valeur égale à **timeSinceLastReset + waitingTime**.

Cela se traduit par la ligne suivante : `while ((GetTime() - startTime) < waitingTime) {}`

Ainsi, le fonctionnement de la fonction d'attente **MESN_Delay** est basé sur la comparaison de temps pour créer une pause d'attente avant de poursuivre l'exécution du programme.

Remarque :

Une question intrigante qui peut se poser concerne l'incrémentation de la variable `timeSinceLastReset` et la possibilité de son dépassement. La réponse est que la déclaration de cette variable est `uint32_t`, ce qui signifie qu'elle peut atteindre une valeur maximale de 4 294 967 295, équivalant à une durée de 119 heures et 18 minutes avant que la variable ne déborde.

Si `timeSinceLastReset` atteint sa valeur maximale (4 294 967 295) et continue à s'incrémenter à partir de là, il débordera et reviendra à zéro. Si la fonction **MESN_Delay** est en attente lorsque ce débordement se produit, cela pourrait entraîner **une attente incorrecte ou une durée d'attente beaucoup plus longue que prévue**, car la condition **(GetTime() - startTime) < waitingTime** ne se remplirait pas correctement.

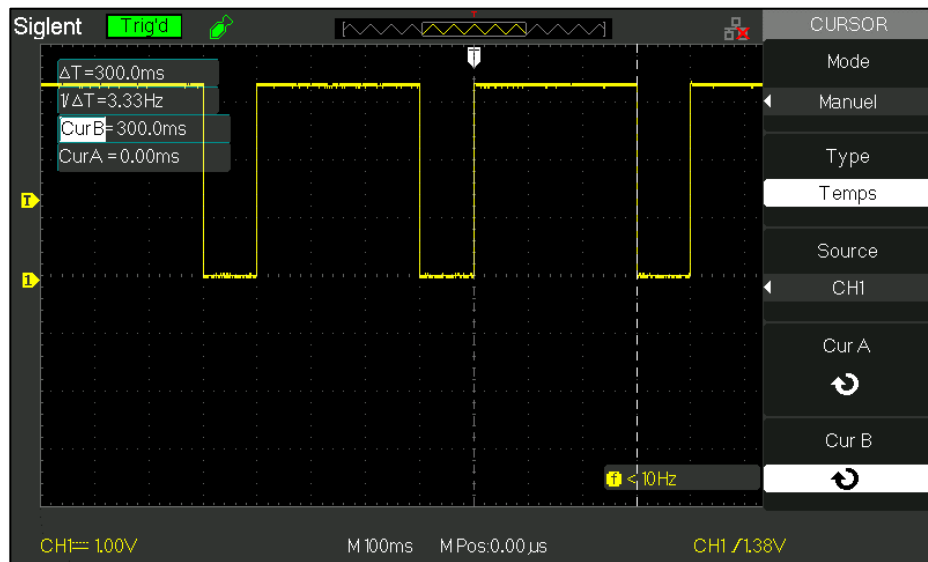
Ce problème réside dans le fait que la variable `timeSinceLastReset` est une variable non signée (`uint32_t`) et qu'elle débordera naturellement lorsqu'elle atteindra sa valeur maximale. Cela peut provoquer des comportements indésirables dans la fonction **MESN_Delay**, car elle ne peut pas détecter ce débordement et continuera d'attendre en fonction des valeurs de **startTime** et **waitingTime**. Pour résoudre ce problème, il faudrait gérer le débordement potentiel de la variable `timeSinceLastReset` dans la fonction **MESN_Delay** pour garantir que l'attente se comporte correctement même en cas de débordement.

3 Test de la bibliothèque MeSN_TimeBase

Grace au code fournie dans le (main) :

```
...
GPIOA->ODR |= (0x01 << 5);    //LED on
MESN_Delay(3000); //300 ms
GPIOA->ODR &= ~(0x01 << 5);    //LED off
MESN_Delay(1000); // 100 ms
...
```

Nous vérifions la broche PA5 à l'oscilloscope et nous observons que :



Cela correspond parfaitement à notre analyse, car le débordement se produit toutes les 100 microsecondes. En conséquence, la broche reste à l'état haut pendant 300 millisecondes, puis s'éteint pendant 100 millisecondes, ce qui donne une période totale de 400 millisecondes, en parfait accord avec les mesures effectuées à l'oscilloscope.

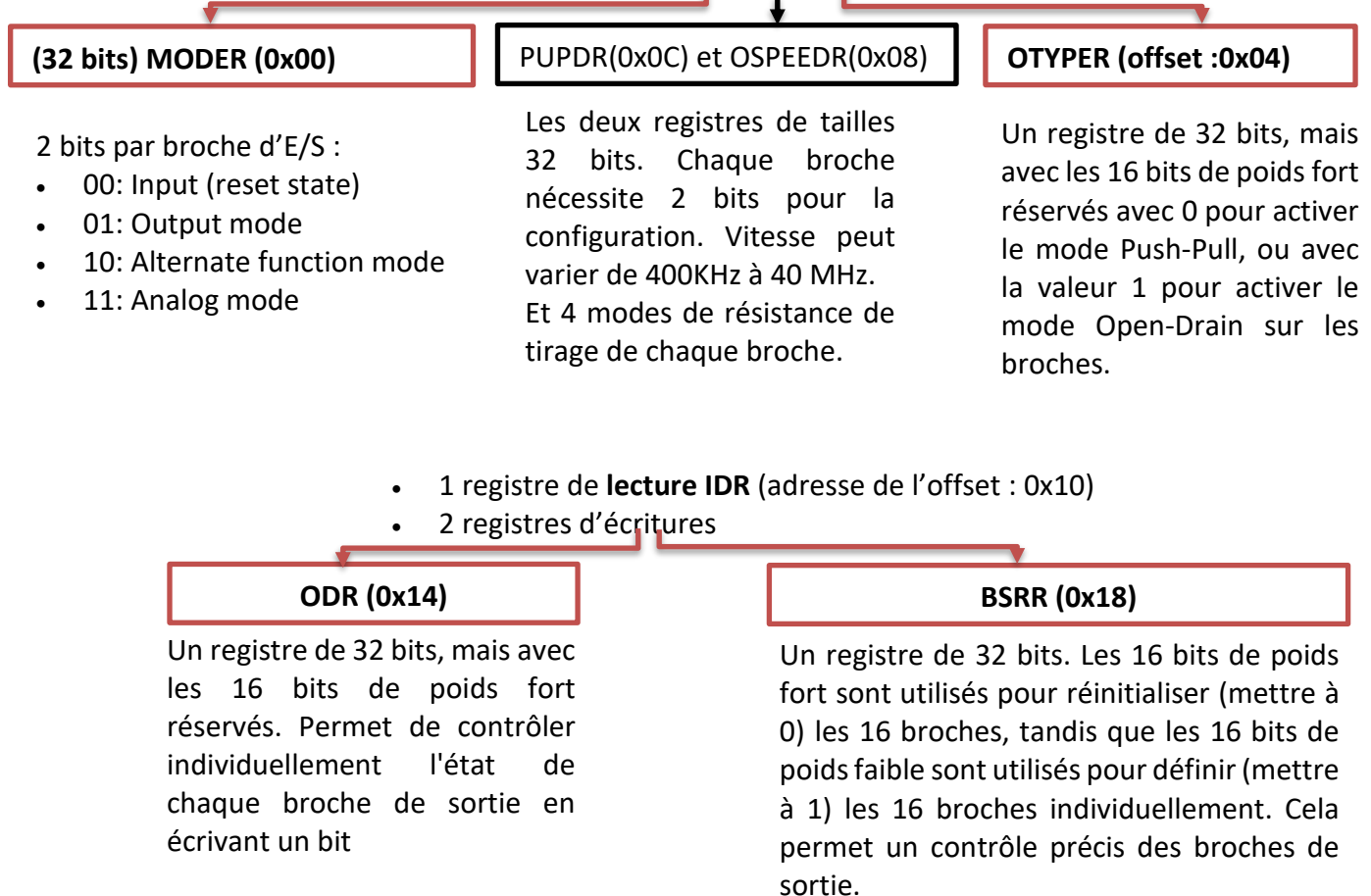
Conclusion :

Cela permet de confirmer que la fonction d'interruption est bien appelée à la fréquence attendue et que la base de temps fonctionne correctement.

TP 2 : Périphériques externes basiques

1 Travail préparatoire

- 8 ports GPIO (A à H)
- Chaque port a 16 broches d'entrée/sortie
- 4 registres de configuration clés.



↳ Pour configurer les broches 4, 5, 6 et 7 du port GPIOA en mode output :

```
GPIOA->MODER |= ( (0x1<<4*2) | (0x1<<5*2) | (0x1<<6*2) | (0x1<<7*2) ) ;
```

2 Driver du clavier matriciel

2.1 Portage du driver

- ↳ Il est préférable d'utiliser "[clavier_port_STM32.h](#)" car il est spécifiquement conçu pour la plate-forme STM32L152RE, optimisé pour son matériel et évite les conflits potentiels. En revanche, "[clavier_port.h](#)" peut être plus adapté à une gamme plus large de cibles matériels.
- ↳ Un aspect particulièrement intéressant concerne l'utilisation des masquages sur les registres, notamment avec les microcontrôleurs 32 bits, qui disposent de registres de grande taille. Dans ce contexte, les décalages jouent un rôle essentiel. L'opération $(x \ll y)$ signifie que vous décalez la valeur x vers la gauche de y positions, tandis que $(x \gg y)$ signifie que vous la décalez vers la droite de y positions.
- ↳ Ainsi "`GPIOA->MODER&=~(0x3<<(9*2)) ;`" /*Cela réinitialiserait les bits correspondant à la broche 9 dans le registre MODER, configurant **broche 9** en mode **input**.*/

2.2 Test du driver

Après l'ajout de l'initialisation, il a été observé que la variable "touche_etat_actuelle" récupérait correctement la valeur saisie au clavier qu'une fois la touche enfoncée. L'objectif est de scanner le clavier et de stocker la frappe d'une touche dans la variable "touche_mem" une fois relâchée. Pour ce faire on ajoute le code suivant :

main.c :

```
...
uint8_t touche_etat_actuelle = 0 ;
uint8_t touche_mem = 0 ;
uint8_t tmp = 0;
/* Initialisation des bibliothèques utilisées */

MESN_InitTimeBase();
clavier_init();
/* Infinite loop */
while (1) {

    touche_etat_actuelle = get_touche() ;

    if( touche_etat_actuelle != 0 ){
        tmp = touche_etat_actuelle ;
    }else if(touche_etat_actuelle == 0 ){
        touche_mem = tmp ;
        tmp = 0 ;
    }

    ...
}
```

Lorsque la touche est enfoncée (c'est-à-dire lorsque "touche_etat_actuelle" est différent de zéro), nous stockons cette touche dans une variable temporaire appelée "tmp". Ensuite, lorsque la touche est relâchée, nous pouvons prendre la valeur de la touche précédemment enfoncée (qui est stockée dans "tmp") et la sauvegarder dans la variable "touche_mem". Enfin, nous réinitialisons "tmp" à zéro pour le prochain appui de touche.

3 Driver de l'afficheur LCD

3.1 Portage du driver

LCD_port_STM32.h :

```
...
#include "timeBase.h" // provide delay routine
...
#define AFFICHEUR_RS_BAS GPIOC->ODR&=~(0x01<<1)
#define AFFICHEUR_POWER_BAS GPIOC->ODR&=~(0x01<<2)

#define AFFICHEUR_EN_HAUT GPIOC->ODR |= (0x01<<0)
#define AFFICHEUR_RS_HAUT GPIOC->ODR |= (0x01<<1)
#define AFFICHEUR_POWER_HAUT GPIOC->ODR |= (0x01<<2)
...
// AFFICHEUR OFF :
#define AFFICHEUR_OFF\
AFFICHEUR_DB|=0xF0;\
AFFICHEUR_RS_HAUT;\
AFFICHEUR_EN_HAUT;\
AFFICHEUR_POWER_BAS;\
GPIOC->MODER&=~((0xF<<0*2) | (0xF<<1*2) | (0xF<<2*2) | 0x0000FF00);\
```

3.2 Test du driver

main.c :

```
//Attention la durée init_LCD() est plus que 50 ms à cause du power on
init_LCD();
/* Infinite loop */
while (1){
...
    switch(touche_mem){
        case '3':
            clrscr_LCD();
            break;
        case '1':
            putC_LCD('3');
            break;
        case '2':
            putC_LCD('1');
            break;
        case '*':
            deinit_LCD();
            break;
        case '#':
            init_LCD();
            break;
    }
...
}
```

TP 3. Génération d'un signal PWM

1 Travail préparatoire

1.1 Modulation à largeur d'impulsion

Un signal PWM (modulation de largeur d'impulsion) est une forme d'onde périodique composée de cycles avec une durée d'impulsion variable. Le rapport cyclique représente le pourcentage de temps pendant lequel le signal est à l'état haut (actif) par rapport à la période totale du signal. Ainsi, en ajustant le rapport cyclique, on peut contrôler la tension moyenne de sortie, fournissant une tension continue variable.

1.2 Timer et PWM

Pour calculer les valeurs des registres **TIM10_ARR** et **TIM10_CCR1** pour générer un signal PWM de fréquence 20 kHz et de rapport cyclique 75% :

$$\text{On a} \quad F_{\text{overflow}} = \frac{F_{\text{tim_base}}}{[\text{PSC} + 1][\text{ARR} + 1]}$$

Donc $\text{TIM10_ARR} = 799$; $\text{PSC} = 0$

Et pour avoir le rapport cyclique %75 de 799 :

$$\text{TIM10_CCR1} = 599$$

Pour ce faire on change les configurations comme suit :

Counter Settings	
Prescaler (PSC - 16 bits value)	0
Counter Mode	Up
Counter Period (AutoReload Register - 16 bits value)	799
Internal Clock Division (CKD)	No Division
auto-reload preload	Disable
PWM Generation Channel 1	
Mode	PWM mode 1
Pulse (16 bits value)	599
Output compare preload	Enable
Fast Mode	Disable
CH Polarity	High

1.3 Assignation d'une broche physique

A partir de la description des broches du MCU, les broches sur lesquelles il est possible de faire sortir la voie OC1 du timer TIM10 (appelée TIM10_CH1) sont : PA6, PB8, PB12 et PE0.

Le numéro de l'alternate function à utiliser **AF3** (plus de détails documentation Figure 20 pages 173). Le e nom du registre concerné GPIO_AFRL[31 :0] (pour les pins de 0 à 7) qui correspond à notre cas.

2 Développement via l'IDE et Utilisation

En résumé, il faut inclure les bibliothèques nécessaires et effectuer les initialisations requises en ajoutant les lignes suivantes dans votre code :

```
19 /* Includes -----
20 #include "main.h"
21 #include "tim.h"
22 #include "gpio.h"
23 #include "stm3211xx_hal.h"
24
```

```
/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_TIM10_Init();
```

Ensuite, pour générer le signal PWM, utilisez la commande suivante dans « main.c » :

```
HAL_TIM_PWM_Start(&htim10, TIM_CHANNEL_1);
```

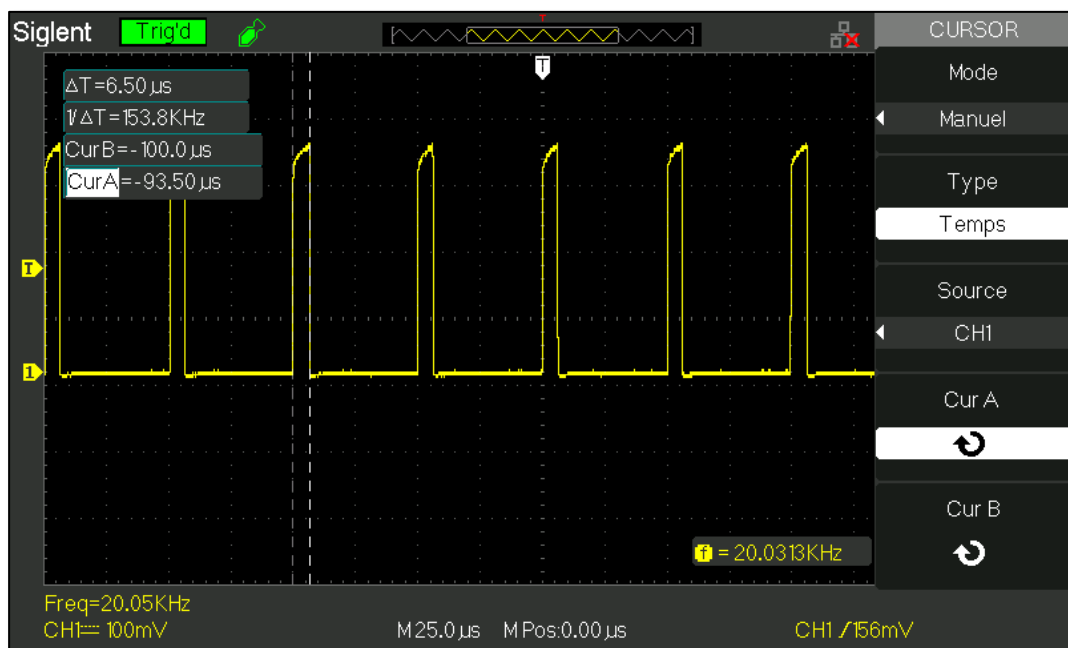
2.1 Utilisation

Nous cherchons à visualiser notre signal PWM sur un oscilloscope en variant les fréquences. Pour ajuster les fréquences, nous utilisons la fonction suivante :

```
HAL_TIM_SET_COMPARE(&htim10, TIM_CHANNEL_1, dutyCycle);
```

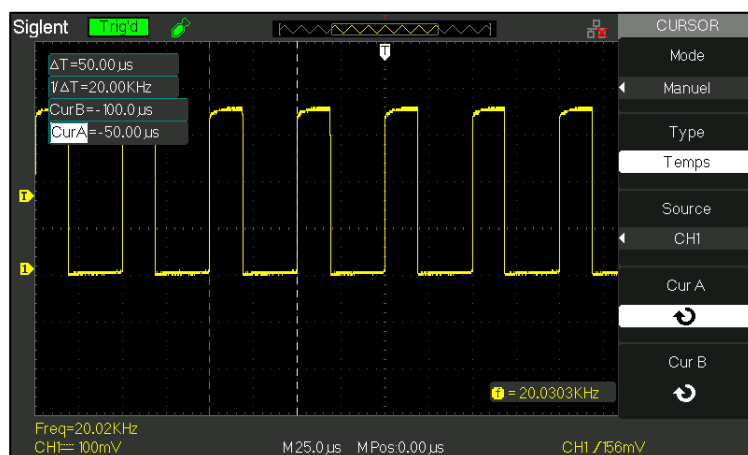
Cette instruction nous permet de changer le rapport cyclique du signal PWM, ce qui influence la fréquence observable sur l'oscilloscope.

Pour un dutyCycle de 100 :

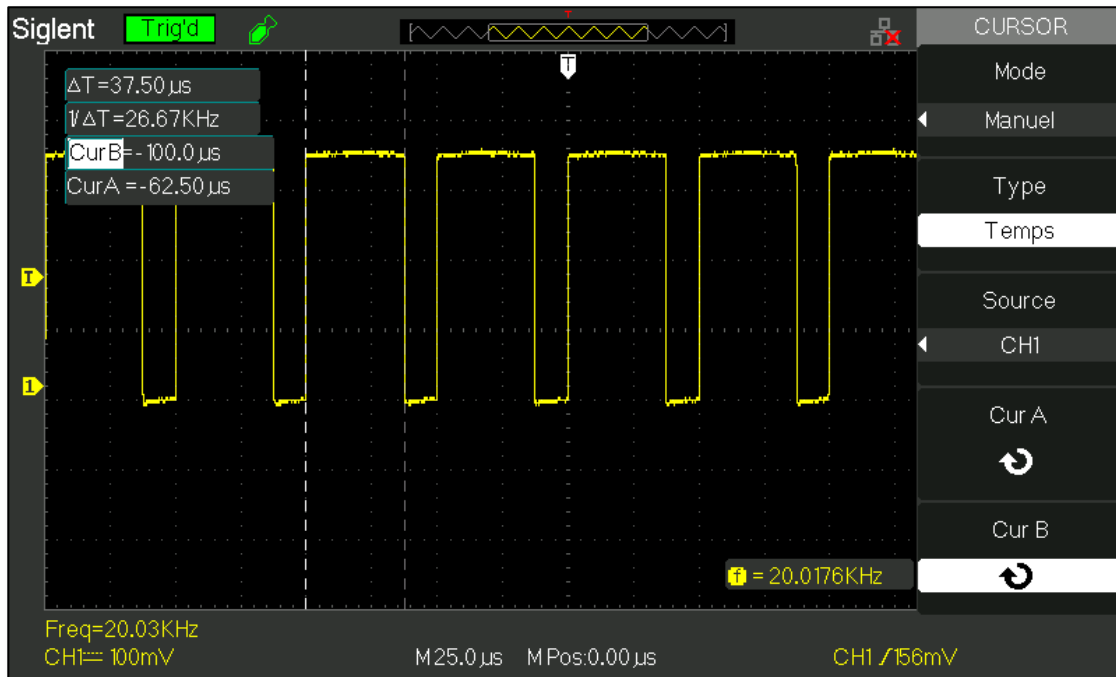


On aura comme tension continue est $[(6.5 \mu s) * (3.3 V) + 0V] / [50 \mu s]$ donc 0.429 V

Pour un dutyCycle de 300 :



On aura comme tension continue est $[(18.5 \mu s) * (3.3 V) + 0V] / [50 \mu s]$ donc 1.221 V
 Pour un dutyCycle de 600 :



On aura comme tension continue est $[(37.5 \mu s) * (3.3 V) + 0V] / [50 \mu s]$ donc 2.475 V

2.2 Conclusion

Le PWM offre un moyen efficace de contrôler la puissance fournie à un dispositif en variant la durée des impulsions électriques. Cela permet de réguler la vitesse des moteurs, la luminosité des LED, et de nombreux autres dispositifs, tout en minimisant la dissipation de chaleur et en conservant une efficacité énergétique élevée.

TP 4. Pilotage d'un Moteur à Courant Continu

1 Travail préparatoire

1.1 Modulation du rapport cyclique

`__HAL_TIM_SET_COMPARE()` de la bibliothèque HAL permet de changer le rapport cyclique d'un signal PWM pendant l'exécution sans réinitialisation. Les paramètres à passer sont la structure du timer **&htim10**, le canal du timer **TIM_CHANNEL_1**, et la nouvelle valeur du rapport cyclique (dutyCycle).

1.2 Étage de puissance


- ↪ Le **TB6612FNG** sert d'interface entre le microcontrôleur et le moteur, offrant des avantages en matière de protection, de contrôle et de gestion du courant. En connectant directement le moteur au signal PWM issu du MCU, il y aurait des risques de dommages au MCU en raison des pics de courant générés par le moteur. Le **TB6612FNG** agit comme une barrière robuste, protégeant le MCU contre de tels incidents tout en fournissant un contrôle bidirectionnel du moteur. De plus, le circuit intégré intègre des fonctionnalités telles que la régulation du courant, assurant un contrôle précis de la puissance envoyée au moteur et contribuant ainsi à une gestion efficace du courant, évitant les surtensions et les court-circuits. En résumé, l'utilisation du **TB6612FNG** optimise la performance et la sécurité du système en fournissant une interface spécialisée entre le MCU et le moteur.
- ↪ Toutes ces connaissances ont été acquises par l'expérience lors du projet de la première année, où nous travaillions avec un moteur à courant continu pour construire une poubelle intelligente. Au départ, nous avons tenté de connecter directement le moteur au signal PWM issu de la carte Intel 8051, mais cette approche a failli endommager la carte.
- ↪ La fréquence maximum du signal PWM admissible par ce composant est 100 KHz. (Référence : <https://www.otronic.nl/fr/module-pilote-moteur-tb6612fng-pour-arduino.html>)
- ↪ Le rôle des 2 entrées In1 et In2 est pour contrôler la direction de rotation du moteur.
- ↪ La structure à 4 transistors en électronique est appelée un "pont en H" (H-bridge). La configuration du pont en H permet de faire circuler le courant dans les deux sens à travers le moteur en utilisant des combinaisons de transistors activés et désactivés. Cela offre un contrôle complet de la direction du moteur, permettant au moteur de tourner dans les deux sens et même de freiner en coupant le courant.

2 Développement de la bibliothèque MotorDriver

2.1 Couche pilote (MotorDriver.c et MotorDriver.h)


- 📖 Le qualificateur **static** confère une portée locale à la fonction avec une durée de vie globale, limitant son accès au fichier source.
- 📖 La valeur du paramètre passé en argument dans la fonction **MotorDriver_SetDir()** est une conversion du paramètre **rotSpeed**. Le calcul ajuste la valeur de **rotSpeed** pour qu'elle soit

proportionnelle à l'échelle du rapport cyclique PWM en fonction de la vitesse maximale du moteur.


 Ajout du code dans la fonction MotorDriver_SetDir() :

```
122 static void MotorDriver_SetDir(RotDir_TypeDef rotDir){
123     /* Always set pin low first before inverting direction to ensure
124        a "brake to ground" state during transition */
125     if (rotDir == CW){
126         //MotorDriver_Port_SetPin_IN2(GPIO_DIRPIN_LOW);
127         MotorDriver_Port_SetPin_IN1(GPIO_DIRPIN_LOW); // ligne ajouter
128         MotorDriver_Port_SetPin_IN1(GPIO_DIRPIN_HIGH);
129     }
130     else if (rotDir == CCW){
131 //TODO A completer : .....
132         //MotorDriver_Port_SetPin_IN2(GPIO_DIRPIN_LOW);
133         MotorDriver_Port_SetPin_IN1(GPIO_DIRPIN_LOW);
134         MotorDriver_Port_SetPin_IN2(GPIO_DIRPIN_HIGH);
135     }
136 }
137 else if (rotDir == STOP){
138 //TODO A completer : .....
139     MotorDriver_Port_SetPin_IN2(GPIO_DIRPIN_LOW);
140     MotorDriver_Port_SetPin_IN1(GPIO_DIRPIN_LOW);
141 }
142 }
143 }
```

2.2 Couche portabilité (MotorDriver_port.c et MotorDriver_port.h)


 En résumé, il faut inclure les bibliothèques nécessaires et effectuer les initialisations requises en ajoutant les lignes suivantes dans votre code :

```
18 //TODO : A completer #include "..." //Permet d'utiliser la lib HAL.
19 #include "stm321xx_hal_tim.h"
20 /* External Variables -----*/
21 //TODO a completer : ... //Peripheral handle for Timer10 (usually htimXX)
22 extern TIM_HandleTypeDef htim10;
```

 Ajout du code dans la fonction MotorDriver_Port_PWM_Init() afin de démarrer la génération du signal PWM en utilisant la fonction HAL dédiée vu au TP précédent :

```
//TODO : A completer .... //Demarre la PWM (fonction HAL, sans interrupt)
HAL_TIM_PWM_Start(&htim10,TIM_CHANNEL_1);

}
```

 Ajout du code dans les fonctions MotorDriver_Port_SetPin_IN1() et MotorDriver_Port_SetPin_IN2() pour contrôler les rotations grâce à ses fonctions après :

```
24 void MotorDriver_Port_GPIO_Init(void) {
25
26     /*Peripheral initialization functions are generated by CubeMX software and
34     //Pull the two dirPin low (motor in STOP configuration)
35     MotorDriver_Port_SetPin_IN1(GPIO_DIRPIN_LOW);
36     MotorDriver_Port_SetPin_IN2(GPIO_DIRPIN_LOW);
37
38 }
```


📖 En utilisant la macro `__HAL_TIM_SET_COMPARE()`, complétez le code de la fonction `MotorDriver_Port_SetPWM()` :

```
void MotorDriver_Port_SetPWM(uint32_t dutyCycle){  
    if(dutyCycle <= PWM_DUTYCYCLE_FULL_SCALE){  
//TODO a completer ... #define __HAL_TIM_SET_COMPARE(__HANDLE__, __CHANNEL__, __COMPARE__)  
    __HAL_TIM_SET_COMPARE(&htim10,TIM_CHANNEL_1,dutyCycle);  
    }  
}
```

2.3 Test de la bibliothèque MotorDriver

↪ Essayons de tester notre moteur pour ce faire :

```
20 #include "stdio.h"  
21  
22 #include "main.h"  
23 #include "tim.h"  
24 #include "gpio.h"  
25  
26 #include <stdint.h>  
27 #include "stm3211xx.h"  
28  
29 //Bibliothèques additionnelles  
30 #include "timeBase.h"  
31 #include "clavier.h"  
32 #include "LCD.h"  
33 #include "MotorDriver.h"  
34
```

```
43 int main(void)  
44 { //Ini des variables :  
45     static uint8_t touche_etat_actuelle = 0 ;  
46     static uint8_t touche_mem = 0 ;  
47     volatile uint8_t tmp = 0;  
48     static uint8_t i = 0;  
49     char tab[5];  
50  
51     int32_t vitesse = 0 ; // varie de -32000 à 32000  
52     int32_t convertisseur = 0;  
53     static unsigned char flag_commander_a_mem = 0 ;  
54  
55     /* Initialize biblio */  
56     HAL_Init();  
57     SystemClock_Config();  
58     /* Initialisation des bibliothèques utilisées */  
59     MX_GPIO_Init();  
60     MX_TIM10_Init();
```

```

61
62     MESN_InitTimeBase();
63     clavier_init();
64     init_LCD();
65     MotorDriver_Init();
66
67
68     HAL_TIM_PWM_Start(&tim10,TIM_CHANNEL_1);

69     while (1){
70         /* meme si la fonction MotorDriver_Move() prend en argument int32_t
71          * la conversion de short à int_32t est implicite dans ce cas */
72
73         MotorDriver_Move(vitesse);
74
75         touche_mem = touche_etat_actuelle;
76         touche_etat_actuelle = get_touche() ;
77         if((touche_etat_actuelle == 0 )&&(touche_mem!=0)){
78             if((touche_mem == '*')||(touche_mem == '#')){
79                 flag_commander_a_mem = 1;
80             }

93         if((flag_commander_a_mem == 1)&&(i<4)){
94             sprintf(tab+i,"%c",touche_mem);
95             i++;
96             putC_LCD(touche_mem);
97         }
98         else if(i > 3){
99             i = 0;
100             flag_commander_a_mem = 0;
101             clrscr_LCD();
102             sscanf(tab+1,"%d",&convertisseur);
103
104             switch (tab[0]){
105                 case '*':
106                     vitesse = (short)-1 * convertisseur;
107                     break;
108                 case '#':
109                     vitesse = convertisseur;
110                     break;
111             }
112         }
113     }
114     MESN_Delay(1000);
115 }
116 }

```

2.4 Conclusion

Le programme présent effectue avec succès le contrôle de la vitesse d'un moteur en fonction des saisies effectuées au clavier matriciel. Tout d'abord, les bibliothèques nécessaires sont correctement initialisées, notamment celles liées au temps, au clavier, à l'écran LCD et au pilote du moteur. La boucle principale du programme s'occupe de réguler la vitesse du moteur en fonction des touches pressées au clavier. Les saisies sont formatées avec un signe (* pour positif, # pour négatif) suivi d'un chiffre formant un nombre à trois chiffres (de 000 à 999). Une fois la saisie complète, la vitesse est convertie en utilisant `sscanf()` et est ensuite appliquée au moteur à l'aide de la fonction `MotorDriver_Move()`. Le programme assure également la gestion des interruptions pour détecter le relâchement des touches

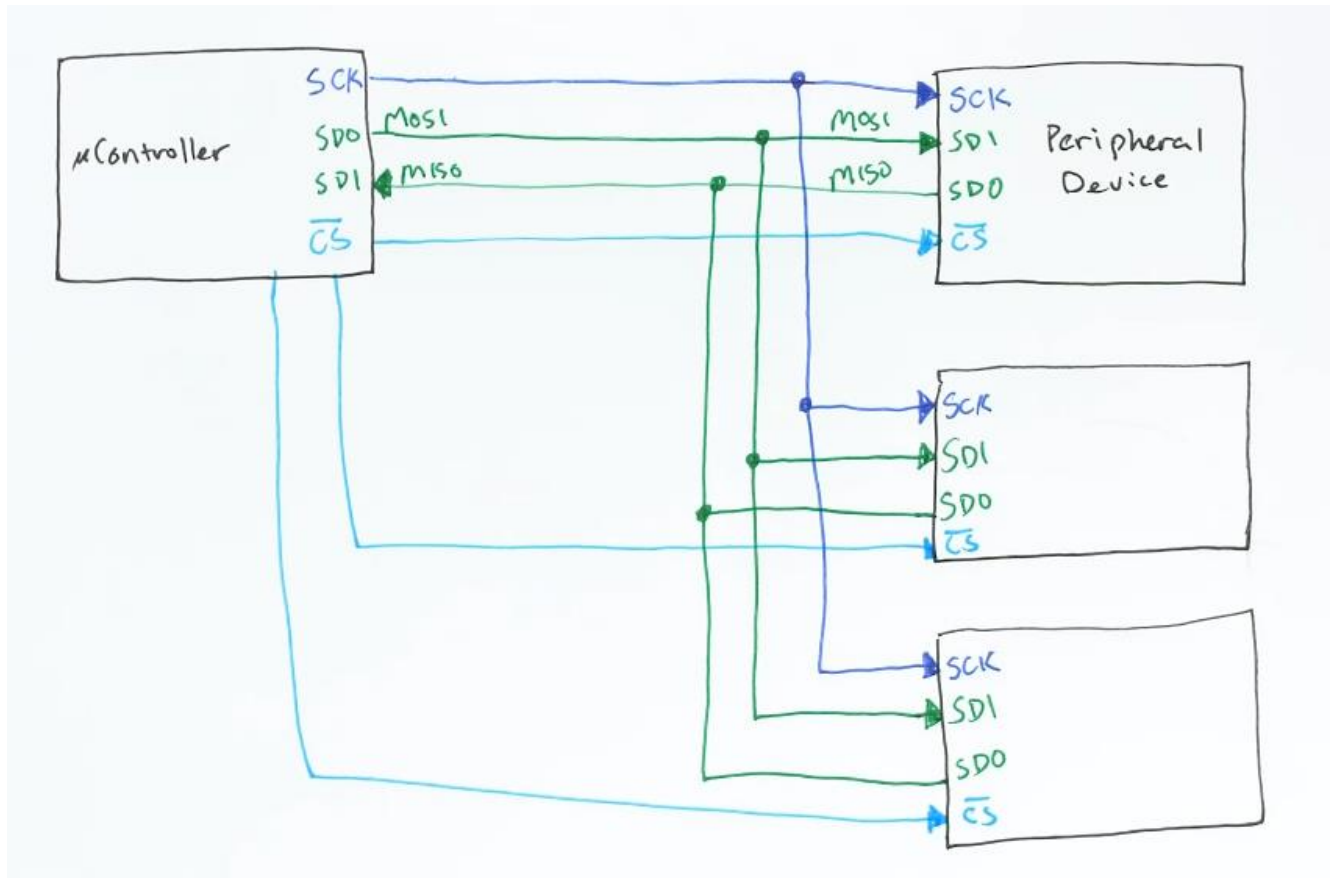
au clavier et met à jour l'affichage sur l'écran LCD en conséquence. En résumé, l'ensemble des fonctionnalités, y compris l'interaction avec le clavier, l'affichage sur l'écran LCD et le contrôle du moteur, fonctionnent de manière harmonieuse.

TP 5. Communication SPI et carte SD

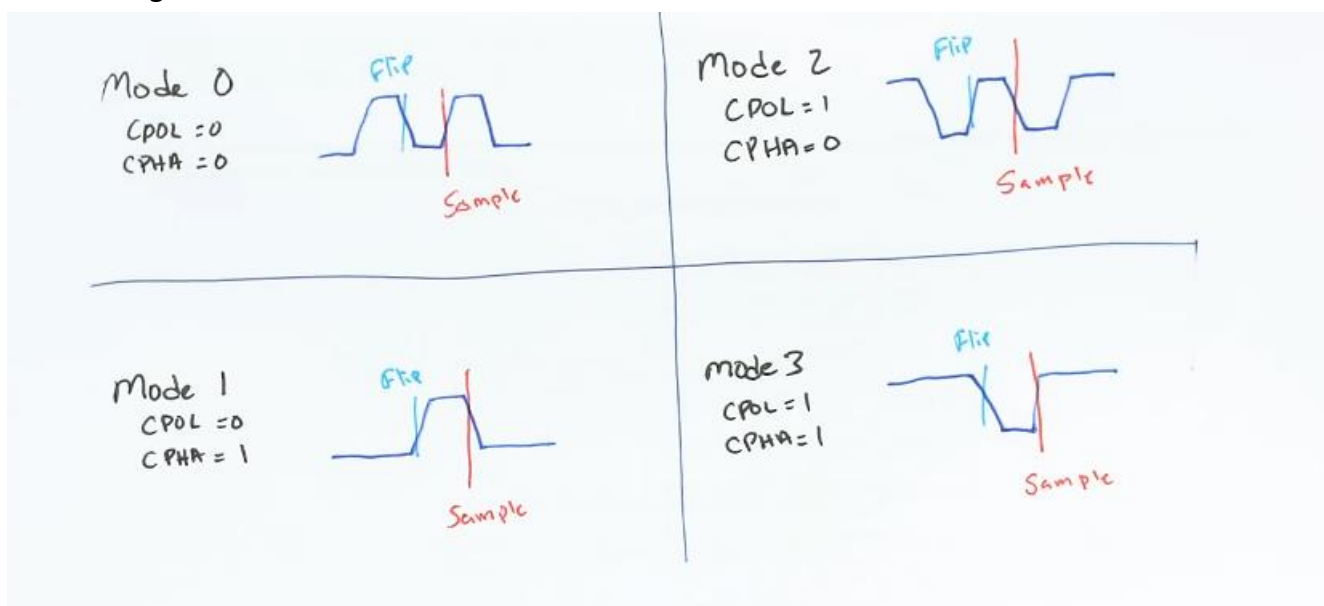
1 Travail préparatoire

1.1 Liaison SPI

↳ Résumé de ce que j'ai compris :



CPOL et CPHA définissent la synchronisation entre le maître et l'esclave en spécifiant le niveau du signal d'horloge inactif et le moment où les données sont échantillonnées par rapport à ce signal d'horloge.



- ↪ Le bit n°7 LSBFIRST dans SPI_CR1 détermine l'ordre de transfert des bits. LSBFIRST à 0 transmet les bits MSB en premier (standard), et LSBFIRST à 1 transmet les bits LSB en premier.

Les IRQ du périphérique SPI peuvent être déclenchées par :

- ↪ Réception de données complète (RX FIFO plein).
- ↪ Transmission de données complète (TX FIFO vide).
- ↪ Erreur de transmission ou réception détectée.
- ↪ Fin de transfert signalée.
- ↪ Conditions spécifiques programmées.
- ↪ Détection de front montant ou descendant.
- ↪ Timeouts dépassés.

1.2 Carte mémoire Secure Digital (SD)

- ↪ Les échanges de données avec la carte SD (lecture/écriture dans la mémoire) se font via les commandes CMD17/CMD18 pour la lecture et CMD24/CMD25 pour l'écriture. Ces commandes impliquent ensuite la transmission d'une trame de données (appelée data packet), illustrée à la figure suivante :

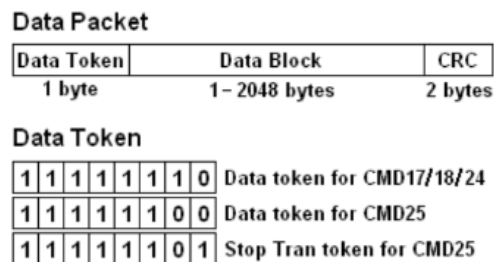


Figure :— Constitution d'une trame de données.

Pour les commandes SD 17, 18, 24 et 25 :

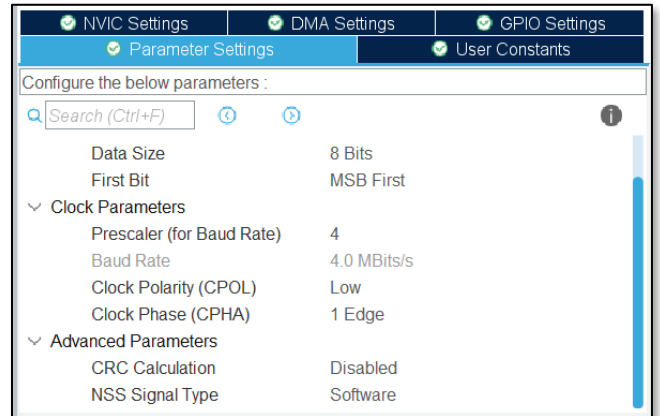
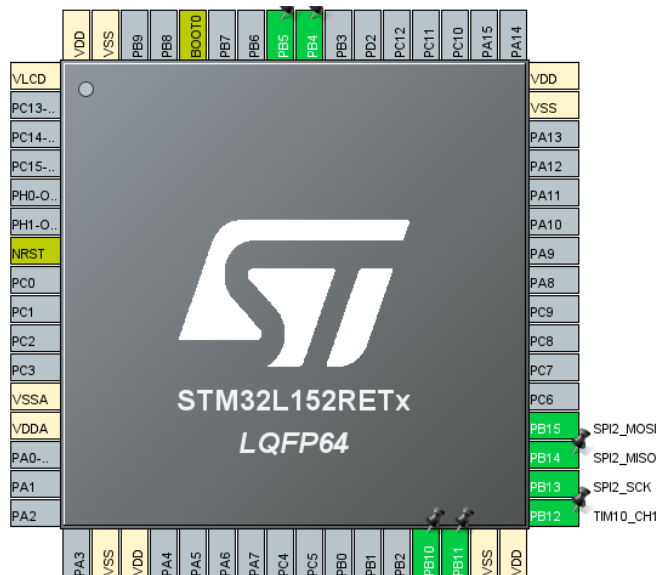
- ↪ Commande 17 (READ_SINGLE_BLOCK) : Argument = Adresse du bloc à lire.
- ↪ Commande 18 (READ_MULTIPLE_BLOCK) : Argument = Adresse du bloc à lire.
- ↪ Commande 24 (WRITE_BLOCK) : Argument = Adresse du bloc où écrire.
- ↪ Commande 25 (WRITE_MULTIPLE_BLOCK) : Argument = Adresse du premier bloc où écrire.

- ↪ CD (Card Detect) et CS (Chip Select) doivent être configurées comme suit :

CD en mode entrée (input) pour détecter la présence de la carte, et CS en mode sortie (output) pour la communication avec la carte SD.

2 Liaison SPI :

Configuration du projet comme suit :



Pin Name	Signal on Pin	GPIO output...	GPIO mode	GPIO Pull-u...	Maximum o...	User Label	Modified
PB4	n/a	n/a	Input mode	Pull-up	n/a		<input checked="" type="checkbox"/>
PB5	n/a	Low	Output Push...	No pull-up a...	High		<input checked="" type="checkbox"/>
PB10	n/a	Low	Output Push...	No pull-up a...	Very Low		<input type="checkbox"/>
PB11	n/a	Low	Output Push...	No pull-up a...	Very Low		<input type="checkbox"/>

2.1 Vérification du fonctionnement

L'envoi, par polling, de données via le périphérique SPI dans « main.c » on effectue les modifications suivante :

```

72     uint8_t touche_etat_actuelle = 0 ;
73     uint8_t touche_mem = 0 ;
74     uint8_t tmp = 0;
75
76     uint32_t t = 10;
77     uint8_t tab[4] = {0x55,0xFF,0x00,0x2A};
78     SD_CardInfo carte_du_tp;
79     /* USER CODE END 1 */
80     MX_SPI2_Init();

```

.....

```

122         switch(touche_mem) {
123
124             case '1':
125                 HAL_SPI_Transmit(&hspi2, tab ,sizeof(tab), 10);
126                 break;
127
128         }

```

Le prototype de la fonction HAL_SPI_Transmit() comme suit :

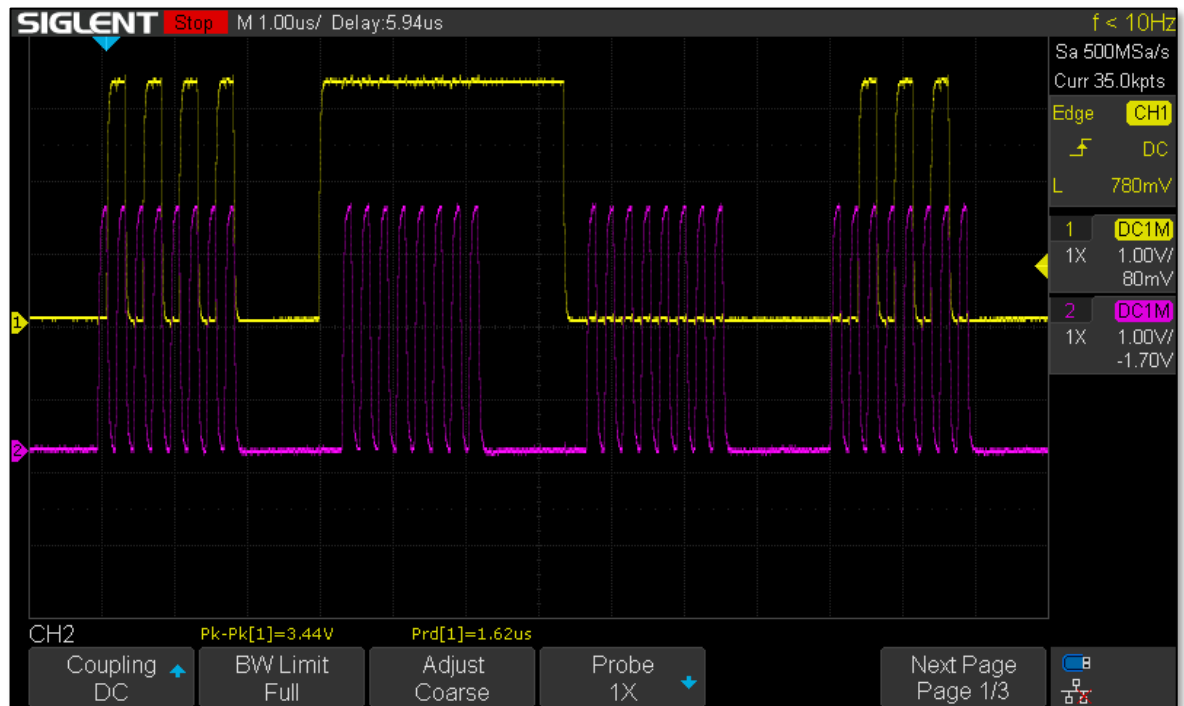
```

HAL_StatusTypeDef HAL_SPI_Transmit(SPI_HandleTypeDef *hspi, uint8_t *pData,
uint16_t Size, uint32_t Timeout)

```

On peut sélectionner un timeout maximal, et la transmission sera automatiquement considérée comme terminée une fois celle-ci accomplie. Alternativement, choisir un timeout suffisant pour la transmission du tableau est possible.

📖 A l'aide de l'oscilloscope, observons les signaux présents sur la broche PB.15 pour la MOSI et la PB.13 pour SCLK lors d'une émission :



3 Carte SD

3.1 Adaptation de la couche basse SD_IO

📖 Le périphérique SPI utilisé SPI2 et on configure les broches CD et CS. Pour ce faire on ajoute le code suivant :

```

22 //TODO : sélectionner le périphérique SPI
23 // #define USE_SPI1 //Use SPI1 peripheral on PA5-6-7 pins
24 #define USE_SPI2 //Use SPI2 peripheral on PB13-14-15 pins
25
26 #define SD_SPI_TIMEOUT_MAX 1000
27
28 /* Pins mapping for CD and CS */
29 //TODO : affectez l'assignation des broches GPIO DC et CS
30 #define SD_CS_PORT GPIOB //CS est la broche PB5
31 #define SD_CS_PIN GPIO_PIN_5 //...
32 #define SD_CD_PORT GPIOB //CD EST LA BROCHE PB4...
33 #define SD_CD_PIN GPIO_PIN_4 //...

```

📖 Le mot-clé `inline` dit au compilateur de copier directement le code de la fonction à l'endroit où elle est appelée, plutôt que de créer un appel de fonction. C'est comme si le contenu de la fonction était copié et collé à chaque endroit où la fonction est utilisée. Cela peut rendre le programme plus rapide car il évite le coût supplémentaire d'un appel de fonction.

↪ Le code des fonctions `SD_CS_LOW()`, `SD_CS_HIGH()` et `SD_CD_State()` permettant la gestion des broches GPIO CD et CS :

```

186 static inline void SD_CS_LOW(void) {
187     //TODO : a completer (en utilisant la fonction STM32-HAL adequate)
188     HAL_GPIO_WritePin(SD_CS_PORT, SD_CS_PIN, GPIO_PIN_RESET);
189 }
190
191
192 static inline void SD_CS_HIGH(void) {
193     //TODO : a completer (en utilisant la fonction STM32-HAL adequate)
194     HAL_GPIO_WritePin(SD_CS_PORT, SD_CS_PIN, GPIO_PIN_SET);
195 }
196
197 static inline GPIO_PinState SD_CD_State(void) {
198     //TODO : a completer (en utilisant la fonction STM32-HAL adequate)
199     return HAL_GPIO_ReadPin(SD_CD_PORT, SD_CD_PIN);
200 }

```

↳ Le code des fonctions SD_SPI_WriteData() et SD_SPI_WriteReadData() permettant l'échange de données sur le bus SPI.

```

223 static void SD_SPI_WriteData(uint8_t *DataIn, uint16_t DataLength)
224 {
225     HAL_StatusTypeDef status = HAL_OK;
226
227     //TODO : a completer (en utilisant la fonction STM32-HAL adequate).
228     status = HAL_SPI_Transmit(&hspi2, DataIn, DataLength, SD_SPI_TIMEOUT_MAX);
229
230     /* Check the communication status */
231     if(status != HAL_OK)
232     {
233         /* Execute user timeout callback */
234         SD_SPI_Error();
235     }
236 }

```

```

245 static void SD_SPI_WriteReadData(const uint8_t *DataIn, uint8_t *DataOut, uint16_t DataLength){
246     //TODO : a completer (en utilisant la fonction HAL_SPI_TransmitReceive() ).
247     HAL_SPI_TransmitReceive(&hspi2, DataIn, DataOut, DataLength, SD_SPI_TIMEOUT_MAX);
248 }

```

4 Test du fonctionnement

↳ Test le fonctionnement du driver en appelant la fonction SD_GetCardInfo() depuis le programme principal comme suit :

↳ SD_GetCardInfo(&carte_du_tp);

Sachant que : carte_du_tp déclarer comme suit :

SD_CardInfo carte_du_tp;

```

120 typedef struct
121 {
122     SD_CSD Csd;
123     SD_CID Cid;
124     uint32_t CardCapacity; /* Card Capacity */
125     uint32_t CardBlockSize; /* Card Block Size */
126 } SD_CardInfo;

```

On obtient on mode debug :

▼ carte_du_tp	SD_CardInfo	{...}
▼ Csd	SD_CSD	{...}
(x) CSDStruct	uint8_t	1 '\001'
(x) Reserved1	uint8_t	0 '\0'
(x) TAAC	uint8_t	14 '\016'
(x) NSAC	uint8_t	0 '\0'
(x) MaxBusClkFrec	uint8_t	50 '2'
(x) CardComdClasses	uint16_t	1461
(x) RdBlockLen	uint8_t	9 '\t'
(x) PartBlockRead	uint8_t	0 '\0'
(x) WrBlockMisalign	uint8_t	0 '\0'
(x) RdBlockMisalign	uint8_t	0 '\0'
(x) DSRImpl	uint8_t	0 '\0'
> version	union csd_version	{...}
(x) EraseSingleBlockEnable	uint8_t	1 '\001'
(x) EraseSectorSize	uint8_t	127 '\177'
(x) WrProtectGrSize	uint8_t	0 '\0'
(x) WrProtectGrEnable	uint8_t	0 '\0'
(x) Reserved2	uint8_t	0 '\0'
(x) WrSpeedFact	uint8_t	2 '\002'
(x) MaxWrBlockLen	uint8_t	9 '\t'
(x) WriteBlockPartial	uint8_t	0 '\0'
(x) Reserved3	uint8_t	0 '\0'
(x) FileFormatGroup	uint8_t	0 '\0'
(x) CopyFlag	uint8_t	0 '\0'
(x) PermWrProtect	uint8_t	0 '\0'
(x) TempWrProtect	uint8_t	0 '\0'
(x) FileFormat	uint8_t	0 '\0'
(x) Reserved4	uint8_t	0 '\0'
(x) crc	uint8_t	40 '('
(x) Reserved5	uint8_t	1 '\001'
> Cid	SD_CID	{...}
(x) CardCapacity	uint32_t	7683072
(x) CardBlockSize	uint32_t	512

Et pour la carte ID :

▼ Cid	SD_CID	{...}
(x) ManufacturerID	volatile uint8_t	156 '\234'
(x) OEM_ApplID	volatile uint16_t	21327
(x) ProdName1	volatile uint32_t	1431520304
(x) ProdName2	volatile uint8_t	48 '0'
(x) ProdRev	volatile uint8_t	2 '\002'
(x) ProdSN	volatile uint32_t	689967973
(x) Reserved1	volatile uint8_t	0 '\0'
(x) ManufactDate	volatile uint16_t	292
(x) CID_CRC	volatile uint8_t	82 'R'
(x) Reserved2	volatile uint8_t	1 '\001'
(x) CardCapacity	uint32_t	7683072
(x) CardBlockSize	uint32_t	512

↪ À l'aide de l'oscilloscope, la trame SPI correspondante à l'envoi de la première commande émise par la fonction SD_GetCardInfo() est :

