Tutoriels STM32F0

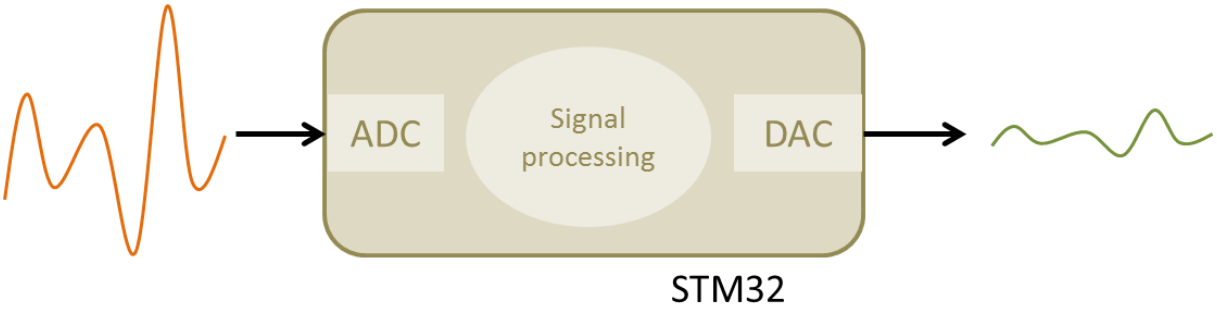
Application : Filtre numérique

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

Table des matières

1. Introduction

Le but de ce tutoriel est de vous guider pour la découverte de la conception d'un filtre numérique. Ce filtre numérique qui utilise le MCU STM32F072RB de la carte Nucleo, recevra un signal analogique appliqué à une des entrées Analogiques (ADC) du MCU STM32F072RB, puis traite les échantillons entrants, et enfin fournit une sortie analogique en utilisant un DAC.



Dans la première partie, nous fournissons quelques informations de base sur les filtres numériques. Vous aurez peut-être besoin de quelques connaissances préalables, principalement sur la transformée de Laplace et la transformée en z, pour bien comprendre certaines parties. Toutefois, vous pourrez tout de même acquérir quelques astuces utiles même si vous êtes nouveau dans le domaine du filtrage numérique.

2. Filtre numérique de premier ordre

2.1 Fonction de transfert & La transformée en *z* (TZ) ou domaine z

La représentation classique de la fonction de transfert pour un filtre passe-bas de premier ordre dans le domaine de Laplace est donnée par l'équation suivante :

$$H(p) = \frac{1}{1 + \tau \times p}$$

La fréquence de coupure d'un tel filtre est donnée par :

$$\omega_c = \frac{1}{\tau} = 2 \times \pi \times f_c$$

Supposons que nous voulons une fréquence de coupure $f_c = 2Hz$ Nous avons donc :

$$\tau = \frac{1}{2 \times \pi \times f_c} \approx 0.08\ s$$

$$H(p) = \frac{1}{1 + 0.08 \times p}$$

La variable de Laplace *p* (ou *s* en France)) est utile pour représenter les fonctions de transfert dans le domaine temporel continu. Lorsqu'il s'agit de systèmes numériques, nous échantillonnons généralement le signal entrant à des intervalles de temps réguliers. La transformée de Laplace n'est pas plus adaptée pour décrire un tel système dans le domaine temporel discret.

L'équivalent de la **transformée de Laplace** pour les signaux échantillonnés est appelé la **transformée z**.

Une approche classique pour passer une fonction de transfert du domaine de Laplace au domaine z consiste à utiliser ce qu'on appelle la "transformée bilinéaire". La transformée bilinéaire est obtenue en cartographiant le plan z en plan p à l'aide de la fonction de transfert de Laplace :

$$z = e^{pT}$$

où *T* est la période d'échantillonnage (en secondes). Dans ce tutoriel, nous utiliserons une période d'échantillonnage de **10 ms** (c'est-à-dire que l'ADC convertira une nouvelle valeur ou échantillons toutes les 10 ms).

La réciproque de la cartographie donne :

$$p = \frac{1}{T}\ln(z)$$

En utilisant le premier terme de développement limité de la fonction *ln*, nous avons :

$$p \approx \frac{2}{T} \times \frac{(z-1)}{(z+1)}$$

L'application de la transformation bilinéaire à notre filtre passe-bas de premier ordre donne alors :

$$H(p) = \frac{1}{1 + \tau \times p}$$

$$H(z) = \frac{1}{1 + \tau \times \frac{2}{T} \times \frac{(z-1)}{(z+1)}}$$

$$H(z) = \frac{1}{\frac{T \times (z+1) + 2 \times \tau \times (z-1)}{T \times (z+1)}}$$

$$H(z) = \frac{T \times z + T}{(T - 2 \times \tau) + z \times (T + 2 \times \tau)}$$

$$H(z) = \frac{\frac{T \times z}{(T + 2 \times \tau)} + \frac{T}{(T + 2 \times \tau)}}{\frac{(T - 2 \times \tau)}{(T + 2 \times \tau)} + z} = \frac{\alpha + \alpha \times z}{\beta + z}$$

Avec $T = 0.01s$ et $\tau = 0.08s$

$\alpha = \frac{T}{(T+2\times\tau)} = 0.0588235$ et $\beta = \frac{(T-2\times\tau)}{(T+2\times\tau)} = -0.8823529$

La fonction de transfert du domaine *z* de notre filtre passe-bas numérique, échantillonné à 100Hz (T=10ms), et ayant une fréquence de coupure de $f_c = 2\ Hz$ est donc :

$$H(z) = \frac{0.0588235 + 0.0588235 \times z}{z - 0.8823529}$$

2.2 Utilisation de Scilab® comme calculateur de fonction de transfert de domaine z

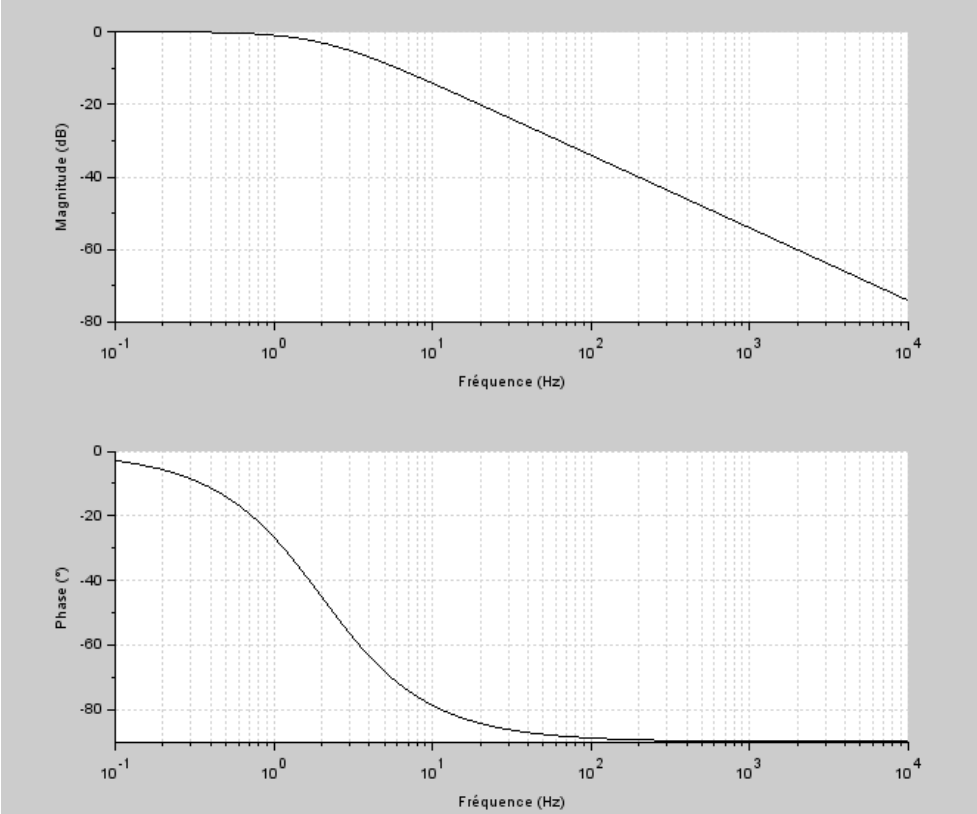
Heureusement pour vous, les logiciels de calcul scientifique tels que Matlab® ou Scilab® sont capables de calculer assez facilement la fonction de transfert des systèmes linéaires du domaine temporel continu au domaine temporel discret (et inversement).

Le script Scilab® ci-dessous effectue la même transformation bilinéaire que celle vue ci-dessus :

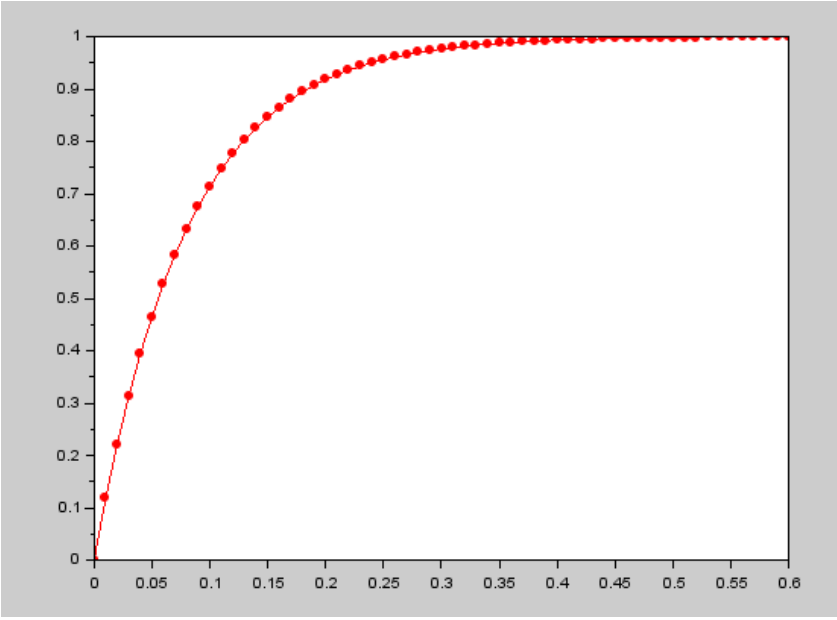
```
clear;

s=poly(0,'s');

sl=syslin('c',(1)/(1+0.08*s));  // Système dans le domaine temporel continu
slss=tf2ss(sl);                 // Tranposition dans espace des états

figure(1);                      // Diagramme de Bode
clf(1);
bode(sl, 0.1, 10000);

t=0:0.01:0.6;                   // Echelle temps
y = csim('step',t,slss);        // Simulation de la réponse à un échélon

figure(2);                      // Tracer la réponse
clf(2);
plot(t,y,'r.-');

sld=cls2dls(slss,0.01);         // Passage du domaine continu au domaine discret
sldt=ss2tf(sld);                // C
disp(sldt);                     // affiche la transformée z
```

L'exécution du script ci-dessus donne les résultats suivants. La fonction de transfert z obtenue confirme celle que nous avons calculée précédemment.

- Diagramme de Bode



- Réponse indicielle



- Fonction de transfert dans le domaine z

$$\frac{0.0588235 + 0.0588235z}{-0.8823529 + z}$$

2.3 De la fonction de transfert du domaine z à l'implémentation en code embarqué

La transformée z est une représentation dans le domaine discret de la fonction de transfert de notre filtre. Elle relie les échantillons de sortie aux échantillons d'entrée par :

$$H(z) = \frac{\alpha + \alpha \times z}{\beta + z} = \frac{Y(z)}{X(z)}$$

Ou $X(z)$ représente les échantillons en entrée du filtre et $Y(z)$ les échantillons en sortie de filtre.

Réécrivons cette équation de sorte qu'elle soit plus pratique du point de vue de la programmation :

$$\frac{Y(z)}{X(z)} = \frac{\alpha + \alpha \times z}{\beta + z}$$

$$Y(z) \times \beta + Y(z) \times z = \alpha \times X(z) + \alpha \times z \times X(z)$$

L'équation ci-dessus est calculée à chaque fois qu'un nouvel échantillon est disponible. Alors que *X(z)* représente l'échantillon actuel du signal entrant, *X(z)z* représente l'échantillon qui viendra (dans le futur) à la prochaine période d'échantillonnage. Plus généralement, multiplier *X(z)* ou *Y(z)* par z^n revient à mettre un indice *n* sur l'échantillon en cours de traitement.

- *n = 0* → échantillon courant
- *n = 1* → prochain échantillon
- *n = -1* → échantillon précédent

Comme aucun filtre ne peut prédire ce que sera l'avenir, nous pouvons diviser toutes les parties de l'équation ci-dessus par *z* pour avoir :

$$Y(z) \times \beta \times z^{-1} + Y(z) = \alpha \times X(z) \times z^{-1} + \alpha \times X(z)$$

$$Y(z) = \alpha \times X(z) \times z^{-1} + \alpha \times X(z) - Y(z) \times \beta \times z^{-1}$$

Le facteur z^{-1} représente un retard d'un échantillon. L'équation ci-dessus peut donc être écrite :

$$y_n = \alpha \times x_{n-1} + \alpha \times x_n - \beta \times y_{n-1}$$

$$y_n = \alpha \times (x_{n-1} + x_n) - \beta \times y_{n-1}$$

où :

- y_n est l'échantillon de sortie calculé à la $n^{ième}$ itération
- y_{n-1} est l'échantillon de sortie calculé à la $(n-1)^{ième}$ itération
- x_n est l'échantillon d'entrée lu à la $n^{ième}$ itération
- x_{n-1} est l'échantillon d'entrée lu à la $(n-1)^{ième}$ itération

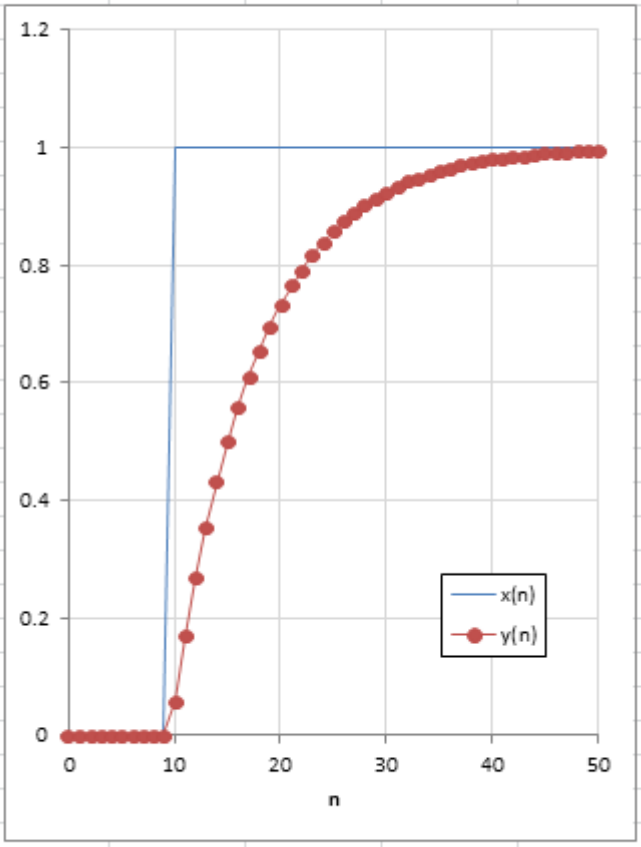
L'équation ci-dessus est équation récursive, et peut être directement appliquée au calcul du filtre dans un programme.

Par exemple, vous pouvez expérimenter l'équation ci-dessus avec Excel (ou tout autre outil de votre choix) :

- Créer une colonne d'exemples d'indices *n*
- Créez une colonne de *x* échantillons d'entrée (certains "0" et d'autres "1" pour simuler l'échelon)
- Créer une colonne pour *y* : résultats de l'équation ci-dessus
- Tracez les données *x* et *y* avec *n* comme axe des abscisses

C3 f_x =(0.0588235*B2)+(0.0588235*B3)-(-0.8823529*C2)

	A	B	C
1	n	x(n)	y(n)
2	0	0	0
3	1	0	0
4	2	0	0
5	3	0	0
6	4	0	0
7	5	0	0
8	6	0	0
9	7	0	0
10	8	0	0
11	9	0	0
12	10	1	0.0588235
13	11	1	0.16955009
14	12	1	0.26725001
15	13	1	0.35345582
16	14	1	0.42951977
17	15	1	0.49663501
18	16	1	0.55585434
19	17	1	0.60810669
20	18	1	0.6542117
21	19	1	0.69489259
22	20	1	0.7307875
23	21	1	0.76245947
24	22	1	0.79040532
25	23	1	0.81506343
26	24	1	0.83682058
27	25	1	0.85601806
28	26	1	0.87295702



En regardant la réponse à l'étape ci-dessus, vous pouvez reconnaître un filtre passe-bas de premier ordre. De plus, vous remarquerez que le temps n'est guère présent puisque chaque échantillon de sortie est calculé après l'échantillon précédent et l'échantillon d'entrée. Cela devrait vous faire comprendre que la constante de temps réelle du filtre τ dépend maintenant de l'intervalle de temps entre chaque calcul d'échantillon, c'est-à-dire de la période d'échantillonnage *T*, de sorte que l'axe des abscisses peut être modifié de *n* l'indice de l'échantillon au temps *t*.

2.4 Simplifions encore un peu

Dans l'équation précédente, nous observons que chaque calcul d'échantillon de sortie nécessite :

- Une addition → $x_n + x_{n-1}$
- 2 multiplications → $\alpha \times (x_n + x_{n-1})$ et $\beta \times y_{n-1}$
- Une autre addition → $\alpha \times (x_n + x_{n-1}) + -\beta \times y_{n-1}$

Si nous supposons que la fréquence d'échantillonnage est élevée par rapport à la fréquence du signal d'entrée (ce qui est généralement le cas), nous pouvons encore simplifier notre équation en supposant que x_{n-1} et x_n ne sont pas très différents.

$$y_n = \alpha \times (x_{n-1} + x_n) - \beta \times y_{n-1}$$

En posant $x_n = x_{n-1}$

on écris :

$$y'_n = 2 \times \alpha \times x_n - \beta \times y_{n-1}$$

Nous pouvons aussi remarquer :

$2 \times \alpha = \frac{2 \times T}{(T+2 \times \tau)}$ et $\beta = \frac{(T-2 \times \tau)}{(T+2 \times \tau)}$

$$\beta + 1 = \frac{(T - 2 \times \tau)}{(T + 2 \times \tau)} + 1 = \frac{(T - 2 \times \tau) + (T + 2 \times \tau)}{(T + 2 \times \tau)} = \frac{2 \times T}{(T + 2 \times \tau)} = 2 \times \alpha$$

$$\beta + 1 \ = 2 \times \alpha$$

L'équation ci-dessus est réécrite :

$$y'_n = (\beta + 1) \times x_n - \beta \times y_{n-1}$$

Posons :

$$\gamma = \ \beta + 1$$

$$y'_n = \gamma \times x_n (1 - \gamma) \times y_{n-1}$$

Avec

$$\gamma = \frac{2 \times T}{(T + 2 \times \tau)}$$

Nous avons supprimé une opération d'addition. Utilisons à nouveau :

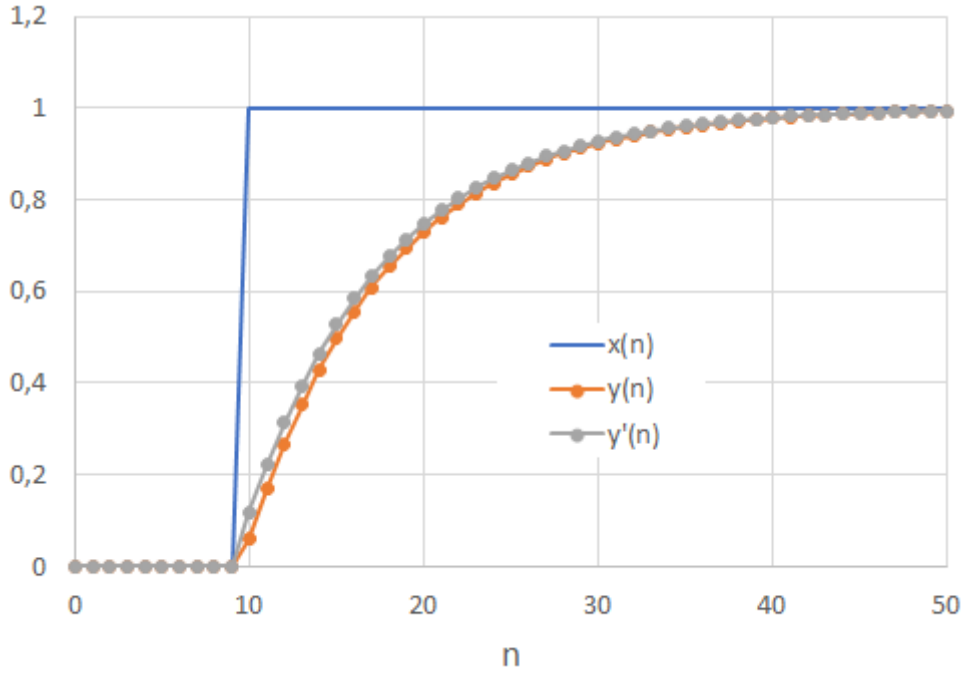
$T = 0.01s$ et $\tau = 0.08s$

Nous obtenons finalement

$$y'_n = 0.11765 \times x_n + 0.88235 \times y_{n-1}$$

Le graphique ci-dessous compare les réponses indicielles de la première version (***y***) et de la version simplifiée (***y'***) de notre filtre. Sur la base de ce résultat, nous pouvons accepter l'hypothèse que $\mathbf{x_{n-1} \approx x_n}$ et utiliser la forme simplifiée du filtre. Nous pouvons également voir que **63% de la valeur finale de sortie** est atteinte à l'échantillon **n°18.** C'est-à-dire 8 échantillons après l'application de l'échelon. Avec une période d'échantillonnage de 0,01s, nous confirmons que τ=0,08. Il est maintenant clair que la période d'échantillonnage détermine fortement la fréquence de coupure du filtre. En d'autres termes, l'exécution

de l'équation ci-dessus à un taux d'échantillonnage différent donnerait une fréquence de coupure différente.



3. Implémentation du filtre : étape 1 – « Suiveur Analogique »

Avant de passer à l'implémentation du filtre, préparons une application qui copie simplement la tension d'entrée d'un canal de l'ADC vers une sortie DAC. Les sections suivantes passent en revue la configuration nécessaire pour les périphériques concernés.

Cloner le projet « **Tutoriels_Interruptions** » en nomma le projet cloné, «**Tutoriel_FiltreNumerique**».

3.1 Configuration de ADC

Vous trouverez ci-dessous la configuration de l'ADC. Il s'agit d'une conversion sur un canal. C'est la même confiuration que celle utilisée dans le tutoriel concernant ADC.

```
/*
 * ADC_Init()
 * Initialise ADC pour la conversion sur un seul canal
 * canal 11 -> pin PC1
 */

void BSP_ADC_Init()
{
	// Activation de horloge du GPIOC
	// Mettre a  '1' le bit b19 du registre RCC_AHBENR
	RCC->AHBENR |= (1<<19);

	// Configure le pin PC1 en mode Analog
	// Mettre à "11" les bits b3b2 du registre GPIOC_MODER
	GPIOC->MODER |= (0x03 <<1);

	// Activation de horloge de ADC
	RCC->APB2ENR |= (1<<9);

	// Reset de la configuration de ADC
	// Mise a zero des regsitres de configuration de ADC
	ADC1->CR 	= 0x00000000;
	ADC1->CFGR1  = 0x00000000;
	ADC1->CFGR2  = 0x00000000;
	ADC1->CHSELR = 0x00000000;

	// Choix du mode de conversion
	// '1' : conversion une en continue
	ADC1->CFGR1 |= (1<<13);

	// Choix de la resolution (nombre de bits des data)
	//	12 bits
	ADC1->CFGR1 &= ~(0x03 <<4);

	// Choix de la source horloge pour ADC
	ADC1->CFGR2 |= (0x01 <<31UL);

	// Choix de la periode echantillonnage
	// Prenons 28.5 ADC clock cycles
	ADC1->SMPR = 0x03;

	// Sélectionner le canal 11 pour la conversion
	ADC1->CHSELR |= ADC_CHSELR_CHSEL11;

	// Activer ADC
	// Mettre a '1' le bit b0 du registre ADC_CR
	ADC1->CR |= (1<<0);

	// Demarrer la converion
	// Mettre a '1' le bit b2 du registre ADC_CR
	ADC1->CR |= (1<<2);
}
```

3.2 Configuration du DAC

Nous utilisons la configuration du DAC utilisée dans le tutoriel DAC. La configuration est rappelée ci-dessous :

```
/*
 * 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;
}
```

3.3 Configuration de la base de temps

Pour le cadencement, nous utilisons le Timer TIM6. Nous activons la demande d'interruption de débordement du TIM6. La période de débordement sera fixée à **10 ms**.

```
/*
 * BSP_TIMER_Timebase_Init()
 * TIM6 cadenser a 48MHz
 * Prescaler   = 48000 -> periode de comptage = 1ms
 * Auto-reload = 10  -> periode de debordement    = 10ms
 */

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);          // le bit b4 est defini comme etant
RCC_APB1ENR_TIM6EN

        // Faire un Reset de configuration du TIM6 : mise a zero des registres
        // 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) 48000 -1;

        // Configuration periode des evenements
        // Prescaler : registre TIM6_ARR
        // 1000 /10 = 100Hz
        TIM6->ARR = (uint16_t) 10 -1;

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

        // Activation de la demande interruption de debordement
        // mettre a '1' le bit b0 du registre TIM6_DIER
        // page 544
        TIM6->DIER|= (1<<0);

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

Nous configurons également le NVIC pour autoriser ou accepter les demandes d'interruptions du Timer TIM6 avec priorité 1 :

```
/*
 * BSP_NVIC_Init()
 * Configuration du controleur NVIC pour autoriser et accepter les sources interruptions activer
 */

void BSP_NVIC_Init()
{
	// Mettre la priorite 1 pour les interruptions du TIM6
	NVIC_SetPriority(TIM6_DAC_IRQn, 1);

	// Autoriser les demandes interruptions du TIM6
	NVIC_EnableIRQ(TIM6_DAC_IRQn);
}
```

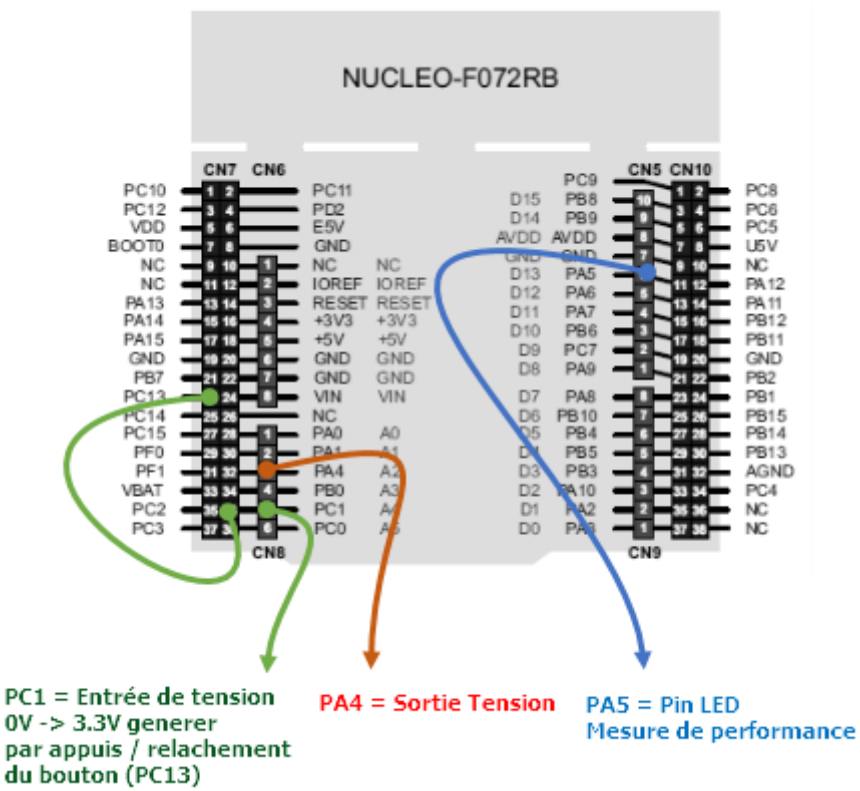
3.4 Implémentation du suiveur

Enfin, la fonction principale **main()** mettant en œuvre le suiveur analogique est donnée ci-dessous. Les variables ***in*** et ***out*** ont été définies comme globales afin de permettre le suivi avec STM-Studio® ou autre méthode :

```
// variables globales
uint8_t timebase_irq = 0;
uint16_t        in, out;

int main(void)
{
        // Configure System Clock for 48MHz from 8MHz HSE
        SystemClock_Config();

        //Initialisation de la Led verte
        BSP_LED_Init();
        // initialisation de la base de temps de 10 ms
        BSP_TIMER_Timebase_Init();
        BSP_NVIC_Init();
        // Initialisation de ADC
        BSP_ADC_Init();
        // Initialisation de DAC
        BSP_DAC_Init();
        while(1)
        {

                // Executer tout les 10 ms
                if (timebase_irq == 1)
                {
                        // Demmarrer la mesure de performance
                        BSP_LED_On();
                        // Lecture entree ADC
                        while ( (ADC1->ISR & ADC_ISR_EOC) != ADC_ISR_EOC );
                        in = ADC1->DR;

                        // etage de suivi : copie de entree ADC sur la sortie DAC
                        out = in;

                        // envoi de la donnee sur la sortie DAC
                        DAC->DHR12R1 = out;

                        // Arret la mesure de performance
                        BSP_LED_Off();
                        timebase_irq = 0;
                }
        }
}
```

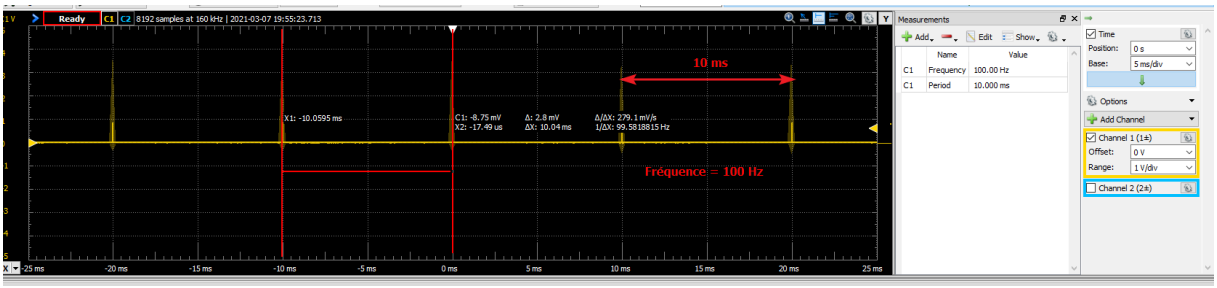
Afin de tester le suiveur analogique, connectons la broche **PC13** du bouton utilisateur à la broche **PC1** de l'entrée analogique de l'**ADC**. C'est une façon pratique de générer des échelons ou pulses de tension pleine échelle (**0V → 3.3V**) à la broche d'entrée du filtre **PC1**. Assurez-vous que vous n'avez pas de code traitant l'événement du bouton poussoir. Vous n'avez même pas besoin d'appeler la fonction ***BSP_PB_Init()*** car toutes les broches sont configurées en entrée (Input) lors de la réinitialisation et nous n'allons pas lire l'état du bouton de toute façon.

Sauvegarder le projet, compiler et téléverser dans le MCU.

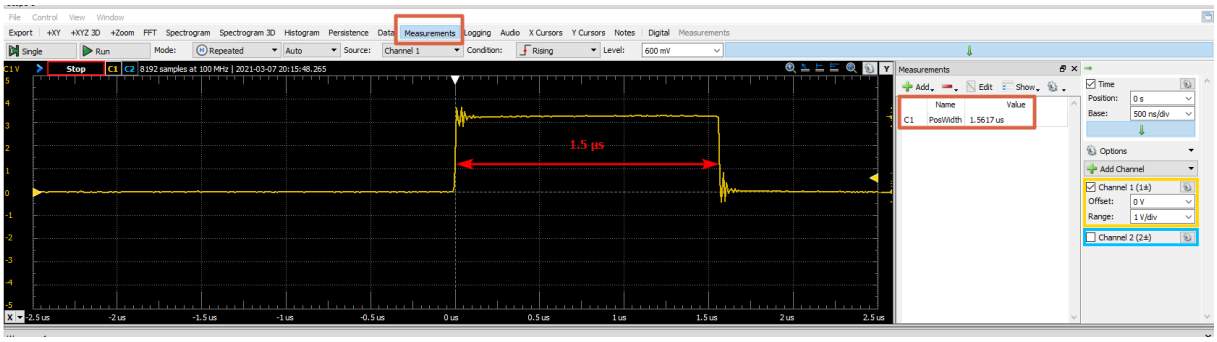


Sauvegarder le projet, compiler et téléverser dans le MCU.

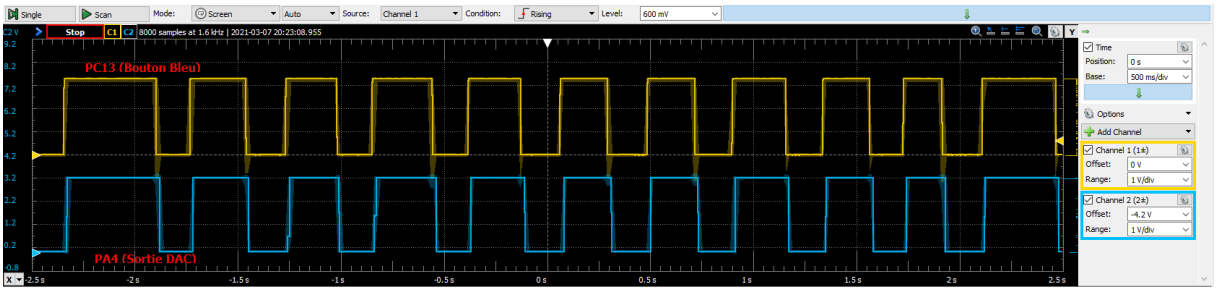
- Avec un oscilloscope sonder le pin PA5, vous devriez voir une courte impulsion qui se produit toutes les 10 ms, selon les réglages de la base de temps :



- En zoomant sur l'axe des abscisses (**base de temps = 500ns/div**), la largeur de l'impulsion **PA5** vous indique le temps nécessaire pour exécuter la boucle de filtrage. Dans cet exemple, le processus prend **1,5 µs** pour se terminer. N'oubliez pas que pour l'instant il n'y a pas de traitement (pas de filtre). Nous ne faisons rien de plus que sortie = entrée (suiveur analogique). Il faut que ce soit court... Utiliser l'outil **measurements** de Waveforms pour mesurer la largeur d'impulsion (demander l'aide de l'encadrant de TP si besoin).



En sondant à la fois **PC1** et **PA4** et en appuyant sur le bouton-poussoir bleu, vous pouvez vous montrer que la sortie **DAC** sur **PA4** est une copie de l'entrée **ADC sur PC1** (tension de **PC13** (Bouton bleu)).



Vous pouvez également surveiller les variables d'entrée et de sortie avec STM-Studio® tout en jouant avec le bouton utilisateur (uniquement à titre d'informations) :

Lorsque vous avez vérifié que tout fonctionne comme prévu, vous êtes prêt à commencer la mise en œuvre du filtre.

4. Implémentation du filtre numérique

Souvenez-vous que notre filtre est décrit par l'équation récursive suivante :

$$y_n = 0.11765 \times x_n + 0.88235 \times y_{n-1}$$

$$y_n = C_1 \times x_n + C_2 \times y_{n-1}$$

Dans un premier temps, nous allons mettre en œuvre cette équation directement en utilisant des nombres de type virgule flottante, float, pour les deux coefficients c_1 et c_2.

4.1 Arithmétique virgule flottante

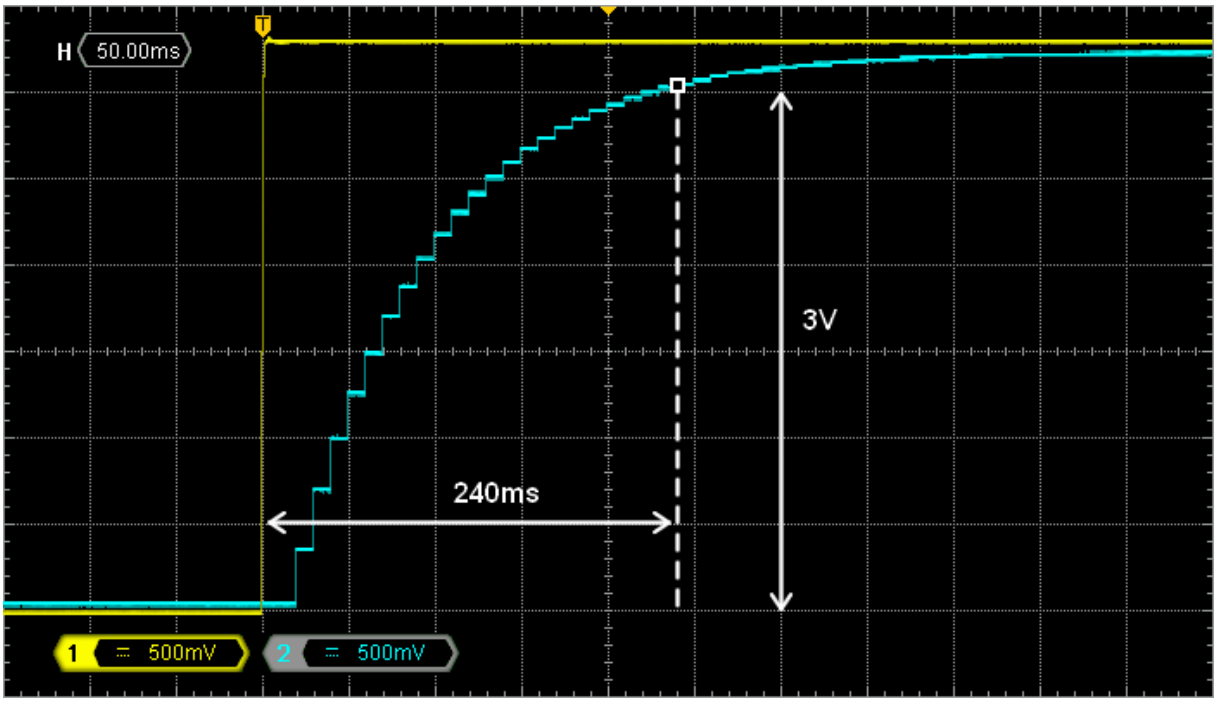
Modifiez la fonction **main()** comme indiqué ci-dessous.

```
int main(void)
{
	float c1, c2;		// les coef du filtre
	float x = 0;			// Entree (Input) du Filtre
	float y = 0;			// Sortie (Output) du filtre

	// configuration des coefs du filtre
	c1 = 0.11765f;
	c2 = 0.88235f;
	// Configure System Clock for 48MHz from 8MHz HSE
	SystemClock_Config();

	//Initialisation de la Led verte
	BSP_LED_Init();
	// initialisation de la base de temps de 10 ms
	BSP_TIMER_Timebase_Init();
	BSP_NVIC_Init();
	// Initialisation de ADC
	BSP_ADC_Init();

	// Initialisation de DAC
	BSP_DAC_Init();
	while(1)
	{
		// Executer tout les 10 ms
		if (timebase_irq == 1)
		{
			// Demmarrer la mesure de performance
			BSP_LED_On();
			// Lecture entree ADC
			while ( (ADC1->ISR & ADC_ISR_EOC) != ADC_ISR_EOC );
			in = ADC1->DR;
			// etage de filtrage
			x = (float)in;
			y = (c1*x) + (c2*y);
			out = (uint16_t)y;
			// envoi de la donnee sur la sortie DAC
			DAC->DHR12R1 = out;
			// Arret la mesure de performance
			BSP_LED_Off();
			timebase_irq = 0;
		}
	}
}
```

Notez que dans le code l'équation **y *= (c1*x) + (c2*y)*** nous avons déjà y_n à gauche de **'='** et y_{n-1} à droite.

Vous pouvez utiliser STM-Studio® pour obtenir un aperçu rapide du résultat :

Pour des mesures précises, il nous utilisera un oscilloscope (analog discovery). Sondez les broches d'entrée (PC1) et de sortie (PA4) :

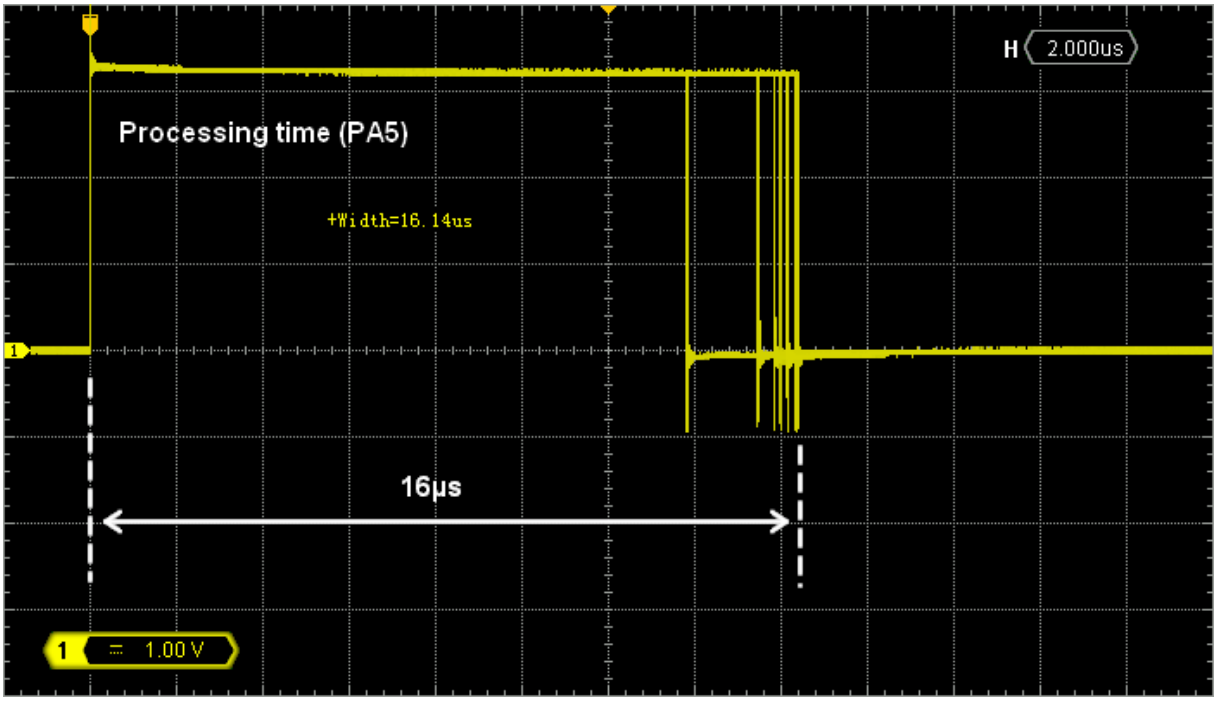


N'oubliez pas que nous voulons **τ =80ms**. Environ 95% de la valeur finale devrait être atteinte dans un délai de 3τ = 240ms. La figure ci-dessus est une vue détaillée de la réponse à l'étape (bouton-poussoir). Nous pouvons voir que le filtre se comporte parfaitement comme prévu. La valeur finale du filtre est presque atteinte après 5τ.

Ensuite, il y a la mesure du temps de traitement du filtre par le pin PA5. Dans la figure ci-dessous, nous constatons tout d'abord que le filtre est traité assez rapidement par rapport à la période d'échantillonnage

(10ms), ce qui signifie que le CPU passe la plupart du temps à ne rien faire, en attendant la prochaine interruption de la base de temps.

Nous pouvons mesurer un temps d'exécution d'environ 16µs :



Si vous jouez avec le bouton-poussoir, vous remarquerez que le temps de traitement n'est pas le même lorsque le signal d'entrée est à l'état haut ou bas. Il est en fait un peu plus rapide (13,4µs) lorsque le signal d'entrée est à 0 (bouton enfoncé) car les opérations arithmétiques sont probablement simplifiées avec des opérandes nuls. Par conséquent, on peut voir que le temps de calcul n'est pas déterministe, et dépend des valeurs des opérandes. C'est une conséquence des branches conditionnelles dans le code assembleur qui résultent de la synthèse arithmétique du type float (virgule flottante).

Enregistrer vos codes et résultats de mesures pour le rapport à soumettre.

4.2 Arithmétique virgule fixe

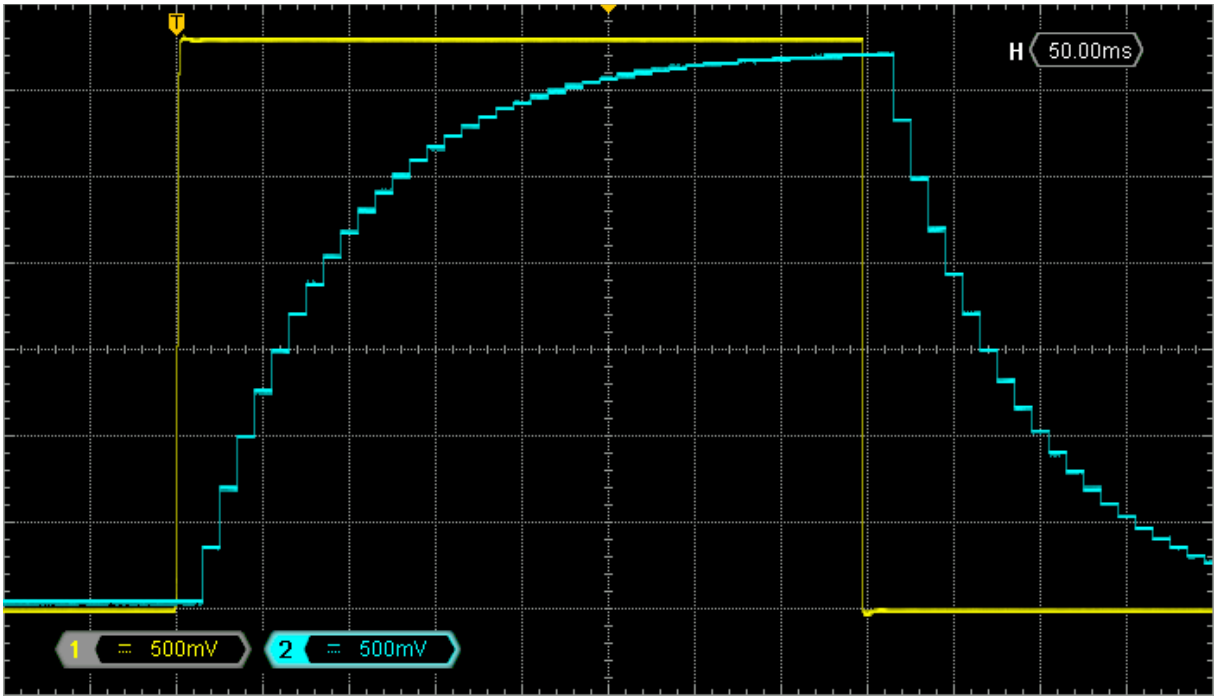
L'utilisation du type **float** pour représenter des nombres réels tels que nos coefficients c1 et c2 est une chose naturelle, mais vous savez peut-être que les nombres réels peuvent également être représentés par des nombres entiers, en supposant une virgule virtuelle. Une telle représentation est appelée **représentation à virgule fixe** ("float" étant pour la représentation à virgule flottante).

Nous savons que le processeur **Cortex-M0** ne dispose pas de matériel dédié pour effectuer des calculs arithmétiques en virgule flottante (**Floating Point Unit, FPU**). Il est donc probable que l'utilisation du type float ralentit l'exécution du code (c'est-à-dire que l'arithmétique en virgule flottante produit de nombreuses lignes de code assembleur à exécuter).

Essayez le code ci-dessous où les variables de filtrage sont représentées **en virgule fixe**.

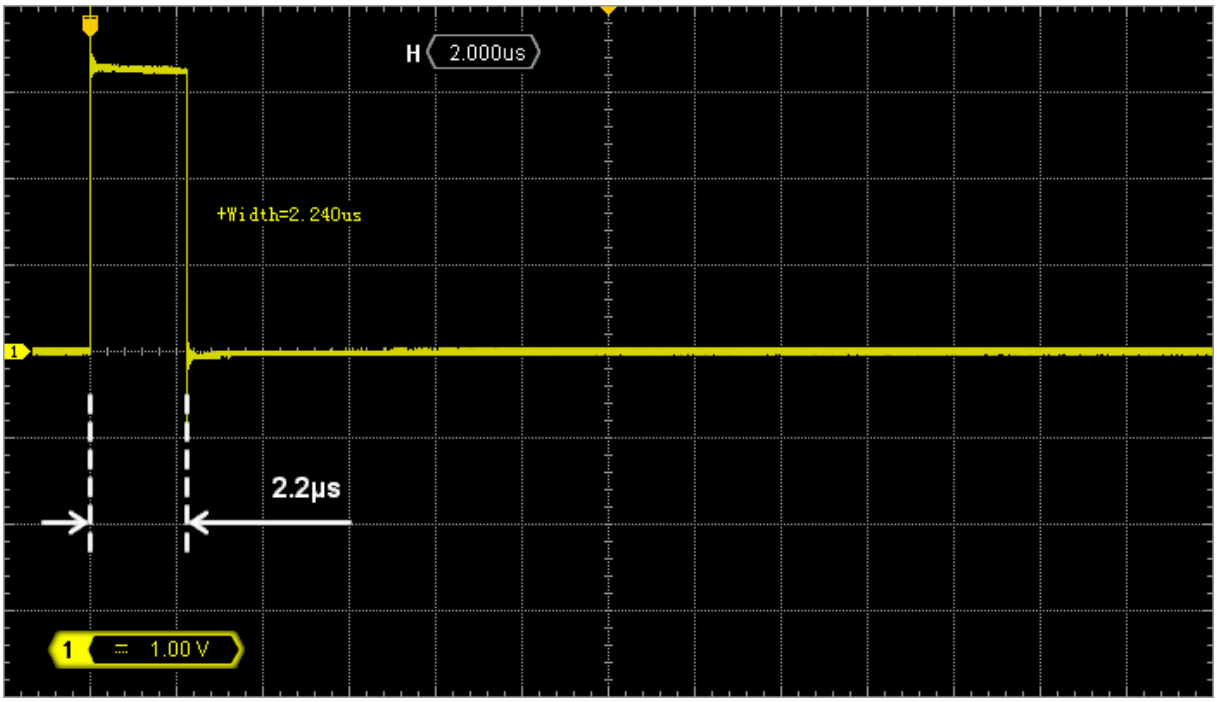
```c
// variables globales
uint8_t timebase_irq = 0;
uint16_t        in, out;
int main(void)
{
        float c1, c2;       // les coef du filtre
        uint16_t x = 0;                 // Entree (Input) du Filtre
        uint32_t y = 0;                 // Sortie (Output) du filtre
        // configuration des coefs du filtre
        c1 = 482;                       // 0.11765f *2^12
        c2 = 3614;                      // 0.88235f *2^12
        // Configure System Clock for 48MHz from 8MHz HSE
        SystemClock_Config();
        //Initialisation de la Led verte
        BSP_LED_Init();
        // initialisation de la base de temps de 10 ms
        BSP_TIMER_Timebase_Init();
        BSP_NVIC_Init();
        // Initialisation de ADC
        BSP_ADC_Init();
        // Initialisation de DAC
        BSP_DAC_Init();
        while(1)
        {
                // Executer tout les 10 ms
                if (timebase_irq == 1)
                {
                        // Demmarrer la mesure de performance
                        BSP_LED_On();
                        // Lecture entree ADC
                        while ( (ADC1->ISR & ADC_ISR_EOC) != ADC_ISR_EOC );
                        in = ADC1->DR;
                        // etage de filtrage
                        x = (float)in;
                        y = (c1*x) + (c2*y);
                        y = (y >>12U);          // retrait de la partie decimale
                        out = y;

                        // envoi de la donnee sur la sortie DAC
                        DAC->DHR12R1 = out;
                        // Arret la mesure de performance
                        BSP_LED_Off();
                        timebase_irq = 0;
                }
        }
}
```

Assurez-vous que le filtre se comporte exactement comme avant (c'est-à-dire lors de l'utilisation du type float) :



Utilisez la broche PA5 pour mesurer le temps de traitement du filtre. Vous constaterez que cette implémentation est environ 8x plus rapide (2µs) que celle qui utilise le type float. Pour le même comportement ! Jouez avec le bouton utilisateur et vous remarquerez également que le temps de traitement est maintenant déterministe (c'est-à-dire que le temps de calcul ne dépend pas des opérandes).

Enregistrer vos codes et résultats pour le rapport.



5. Résumé

Dans ce tutoriel, vous avez mis en place un filtre passe-bas numérique. Avec un peu d'aide de Scilab®, vous devriez être capable de synthétiser dans le domaine numérique n'importe quel filtre tant que vous avez sa fonction de transfert.

Le tutoriel vous a fait découvrir deux approches de codage, la seconde étant 8x plus rapide que la première pour un résultat identique.

L'efficacité du traitement dépend de vos compétences en matière de codage. Elle a des conséquences sur les performances de l'application et plus généralement sur l'entreprise (argent).

Un code plus efficace vous le permet :

- d'acheter un processeur moins puissant et moins cher pour la même tâche
- de réduire la vitesse de l'horloge du MCU et donc de réduire la consommation d'énergie (c'est-à-dire augmenter la durée de vie de la batterie dans le cas d'une application mobile, réduire les besoins de dissipation de chaleur...)
- passer plus de temps dans les modes à faible consommation d'énergie

Ces avantages ne sont pas seulement la "cerise sur le gâteau". Il est fondamental que vous produisiez un code efficace chaque fois que vous le pouvez.