Tutoriels STM32F0

Périphériques standards

Convertisseur Numérique vers Analogique CNA

Arouna DARGA – enseignant chercheur Sorbonne Université
arouna.darga@sorbonne-universite.fr

Table des matières

1. Introduction

Une sortie PWM peut facilement être transformée en un signal "**analogique**" grâce à un filtre passe-bas. C'est la raison pour laquelle dans la plateforme Arduino la sortie **PWM** est dite "analogique". Néanmoins, le MCU STM32F072 dispose d'un véritable **convertisseur numérique-analogique (CNA)** ou Digital to Analog Converter, DAC en anglais. En utilisant le **DAC**, vous vous rapprochez d'une vraie sortie analogique comme nous le verrons dans ce tutoriel.

2. Configuration du DAC du STM32F072

Selon la fiche technique du MCU, il y a deux broches qui peuvent être utilisées comme sortie DAC : **PA4** et **PA5**. Comme **PA5** est déjà utilisé par la LED de la carte, nous ne considérerons que le canal **DAC 1**, relié à **PA4**. ***Voir tableau 13, page 35 de la notice technique du MCU STM32F072RB***

M3	29	20	14	G6	PA4	I/O	TTa		SPI1_NSS, I2S1_WS, TIM14_CH1, TSC_G2_IO1, USART2_CK	COMP1_INM4, COMP2_INM4, ADC_IN4, DAC_OUT1
K4	30	21	15	F5	PA5	I/O	TTa		SPI1_SCK, I2S1_CK, CEC, TIM2_CH1_ETR, TSC_G2_IO2	COMP1_INM5, COMP2_INM5, ADC_IN5, DAC_OUT2

Pour une utilisation de base, la configuration du **DAC** est très simple. Il suffit de configurer la broche **PA4** comme **analogique**, puis **d'activer l'horloge du DAC** et **d'activer le DAC**. Ajoutez la fonction suivante à votre fichier de code source **bsp.c** et la déclaration du prototype associé au fichier d'en-tête **bsp.h** :

```
/*
 * DAC_Init()
 * Initialise le DAC une seule sortie
 * sur channel 1 -> pin PA4
 */

void BSP_DAC_Init()
{
	// activer horloge du GPIOA
	RCC->AHBENR |= RCC_AHBENR_GPIOAEN;

	// Configurer le pin PA4 en mode Analog
	GPIOA->MODER &= ~GPIO_MODER_MODER4_Msk;
	GPIOA->MODER |= (0x03 <<GPIO_MODER_MODER4_Pos);

	// Activer horloge du DAC
	RCC->APB1ENR |= RCC_APB1ENR_DACEN;

	// Reset de la configuration du DAC
	DAC->CR = 0x00000000;

	// Activer le canal 1 du DAC
	DAC->CR |= DAC_CR_EN1;
}
```

3. Configuration de la période d'échantillonnage

Nous voulons générer un échantillon de signal analogique du DAC à intervalles réguliers. Comme nous ne sommes pas sûrs du temps nécessaire au calcul de l'échantillon, nous utiliserons un événement de mise à jour d'un Timer pour déclencher de nouveaux échantillons à intervalles réguliers. Ajustez la fonction de gestion du temps du TIM6 précédemment développée de sorte que l'événement de mise à jour (détecté par scrutation du drapeau UIF) se produise toutes les 200µs : ***Voir fonction BSP_TIMER_Timebase_Init() dans le fichier de code source bsp.c***

```
/*
 * BSP_TIMER_Timebase_Init()
 * TIM6 cadenser a 48MHz
 * Prescaler   = 48 -> periode de comptage = 1us
 * Auto-reload = 200  -> periode de mis a jour   = 200 us
 */

void BSP_TIMER_Timebase_Init()
{
	// activer horloge du peripherique TIM6
	// mettre a '1' le bit b4 (TIM6EN) du registre RCC_APB1ENR
	// voir page 131 du manuel de reference
	RCC->APB1ENR |= (1<<4);

	// Faire un Reset de configuration du TIM6 :
	// TIM6_CR1 et TIM6_CR2
	// voir page 543 a 544 du manuel de reference
	TIM6->CR1 = 0x0000;
	TIM6->CR2 = 0x0000;

	// Configuration frequence de comptage
	// Prescaler : registre TIM6_PSC
	// Fck = 48MHz -> /48000 = 1KHz frequence de comptage
	TIM6->PSC = (uint16_t) 48 -1;

	// Configuration periode des evenements
	// Prescaler : registre TIM6_ARR
	// 1000 /1000 = 1s
	TIM6->ARR = (uint16_t) 200 -1;

	// Activation auto-reload preload : prechargement
	// mettre a '1' le bit b7 du registre TIM6_CR1
	TIM6->CR1 |= (1<<7);

	// Demarrer le Timer TIM6
	// Mettre a '1' le bit b0 du registre TIM6_CR1
	TIM6->CR1 |= (1<<0);
}
```

4. Génération de signaux sinusoïdaux

Dans l'exemple ci-dessous, le DAC est utilisé pour produire une signal sinusoïdal.

```
#include "stm32f0xx.h"
#include "bsp.h"
#include "delay.h"
#include "main.h"

#include <math.h>

static void SystemClock_Config(void);

int main(void)
{
	float 			angle, y;
	uint16_t	output;

	// Configurer horloge du Systeme
	SystemClock_Config();

	//Initialiser le pin PA5 : LED
	BSP_LED_Init();

	//Initialiser le DAC
	BSP_DAC_Init();

	//Initialiser le TIM6 pour debordement chaque 200µs
	BSP_TIMER_Timebase_Init();
	// Initialiser la variable angle
	angle = 0.0f;
	while(1)
	{
		// Commencer la mesure du temps : mise a '1' de pin PA5
		BSP_LED_On();

		// Incrementer la valeur de angle value modulo 2*PI
		angle = angle + 0.01f;
		if (angle > 6.28f) angle = 0.0f;

		// caculer le sinus(angle)
		y = sinf(angle);

		// Offset et mise a echelle pour la sortie du DAC :
		// le registre du DAC est unsigned 12-bit
		output = (uint16_t)(0x07FF + (int16_t)(0x07FF * y));
		// Arreter la mesure du temps : mise a '0' de pin PA5
		BSP_LED_Off();
		// Envoyer la valeur calculer le DAC
		DAC->DHR12R1 = output;
		// Attendre le debordement du Timer TIM6 : (1 echantillon
chaque 200µs)
		while ((TIM6->SR & TIM_SR_UIF) == 0);
		TIM6->SR &= ~TIM_SR_UIF;
	}

}
```

Par soucis de clarté le corps de la fonction SystemClock_Config(); n'est pas répété ici.

Les échantillons sont calculés à la volée en utilisant la fonction **sinf()** de la bibliothèque **<math.h>. sinf()** est plus rapide que **sin()** car elle est écrite en utilisant le type **float simple précision** au lieu du type **float double précision**.

La variable d'angle va de 0 à 6,28 (2Pi, pour ceux qui se posent des questions) par pas de 0,01, ce qui nous donne 628 échantillons pour une période. Étant donné que nous avons une période de 200µs entre les échantillons, la période de l'onde sinusoïdale est :

$$f = \frac{1}{628 \times 200\mu} = 7.9617\ Hz$$

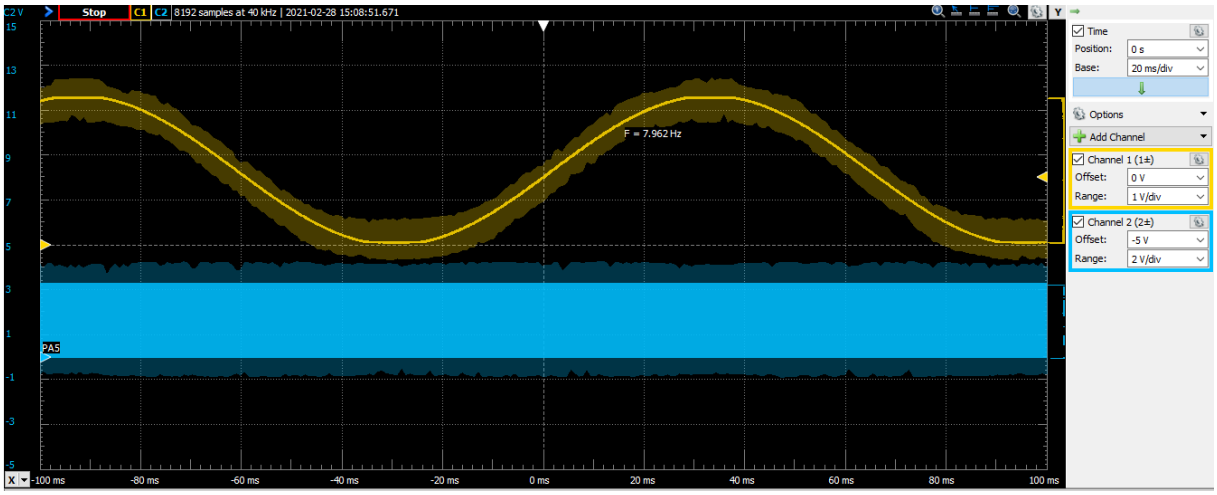
La fonction **sinf()** renvoie un nombre de type float **y** dans l'intervalle [-1 : 1]. Le registre de données DAC est un nombre de 12 bits non signé [0 : 4095] qui représente la valeur de la tension entre 0 et 3,3V. Par conséquent, pour une onde sinusoïdale de 3,3Vpp à pleine échelle, nous avons :

$$DAC_{output} = 2047 + (y \times 2047)$$

Enfin, notez que la LED sur le pin PA5 est utilisée pour mesurer le temps nécessaire au processeur pour calculer et sortir l'échantillon sinusal suivant. Nous pouvons l'utiliser pour nous assurer que le temps de traitement est inférieur à la période d'échantillonnage de 200µs. Si ce n'est pas le cas, nous avons un problème...

Tout devrait être clair maintenant. Sauvegardez tout, compiler et exécutez l'application pour tester.

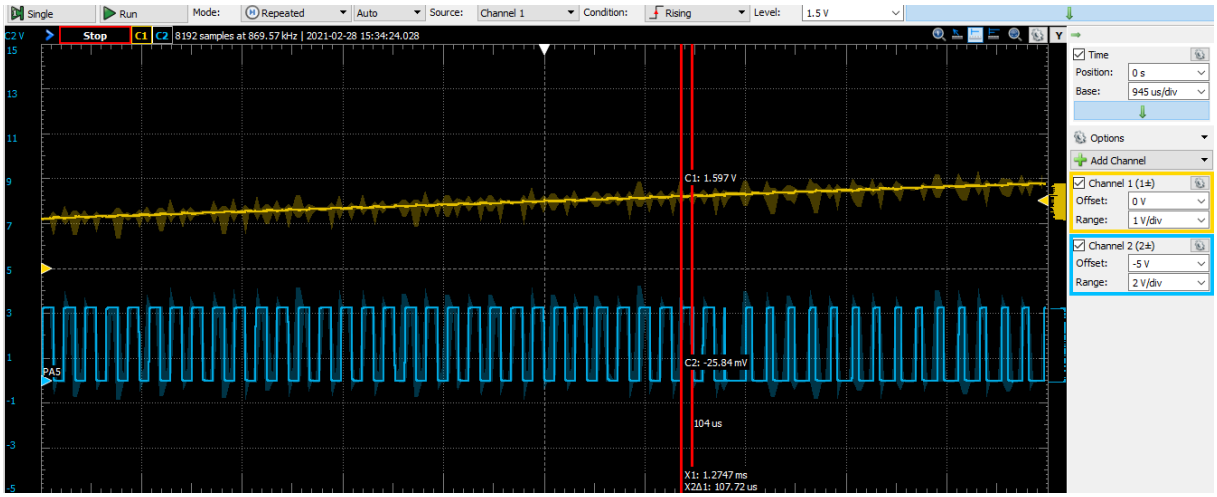
Sondez les pattes PA4 et PA5 avec un oscilloscope, et vous devriez avoir les graphiques de la figure ci-dessous. La fréquence et l'amplitude de l'onde sinusoïdale sont en parfait accord avec les valeurs prévues.



L'application utilise la broche de la LED pour fournir une mesure du temps nécessaire au calcul d'un nouvel échantillon. Cette méthode de mesure du temps d'exécution est simple et précise. Vous devez l'utiliser quand vous en avez besoin.

La sonde **PA5 (**canal 2 de l'oscilloscope) révèle que la partie mathématique du code nécessite entre 56µs et 104µs pour être complète selon la valeur de l'angle. Le temps que prend la fonction **sinf()** pour fournir le résultat n'est donc pas déterministe et dépend de son opérande. Il s'agit là d'un élément habituel que vous devez connaître.

L'application s'exécutera comme prévu tant que la période d'échantillonnage correspond au temps de traitement de l'échantillon le plus long. Ici, 200µs sont suffisants pour le calcul de l'échantillon. Dans le cas contraire, l'uniformité de l'échantillonnage est perdue.



Si vous voulez essayer la fonction **sin()** standard, vous devez modifier la base de temps pour permettre environ 300µs entre les événements de mise à jour. La fonction **sin()** prend entre **100µs et 200µs** pour être exécutée.

C'est un temps assez long, et c'est la raison pour laquelle les fonctions mathématiques doivent être utilisées avec précaution lorsque les contraintes de temps sont serrées. Sinon, vous pouvez utiliser une table de lookup (nous y reviendrons plus tard) ou, si vous êtes riche, vous acheter un meilleur CPU avec une unité à virgule flottante matérielle (FPU). Les processeurs Cortex-M4 en ont un...