

projet document

Louison GOUY Yiying WEI

15 janvier 2022

name

abstract

"It's not you can use C to generate good code for hardware. If you think like a computer writing C actually makes sense."

Linus Torvalds

Table des Matières

1	Glossaire	5
2	Le langage C	6
3	Programme de base	7
3.1	Vue globale	7
3.2	Affinage	7
4	Programmation d'un timer	9
4.1	Fonctionnement d'un timer	9
4.2	Configuration TC6	11
4.2.1	Configuration du GCLK	11
4.2.2	Configuration du PM	13
4.2.3	Configuration du PORT	13
4.2.4	Configuration du TC6	13
5	Fonction sous interruption	14
5.1	Présentation du problème	14
5.2	Fonctionnement d'une interruption	15
5.3	Configuration interruption	16
5.3.1	Timer TC6	16
5.3.2	NVIC	16
5.3.3	Routine	17
6	Implémentation fonction calcul gabarit	19
6.1	Conception fonctionnelle	19
6.2	Implémentation logicielle	20
6.3	Test calcul gabarit et configuration DAC	22

Liste des figures

1	Mode de fonctionnement en WAVEFORM pour les compteurs	9
2	Fonctionnement timer configuré en MFRQ	10
3	Configuration du compteur TC	10
4	Fonctionnement du Generic Clock Controller	11
5	Configuration des différents bits du GENCTRL	12
6	Configuration des différents bits du CLKCTRL	13
7	Ordonnancement tâches avec polling	14
8	Schéma bloc NVIC	15
9	Registre INTENSET	16
10	Registre NVIC_IUSER ARMv6-M	17
11	Oscillogramme vérification procédure interruption	18
12	Description fonctionnelle calcul gabarit (FSM)	19
13	Description diagramme bloc DAC	22
14	Description registre DAC CTRLA	22
15	Description registre DAC CTRLB	23
16	Description registre DAC CTRLB	23

1 Glossaire

Requête d'interruption : (IRQ : interrupt request) Signal matériel indiquant qu'une interruption est requise.

Polling : approche d'ordonnancement dans laquelle le logiciel répète un test sur une condition pour déterminer s'il doit exécuter une tâche. [1]

Une **machine état ou automate fini** est une construction mathématique abstraite, susceptible d'être dans un nombre fini d'*états* , mais étant un moment donné dans un seul état à la fois. Le passage d'un état à un autre se fait par une *transition*.

2 Le langage C

Le langage C est un langage combiné, il a les caractéristiques des langages évolués (boucles itératives etc.) associé à des fonctionnalités des langages assemblés (décalage de bit, adressage indirect généralisé etc.). C'est la combinaison de ces deux caractéristiques qui font la force du langage [2]. Sa proximité avec l'assembleur le rendant très efficace, il est ainsi devenu le langage indispensable dans la programmation des applications comme l'automatique, la robotique, les OS ect. Cette même proximité impose peu de contraintes à l'utilisateur sur la structure de son programme. Aussi, il est possible d'écrire des fonctions avec plusieurs points de sorties, ou encore, d'échapper à une boucle avant son terme. Là où certains trouveront une grande souplesse, les critiques le considéreront trop permissif. On notera qu'un certain nombre d'organismes officiels proposent un ensemble de règles visant, tout en conservant son efficacité, à éviter les problèmes liés à une programmation peu soignée. L'Agence Nationale de la Sécurité des Systèmes Informatiques française (ANSSI) propose un rapport complet, *Règles de programmation pour le développement sécurisé de logiciels en langage C* [3], visant à "favoriser la production de logiciels C plus sécurisés, plus sûrs, d'une plus grande robustesse et portables". Il servira de référence durant ce projet.

3 Programme de base

Dans un premier temps un nouveau projet est créé de type "GCC C ASF Board project". Microchip studio génère alors une arborescence de fichiers dont un `main.c`. Ce dernier est étudié de manière globale puis affinée par étape dans la section suivante.

3.1 Vue globale

Cette partie détaille le fonctionnement du programme de base.

```
1 #include <asf.h>
2
3 int main (void)
4 {
5     system_init();
6
7     /* Insert application code here, after the board has been initialized. */
8
9     /* This skeleton code simply sets the LED to the state of the button. */
10    while (1) {
11        /* Is button pressed? */
12        if (port_pin_get_input_level(BUTTON_0_PIN) == BUTTON_0_ACTIVE) {
13            /* Yes, so turn LED on. */
14            port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);
15        } else {
16            /* No, so turn LED off. */
17            port_pin_set_output_level(LED_0_PIN, !LED_0_ACTIVE);
18        }
19    }
```

La première ligne permet d'inclure la bibliothèque `asf` et ainsi de profiter du niveau d'abstraction mis à disposition par Microship. La suivante, `int main ()` bien connue des développeurs C, est le point d'entrée du programme. C'est la première fonction exécutée. La ligne 5 `system_init()`; a été générée automatiquement par le logiciel à la création du projet. C'est elle qui nous offre ce niveau d'abstraction en initialisant les horloges et les entrées/sorties etc. Elle est spécifique à la cible utilisée, dans notre cas la carte Microchip SAMD21 Xplained Pro. La ligne 10 correspond à l'implémentation d'une boucle infinie. Cette dernière permet de lire l'état du bouton (ligne 12) en "continu". La condition suivante d'éclanche le résultat souhaité : allumage ou extinction de la LED0.

3.2 Affinage

Include ASF

L'Advanced Software Framework (ASF) fournit un riche ensemble de pilotes éprouvés et de modules de code développés par des experts pour réduire le temps de conception. Il simplifie l'utilisation des microcontrôleurs en fournissant une abstraction au matériel. ASF est une bibliothèque de code gratuite et open-source conçue pour être utilisée lors des phases d'évaluation, de prototypage, de conception et de production. Elle sera utilisée tout au long de ce TP et fera l'objet de nombreuses références.

System_init

Au début du `main` la fonction `system_init()` est appelée. Comme son nom l'indique elle a pour but d'initialiser le système. Elle est définie dans le fichier `system.c` et consiste en un simple appel successif à cinq fonctions de configuration : `system_clock_init()`; `system_board_init()`;

`_system_events_init(); _system_extint_init();` et `_system_divas_init();`. Elles jouent chacune un rôle essentiel dans l'initialisation de carte.

Boucle infinie

Implémenté à travers un `tant que VRAI`, cette ligne n'est pas difficile à comprendre mais il peut être intéressant d'en établir le contexte. Le guide des bonnes de pratiques de l'ANSSI [3] indique toutefois que la forme d'une boucle infinie est bien `while(1)` et non `for(;;)`

De manière générale, le bouclage répète un jeu d'instruction jusqu'à se qu'une condition particulière soit atteinte. On définit une boucle infinie dès lors que cette condition n'arrive jamais en raison d'une caractéristique inhérente à la boucle. Dans notre cas la condition de sortie serait `VRAI=FAUX`. C'est impossible!

Du point de vue matériel l'utilisation d'une boucle infinie permet de borner le programme compteur (PC) dans un espace mémoire bien défini. Le compilateur devrait l'interpréter par un `jump` ou `jmp`. Le mieux est probablement de le vérifier. Un fichier `loop.c` est créé, volontairement le plus simple possible.

```
1  /* file loop.c */
2  void main(void){ while(1); }
3
```

Puis la commande `gcc -S -fverbose-asm loop.c` est exécutée dans un terminal linux. Un fichier `loop.s` apparaît. L'option `-S` indique la génération du code assembleur et `-fverbose-asm` ajoute des commentaires tel que la ligne C correspondant à l'instruction. On extrait du résultat la partie qui nous intéresse :

```
    ; file loop.s
.L2:
# loop.c:2:      while (1);
jmp     .L2      #
```

Le compilateur gcc a bien implémenté la boucle infinie via une instruction `jump` indiquant un saut du PC. Dans cette exemple, la boucle étant vide, le PC saute au même endroit. Il est intéressant de faire le parallèle avec l'assembleur. Cet exemple reste toutefois approximatif puisque ce n'est pas le jeux d'instruction du CORTEXM0+ qui a été utilisé. Prenons le comme une introduction.

Condition sur E/S

Les lignes suivantes implémentées via une structure `if else` traduisent le comportement souhaité du point de vue utilisateur. A savoir, le maintien en position enfoncé du bouton provoque l'illumination de la LED0. La lecture de son état est permis grâce à la fonction `port_pin_get_input_level` retournant un entier de valeur `XX` ou `XX`. Elle est alors comparé à `LED_0_ACTIVE` défini comme `XX`. Si la condition est vrai la fonction `port_pin_set_output_level` est appelé avec comme paramètre `LED_0_ACTIVE` sinon `!LED_0_ACTIVE`.

4 Programmation d'un timer

Cette étape vise à générer un signal carré de période 1ms sur une des sorties timer du microcontrôleur. Il s'agit donc de préparer l'implantation de la fonction Horloge. Cette fonction sera donc réalisée par une ressource matérielle du microcontrôleur ; un timer.

4.1 Fonctionnement d'un timer

Le microcontrôleur SAMD21 possède 5 timers/counters allant de TC3 à TC7. Il est possible de les paramétrer en fonction de l'utilisation qu'il en sera fait. Dans notre cas, le timer TC6 est imposé par le sujet du TP.

Chaque timer peut prendre 3 configurations possibles : 8, 16 ou 32 bits¹. Le nombre de registres associés à chacune des configurations est différent. Nous utiliserons le mode 16 bits (65536 valeurs possibles).

Fonctionnement du TC en mode waveform

Les timers/counters (TC) du microcontrôleur SAMD21 proposent un mode de fonctionnement adapté à la production de signaux logiques : le mode *waveform*. La sélection du mode se fait via la configuration de certains registres. L'objectif est de générer un signal rectangulaire de rapport cyclique quelconque.

Il existe 4 modes de fonctionnement pour les compteurs en mode WAVEFORM présenté par la figure ci-dessous.

Name	Operation	TOP	Update	Output Waveform		OVFIF/Event	
				On Match	On Update	Up	Down
NFRQ	Normal Frequency	PER	TOP/ ZERO	Toggle	Stable	TOP	ZERO
MFRQ	Match Frequency	CC0	TOP/ ZERO	Toggle	Stable	TOP	ZERO
NPWM	Single-slope PWM	PER	TOP/ ZERO	See description above.		TOP	ZERO
MPWM	Single-slope PWM	CC0	TOP/ ZERO	Toggle	Toggle	TOP	ZERO

FIGURE 1 – Mode de fonctionnement en WAVEFORM pour les compteurs

Le mode Match Frequency Generation (MFRQ) est le plus adapté à l'application. En effet, la fréquence n'est fixée qu'avec un seul paramètre CC0. D'après la datasheet du SAMD21, le fonctionnement du mode MFRQ est le suivant.

1. Le timer 32bits fonctionne en assemblant 2 timers 16 bits en cascade

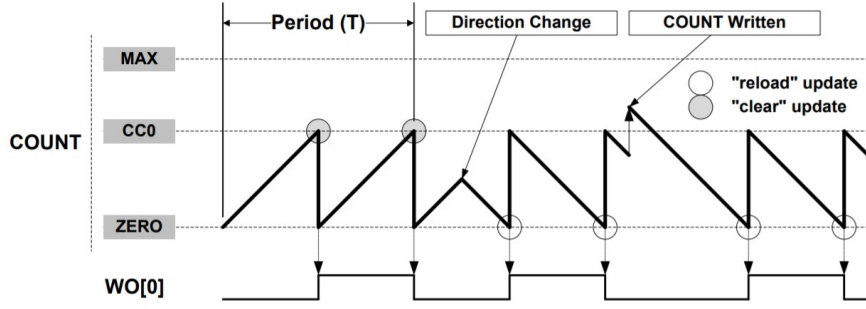


FIGURE 2 – Fonctionnement timer configuré en MFRQ

La période T du signal est contrôlée par le registre $CC0$. Le signal de sortie est numérique, sa valeur se trouve dans $WO[0]$. A chaque fois que le compteur $COUNT$ atteint la valeur du registre $CC0$. Le signal de sortie $WO[0]$ est permuté. La valeur MAX correspond à la résolution du compteur : ici 16 bits donc 65536 valeurs possibles. Il faudra être vigilant car la valeur du compteur vaut deux fois celle du signal de sortie.

Calculs pour une fréquence de 1kHz

Pour obtenir une fréquence de 1kHz il faut déterminer la valeur de $CC0$ comme expliqué précédemment. Pour faire cela il est primordial de bien comprendre son fonctionnement et les registres impliqués dans la configuration. La figure ci-dessous donne la fréquence de comptage.

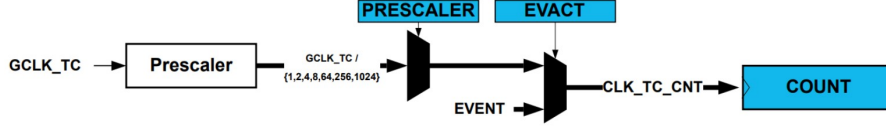


FIGURE 3 – Configuration du compteur TC

L'horloge count est fournie à partir de l'horloge $GCLK_TC$ (Generic clock for TC). Elle est l'horloge de référence pour les TC. Elle a une fréquence de 8MHz. Cette horloge peut être divisée en y appliquant un prescaler afin d'obtenir CLK_TC_CNT . N est une pré division de l'horloge du timer. Dans notre cas, le prescaler n'est pas appliqué et prendra la valeur $N = 1$. L'équation ci-dessous présente la fréquence à laquelle sera effectué le comptage.

$$T_{GCLK_TC} = N * T_{CLK_TC_CNT} = T_{CLK_TC_CNT} \quad (1)$$

Donc

$$f_{GCLK_TC} = f_{CLK_TC_CNT} \quad (2)$$

La fréquence souhaitée est établie à partir de l'équation suivante :

$$f_{WO[0]} = \frac{f_{CLK_TC_CNT}}{2 * (CC0 + 1)} \quad (3)$$

On sait que f_{GCLK_TC} est égale à 8 MHz. Pour une fréquence $f_{WO[0]}$ de 1kHz, on obtient

$$CC0 = \frac{f_{GCLK_TC}}{2 * f_{WO[0]}} - 1 = 3999 \quad (4)$$

La valeur chargée dans le registre CC0 sera donc 3999.

4.2 Configuration TC6

Cette partie détaille les configurations nécessaires à la génération d'un signal carré de 1kHz grâce à TC6. Les éléments suivants seront configuré : le generic clock controller, le power manager, un port d'entrée sortie et finalement TC6.

4.2.1 Configuration du GCLK

Chaque périphérique de la carte SAMD21 nécessite une horloge de fonctionnement interne. Pour notre périphérique du Timer, il s'agit de GCLK_TC6 (Generic clock for TC6).

La figure ci-dessous présente la génération des signaux de l'horloge périphérique et de l'horloge principale. Le Generic Clock Controller est composé de 9 générateurs et de multiplexeurs.

On remarque que le Generic Clock Controller est divisé en deux parties. D'une part le Generic

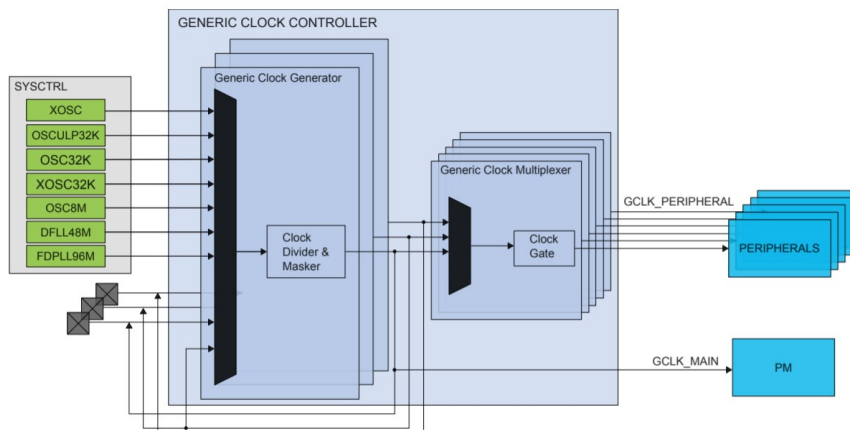


FIGURE 4 – Fonctionnement du Generic Clock Controller

Clock Controller est configuré par le registre GENCTRL. D'autre part le Generic Clock Multiplexer est configuré par le registre CLKCTRL.

Configuration du registre GENCTRL

Le détail de Generic Clock Generator Control (GENCTRL) est donné dans la figure ci-dessous :

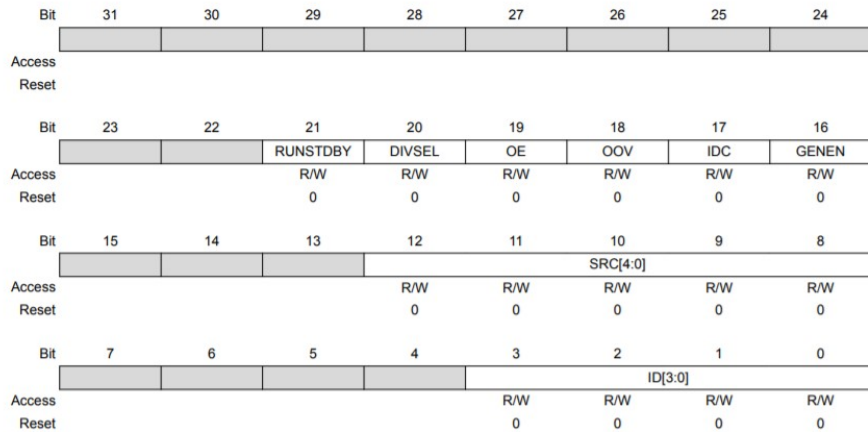


FIGURE 5 – Configuration des différents bits du GENCTRL

- **RUNSTDBY** : Fonctionnement en mode Standby ou non. Dans notre cas, nous voulons la désactiver donc il faut mettre **0** < 21 dans ce champ.
- **DIVSEL** : Définit le facteur de division de l'horloge. Nous ne voulons pas la diviser donc il faut mettre la valeur **0** < 20 dans ce champ.
- **OE** : Permet d'autoriser l'activation sur une sortie de GCLK. Nous ne voulons pas activer cette option donc il faut mettre la valeur **0** < 19 dans ce champ.
- **OOV** : : Définit la valeur de la sortie de GCLK. Lorsque l'OE est à 0 il faut mettre également 0 dans ce champ donc la valeur **0** < 18.
- **IDC** : Définit du rapport cyclique en cas de division impaire. Dans notre cas, il faut mettre la valeur **0** < 17 dans ce champ.
- **GENEN** : Validation ou non du générateur d'horloge. Nous voulons l'activer donc il faut mettre la valeur **1** < 16 dans ce champ.
- **SRC[4 :0]** : Choix de la source d'horloge. Nous voulons choisir la source OSC8M donc d'après la datasheet il faut mettre la valeur **6** < 8 dans ce champ.
- **ID[3 :0]** : Définit le numéro du générateur que l'on configure (0 à 8). Nous choisissons la générateur 0 donc il faut mettre la valeur **0** < 0.

Configuration du registre CLKCTRL

Ce registre permet de choisir parmi les 9 générateurs décrit précédemment. Le détail de Generic Clock Control (CLKCTRL) est donné dans la figure ci-dessous :

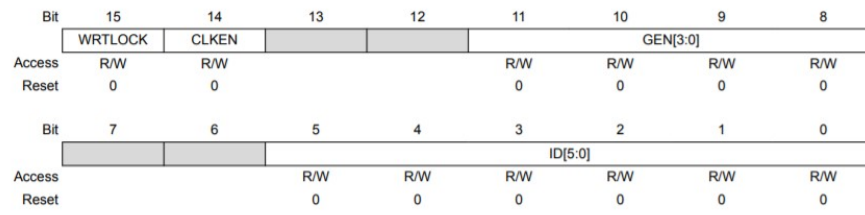


FIGURE 6 – Configuration des différents bits du CLKCTRL

4.2.2 Configuration du PM

4.2.3 Configuration du PORT

4.2.4 Configuration du TC6

5 Fonction sous interruption

Les interruptions sont des outils essentiels pour concevoir des systèmes réactifs (en : responsive) dès lors qu'ils doivent exécuter des opérations logicielles et matérielles simultanément. Avant de présenter le fonctionnement et la configuration du système d'interruption du Cortex-M il est préférable de donner un aperçu du problème.

5.1 Présentation du problème

L'environnement avec lequel interagit le microcontrôleur est dit asynchrone. Il ne peut à priori par connaître l'instant d'apparition d'un événement. Il est alors obligé de le scruter fréquemment pour être averti rapidement d'un changement. C'est précisément ce que fait l'étape 1. La lecture du bouton est placée dans une boucle infinie, si l'utilisateur appui sur ce dernier l'état de la led est modifié. Bien que ce programme soit inefficace, il fonctionne bien et offre une faible latence. Ce mode de fonctionnement est appelé "polling". Le problème survient dès lors qu'une tâche logicielle est ajoutée au programme. Le processeur n'est plus seulement occupé à contrôler l'état du bouton mais également à exécuter une autre partie du programme. La réactivité n'est alors plus garantie. Dans certains cas, si la tâche logicielle est importante, le changement d'état du bouton pourra être invisible pour le programme.

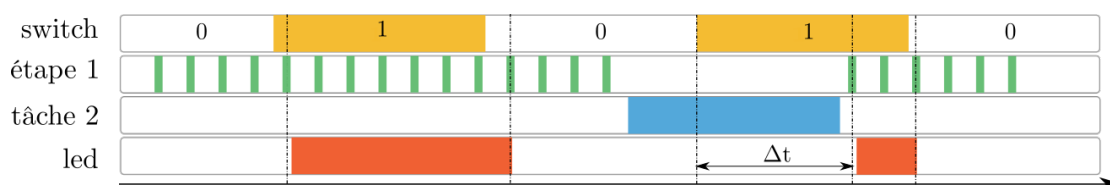


FIGURE 7 – Ordonnancement tâches avec polling

La figure ci-dessus représente l'exécution² d'un programme comme celui de l'étape 1 auquel une seconde tâche a été ajoutée (tâche 2). Deux cas sont illustrés dans cet exemple. Le premier, à gauche, correspond au bon déroulement, le processeur lit de manière successive l'état du switch (étape 1), lorsque une lecture vaut '1' l'état de la led est mis à '1' et même chose pour '0'. Une petite (latence) est présente mais elle sera généralement négligée. Dans le second cas, à droite, la tâche 2 est en cours d'exécution et empêche la lecture du bouton. Une fois terminée la mise à jours de la led peut s'opérer comme précédemment. L'exécution de la tâche 2 a généré un retard qui n'est cette fois-ci plus négligeable du point de vue de l'utilisateur. Si la tâche 2 avait duré un instant de plus le système n'aurait pas réagi à la stimulation. Un tel cas ne doit, en aucun cas, se produire pour un système dont la sécurité est critique.

En pratique ce problème survient fréquemment. Les interactions utilisateurs, tel que l'appui sur un bouton, sont relativement lentes comparés à la vitesse de fonctionnement des processeurs. Un "polling" pourra suffire dans certains cas. Mais pour ce qui est des événements matériels, comme un timer ou un protocole série, la situation est plus critique. Il n'y a alors pas d'autres solutions que d'utiliser une interruption. De manière générale les logiciels utilisant des interruptions fonctionnent plus efficacement que ceux basés sur le "pooling". Aucun temps n'est perdu à contrôler l'apparition d'un événement. L'approche par déclenchement sur événements (en : even-trigger) offre également une meilleure réactivité. [1]

2. Aucune échelle précise n'est choisie le but étant une simple présentation fonctionnelle.

5.2 Fonctionnement d'une interruption

Une interruption est définie comme une suspension temporaire de l'exécution d'un programme informatique par le microprocesseur afin d'exécuter un programme prioritaire (routine d'interruption). La dite suspension est déclenchée par une source externe.

La gestion d'une interruption s'effectue comme tel : pause du programme, sauvegarde du contexte, identification de la routine d'interruption à exécuter (en fonction de sa source), exécution de la routine et rétablissement du contexte. Ces actions sont gérées par le NVIC, Nest Vectored Interrupt Controller faisant l'interface entre les périphériques et le CPU.

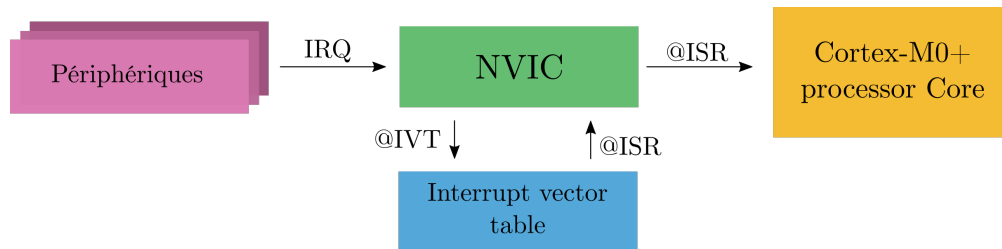


FIGURE 8 – Schéma bloc NVIC

Lorsqu'une interruption x se produit, la demande d'interruption (IRQ) est envoyée au NVIC. Si le NVIC accepte la demande, l'étape suivante du NVIC consiste à trouver l'adresse de départ de la routine d'interruption (@ISR). Cette adresse est stockée dans la table de vecteurs d'interruption (IVT). Le NVIC utilise le numéro d'interruption x pour calculer l'adresse de l'exception dans la table des vecteurs d'interruption. Il utilise ensuite le contenu de cette adresse mémoire pour exécuter le gestionnaire d'exception. Le compteur de programme est alors chargé avec l'adresse de l'ISR et le CPU commence à exécuter la routine d'exception. Un rappel du vocabulaire utilisé dans la figure 8.

@	adresse mémoire
IRQ	Interrupt Service Routine
NVIC	Nest Vectored Interrupt Controller
IVT	Interrupt Vector table
ISR	Interrupt Service Routine

TABLE 1 – Terminologie mécanisme d'interruption

La table des vecteurs d'interruptions, est une structure de données et, comme son nom l'indique, un tableau contenant des vecteurs. En programmation embarquée, un vecteur désigne une adresse mémoire. Par conséquent, une table de vecteurs est une table contenant des adresses mémoire. **La table des vecteurs d'interruption contient les adresses (pointeurs de fonction) des routines d'interruption et des fonctions de gestion des exceptions.** Les exceptions désignent les événements en mesure d'interrompre le déroulement normal du programme pour effectuer une courte tâche. Les interruptions sont donc partie des exceptions. Elles ne sont pas les seules le processeur lui même peut déclencher le mécanisme d'exception lorsque on lui demande de faire quelque chose qu'il ne parvient pas à faire. Par exemple la lecture d'un mot de 32 bits à une adresse impaire provoquera une exception à l'exécution.

5.3 Configuration interruption

Pour qu'une interruption se produise il faut que le périphérique soit configuré de sorte à réagir à l'occurrence d'un événements spécifique. Dans le contexte du projet nous devons calculer le gabarit toutes les 1 ms. Cette partie présente la configuration à effectuer pour que le timer TC6 déclenche une routine d'interruption contenant la fonction calcul gabarit. Elle détaillera également l'autorisation de l'interruption par le NVIC.

5.3.1 Timer TC6

L'activation de l'interruption sur le périphérique TC6 est documentée en détaille dans la datasheet. [4] Chaque source d'interruption est associée à un drapeau d'interruption. Le drapeau d'interruption (en : interrupt flag), dans le registre (INTFLAG), est activé lorsque la condition d'interruption en vrai. Chaque interruption peut être activée individuellement en écrivant un '1' au bit correspondant dans le registre d'activation Interrupt Enable Set (INTENSET) et désactivée en écrivant un '1' au bit correspondant dans le registre de désactivation Interrupt Enable Clear (INTENCLR). Le registre INTENSET se présente comme tel :

Bit	7	6	5	4	3	2	1	0
			MC1	MC0	SYNCRDY		ERR	OVF
Access			R/W	R/W	R/W		R/W	R/W
Reset			0	0	0		0	0

FIGURE 9 – Registre INTENSET

Ayant programmé le timer en mode MFRQ dans la partie précédente, seul le bit 4 nous intéresse. Il est chargé à '1' pour activer l'interruption sur correspondance de COUNT et MC0. L'implémentation en C est intuitive :

```
1 void configure_TC6_IT(void){
2     Tc *ptr_TC = TC6;
3     ptr_TC->COUNT16.INTENSET.reg = TC.INTFLAG_MC0;
4 }
5
```

Comme l'indique la datasheet après la configuration du périphérique il faut activer l'interruptions correspondante dans NVIC.

"The interrupt request line is connected to the Interrupt Controller. In order to use interrupt requests of this peripheral, the Interrupt Controller (NVIC) must be configured first." [4](page 557)

5.3.2 NVIC

Le NVIC est un élément interne au processeur. La documentation de Microchip donne pour chaque périphérique le numéro d'interruption associée mais renvoie à la documentation de Arm pour la configuration du NVIC. Le numéro du TC6 est 21. Comme nous avons pu l'évoquer plus haut le NVIC peut accepter ou non les interruptions provenant des périphériques. Il consulte pour cela un registre nommé NVIC.ISER contenant un seul champ de 32 bits SETENA littéralement set enable. L'extrait de la documentation [5] Arm ci-dessous nous donne la valeur des bits à charger.

Bits	Name	Function
[31:0]	SETENA	Enables, or reads the enabled state of one or more interrupts. Each bit corresponds to the same numbered interrupt:
	On reads	0 the associated interrupt is disabled. 1 the associated interrupt is enabled.
	On writes	0 no effect. 1 enable the associated interrupt.

FIGURE 10 – Registre NVIC_ISER ARMv6-M

Ce registre permet de lire l'activation ou d'activer l'interruption sur chaque ligne. A noter : il faut utiliser un autre registre pour la désactivation, écrire un '0' n'aura aucun effet. **Nous devons dans notre cas placer un '1' sur le bit 21.** Cette configuration est grandement simplifiée par l'ASF. Mais il est essentiel de comprendre et de retenir qu'une activation supplémentaire au périphérique est nécessaire. Le code ajouter à la fonction main est le suivant :

```
1 system_interrupt_enable(SYSTEM_INTERRUPT_MODULE_TC6);
```

L'appel de cette fonction est simple et permet l'abstraction du matériel. Le paramètre passé est le numéro d'interruption TC6_IRQn comme le montre la définition dans `system_interrupt_features.h`

```
1 SYSTEM_INTERRUPT_MODULE_TC6 = TC6_IRQn
```

TC6_IRQn est défini plus haut et dépend de la cible utilisée dans notre cas le SAMD21J18A.

```
1 TC6_IRQn = 21
```

Sa valeur est bien 21. Nous venons d'observer la cohérence entre la datasheet et l'ASF. Pour finaliser le raisonnement regardons l'implémentation de la fonction `system_interrupt_enable()`.

```
1 static inline void system_interrupt_enable(  
2 const enum system_interrupt_vector vector)  
3 {  
4     NVIC->ISER[0] = (uint32_t)(1 << ((uint32_t)vector & 0x0000001f));  
5 }
```

Cette fonction est particulièrement intéressante d'un point de vue pédagogique par l'utilisation de static, inline et du masquage sur le vecteur. D'un point de vue fonctionnel, elle place un '1' décalé du numéro d'interruption sur le registre ISER de NVIC. C'est précisément ce qui doit être fait pour activer les interruptions comme le montre le figure 10.

5.3.3 Routine

La routine d'interruption est la fonction appelée automatiquement par le NVIC lors d'un événement. L'adresse stocké à l'indice TC6_IRQn de la table des vecteurs d'interruption coreposn- dant à cette fonction est chargée dans PC. Elle est exécutée puis le PC retourne à sa valeur d'origine. La définition de TC6_Handler est réalisée par l'Asf, il n'est pas nécessaire de déclarer son prototype.

Dans la suite du projet la fonction calcul gabarit est exécutée par la routine d'interruption. N'étant pas encore développée, le changement d'état de PB6, jouera le rôle de témoin pour valider le bon déclenchement toutes les 1ms.

```
1 void procedure_traitement_IT(void){  
2     Port *ptr_Port = PORT;  
3     ptr_Port->Group[1].OUTTGL.reg = PORT_PB06;  
4 }
```

```

5
6 void TC6_Handler(void){
7     Tc *ptr_TC = TC6;
8
9     procedure_traitement_IT();
10
11     ptr_TC->COUNT16.INTFLAG.reg = TC_INTFLAG_MC0;
12 }
13

```

Cette fonction d'interruption témoin commence par la déclaration du pointeur TC6. Mais si cette variable ne sert qu'à la dernière ligne il est impératif de l'initialisation au début de la fonction comme l'indique la recommandation 53 de guide de référence de l'ANSI [3]. Ensuite, la procédure de traitement est appelée. Elle consiste à inverser (en :toggle) l'état du pin PB6. Pour finir, le drapeau d'interruption est remis à zéro grâce au registre Flag Status and Clear (INTFLAG) dont la structure est identique à celle de la figure 9. Il peut être intéressant de noter que la fonction `procedure_traitement_IT()` ; n'étant appelée qu'une seule fois dans le programme sera insérée directement dans le code en assembleur. C'est l'équivalent d'un `inline` réalisé automatiquement par le compilateur.

ajouter code assembleur pour le prouver et capture d'ecran frequence IT

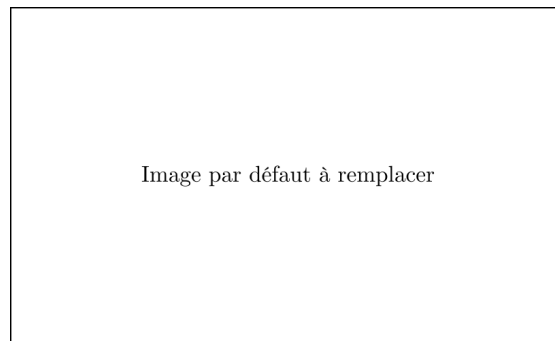


FIGURE 11 – Oscillogramme vérification procédure interruption

La fréquence du signal observé est de XXHz.

Cette partie a permis de soulever un problème lié à une programmation conjointe logiciel matériel et de proposer une solution, les interruptions. Le fonctionnement global du système d'interruption des microcontrôleurs basés sur un Cortex-M a pu être détaillé. La configuration à travers un exemple concret utilisant le TC6 a finalement illustré ces explications.

6 Implémentation fonction calcul gabarit

Cette partie présente la conception de la fonction calcul gabarit conformément au cahier des charges donné dans le sujet. Elle a pour fonction de déterminer la vitesse de rotation du moteur, comprise entre 0 et 5000 tr/min, en fonction du temps. Le gabarit se décompose en trois phases :

- accélération d'une durée de 128 ms
- vitesse constante d'une durée de 512 ms
- décélération d'une durée de 128 ms

La valeur du palier correspondant à la vitesse constante est un paramètre variable fournis par un bloc externe avec une précision de 10 tr/min. Pour simplifier le problème elle ne pourra être modifiée au cours d'un cycle. A ce niveau dans le projet les blocs externes ne sont pas encore développés. Pour faciliter l'observation lors de tests le gabarit sera généré en boucle. Un état Repos doit être ajouté pour les délimiter, sa durée sera de 64 ms.

6.1 Conception fonctionnelle

L'étape de conception fonctionnelle a pour objectif de définir une solution en ne considérant que l'aspect fonctionnel et sans tenir compte des contraintes technologiques. Il est donc question de trouver une solution en mesure de générer **le gabarit constitué des quatre étapes : repos, accélération, constante et décélération**. Ainsi la description sous forme d'un automate à état finis s'impose par la séquentialité du problème. La traduction donne le résultat suivant :

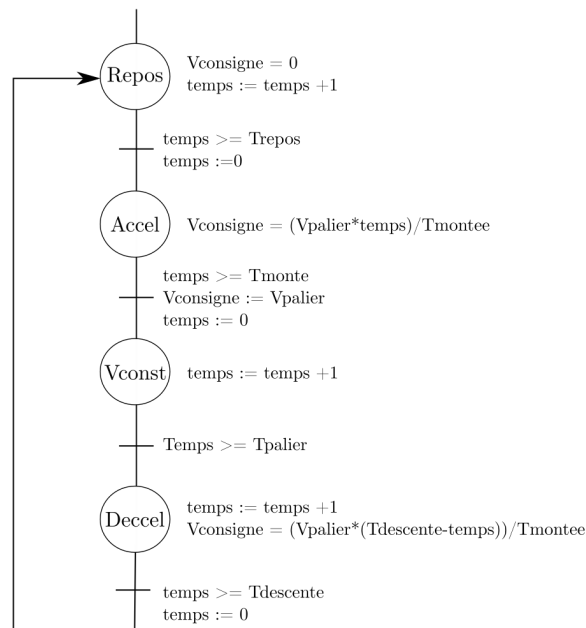


FIGURE 12 – Description fonctionnelle calcul gabarit (FSM)

Cet automate est constitué des quatre états repos, accélération, constante et décélération et de leur transition respective. La variable V_{palier} correspond au paramètre d'entrée de la fonction. Le résultat de l'exécution, placé dans $V_{consigne}$, devra être communiqué au reste du système.

6.2 Implémentation logicielle

La conception fonctionnelle précédente a permis de clarifier le problème. La première étape de la traduction en C est la déclaration des constantes. Elle se fait par l'instruction préprocesseur `define` ayant pour fonction de remplacer les instances suivantes de l'identifieur par la suite de lexèmes³. [6]

```
1 #define identifieur suite-de-lexemes
```

Ce procédé est utilisé pour définir les constantes utilisées ci-dessous. Le code C est le suivant :

```
1 #define TMontee 128
2 #define TPalier 512
3 #define TDescente 128
4 #define TRepos 64
```

Les valeurs numériques proviennent directement du cahier des charges.

Avant de traduire l'automate figure 12 il est important de définir précisément le type des variables utilisées.

nom	min	max	type générique	type C
temps	0	512	entier	uint16_t
Vconsigne	0	5000	entier	uint16_t
Vpalier	0	5000	entier	uint16_t

TABLE 2 – Types des variables fonction calcul gabarit

Le tableau ci dessus rassemble les informations permettant de choisir efficacement les types des variables. Elles sont toutes les trois entières, leur valeur minimum est supérieur à zéro et leur maximum compris entre 255 et 65535. Le type C correspondant est donc nécessairement `uint16_t`. Il existe une autre variable implicite à l'implémentation d'un automate. C'est l'état courant `state` de type énumération. Il peut prendre pour valeur chacun des états repos, accélération, constante et décélération. Sa définition en langage C est la suivante :

```
1 enum state_t {Repos , Accel , Vconst , Decel };
```

Le type énumération correspond à un entier dont la taille peut contenir le nombre d'élément. Par défaut le premier élément vaut 0, et les suivants sont supérieures de 1 par rapport à la valeur de la constante précédente. Il est possible d'affecter une valeur manuellement. Le guide des bonnes pratiques de l'ANSSI [3] indique par la règle 68 qu'il ne faut pas mélanger des constantes explicites et implicites dans une énumération pour éviter tout comportement indéterminé. Toute tentative d'affectation à une valeur non définie dans l'énumération provoquera une erreur à la compilation. Dans notre cas nous laisserons la définition à la charge du compilateur. Tel qu'est construit la machine état il est nécessaire de sauvegarder les valeurs des variables d'une exécution à l'autre. Il faudra donc ajouter le qualificatif `static`.

Une fois les variables et constantes déclarées le corps de la fonction peut être implémenté. Il convient de convertir la machine état figure 12 en langage C. Cela se fait simplement autour d'un `"switch case"`. Il est nécessaire de déclarer la variable `state` de type `state_t`. Un cas par défaut a été ajouté à la description fonctionnelle pour couvrir les cas d'erreur. Lors d'une exécution normale la variable `state` ne peut prendre d'autres états que ceux déclarés dans l'énumération. Si toutefois, pour une raison inconnue, elle venait à prendre une autres valeurs, le programme se trouverait dans un état indéterminé et y resterait. Avec le cas défaut le programme peut se trouver dans un état non défini, mais il sera alors remis dans son état d'origine et l'exécution pourra continuée. L'implémentation en C est la suivante :

3. Unité minimale de signification appartenant au lexique (Larousse)

```

1  /* file : calcul_gabarit.c */
2
3  uint16_t calcul_gabarit (uint16_t VPalier){
4
5      static uint16_t Temps = 0;
6      static enum state_t state = Repos;
7      static uint16_t Vconsigne = 0;
8
9      switch (state)
10     {
11         case Repos :
12             Vconsigne = 0;
13             Temps ++;
14             if(Temps >= 64){
15                 state = Accel;
16                 Temps = 0;
17             }
18             break;
19
20         case Accel :
21             if(Temps>=TMontee){
22                 state = Vconst;
23                 Temps = 0;
24                 Vconsigne = VPalier;
25             }
26             else{
27                 Temps ++;
28                 Vconsigne = (VPalier * Temps) / TMontee;
29             }
30             break;
31
32         case Vconst :
33             if (Temps>=TPalier){
34                 state = Decel;
35                 Temps = 0;
36             }
37             else{
38                 Temps ++;
39             }
40             break;
41
42         case Decel:
43             if(Temps >= TDescente){
44                 state = Repos;
45                 Temps = 0;
46                 Vconsigne = 0;
47             }
48             else{
49                 Temps ++;
50                 Vconsigne = (VPalier * (TDescente - Temps)) / TDescente;
51             }
52             break;
53
54         default :
55             state = Repos;
56             Temps = 0;
57             Vconsigne =0;
58             break;
59     }
60     return Vconsigne;
61 }
62 }

```

6.3 Test calcul gabarit et configuration DAC

La fonction calcul gabarit a été implémenté et doit donc être testée. Le résultat retourné est **Vconsigne**, dont la valeur analogique est comprise entre 0 et 5000. Pour visualiser son évolution dans le temps on peut s'appuyer sur un périphérique présent dans le SAMD21, le convertisseur numérique vers analogique (DAC : Digital Analog Converter). Ainsi il sera possible de visualiser le gabarit généré sous sa forme temporel tel qu'il a été décrit initialement dans le cahier des charges. Le DAC converti une valeur numérique en tension. Celui présent dans le microcontrôleur utilisé dispose d'une voix 10 bits sur 3,3V, soit une précision de 3,32mV. Sa représentation structurelle sous forme d'un diagramme bloc est la suivante :

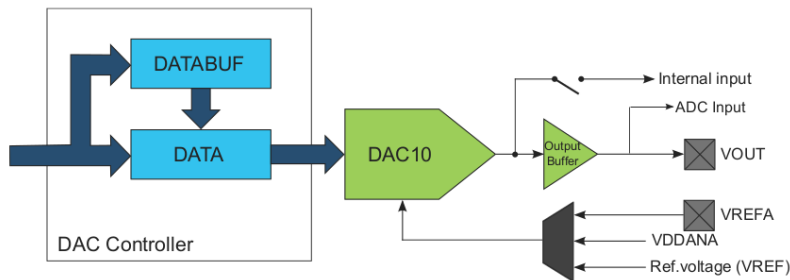


FIGURE 13 – Description diagramme bloc DAC

Les données à convertir transitent sur le bus APB BRIDGE C et doivent être dans le registre **DATA** ou **DATABUF**. Ils correspondent respectivement à la valeur actuellement convertie et à la prochaine valeur convertie. Dans notre cas nous n'utiliserons que le registre **DATA**. Sa valeur est ensuite transmise à l'unité de conversion 10 bits dont la tension de référence, tension maximum, peut être sélectionnée parmi plusieurs sources. Le résultat sort finalement sur une sortie interne, un buffer capable de débiter un courant plus important. Comme pour les autres périphériques un certain nombre de registres sont à disposition pour configurer ses nombreuses fonctionnalités.

Configuration du registre CTRLA

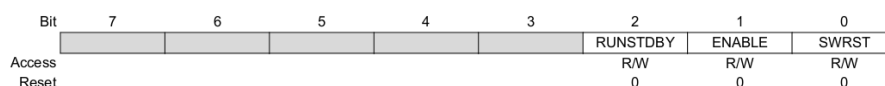


FIGURE 14 – Description registre DAC CTRLA

- **SWRST** - Software Reset - Ce champ permet de remettre les registres du DAC dans leur état initial. Nous ne souhaitons pas faire cela, on écrira dans le registre **0 << 0**
- **ENABLE** - Enable DAC Controller - Ce champ permet d'activer le Dac il faut donc écrire **1 << 1** dans le registre.
- **RUNSTDBY** - Run in Standby - Ce champ permet d'activer ou de désactiver le buffer du DAC et mode veille. Dans notre cas cela a peu d'importance ce bit peut être mis à 1 le laisser activé dans tous les cas : **1 << 2**.

Configuration du registre CTRLB

Bit	7	6	5	4	3	2	1	0
	REFSEL[1:0]			BDWP	VPD	LEFTADJ	IOEN	EOEN
Access	R/W	R/W		R/W	R/W	R/W	R/W	R/W
Reset	0	0		0	0	0	0	0

FIGURE 15 – Description registre DAC CTRLB

- **EOEN** - External Output Enable - Ce champ permet d'activer ou de désactiver la sortie du buffer. Nous ne souhaitons appliquer la tension sur le pin Vout, on écrit dans le registre $1 \ll 0$
- **IOEN** - Internal Output Enable - La sortie interne n'est pas utilisée on écrit $0 \ll 1$ dans le registre.
- **LEFTADJ** Left-Adjusted Data - La taille du registre **DATA** est de 16 bits mais seul les 10 premiers sont utilisés. Ce champs permet de sélectionner les bits en partant de la droite ou de la gauche. Dans notre cas les poids faibles sont à droite on choisira l'option d'alignement à droite. On écrit dans le registre $0 \ll 2$
- **VPD** - Voltage Pump Disabled - Cette option permet d'économiser de l'énergie. Sa gestion est conservée comme automatique par l'écriture de $1 \ll 3$ dans le registre.
- **BDWP** - Bypass DATABUF Write Protection - Le registre DATABUF n'est pas utilisé, il n'est donc pas nécessaire de le protéger, on écrit $1 \ll 4$ dans le registre.
- **REFSEL[1:0]** - Reference Selection - Ce registre permet de sélectionner la source de tension de référence. Nous prendrons l'alimentation destinée aux circuits analogiques en écrivant $0x1 \ll 6$ dans le registre.

Initialisation du registre DATA

Bit	15	14	13	12	11	10	9	8
	DATA[15:8]							
Access	W	W	W	W	W	W	W	W
Reset	0	0	0	0	0	0	0	0

Bit	7	6	5	4	3	2	1	0
	DATA[7:0]							
Access	W	W	W	W	W	W	W	W
Reset	0	0	0	0	0	0	0	0

FIGURE 16 – Description registre DAC CTRLB

Ce registre contient la valeur 10 bits à convertir dont l'alignement a été sélectionné avec le registre **LEFTADJ**. Pour tester la bonne configuration se champs pourra être chargé à $1023/2$. Une tension de $3,3/3V$ devrait pouvoir être observée.

La configuration du DAC traduite en langage C donne le résultat suivant :

```
1 void config_DAC(void){
2     Dac *ptr_DAC = DAC ;
3
4     ptr_DAC -> CTRLA.reg = DAC_CTRLA_RUNSTDBY |
5                           DAC_CTRLA_ENABLE   |
6                           DAC_CTRLA_SWRST_NO ;
7
8     ptr_DAC -> CTRLB.reg = DAC_CTRLB_REFSEL_AVCC |
9                           DAC_CTRLB_BDWP_NO    |
10                          DAC_CTRLB_VPD_NO     |
11                          DAC_CTRLB_LEFTADJ_NO  |
12                          DAC_CTRLB_IOEN_NO     |
13                          DAC_CTRLB_EOEN ;
14     /* Initialisation pour test */
15     ptr_DAC -> DATA.reg = 1024/2;
16     /* DATA.reg valeur 10bits max */
17 }
```

De nouveau l'utilisation des structures de l'ASF facilite grandement le travail. Cela améliore aussi la compréhension et donc la modification par un tiers.

Références

- [1] Alexander G. Dean. *Embedded Systems Fundamentals with ARM Cortex-M based Microcontrollers*. ARM Education Media UK, March 2017.
- [2] Pompignac Hernando Monteil, Nicomette. *Du langage C au C++*. Pour l'ingénieur, 2012.
- [3] ANSSI. Règles de programmation pour le développement sécurisé de logiciels en langage c, November 2021.
- [4] Microchip. complet datasheet sam d21/da1 family. Technical report, Microchip Technology Inc., 2021. DS40001882G.
- [5] Arm. *ARMv6-M ArchitectureReference Manual*. Arm, 2010.
- [6] Brian W. Kernighan Denis M. Ritchie. *Le Langage C : Norme ANSI*. Dunod, Paris, 2000.