1 TABLE DES MATIERES

2 Expérimentation avec le DAC N°1 d'une carte Nucléo-L476RG 2

3 Initialisation de l'environnement 2

4 Informations à connaître avant d'utiliser un périphérique 3

4.1 Première source d'information : les documents pdf sur le site st.com 3

4.1.1 Description de quelques chapitres du document EM1884 3

4.1.2 Comment utiliser les API du DAC 5

4.2 Deuxième source d'information : commentaires des fichiers « .c » et « .h » 5

4.2.1 Sur les DACs 5

4.2.2 Sur le mode d'utilisation des triggers 6

4.2.3 Sur la fonction buffer 6

4.2.4 Pour utiliser la sortie DAC vers un composant interne du microcontrôleur 6

4.2.5 Interférences entre les deux DAC 6

4.2.6 Mode normal et mode échantillonnage et maintient 6

4.2.7 Fonctions de génération de formes d'ondes 7

4.2.8 Format des données utilisées 7

4.2.9 Correspondance entre les valeurs numériques et les tensions. 7

4.2.10 Comment utiliser le driver DAC 7

5 L'étape 1 8

5.1 Conception 8

5.2 Programmation 8

6 l'étape 2 8

6.1 Conception 8

6.2 Programmation 9

8 l'étape 3 10

8.1 Conception 10

8.2 Nouvelle configuration dans STM32CubeMX 10

8.3 Programmation 11

2 EXPERIMENTATION AVEC LE DAC N°1 D'UNE CARTE NUCLEO-L476RG

Nous allons progresser par étapes :

Etape 1 : mise en œuvre du DAC qui délivrera une tension constante de 2,5V.

Etape 2 : mise en œuvre du DAC avec évolution progressive de la tension entre deux bornes.

Etape 3 : mise en œuvre du DAC et du DMA qui déchargera le processeur de la gestion du DAC.

3 INITIALISATION DE L'ENVIRONNEMENT

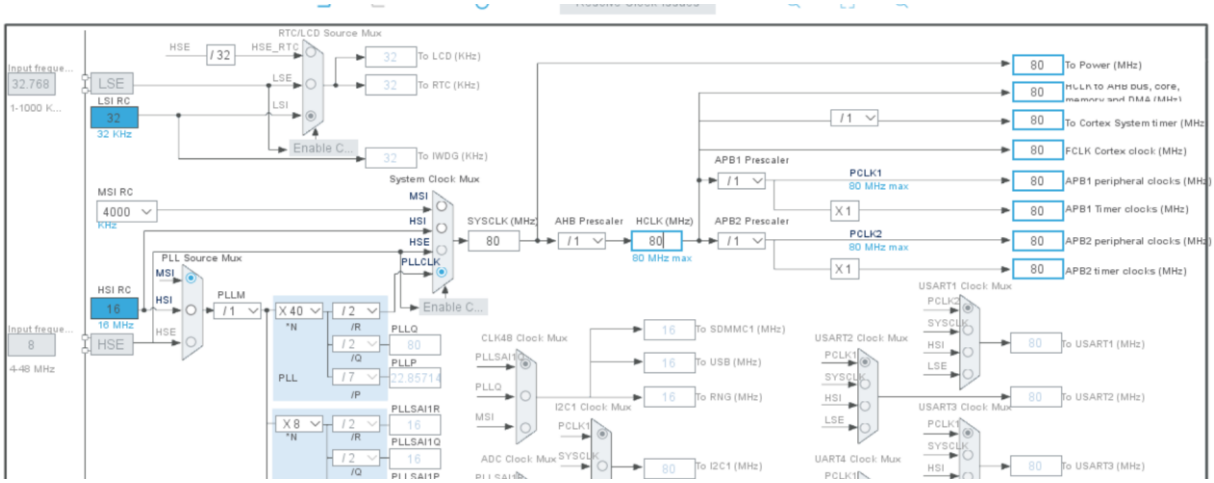
STM32CubeMX peut grandement soulager l'utilisateur en générant le maximum de code à sa place, il suffit pour cela de lui indiquer nos choix en matière de :

- carte Nucléo, ici ce sera le modèle Nucléo-L476RG ;
- préférences pour la génération automatique de code ;
- périphérique à mettre en œuvre, ici ce sera le DAC1 ;
- fréquence d'horloge, on prendra la fréquence maximale du STM32L476RG : 80 MHz ;
- nom et endroit de stockage du projet.
- Choix de l'IDE, ici ce sera MDK-ARM.

Voici comment lui indiquer ce que nous souhaitons :

1) Lancer STM32CubeIDE.
2) FILE/NewProject.
3) Dans « Part Number Search », choisir STM32L476RG.
4) Cliquer dans la fenêtre de droite sur la carte Nucléo correspondante.
5) Cliquer sur le bouton « Start Project ».
6) « *Initialize all peripherals with their default Mode* » ➔ répondre Yes
7) Dans l'onglet « Pinout & Configuration »
    a. Développer Analog et cliquer sur DAC1
        i. Dans la partie Mode
            1. Out1 mode choisir Connected to external pin only.
        ii. Dans la partie Configuration
            1. Cliquer sur l'onglet Parameter Settings ➔ dans la partie pinout view vérifier que DAC1_OUT1 est assigné à PA4.
            2. Vérifier aussi qu'il n'y a pas de trigger positionné, on s'en occupera lors d'une prochaine étape.

8) Onglet Clock Configuration
    a. Dans le cadre HCLK (MHz) au centre de l'écran, entrer la fréquence maximale 80 MHz et valider par ENTER.
    b. Laisser le système chercher la meilleure configuration pour les horloges. Dans le cas du STM32L476RG, on obtient 80 MHz pour la fréquence APB1 sur la droite du schéma (*c'est la fréquence choisie pour le timer que nous utiliserons dans une prochaine étape*).



9) Onglet Project Manager
    a. Onglet horizontal « Project » ➔ choisir l'IDE Toolchain/IDE: MDK-ARM pour ce qui me concerne.
    b. Onglet horizontal « Code Generator » ➔ cocher « *Generate peripheral initialization as a pair of '.c/h' files per peripheral* »
10) Fenêtre principale de STM32CubeMX
    a. Faire « FILE/SAVE » et donner un nouveau nom au projet actuel : D:\Documents\Informatique\Programmes\STM32\DAC_Test2
    b. Appuyer sur le bouton Generate Code.
    c. Dans le popup qui s'affiche appuyer sur le bouton « Open Project »…
11) L'IDE choisi s'ouvre…
    a. Dans l'arborescence DAC_Test2/Application/User/, cliquer sur main.c
    b. Visualiser le fichier main.c qui a été généré.

4 INFORMATIONS A CONNAITRE AVANT D'UTILISER UN PERIPHERIQUE

4.1 PREMIERE SOURCE D'INFORMATION : LES DOCUMENTS PDF SUR LE SITE ST.COM

La documentation est très fournie, mais ça demande une grande attention. Nous verrons qu'il y a une autre façon d'obtenir des informations intéressantes pour la mise en œuvre, comme les commentaires présents dans les divers fichiers générés par STM32CubeMX. Prenons quand-même un peu de temps pour parcourir la description des API.

Les acronymes utilisés ci-dessous sont décrits dans le fichier UM1884 que vous trouverez facilement sur le site st.com.

Pour utiliser le DAC, mais c'est également valable pour les autres périphériques, il faut aller dans la documentation UM1884 décrivant les API HAL.

4.1.1 Description de quelques chapitres du document EM1884

a. Le chapitre 2.1.2 indique que STM32Cube génère automatiquement un certain nombre de choses pour nous :

i. Les sources HAL + CMSIS et BSP drivers qui sont les composants minimaux pour développer du code avec une carte donnée.
ii. Les chemins vers tous les composants utilisés.
iii. La configuration des périphériques.
iv. L'appel de la fonction de HAL_Init().
v. Implémentation de SysTickISR pour la fonction HAL_GetTick(). *SysTick = génération d'une interruption à intervalle régulier à l'usage des ISR = Interrupt Service Routine*.
vi. La configuration des horloges systèmes conformément aux choix faits dans l'onglet « *clock configuration* » de STM32Cube.

b. Le chapitre 2.2 détaille les structures de données pour chaque driver HAL.
c. Le chapitre 2.4, est une matrice permettant d'identifier les périphériques disponibles selon les différents modèles de microcontrôleur Cortex. Par exemple on y trouvera les IP/modules relatifs aux DACs :
    i. stm32l4xx_hal.c ;
    ii. stm32l4xx_hal_dac.c ;
    iii. stm32l4xx_hal_dac_ex.c ➔ Uniquement si on utilise les 2 DAC à la fois.
d. Le chapitre 2.5.1, donne les règles de nommage des API concernant les :
    i. Les noms de fichiers.
    ii. Les noms de modules.
    iii. Les noms de fonctions.
    iv. Les noms de handles.
    v. Les noms des structures d'initialisation.
    vi. Les noms des énumérations.
e. Le chapitre 2.5.2, donne les règles de nommage générales HAL.
f. Le chapitre 2.5.3, indique que les gestionnaires d'interruption sont générés dans le fichier stm32l4xx_it.c. Chaque fonction « user callback » y doit être définie par l'utilisateur.
g. Le chapitre 2.6 présente les API génériques des fonctions s'appliquant à tous les périphériques.
    i. Fonctions **d'initialisation/désinitialisation**
        1. **Initialisation** : HAL_PPP_Init() qui permet d'initialiser un périphérique et de configurer les ressources de bas niveau, principalement les horloges, GPIO, fonctions alternatives (AF) et éventuellement le DMA.
        2. **désinitialisation** : HAL_PPP_Deinit() qui permet d'interrompre un périphérique en restaurant son état, en libérant les ressources de bas niveau et en supprimant toute dépendance directe avec le matériel.
    ii. Fonctions d'**entrée/sortie**
        1. HAL_PPP_Read()
        2. HAL_PPP_Write()
        3. HAL_PPP_Transmit()
        4. HAL_PPP_Receive()
    iii. Fonctions de **contrôle** qui permettent de changer dynamiquement la configuration et le mode de fonctionnement d'un périphérique
        1. HAL_PPP_Set()
        2. HAL_PPP_Get()
    iv. Fonctions d'**états et d'erreurs**
        1. HAL_PPP_GetState()
        2. HAL_PPP_GetError()

h. Le chapitre 2.7 décrit les concepts des HAL Extension API qui fournissent des API non plus générales, mais spécifiques à une famille de microcontrôleur ou bien spécifiques à un modèle particulier de processeur. Ces API se situent dans les fichiers stm32l4xx_hal_ppp_ex.c et stm32l4xx_hal_ppp_ex.h
i. Le chapitre 2.12 détaille l'utilisation des HAL drivers.
j. La figure 7 en 2.12.1 montre les interactions entre l'application utilisateur, les HAL drivers et les interruptions. Les rectangles à bords rouges montrent les fonctions msp implémentées dans l'application de l'utilisateur.

4.1.2 Comment utiliser les API du DAC

Le chapitre 14.2.2 décrit comment utiliser les API du DAC. Il faut :

a. valider DAC APB clock en appelant HAL_DAC_Init() pour obtenir le droit d'écrire dans le registre du DAC ;
k. Configurer DAC_OUT1 :PA4 ou DAC_OUT2 :PA5 en mode analogique ;
l. Configurer le canal du DAC en appelant la fonction HAL_DAC_ConfigChannel() ;
m. Valider le canal du DAC en appelant HAL_DAC_Start() ou HAL_DAC_Start_DMA().

⇨ STM32CubeMX réalise les étapes a,b et c pour nous, il nous revient donc la mise en œuvre de l'étape d. Pour information voici où sont réalisées les étapes a,b et c :

- ✓ **L'étape a**. ➔ dans dac.c où l'on trouve l'appel « if (HAL_DAC_Init(&hdac1) != HAL_OK) …»
- ✓ **L'étape b**. ➔ dans dac.c au sein de la fonction void HAL_DAC_MspInit(DAC_HandleTypeDef* dacHandle) qui est appelée via la fonction HAL_DAC_Init() qui se trouve dans stm32l4xx_hal_dac.c
- ✓ **L'étape c**. ➔ dans dac.c au sein de la fonction MX_DAC1_Init() qui appelle donc HAL_DAC_ConfigChannel(&hdac1, &sConfig, DAC_CHANNEL_1), sachant que MX_DAC1_Init() est appelée dans la fonction main() du fichier main.c.

4.2 DEUXIEME SOURCE D'INFORMATION : COMMENTAIRES DES FICHIERS « .C » ET « .H »

Dans un premier temps nous voulons juste que le DAC sorte une tension continue...

➔ Dans l'IDE, ouvrons le fichier stm32l4xx_hal_dac.c qui se trouve dans Drivers/STM32L4xx_HAL_Driver.

Nous voyons que STM32CubeMX a généré beaucoup de commentaires pour nous faciliter la vie ! Et voici tout ce que l'on apprend :

4.2.1 Sur les DACs

La famille STM32L4 dispose d'un ou deux DAC de 12 bits de résolution. Les microcontrôleurs qui n'ont qu'un DAC n'ont qu'un *channel* tandis que les microcontrôleurs qui en ont deux disposent de deux *channels* : il revient au même de parler de DAC ou de channel.

Le modèle STM32L476RG qui nous intéresse en comprend deux.

Quand deux *channels* sont présents, les deux DACs peuvent être utilisés indépendamment ou simultanément (*dual mode*).

DAC channel1 :

- avec DAC_OUT1 (PA4) comme sortie ;
- ou connecté à un autre composant en interne du microcontrôleur.

DAC channel2 :

- avec DAC_OUT2 (PA5) comme sortie ;
- ou connecté à un autre composant en interne du microcontrôleur.

4.2.2 Sur le mode d'utilisation des triggers

La conversion analogique peut fonctionner en mode « non déclenché » en utilisant la configuration DAC_TRIGGER_NONE, dans ce cas DAC_OUT1/DAC_OUT2 est disponible dès l'écriture dans le registre DHRx.

La conversion analogique peut fonctionner en mode « déclenché » :

- par un évènement externe : EXTI Line 9 (tout GPIOx_PIN_9) utilisant DAC_TRIGGER_EXT_IT9, dans ce cas la pin (GPIOx_PIN_9) utilisée doit être configurée en mode input.
- Par un timer TRGO : TIM2, TIM3, TIM4, TIM5, TIM6 and TIM7 (DAC_TRIGGER_T2_TRGO, DAC_TRIGGER_T3_TRGO...)
- Par software en utilisant DAC_TRIGGER_SOFTWARE

4.2.3 Sur la fonction buffer

Chaque channel DAC dispose d'un buffer permettant de réduire l'impédance de sortie et permettant de piloter des charges externes directement sans avoir à ajouter d'amplificateur opérationnel. Pour activer ce buffer, il faut utiliser sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;

⇨ Se référer au datasheet pour plus de détails sur l'impédance de sortie avec et sans ce buffer de sortie.

4.2.4 Pour utiliser la sortie DAC vers un composant interne du microcontrôleur

Utiliser sConfig.DAC_ConnectOnChipPeripheral = DAC_CHIPCONNECT_ENABLE;

4.2.5 Interférences entre les deux DAC

Quand l'un des DAC est utilisé (ex channel1 sur PA4) et l'autre pas (ex channel2 sur PA5 et configuré en analogique et désactivé), dans ce type de situation le channel1 peut perturber le channel2 par effet de couplage.

Dès que le channel 2 est activé (sur on) alors le couplage cesse.

Les interférences peuvent être supprimées :

- quand PA5 est configuré en INPUT PULL-UP ou en INPUT PULL-DOWN ;
- quand PA5 est configuré en ANALOG et qu'il est mis sur ON juste après.

4.2.6 Mode normal et mode échantillonnage et maintient

Pour chaque convertisseur, 2 modes sont pris en charge : le mode normal et le mode « Echantillonnage et maintien » (c'est-à-dire mode basse consommation).

Dans le mode d'échantillonnage et de maintien, le noyau DAC convertit les données, puis maintient la tension convertie grâce à un condensateur. Lorsqu'ils ne sont pas en cours de conversion, les cœurs DAC et tampon sont complètement désactivés et la sortie DAC passe en

haute impédance, réduisant ainsi la consommation d'énergie. Une nouvelle période de stabilisation est nécessaire avant chaque nouvelle conversion.

Le mode échantillonnage et maintien permet de régler la tension interne ou externe à un niveau de faible consommation d'énergie.

Pour activer le mode échantillonnage et maintient, activer LSI à l'aide de HAL_RCC_OscConfig avec les paramètres RCC_OSCILLATORTYPE_LSI & RCC_LSI_ON.

Utiliser DAC_InitStructure.DAC_SampleAndHold = DAC_SAMPLEANDHOLD_ENABLE ;

& DAC_ChannelConfTypeDef.DAC_SampleAndHoldConfig.DAC_SampleTime,

DAC_HoldTime & DAC_RefreshTime;

4.2.7 Fonctions de génération de formes d'ondes

Les DACs peuvent être utilisés pour générer des formes d'onde de type :

- Bruit aléatoire ;
- Triangles.

4.2.8 Format des données utilisées

Les données soumises au DAC peuvent être dans l'un des 3 formats :

- (#) 8-bit alignés à droite en utilisant DAC_ALIGN_8B_R

- (#) 12-bit alignés à gauche en utilisant DAC_ALIGN_12B_L

- (#) 12-bit alignés à droite en utilisant DAC_ALIGN_12B_R

4.2.9 Correspondance entre les valeurs numériques et les tensions.

Elle est déterminée par cette équation : DAC_OUTx = VREF * DOR / 4095

Avec :

DOR = Data Output Register ;

VREF = la référence de voltage en entrée (cf le datasheet)

4095 = 2^12 – 1

Application numérique 1 : si VREF = 3,3 V et DOR = 868, la tension en sortie du DAC sera égale à 3,3*868/4095 = 0,7V

Application numérique 2 : on veut obtenir 1,2V en sortie avec VREF = 3,3V, il faut donc programmer DOR avec la valeur (1,2*4095/3,3) = 1489

4.2.10 Comment utiliser le driver DAC.

On retrouve dans les commentaire de stm32l4xx_hal_dac.c, les mêmes informations que nous avons trouvées dans le fichier pdf *UM1884,* nous ne répèterons donc pas ces informations, nous savons qu'il nous faut simplement démarrer le driver avec la fonction **`HAL_DAC_Start()`**, le reste étant généré automatiquement par STM32CubeMX.

5 L'ETAPE 1

5.1 CONCEPTION

Pour ce tout premier test, nous souhaitons générer une tension de 2,5V en sortie du DAC1, tension que nous contrôlerons avec un voltmètre à aiguille.

Il va donc falloir programmer le DOR avec la valeur suivante : 2,5*4095/3,3 = 3102

5.2 PROGRAMMATION

Comme vu dans auparavant, on démarre le driver au bon endroit dans le code :

```
/* Initialize all configured peripherals */
/* USER CODE BEGIN 2 */
  HAL_DAC_Start(&hdac1, DAC_CHANNEL_1);
/* USER CODE END 2 */
```

Et juste avant la boucle principale qui restera vide, il suffit d'écrire :

```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
HAL_DAC_SetValue(&hdac1,DAC_CHANNEL_1,DAC_ALIGN_12B_R,3102)
;

while (1)
{
  /* USER CODE END WHILE */
  /* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */
```

Il est difficile de faire plus simple pour ce premier test !

6 L'ETAPE 2

6.1 CONCEPTION

Sachant que mon voltmètre dispose d'un calibre 2,5V, je souhaite observer la montée progressive de l'aiguille depuis le 0V jusqu'à la graduation 2,5V en un temps égal à 60 secondes, Après quoi l'aiguille devra repartir en sen inverse, jusqu'à la graduation zéro et ainsi de suite.

Nous avons déjà calculé la valeur de DOR correspondant à 2,5 V ➔ 3102.

Puisque nous souhaitons qu'une phase de montée ou de descente d'aiguille dure 60 s, comme il y a 3103 valeurs entre 0 et 3102, nous aurons 3102 intervalles de temps dont chacun devra durer 60000/3102 = 19,34 ms. Nous donnerons la valeur 19 à la fonction HAL_Delay entre chaque conversion d'une phase montante ou descendante qui durera donc 19*3102 ms, soit environ 59 s.

Nous aurons besoin d'une variable « val » qui contiendra la valeur à envoyer dans le registre DOR.

Nous aurons aussi besoin d'une variable « inc » qui sera positionnée à 1 ou -1 au rythme des phase de montée ou de descente.

Avant la boucle infinie :

Déclarer val et l'initialiser à 0 ;

Déclarer inc et l'initialiser à -1 ;

…

Démarrer le DAC1 ;

Dans la boucle infinie :

Envoyer val dans le registre DOR ;

Attendre 19 ms ;

si (val = 3102) ou (val = 0) alors inc = -1*inc ;

val = val + inc ;

6.2 PROGRAMMATION

```
  /* USER CODE BEGIN 1 */
     uint32_t val=0;
     int inc = -1;
  /* USER CODE END 1 */
…
  /* USER CODE BEGIN 2 */
     HAL_DAC_Start(&hdac1, DAC_CHANNEL_1);
  /* USER CODE END 2 */
…
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
          HAL_DAC_SetValue(&hdac1,DAC_CHANNEL_1,DAC_ALIGN_12B_R,val);
          HAL_Delay(19);
          if((val==3102)||(val==0)) inc=-1*inc;
          val+=inc;
  }
  /* USER CODE END 3 */
```

8 L'ETAPE 3

8.1 CONCEPTION

Nous souhaitons créer une forme d'onde sinusoïdale.

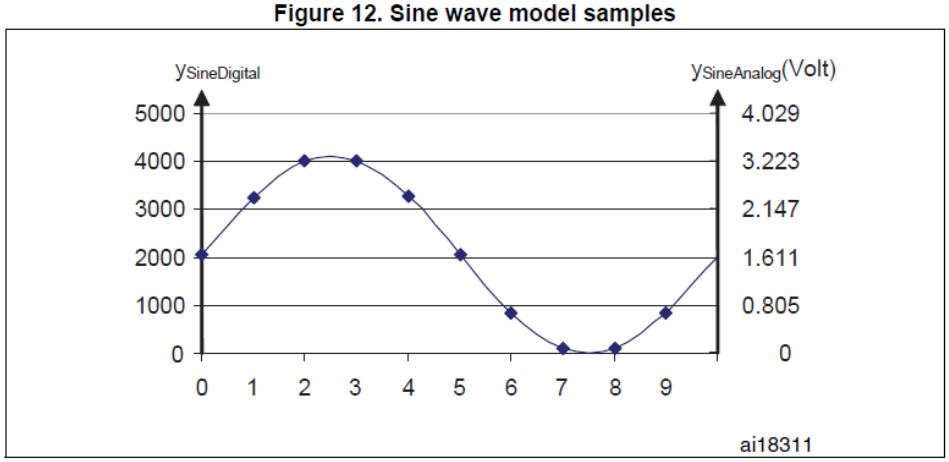
1) Ouvrir la note d'application fournie par ST : AN3126.
2) En 2.1.1 se trouve l'explication pas à pas de la méthode à suivre pour créer une forme sinusoidale.

2.1.2 Waveform preparation

To prepare the digital pattern of the waveform, we have to go through some mathematics.

Our objective is to have ten digital pattern data (samples) of a sine wave form that varies from 0 to 2π.

Figure 12. Sine wave model samples



The sampling step is $2\pi / n_s$ (number of samples).

The value of sin(x) varies between -1 and 1, we have to shift it up to have a positive sine wave with samples varying between 0 and 0xFFF (corresponding to the 0 to 3.3 V voltage range, where V_{REF} is set to 3.3 V).

$$y_{SineDigital}(x) = \left(\sin\left(x \cdot \frac{2\pi}{n_s}\right) + 1\right)\left(\frac{(0xFFF + 1)}{2}\right)$$

3) Page 16 on retrouve la formule permettant de calculer la sortie du DAC

Digital inputs are converted to output voltages on a linear conversion between 0 and V_{REF+}.

The analog output voltages on each DAC channel pin are determined by the equation

$$DAC_{Output} = V_{REF}\frac{DOR}{DAC\_MaxDigitalValue + 1}$$

Vref = 3,3 V

DOR est le registre dans lequel on met la valeur relative du signal à convertir en tension.

DAC_MaxDigitalValue dépend de la résolution que l'on utilise (12 bits ou 8 bits)

8.2 NOUVELLE CONFIGURATION DANS STM32CUBEMX

Il faut retourner dans STM32CubeMX et :

Partie « Configuration » du DAC1, dans l'onglet « parameter settings » ➔ choisir « Timer2 Trigger Out Event » ;
Nous allons utiliser le DMA pour soulager le CPU :
Partie « Configuration » du DAC1 », dans l'onglet « DMA settings » :

➔ Cliquer sur le bouton « Add » ;
➔Dans la boîte « select » qui s'ouvre, choisir « DAC_CH1 »
➔Dans la partie « DMA Request Settings »

- choisir le Mode « Circular »
- Data Width : « Word » (du côté Peripheral)
- Data Width : « Word » (du côté Memory).

Il reste à configurer le timer2 que nous avons choisi :
Partie « Pinout & Configuration » ➔ choisir Tim2.
Partie « Tim2 Mode & Configuration », faire les réglages suivants :

➔ « Clock Source » ➔ « Internal Clock » ;

Ensuite Il faudra régler le « Prescaler (PSC – 16 bit value) » du timer2 qui est connecté à l'horloge APB1 (si on a oublié la valeur que nous avions configurée dans l'onglet « Clock Configuration », on peut y retourner pour en prendre connaissance car nous avons besoin de cette valeur de manière critique ! Nous l'avions positionnée à 80 MHz).

➔ « Prescaler (PSC – 16 bit value) » ➔ 80-1. Ceci a pour effet de diviser la fréquence d'horloge par 80, ce qui fait qu'en sortie du prescaler nous aurons 1 MHz en guise de fréquence d'horloge pour le timer2.
➔ « Counter Period (AutoReload Register -32 bit value) » ➔ 100-1. Ceci a pour effet de générer une interruption tous les 100 coups d'horloge et donc d'obtenir une fréquence de 10 kHz pour le DMA.
➔ « Trigger Event Selection » ➔ « update event ».
➔ Dans l'onglet « NVIC Settings » cocher « Enabled » en face de « TIM2 blobal interrupt ».

Une fois cette programmation effectuée dans STM32CubeMX, il faut re générer le code ➔ Bouton « Generate Code »

8.3 PROGRAMMATION

Passer en commentaires ou supprimer tout ce que l'on a fait jusqu'à maintenant.

Inclure « **math.h** » pour avoir accès à la fonction sinus.

Il faut maintenant créer un tableau de 100 échantillons. Puisque nous aurons une interruption dont la fréquence sera de 10 kHz, chaque échantillon sera produit toutes les 100 µs et comme nous en aurons 100 par forme d'onde, la période de l'onde générée sera de 100*100µs = 10 ms, c'est dire que la fréquence de l'onde générée sera de 1/0,01 = **100 Hz**.

Ce que confirme d'ailleurs la note d'application :

2.1.3 Setting the sine wave frequency

To set the frequency of the sine wave signal, the user has to set the frequency ($f_{TimerTRGO}$) of the timer trigger output.

The frequency of the produced sine wave is

$$f_{Sinewave} = \frac{f_{TimerTRGO}}{n_s}$$

Nous aurons besoin de la constante **PI = 3,1415926**.

Nous créerons une fonction **get_sineval()** dont le but sera de remplir le tableau d'échantillons avec les valeurs définies par la formule de la note d'application AN3126 (cf plus haut).

$$y_{SineDigital}(x) = \left(\sin\left(x \cdot \frac{2\pi}{n_s}\right) + 1\right)\left(\frac{(0xFFF + 1)}{2}\right)$$

Dans notre cas, le nombre de samples **ns** sera de 100 , et 0xFFF correspond à la résolution du DAC sur 12 bits. 0xFFF+1 = 4096 en décimal et donc (0xFFF+1)/2=2048.

Dans la fonction main() il faudra

- démarrer le timer tim2 en appelant l'API HAL_TIM_Base_Start(&htim2) ;
- appeler la fonction get_sineval() qui servira une seule fois pour remplir la zone mémoire avec les 100 valeurs de samples.
- démarrer le DAC avec l'API spéciale liée au DMA : HAL_DAC_Start_DMA(&hdac1,DAC1_CHANNEL_1,sine_val,100,DAC_ALIGN_12B_R);

Dans cette API, on passe :

- en 3ème paramètre, le pointeur sur la zone où sont stockés les échantillons ;
- en 4ème paramètre, le nombre d'échantillons.

```
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "math.h"  // pour utilisation de la fonction sinus
/* USER CODE END Includes */
…
/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
#define PI 3.1415926 // pour utilisation de la fonction décrite dans AN3126
#define NS 100 // Nb Samples
/* USER CODE END PD */
…
/* Private variables ---------------------------------------------------------*/
/* USER CODE BEGIN PV */
uint32_t sine_val[NS];
```

```
/* USER CODE BEGIN 2 */
  HAL_TIM_Base_Start(&htim2) ;
  get_sineval();
  HAL_DAC_Start_DMA(&hdac1,DAC1_CHANNEL_1,sine_val,100,DAC_ALIGN_12B_R);
/* USER CODE END 2 */
/* Infinite loop */
/* USER CODE BEGIN WHILE */

while (1)
{
 /* USER CODE END WHILE */
 /* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */
```

Et c'est tout ! Il n'y a rien du tout à écrire dans la boucle infinie, puisque c'est la fonction DMA qui se charge de tout envoyer au DAC !

```
/* Private user code ---------------------------------------------------
----*/
/* USER CODE BEGIN 0 */
  void get_sineval() // usage unique pour valorisation des samples
  {// Cf formule note d'application AN3126
    for (int i=0;i<NS;i++) sine_val[i]=((sin(i*2*PI/NS)+1)*2048);
   }
/* USER CODE END 0 */
```

Comme on peut le voir, nous n'avons absolument rien codé dans la boucle infinie, le processeur n'a donc aucun traitement à assurer de notre part, c'est le DMA qui s'occupe de tout.