projet
document

Louison GOUY Yiying WEI

28 novembre 2021

name

abstract

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

Linus Torvalds

Table des Matières

Liste des figures

1 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 [**?**]. 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 considérerons trop permissive. 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 National de la Sécurité des Systèmes Informatiques française (ANSSI) propose un rapport complet, *Règles de programmation pour le développement sécurisé de logiciels en langage C* [**?**], visant à "favoriser la production de logiciels C plus sécurisés, plus sûrs, d'une plus grande robustesse et portables". Il sera la référence durant ce projet.

2 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.

2.1 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   }
20 }
```

La première ligne permet d'inclure la bibliothèque asf et ainsi de profiter du niveau d'abstraction mis à disposition par Microship. La suivante, bien connu des développeurs C, est le point d'entré 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. 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) à chaque exécution de la boucle. La condition suivante d'éclanche alors le résultat souhaité l'allumage ou l'extinction de la LED0.

2.2 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 par le biais de pilotes et de middlewares à forte valeur ajoutée. 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.

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 comme l'horloge ou les entrées sorties par exemple.

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 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 particulaire 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 définit. 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);}
```

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.

Condition sur E/S

Le lignes suivantes implémenté via une structure `if else` traduise le comportement souhaité du point de vue utilisateur. A savoir, le maintient 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` définit 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`.

3 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.

3.1 Fonctionnement d'un timer

Le microcontrôleur SAMD21 utilisé possède 5 counter/timer utilisable allant de TC3 à TC7. Il est ensuite possible de les paramétrer en fonction de l'utilisation qu'il en sera fait. Pour notre cas, le timer TC6 est imposé pour cette fonction Horloge.
Chaque timer peut prendre 3 configurations possibles : 8, 16 ou 32 bits (ce dernier fonctionne avec 2 timers 16 bits cascadés). Le nombre de registres associés à chacune des configurations est différent. Pour notre cas, on utilisera 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*. Afin de configurer ce mode il va falloir configurer certains registres du microcontrôleur pour obtenir un signal rectangulaire de rapport cyclique quelconque.
Il existe 4 modes de fonctionnement pour les compteurs en WAVEFORM donnés en 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 MFRQ, plus adapté afin d'obtenir la fréquence voulue, sera donc choisi pour la génération du signal logique stable et périodique. D'après la datasheet du SAMD21, le fonctionnement du mode MFRQ est le suivant.

La période T du signal est contrôlée par le registre CC0 qu'il faudra modifier. Le signal de sortie est un signal numérique dont la valeur se trouve sur WO[0]. Le signal de sortie WO[0] est permuté à chaque fois que le compteur atteint la valeur du registre CC0. La valeur MAX correspond à la résolution du compteur : ici 16 bits donc 65536 valeurs possibles.

Calculs pour une fréquence de 1kHz
Pour obtenir une fréquence de 1kHz il faut déterminer la valeur de CC0 comme expliqué précédemment. Afin de déterminer cette valeur il faut dans un premier temps comprendre comment le comptage

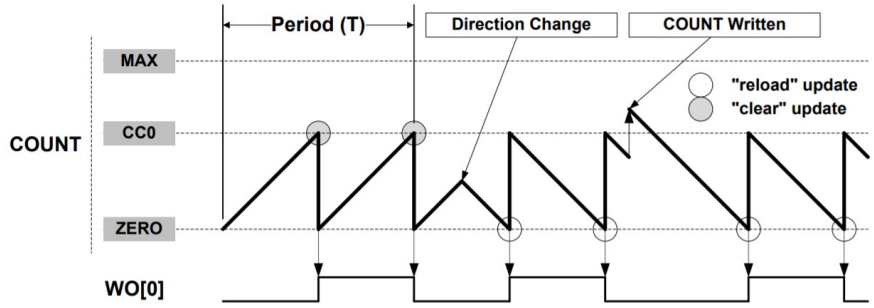

FIGURE 2 – Match Frequency Operation(MFRQ)

est effectué et à quelle fréquence. La figure ci-dessous donne la fréquence de comptage.

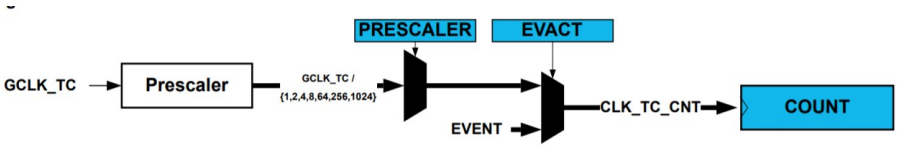

FIGURE 3 – Configuration du compteur TC

L'horloge count est fournie à partir de l'horloge GCLK_TC (Generic clock for TC) qui 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 prédivision 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.

$$T_{GCLK\_TC} = N * T_{CLK\_TC\_CNT} = T_{CLK\_TC\_CNT}$$

Donc

$$f_{GCLK\_TC} = f_{CLK\_TC\_CNT}$$

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

$$f_{WO[0]} = \frac{f_{CLK\_TC\_CNT}}{2 * (CC0 + 1)}$$

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

$$CC0 = \frac{f_{GCLK\_TC}}{2 * f_{WO[0]}} - 1 = 3999$$

3.2 Configuration TC6

Configuration du TC6.

Références