



UNIVERSIDADE FEDERAL DE MINAS GERAIS - UFMG
DEPARTAMENTO DE ENGENHARIA ELETRÔNICA - DEE

Programação de Sistemas Embarcados

Prof. Ricardo de Oliveira Duarte

Autores: Renan Moreira (renanmoreira@ufmg.br) e Rodolfo de Albuquerque Lessa
Villa Verde (rodolfoalvv@ufmg.br)

Application Note AN01000 versão 1.0

Modulação de frequência com PWM

Esta AN foi desenvolvida como trabalho da disciplina de Programação de Sistemas Embarcados da UFMG – Prof. Ricardo de Oliveira Duarte – Departamento de Engenharia Eletrônica.

Introdução

Essa application note descreve como usar o PWM - Pulse Width Modulation - para modular frequência em um microcontrolador da família STM32, com destaque para o stm32f103RB. A modulação da frequência será feita de duas maneiras: através da modulação do período e através da amplitude.

Conteúdo

Introdução	1
Conteúdo	1
Figuras:	2
Introdução ao PWM	3
Conceito	3
Parâmetros	3
Timer Input Clock - TIC	4
Prescaler - PSC	4
Counter Period - CP	4
Pulse	5
Configuração no CubeMX	5
Funções da HAL	5
Modulação de frequência por período	6

Perspectiva Teórica	6
Perspectiva Prática	6
Modulação de frequência por amplitude	9
Perspectiva Teórica	9
Perspectiva Prática	16
Licença	28
Referências	28

Figuras:

<i>Figura 1 - Exemplo de pulsos com diferentes duty cycles com a mesma frequência</i>	3
<i>Figura 2 - Árvore de Clock do CubeMX</i>	4
<i>Figura 3 - Configuração do TIM4</i>	7
<i>Figura 4 - Árvore de clock com os parâmetros setados</i>	8
<i>Figura 5 - Código com função para setar parâmetros do PWM</i>	8
<i>Figura 6 - Código com variação da frequência do PWM em 1000, 500 e 100 respectivamente</i>	9
<i>Figura 7 - Representação de uma onda senoidal e o duty-cycle necessário para cada período do pwm para produzi-la</i>	10
<i>Figura 8 - Representação em blocos da junção do sistema de DDS com o sistema de geração de forma de onda por PWM.</i>	11
<i>Figura 9 - Demonstrativo de como a fidelidade do sinal aumenta quantos maior a resolução. Mas em detrimento da memória.</i>	14
<i>Figura 10 - Configuração da árvore de clock. Tanto o APB1 quanto o APB2 foram definidos para 72MHz.</i>	16
<i>Figura 11 - Configuração do TIM2 para o timer de execução do DDS</i>	17
<i>Figura 12 - Ativa interrupções globais para o TIM2</i>	17
<i>Figura 13 - Configuração do TIM3 para gerar PWM.</i>	18
<i>Figura 14 - Inicialização do Timer e do PWM.</i>	19
<i>Figura 15 - Código em python para geração de FTW.</i>	19
<i>Figura 16 - Saída gerada do código apresentado.</i>	19
<i>Figura 17 - Lookup table das FTWs geradas pelo script Pyhton.</i>	20
<i>Figura 18 - Tela de configuração para o gerador de lookup table para uma senóide.</i>	21
<i>Figura 19 - LUT da senóide abaixo da dos FTWs.</i>	22
<i>Figura 20 - Interface da engine de DDS.</i>	22
<i>Figura 21 - Definição das variáveis.</i>	23
<i>Figura 22 - Implementação das funções da interface.</i>	24
<i>Figura 23 - Estrutura do callback para implementação do DDS.</i>	25

Todos os códigos utilizados nessa AN estão disponibilizado no seguinte repositório do GitHub: <https://github.com/renanmoreira17/STM32DDS>.

Introdução ao PWM

Conceito

PWM é uma técnica de controle de potência através da tensão criada nos anos 60, que permite ter um controle 100% digital e com menos perdas de energia. A técnica consiste em variar um pulso mantendo em nível lógico alto durante um tempo num ciclo em uma frequência.

O PWM possui 2 parâmetros moduláveis para o seu funcionamento:

- Duty Cycle: é o percentil do tempo no qual o sinal de saída permanece em nível lógico alto sobre o período. O duty cycle pode ser fixo ou variável, depende da sua aplicação, sendo responsável pela tensão média aplicada na saída.
- Frequência: quantidade de ciclos realizados em em segundo.

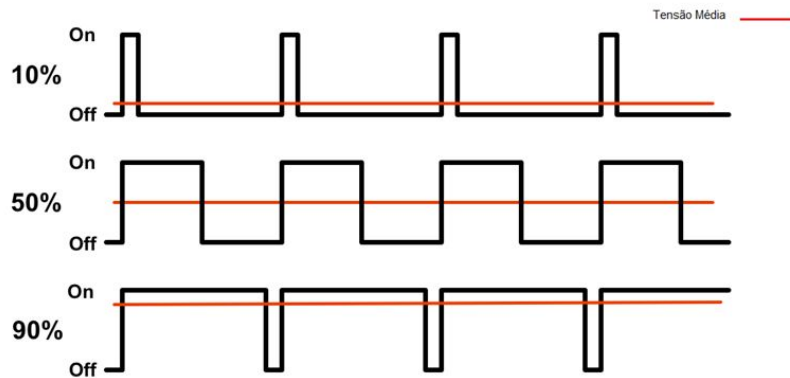


Figura 1 - Exemplo de pulsos com diferentes duty cycles com a mesma frequência

Parâmetros

O recurso de PWM de saída está associado aos timers em MCU, para aplicação em STM32, a frequência está associada ao *TIMx_ARR* (Auto Reload Register) e o duty cycle pelo registrador *TIMx_CCRx* (Compare and Capture Register).

Existem diversas formas de configurar o recurso de PWM no STM32. Para essa AN, utiliza-se timers para essa configuração.

Timer Input Clock - TIC

O *TIC*(clock interno dos timers) do *TIM3* e *TIM4* é representado pelo *APB1 Timer clocks* (MHz), enquanto o *TIC* do *TIM1* no *APB2 timer clocks*. *TIC* é a frequência do pulso de clock dos timers.

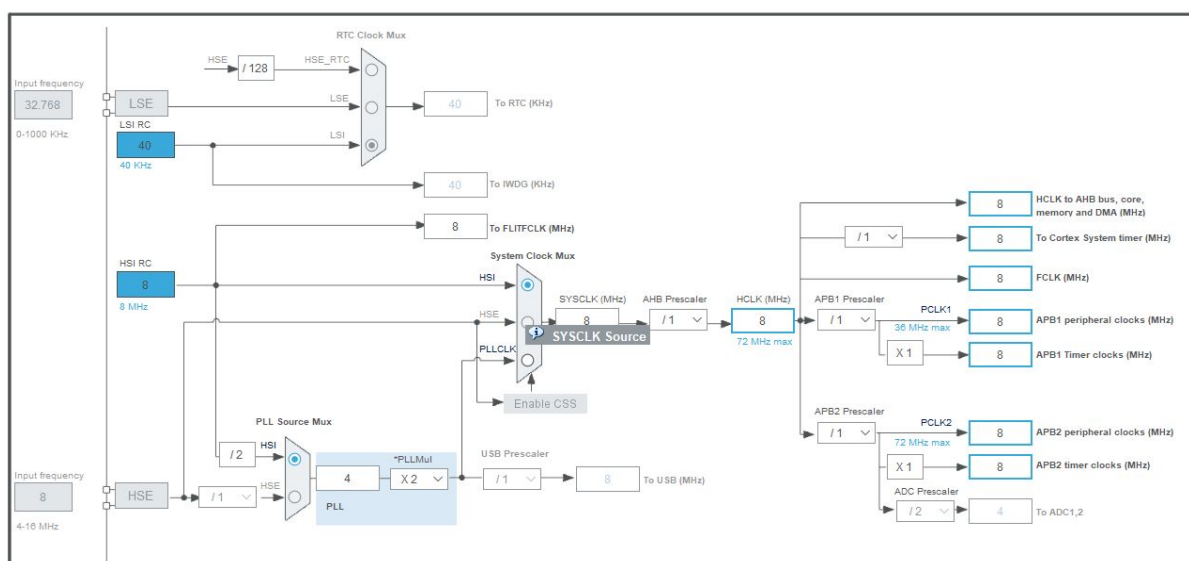


Figura 2 - Árvore de Clock do CubeMX

Prescaler - PSC

Valor de até 16 bits, é o divisor de frequência exclusivo do periférico *TIM* para um certa entrada do sinal de clock do timer. Assim, a frequência do periférico é definida pelo *TIC* dividido pelo *Prescaler* mais 1. Por default, o prescaler é setado em 0, é o mesmo que divisão por 1.

$$frequência\ do\ TIM = \frac{TIC}{PSC + 1}$$

Counter Period - CP

AN01000 - Modulação de Frequência através de PWM

Valor de até 16 bits, é o número de pulsos de clock no qual representará um ciclo do PWM. Portanto, a frequência do PWM é definida pela frequência do TIM dividido pela CP mais 1.

$$frequência\ do\ PWM = \frac{frequência\ do\ TIM}{CP + 1}$$

Pulse

Valor de até 16 bits, é o número que representa o percentil em relação ao Counter Period em que a saída de PWM vai estar em nível alto lógico. Assim, é através do Pulse que é definido o duty cycle. O Pulse é definido pelo duty cycle multiplicado pelo Counter Period.

$$Duty\ Cycle = \frac{Pulse}{CP}$$

Configuração no CubeMX

- 1) Escolher o timer e o canal de saída PWM a ser utilizado na aplicação.

Os timers TIM3 e TIM4 possuem pinos de GPIOs exclusivos para aplicação de PWM.

- 2) Definir o TIC na árvore de clock.
- 3) Setar o PSC, o CP e o Pulse

Exemplo:

- TIC = 8 MHz;
- Prescaler = 15;
- Counter Period = 499;
- Pulse = 249;

Assim obtém-se:

- Duty Cycle = 50%;
- Frequência do TIM = 500 kHz;
- Frequência do PWM = 1000 Hz

Funções da HAL

Para configuração do PWM, a HAL disponibiliza as seguintes APIs:

- *HAL_TIME_PWM_Init()* → Inicializa e configura o TIM PWM;
- *HAL_TIM_PWM_DeInit()* → De-inicializa o TIM PWM;
- *HAL_TIM_PWM_MspInit()* → Inicializa o MCU Support Package do PWM;
- *HAL_TIM_PWM_MspDeInit()* → De-inicializa o MCU Support Package do PWM;
- *HAL_TIM_PWM_Start()* → Começa o Timer PWM;
- *HAL_TIM_PWM_Stop()* → Pausa o Timer PWM;
- *HAL_TIM_PWM_Start_IT()* → Começa o Timer PWM e habilita a interrupção;
- *HAL_TIM_PWM_Stop_IT()* → Pausa o Timer PWM e desabilita a interrupção;
- *HAL_TIM_PWM_Start_DMA()* → Começa o Timer PWM e habilita a transferência DMA;
- *HAL_TIM_PWM_Stop_DMA()* → Pausa o Timer PWM e desabilita a transferência DMA.

Modulação de frequência por período

Perspectiva Teórica

A frequência de um PWM é definida, como dito anteriormente, pelos parâmetros: TIC, prescaler, Counter Period. Portanto, para a manipulação da frequência em código, é modificado o registrador do Auto Reloaded, enquanto o TIC e o prescaler permanecem inalterados.

$$f_{PWM} = \frac{TIC}{(prescaler + 1)(counter\ period + 1)}$$

Dessa forma, basta realizar o cálculo para a frequência desejada e atualizar o novo valor do Counter Period. Vale ressaltar que é necessário alterar o Pulse para não alterar o duty cycle.

Por exemplo, com um CP de 999, obtém-se uma frequência de 1000Hz. Com um Pulse de 499, tem-se um duty cycle de 50%. Entretanto, caso altere-se o CP para 499, mantendo o mesmo Pulse, o duty cycle passa a ser de 100%! Dessa forma, faz-se necessário alterar o Pulse para 249, com intuito de manter o duty cycle a 50%.

Essa metodologia permite que varie a frequência de maneira simples, porém estática. Para gerar funções, ou ondas, é preferível utilizar a modulação de frequência por amplitude.

Perspectiva Prática

AN01000 - Modulação de Frequência através de PWM

Para exemplificar a modulação de frequência pelo período, será abordado uma mudança de frequência de 1000Hz, para 500Hz e 100Hz. O microcontrolador utilizado é o stm32f103RB. Primordialmente, calcula-se os parâmetros a serem utilizados no PWM. Nesta aplicação, utiliza-se o clock interno do MCU de 8MHz e o TIM4. Para manipularmos com mais facilidade o Counter Period, determinou-se um Prescaler de 7999. A frequência de 1000Hz é obtida com um Counter Period de 999.

Para obter as frequências de 500Hz e 100Hz, seta-se o Counter Period em 1999 e 9999, e o Pulse em 999 e 4999 respectivamente.

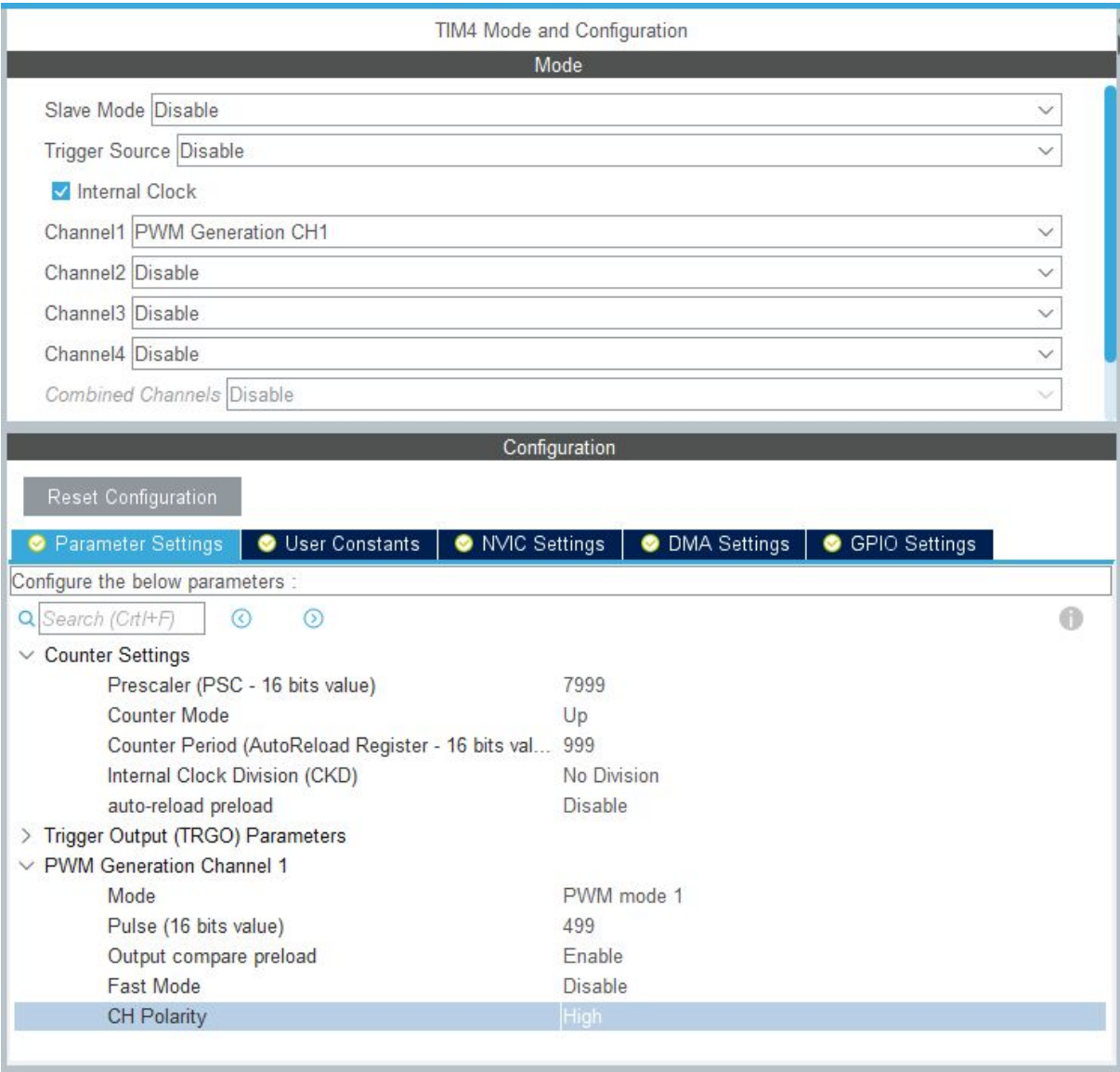


Figura 3 - Configuração do TIM4

AN01000 - Modulação de Frequência através de PWM

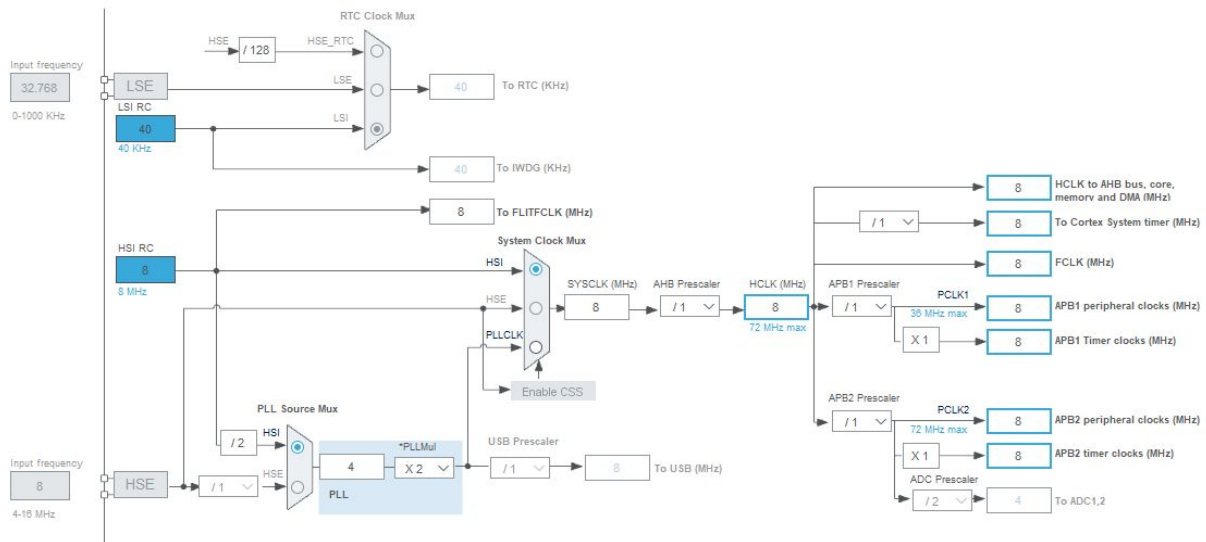


Figura 4 - Árvore de clock com os parâmetros setados

Após a configuração correta dos parâmetros, basta realizar a manipulação em código do Counter Period e do Pulse já calculados.

Para facilitar a manipulação dos parâmetros do PWM, foi criada a função `set_PWM`, que também inicializa o timer, como é possível observar na imagem:

```
212
213 /* USER CODE BEGIN 4 */
214 void setPWM(TIM_HandleTypeDef timer, uint32_t channel, uint16_t period, uint16_t pulse)
215 {
216     HAL_TIM_PWM_Stop(&timer, channel); // stop generation of pwm
217     TIM_OC_InitTypeDef sConfigOC;
218     timer.Init.Period = period; // set the period duration
219     HAL_TIM_PWM_Init(&timer); // reinitialise with new period value
220     sConfigOC.OCMode = TIM_OCMODE_PWM1;
221     sConfigOC.Pulse = pulse; // set the pulse duration
222     sConfigOC.OCpolarity = TIM_OCPOLARITY_HIGH;
223     sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
224     HAL_TIM_PWM_ConfigChannel(&timer, &sConfigOC, channel);
225     HAL_TIM_PWM_Start(&timer, channel); // start pwm generation
226 }
227 /* USER CODE END 4 */
```

Figura 5 - Código com função para setar parâmetros do PWM

Dessa forma, basta setar os parâmetros desejados. Com intuito de demonstração, iremos variar a cada 10 segundos a frequência do PWM.


```
97  while (1)
98  {
99      /* USER CODE END WHILE */
100     setPWM(htim4, TIM_CHANNEL_1, 999, 499);
101     HAL_Delay(10000);
102     setPWM(htim4, TIM_CHANNEL_1, 1999, 999);
103     HAL_Delay(10000);
104     setPWM(htim4, TIM_CHANNEL_1, 9999, 4999);
105     HAL_Delay(10000);
106     /* USER CODE BEGIN 3 */
107
108 }
109 /* USER CODE END 3 */
110 }
```

Figura 6 - Código com variação da frequência do PWM em 1000, 500 e 100 respectivamente

Modulação de frequência por amplitude

Perspectiva Teórica

Existe um outro método para gerar frequência por meio de PWM. Viu-se que ao variar o valor do registrador de estouro do timer, varia-se o período de contagem do periférico - o que muda consequentemente a frequência. Esse outro método consiste em realizar variações no duty-cycle do pwm ao longo do tempo, de modo que essa variação seja periódica com um dado período T . A variação do duty-cycle, se vista como uma média no tempo, se trata nada mais do que a variação da amplitude. Dessa forma, teremos um outro sinal com uma frequência $f=1/T$. Pode-se visualizar esse fenômeno com a seguinte figura:

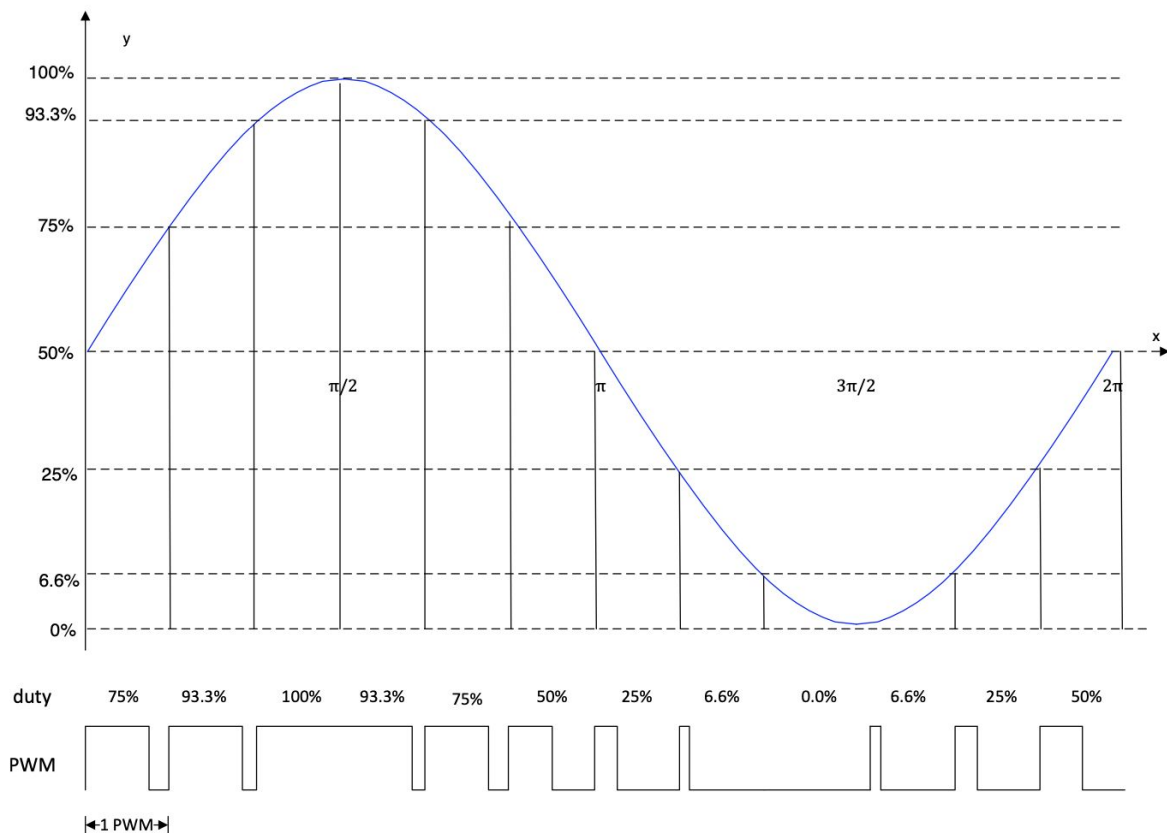


Figura 7 - Representação de uma onda senoidal e o duty-cycle necessário para cada período do pwm para produzi-la

Vale ressaltar que essa figura é apenas uma representação. De maneira a reproduzir fielmente uma forma de onda a partir de uma variação de duty-cycles, é desejável que haja diversos períodos do PWM em um ciclo da forma de onda desejada. Também é importante destacar que, para esse caso, definimos um duty-cycle de 50% como se fosse o 0 da forma de onda, já que, nativamente com o STM32, é impossível gerar um PWM com tensão negativa.

Para fins de referência, pode-se consultar este seguinte material contendo uma explicação teórica da técnica:

https://www.ti.com/lit/an/spna217/spna217.pdf?ts=1603894398721&ref_url=https%253A%252F%252Fwww.google.com%252F (Material da texas instruments exemplificando como gerar formas de onda com PWM).

Esse último material trata da geração da forma de onda de uma maneira mais rudimentar, mas ainda assim tem explicações sólidas do método para fins de compreensão. Uma maneira mais robusta de conseguir a geração de uma forma de onda é a partir de um método chamado DDS (Direct Digital Synthesis), que consiste no uso de duas lookup tables, uma mapeando o valor de amplitude de um período arbitrário de uma forma de onda, e outra guardando o valor de incremento para um acumulador - esse acumulador tem seu conteúdo plugado na lookup table da forma de onda a cada pulso de clock, com o resultado enviado à saída. Dessa forma, para diferentes escolhas do incremento do acumulador, obtemos diferentes frequências de saída. Esse método é mais robusto por alguns fatores:

AN01000 - Modulação de Frequência através de PWM

- A frequência de entrada do sistema é fixa, e com ela pode-se gerar diversas frequências de saída. Isso é especialmente útil para sistemas que não permitam uma variação trivial da frequência de entrada, além de garantir uma sincronia melhor.
- O ruído de fase é baixíssimo, sendo útil para aplicações que necessitem alta fidelidade do sinal gerado
- Alta performance de execução. Não são feitas longas e complexas operações matemática de ponto flutuante. Isso significa pouco gasto de performance para alto desempenho, especialmente interessante para sistemas com baixa capacidade de processamento.
- Pode-se ter uma infinidade de formas de ondas diferentes, não se limitando a ondas quadradas.

Informações mais detalhadas acerca do método de DDS pode ser encontrado neste material, da Analog Device: <https://www.analog.com/media/en/training-seminars/tutorials/MT-085.pdf>.

O material acima trata da saída do DDS ligada a um DAC, porém, no nosso caso, usaremos PWM. No final, trata-se de uma junção do método do segundo material com o do primeiro.

O diagrama do modelo que realizaremos aqui pode ser observado a seguir:

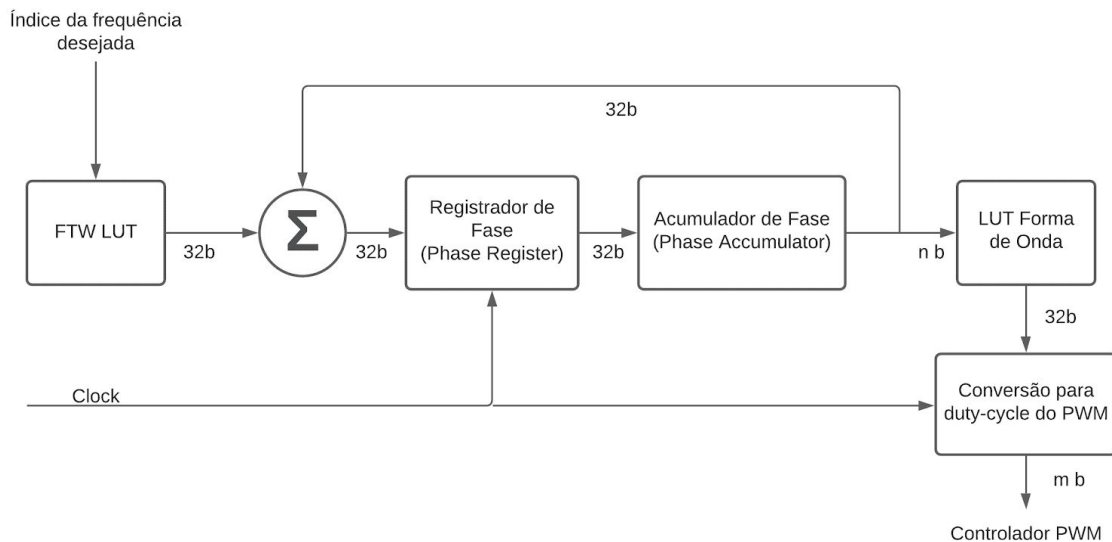


Figura 8 - Representação em blocos da junção do sistema de DDS com o sistema de geração de forma de onda por PWM.

Vale ressaltar que, nesse diagrama, FTW LUT significa “Frequency Tuning Word Lookup Table”, que se trata da lookup table que indexa todos os incrementadores calculados. Dessa forma, a quantidade de frequências que podem ser produzidas fica limitada a quantos valores de incrementadores foram calculados e registrados nessa lookup table.

De fato, não é estritamente necessário que se tenha, obrigatoriamente, essa lookup table de incrementadores, contudo calcular esses incrementadores é uma operação um pouco custosa, então, se as frequências que o sistema produzir forem de número limitado e já conhecidas, aconselha-se a calcular previamente esses incrementadores e definir uma lookup table os contendo.

Para se calcular o valor do incrementador, utiliza-se da seguinte fórmula:

$$FTW = \frac{f_{out} \cdot 2^k}{f_{in}} \quad (1)$$

onde f_{out} é a frequência que se deseja produzir, k é a quantidade de bits que o acumulador de fase armazena (no caso do STM32, k pode ser até 32 bits) e f_{in} é a frequência de entrada do sistema.

A frequência de entrada é um conceito importante para um sistema DDS. Primeiramente, essa frequência nos diz que o algoritmo do DDS precisa ser executado em uma frequência fixa (e isso é efetivamente necessário porque nossa FTW é calculada a partir dessa frequência). Para atingir esse objetivo, há diversos caminhos, no entanto, o mais direto e confiável é a partir do uso de timers. Deve-se configurar o timer para gerar interrupções em um período fixo de tempo, referente a frequência de entrada.

Quanto ao valor da frequência de entrada, essa deve ser definida de maneira a atender os requisitos da aplicação. O teorema de amostragem de Nyquist tem um papel importante aqui: a frequência de entrada precisa ser, pelo menos, duas vezes maior que a máxima frequência que se deseja sintetizar. Entretanto, dependendo da aplicação, pode ser desejável que esse fator seja um pouco maior para garantir uma maior qualidade do sinal gerado. No caso de geração de áudio, por exemplo, nota-se que uma frequência 10 vezes maior é recomendada para uma qualidade boa. De forma geral, para garantir um sinal aceitável, defina a frequência de entrada como 5 vezes maior do que a frequência máxima da aplicação.

$$f_{in} = 5f_{max} \quad (2)$$

Definida essa frequência, gera-se então as FTWs com (1) para cada f_{out} desejada (ou implementa-se um algoritmo para calcular essa FTW em tempo de execução).

Seguindo o diagrama, o outro estágio importante se trata da lookup table da forma de onda desejada. Por padrão, usa-se o método de DDS para gerar senóides, embora possa-se gerar outras formas de onda sem complicações. É necessário gerar a discretização dessa forma de onda, e sua geração obedece alguns critérios. Primeiramente, precisamos definir a quantização da forma de onda. Para isso escolhe-se um valor 2^n para ser o número de amostras. Com isso, temos que fazer um período da nossa forma de onda caber nessas 2^n amostras; isso pode ser atingido espaçando 2^n amostras de $a = T/2^n$, onde T é o período da forma de onda. Plugamos $a.t$ na função matemática f_{onda}

representando nossa forma de onda, onde $t = 0, 1, 2 \dots 2^n - 1$, e guardamos o resultado da função na t ésima posição da lookup table. O resultado da função precisa ser normalizado para se estender no tamanho do tipo de variável da lookup table; essa normalização é feita de maneira que $\min(f_{\text{onda}}) = 0$, $\max(f_{\text{onda}}) = 2^k - 1$, onde k é o tamanho da variável da LUT. Além disso, os valores x em que $f_{\text{onda}}(x)$ era igual seu valor médio, agora obedecem $f_{\text{onda}}(x) = 2^{k-1} - 1$. Esse condicionamento é importante para que se possa conseguir uma resolução mais alta possível para a amplitude do sinal.

De maneira a melhor visualizar essas afirmações, vamos supor um cenário em que: a forma de onda a ser sintetizada é uma senóide, a lookup table da senoide terá 1024 posições (2^{10} , $n = 10$), o tipo variável da lookup table é um uint16_t ($k = 16$). Sabemos que a senóide é uma função periódica centrada em 0 que varia de -1 a 1, com período igual a 2π . Para quantizar essa função, temos que realizar 1024 amostras espaçadas de $2\pi/1024$ da função $f_{\text{onda}} = \sin(x)$. Além disso, temos que encontrar uma maneira para que $\min(f_{\text{onda}}) = 0$ e $\max(f_{\text{onda}}) = 2^{16} - 1 = 65535$, além de que $f_{\text{onda}}(\{0, \pi\}) = 2^{16-1} = 32768$; como a função $\sin(x)$ tem o máximo sendo 1 e o mínimo sendo -1, podemos simplesmente multiplicá-la por $2^{15} - 1$ e somar o resultando com $2^{15} - 1$. Com isso, teremos que $f_{\text{quant}}(x) = (2^{15} - 1)\sin(x) + 2^{15} - 1 = 32767.\sin(x) + 32767$. Nota-se que, dessa forma, atende-se aos critérios estabelecidos. Com isso, instanciamos uma LUT de 1024 posições e, para cada dada posição t da LUT atribuímos o valor $f_{\text{quant}}(t.2\pi/1024) = 32767.\sin(t.2\pi/1024) + 32767$ truncado para um inteiro.

Há, contudo, formas mais práticas fáceis de criar uma LUT para um sinal, como o seguinte gerador online de LUT: <https://www.daycounter.com/Calculators/Sine-Generator-Calculator.phtml>.

É necessários e atentar com o fato de que, como a LUT é de 2^n posições e o sinal de saída do Phase Accumulator é de 2^k , e normalmente $k \geq n$, deve-se truncar a saída do Phase Accumulator de maneira a pegar os n bits mais significantes de k e usá-los para extrair o valor correspondente em índice da lookup table.

Também vale destacar que, embora uma LUT seja mais precisa e fiel quanto mais posições tiver, esbarra-se numa questão de limitação de memória. Um valor de $n = 10$ já costuma ser suficiente, e já gasta 1kb de flash. O consumo aumenta exponencialmente para valores maiores de n .

A seguir temos um demonstrativo, para uma senóide, como o aumento do número de amostras melhora a qualidade da quantização:

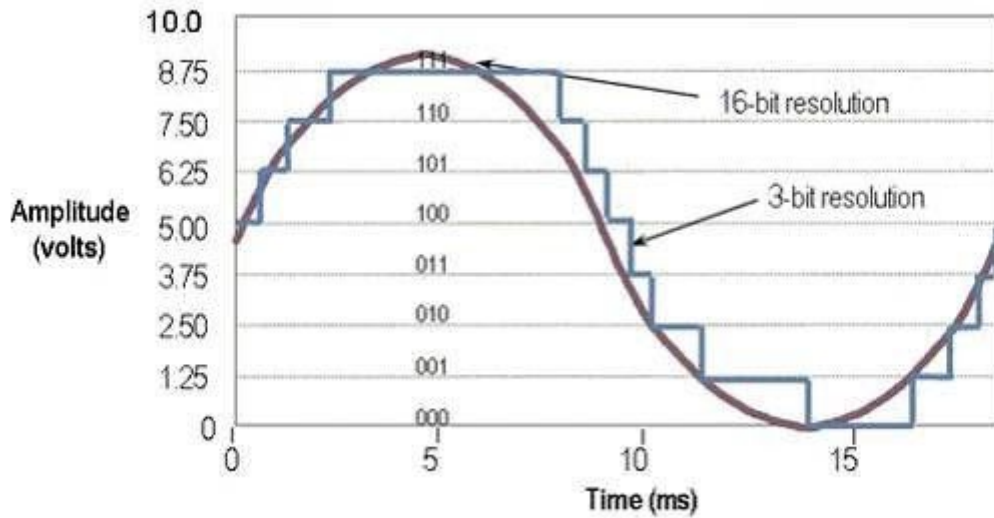


Figura 9 - Demonstrativo de como a fidelidade do sinal aumenta quantos maior a resolução. Mas em detrimento da memória.

Finalmente, para converter a saída do bloco de geração de forma de onda para ser utilizada no PWM, precisa-se ter alguns parâmetros em mente. A princípio, é necessário saber qual a resolução do PWM. No caso do STM32, diferentes timers possuem diferentes resoluções, de 16 ou 32 bits. E, como visto na primeira parte dessa Application Note, podemos configurar o registrador de AutoReload para ter qualquer resolução, contanto que o valor do AutoReload seja menor que o tamanho do registrador de contagem do timer (16 ou 32 bits). Essa resolução definida pelo registrador de AutoReload impacta diretamente na frequência de operação do PWM, de modo que essa frequência é definida por:

$$f_{pwm} = \frac{f_{clk}}{(Prescaler+1)(AutoReload+1)} \quad (3)$$

onde f_{clk} é a frequência do barramento em que o timer que opera o PWM está conectado. O valor do Prescaler é dependente somente da frequência de entrada, caso seja muito alta e deseja-se diminuí-la. Nesse caso, usaremos $Prescaler = 0$.

Essa frequência de operação do PWM precisa, também, conformar com o Teorema da Amostragem de Nyquist, de maneira que essa frequência seja pelo menos duas vezes maior que a frequência máxima que deseja-se representar. Dessa forma, devemos escolher *AutoReload* de maneira que $f_{pwm} \geq 2f_{in}$, definida em (2). Para facilitar os passos adiantes, deve-se escolher *AutoReload* como uma potência m de 2, de maneira que:

$$AutoReload = 2^m - 1 \quad (4)$$

Assim, combinando (4) com (3), (2) e a condição de frequência do AutoReload, temos que, para esse valor de m :

$$m \leq \lfloor \log_2 \left(\frac{f_{clk}}{10f_{max}} \right) \rfloor \quad (5)$$

Onde f_{max} é a frequência máxima que se deseja sintetizar com o sistema, e $\lfloor x \rfloor$ representa o valor inteiro menor e mais próximo de x — a função floor(x).

Com isso, basta substituir o valor de m encontrado em (5) em (4), e setar o parâmetro de AutoReload do timer com essa quantia. Assim, garantimos que o PWM gerado tenha a frequência necessária para uma geração fiel do sinal.

O foco desse bloco é, a partir do sinal de entrada do gerador de forma de onda, definir o duty-cycle do PWM que represente essa amplitude instantânea do sinal. Como essa amplitude é definida para cada ciclo de operação do DDS, essa variação periódica de duty-cycle (amplitude) gerará uma frequência como consequência na forma de onda desejada. E, para variar o duty-cycle, precisa-se condicionar o sinal proveniente do gerador de forma de onda e setá-lo no registrador de comparação do PWM.

Assim como foi feito para o bloco anterior, de geração da forma de onda, percebe-se que a entrada possui um número de bits (k) maior que a saída para o PWM (m). Dessa forma, fazemos do mesmo jeito que anteriormente: truncamos os m bits mais significativos de k . Depois, esse resultado truncado é setado no registrador de comparação. De fato, esse bloco de conversão para sinal de PWM possui a única função de truncar o sinal de saída do gerador de forma de onda para se conformar com o PWM e setá-lo no registrador, mas é necessário levar em consideração todos os pontos citados nessa seção para que se consiga representar o sinal com sucesso.

Explicado passo a passo o funcionamento desse sistema baseado em DDS, pode-se fazer uma síntese de como o sistema atua:

1. Gera-se os valores de FTW para cada frequência desejada e os armazena em uma lookup table. Caso seja desejado gerar dinamicamente o FTW, pula-se essa etapa.
2. Cria-se a lookup table da forma de onda seguindo os passos discutidos anteriormente.
3. Cria-se uma variável de tamanho k bits (recomenda-se 32 bits) para ser o phase accumulator, e a inicializa com 0.
4. Cria-se outra variável de mesmo tamanho para armazenar a FTW, que recebe um dos valores calculados no primeiro item, dependendo da frequência desejada.
5. Configura-se um timer separado para gerar interrupções em uma frequência fixa. Essa frequência é a discutida pela equação (2).
6. Na função de callback da interrupção do timer, implementa-se a lógica do DDS:
 - 6.1. Incrementa-se o phase accumulator com o valor atual do FTW.
 - 6.2. Trunca o novo valor do phase accumulator para os n bits da LUT do gerador de forma de onda como discutido anteriormente.
 - 6.3. Busca, na lookup table da forma de onda, o valor correspondente ao truncamento do phase accumulator.
 - 6.4. Trunca esse último resultado novamente para se conformar com os m bits do PWM.
 - 6.5. Seta o registrador de comparação do PWM com o valor truncado.
7. A qualquer momento, o usuário pode alterar o valor da FTW para gerar a frequência desejada. Seja passando o índice para a lookup table de FTW, ou calculando dinamicamente a FTW. O sistema passa a gerar a nova frequência automaticamente sem nenhum passo adicional.

Perspectiva Prática

Discutida a base teórica para o funcionamento de um sistema DDS, pode-se, agora, exemplificar como seria uma implementação desse sistema utilizando os microcontroladores STM32.

A nossa aplicação de exemplo vai se basear em um sistema que gera ondas senoidais de frequências múltiplo de 100Hz, de 100 a 1000Hz. Com isso, podemos começar a definir nossos parâmetros. Começemos pela frequência de execução do DDS: nossa f_{max} pode ser definida por $f_{max} = 1kHz$; logo, por (2), temos que $f_{in} = 5.1kHz = 5kHz$.

Para começarmos a implementação, primeiramente cria-se o projeto normalmente. Após isso, deve-se realizar as configurações no CubeMX para os timers necessários. Como $f_{in} = 5kHz$, temos que configurar a árvore de clock e os parâmetros do timer para que ele gere interrupções nessa frequência. Se definirmos o clock de entrada do timer para 72MHz, teremos que obedecer $f_{in} = \frac{f_{clk}}{(Prescaler+1)(AutoReload+1)}$ para saber qual valor definir no Prescaler ou no AutoReload. Se definirmos $Prescaler = 0$, teremos que $AutoReload = 14399$. Logo, vamos escolher um timer (nesse caso o TIM2) e fazer essas configurações:

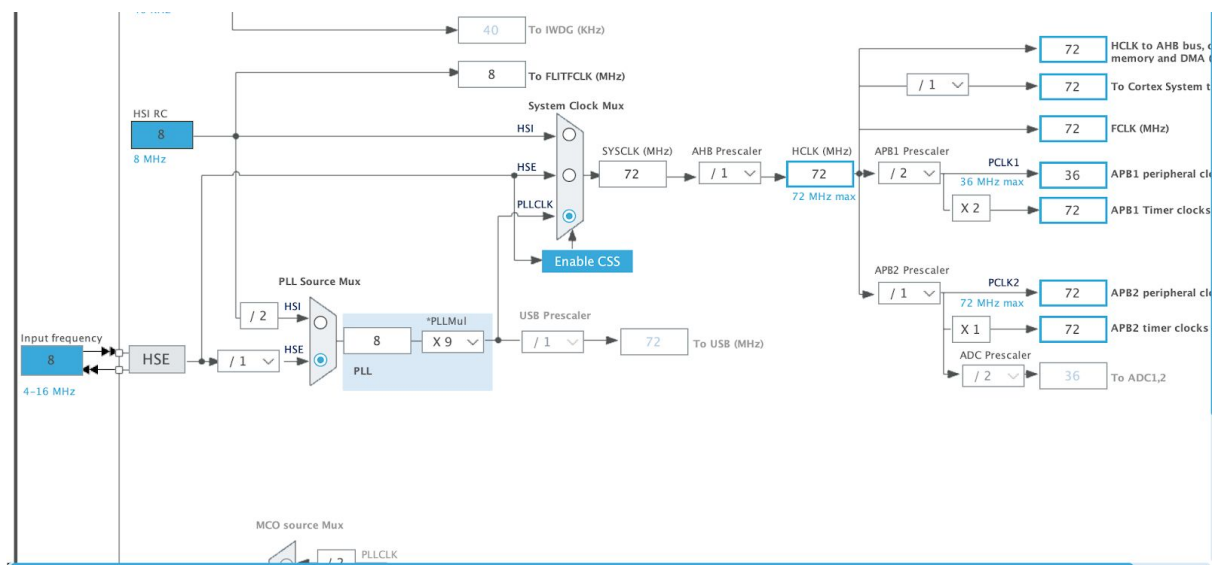


Figura 10 - Configuração da árvore de clock. Tanto o APB1 quanto o APB2 foram definidos para 72MHz.

AN01000 - Modulação de Frequência através de PWM

Mode

Slave Mode

Disable

Trigger Source

Disable

Clock Source

Internal Clock

Channel1

Disable

Channel2

Disable

Channel3

Disable

Channel4

Disable

Combined Channels

Disable

☐ Use ETR as Clearing Source

Configuration

Reset Configuration

Parameter Settings

User Constants

NVIC Settings

DMA Settings

Configure the below parameters :

Search (Ctrl+F)

Counter Settings

Prescaler (PSC – 16 bits value)

0

Counter Mode

Up

Counter Period (AutoReload Register – 16 bits value)

14399

Internal Clock Division (CKD)

No Division

auto-reload preload

Disable

Trigger Output (TRGO) Parameters

Master/Slave Mode (MSM bit)

Disable (Trigger input effect not delayed)

Trigger Event Selection

Reset (UG bit from TIMx_EGR)

Figura 11 - Configuração do TIM2 para o timer de execução do DDS

É importante notar que, para configuração do TIM2, define-se o campo Clock Source como “Internal Clock”. O campo Counter Period (AutoReload) foi definido com o valor calculado. Também é necessário ativar as interrupções globais para o TIM2, como pode-se observar a seguir:

TIM2 Mode and Configuration

Mode

Slave Mode

Disable

Trigger Source

Disable

Clock Source

Internal Clock

Channel1

Disable

Channel2

Disable

Channel3

Disable

Channel4

Disable

Combined Channels

Disable

☐ Use ETR as Clearing Source

Configuration

Reset Configuration

Parameter Settings

User Constants

NVIC Settings

DMA Settings

NVIC Interrupt Table

Enabled

Preemption Priority

Sub Priority

TIM2 global interrupt

☒

0

0

Figura 12 - Ativa interrupções globais para o TIM2

AN01000 - Modulação de Frequência através de PWM

Feito isso, temos que configurar o timer responsável por gerar o sinal PWM. Recomenda-se escolher o timer e canal que melhor atende às necessidades do usuário quanto à resolução e local do pino de saída PWM. Escolheu-se o canal 1 do TIM3, que tem como pino de saída o PA6. O TIM3 é de 16 bits, logo, temos que $m \leq 16$. Usamos, então, a equação (5):

$$m \leq \lfloor \log_2\left(\frac{f_{clk}}{10f_{max}}\right) \rfloor \Rightarrow m \leq \lfloor \log_2\left(\frac{72 \cdot 10^6}{10 \cdot 10^3}\right) \rfloor$$
$$m \leq 12$$

Então, definimos $m = 12$ e aplicamos a equação (4):

$$AutoReload = 2^m - 1 \Rightarrow AutoReload = 2^{12} - 1$$
$$AutoReload = 4095$$

Logo, configuramos o TIM3 de acordo:

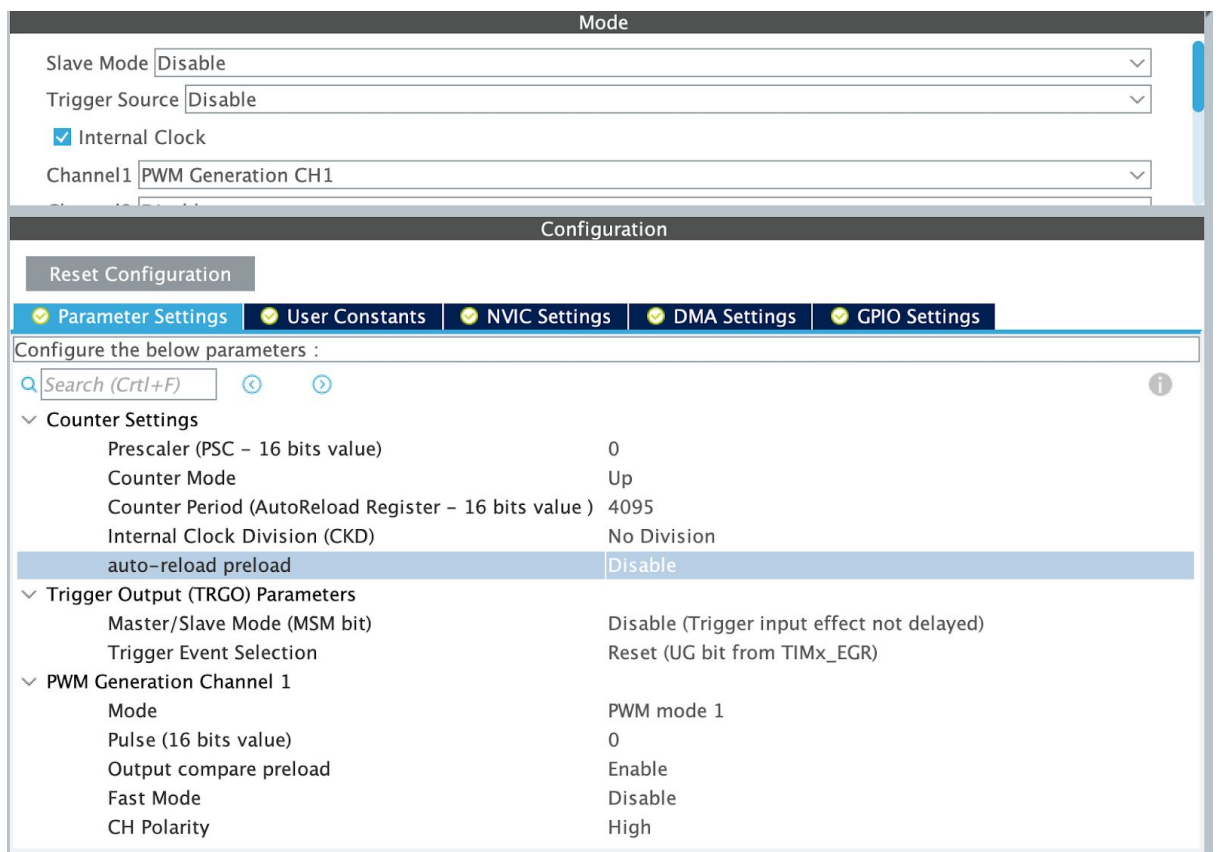
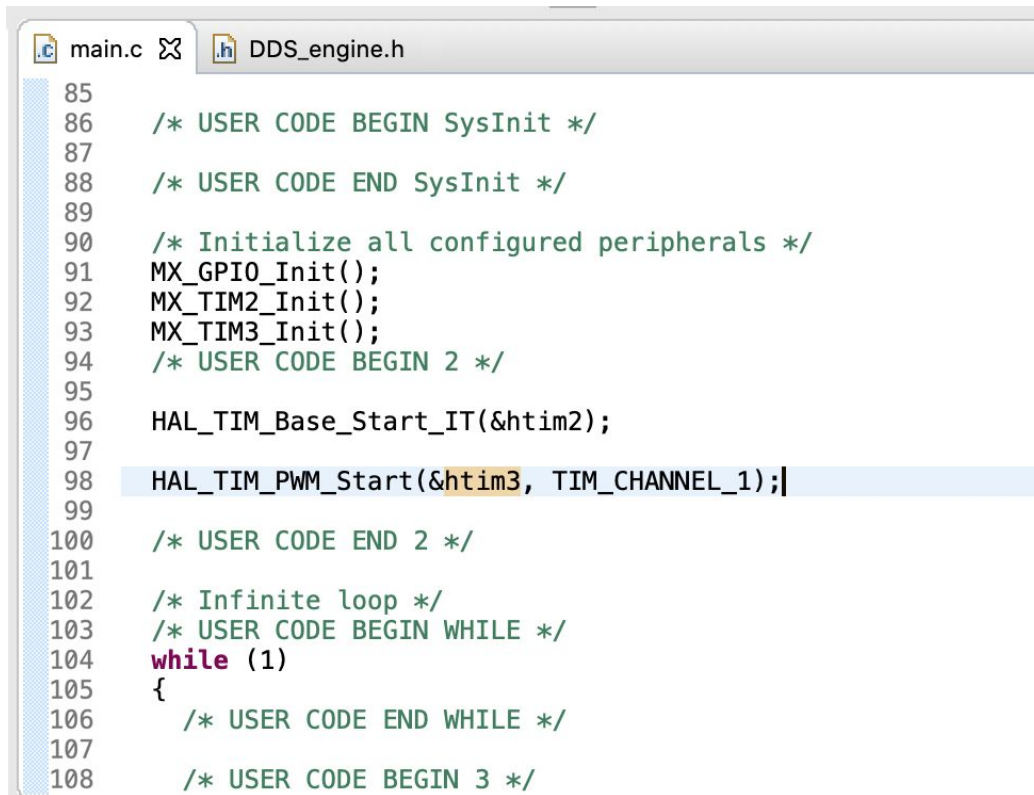


Figura 13 - Configuração do TIM3 para gerar PWM.

Para essa configuração, definiu-se o “Channel 1” como “PWM Generation”, ativou-se a opção “Internal Clock” e setou o AutoReload com o valor calculado. Não é necessário ativar as interrupções globais nesse caso.

AN01000 - Modulação de Frequência através de PWM

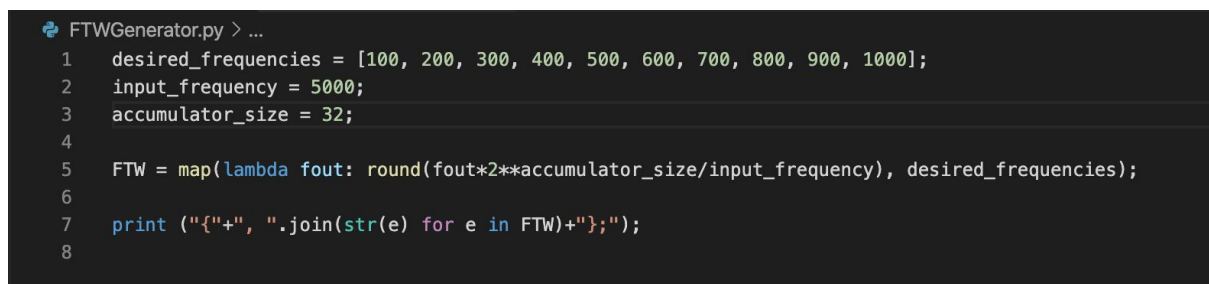
Agora, precisamos inicializar os periféricos. Na main.c, executamos o código de inicialização do Timer e do PWM.



```
main.c DDS_engine.h
85
86 /* USER CODE BEGIN SysInit */
87
88 /* USER CODE END SysInit */
89
90 /* Initialize all configured peripherals */
91 MX_GPIO_Init();
92 MX_TIM2_Init();
93 MX_TIM3_Init();
94 /* USER CODE BEGIN 2 */
95
96 HAL_TIM_Base_Start_IT(&htim2);
97
98 HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_1);
99
100 /* USER CODE END 2 */
101
102 /* Infinite loop */
103 /* USER CODE BEGIN WHILE */
104 while (1)
105 {
106     /* USER CODE END WHILE */
107
108     /* USER CODE BEGIN 3 */
```

Figura 14 - Inicialização do Timer e do PWM.

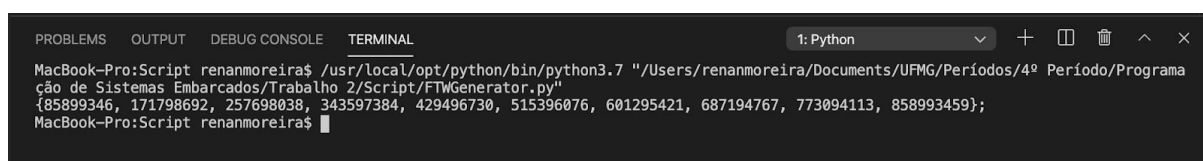
Com isso feito, pode-se partir para a codificação. Primeiramente, como discutido na síntese de funcionamento da perspectiva teórica, precisamos calcular os valores de FTW para cada frequência desejada. Para esse fim, foi desenvolvido um curto script em Python 3 para esse cálculo, como pode-se observar na figura abaixo:



```
FTWGenerator.py > ...
1 desired_frequencies = [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000];
2 input_frequency = 5000;
3 accumulator_size = 32;
4
5 FTW = map(lambda fout: round(fout*2**accumulator_size/input_frequency), desired_frequencies);
6
7 print ("{" + ", ".join(str(e) for e in FTW) + "}");
8
```

Figura 15 - Código em python para geração de FTW.

Foi produzida a seguinte saída:

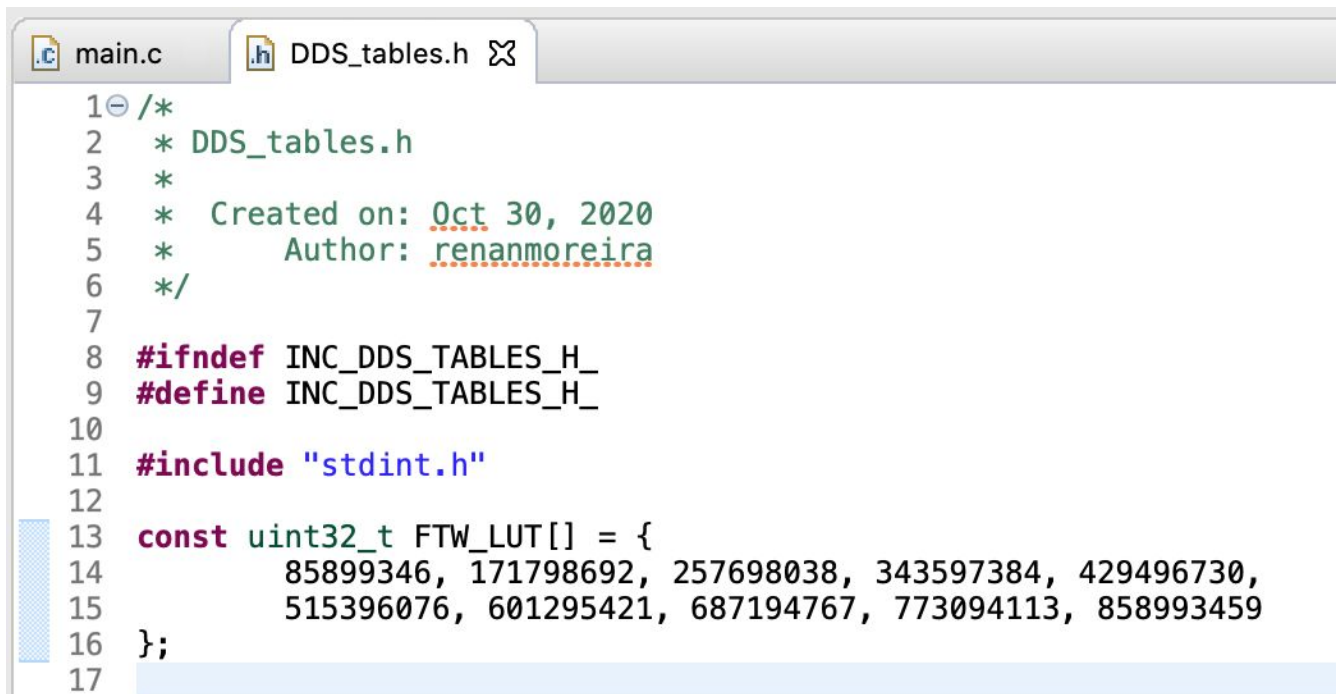


```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: Python
MacBook-Pro:Script renanmoreira$ /usr/local/opt/python/bin/python3.7 "/Users/renanmoreira/Documents/UFGM/Períodos/4º Período/Programa
ção de Sistemas Embarcados/Trabalho 2/Script/FTWGenerator.py"
{85899346, 171798692, 257698038, 343597384, 429496730, 515396076, 601295421, 687194767, 773094113, 858993459};
MacBook-Pro:Script renanmoreira$
```

Figura 16 - Saída gerada do código apresentado.

Com esse resultado gerado, basta copiar a saída para o código.

Então, criou-se um arquivo de header para as lookup tables da aplicação e definiu-se uma LUT para esses FTW calculados. É importante declará-la como *const* para garantir que o compilador entenda essa array como imutável e a guarde na memória não-volátil.



```
1 /*
2  * DDS_tables.h
3  *
4  * Created on: Oct 30, 2020
5  * Author: renanmoreira
6  */
7
8 #ifndef INC_DDS_TABLES_H_
9 #define INC_DDS_TABLES_H_
10
11 #include "stdint.h"
12
13 const uint32_t FTW_LUT[] = {
14     85899346, 171798692, 257698038, 343597384, 429496730,
15     515396076, 601295421, 687194767, 773094113, 858993459
16 };
17
```

Figura 17 - Lookup table das FTWs geradas pelo script Python.

De maneira semelhante, gerou-se a lookup table da forma de onda desejada. Como queremos uma senóide, usaremos o website sugerido na perspectiva teórica para a geração. Abaixo, temos as configurações utilizadas no website para a geração. Como discutido anteriormente, um valor usual para a quantidade de amostras da LUT é 2^{10} , então usaremos essa quantia. O tamanho da variável escolhido é de 16 bits, mas, caso deseje-se utilizar 32 bits, basta substituir o valor de “Max Amplitude” para 2^{32} .

Sine Look Up Table Generator Calculator

This calculator generates a single cycle sine wave look up table. It's useful for digital synthesis of sine waves.

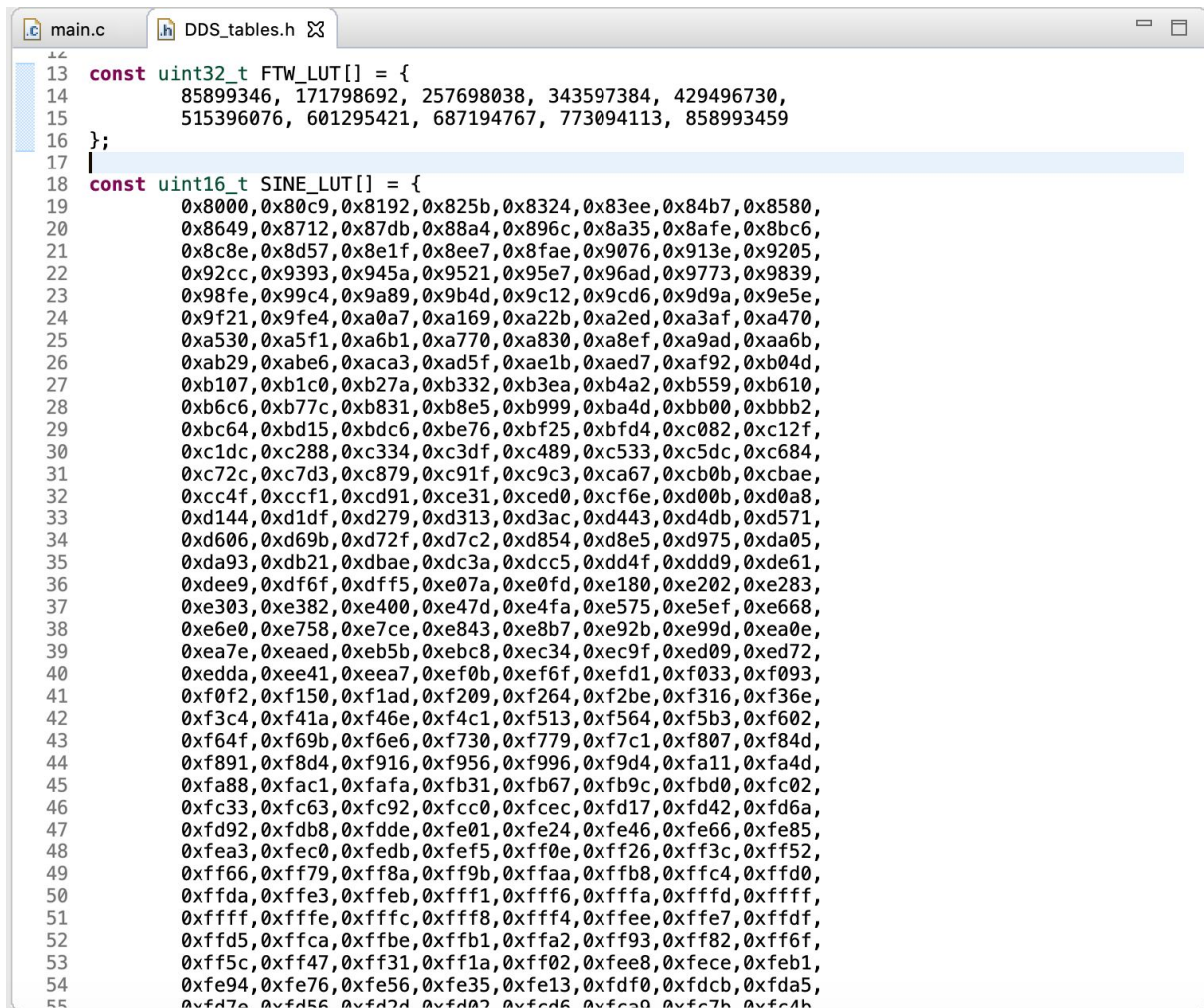
Sine Look Up Table Generator Input	
Number of points	<input type="text" value="1023"/>
Max Amplitude	<input type="text" value="65535"/>
Numbers Per Row	<input type="text" value="8"/>
<input checked="" type="radio"/> Hex	<input type="radio"/> Decimal
<input type="button" value="Submit"/>	

Figura 18 - Tela de configuração para o gerador de lookup table para uma senóide.

Esse mesmo site permite a geração de outras formas de ondas, como onda triangular, se desejado. Também, pode-se implementar um script para a geração da LUT, caso a forma de onda que deseja-se gerar ter uma outra forma mais complexa.

Assim, copia-se os valores gerados para o código, de maneira semelhante como foi feito com o FTW:

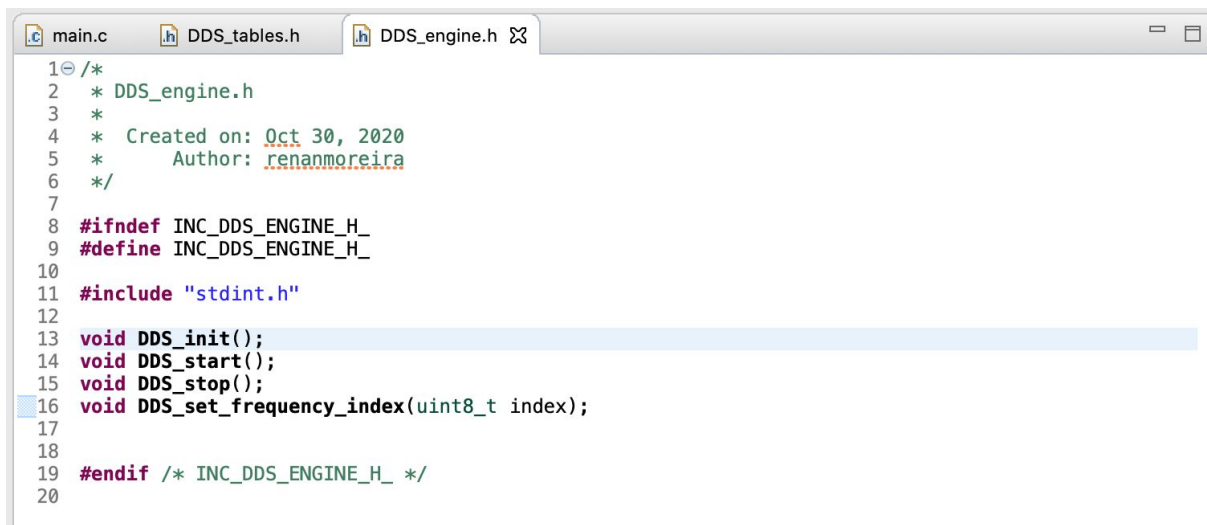
AN01000 - Modulação de Frequência através de PWM



```
13 const uint32_t FTW_LUT[] = {
14     85899346, 171798692, 257698038, 343597384, 429496730,
15     515396076, 601295421, 687194767, 773094113, 858993459
16 };
17
18 const uint16_t SINE_LUT[] = {
19     0x8000, 0x80c9, 0x8192, 0x825b, 0x8324, 0x83ee, 0x84b7, 0x8580,
20     0x8649, 0x8712, 0x87db, 0x88a4, 0x896c, 0x8a35, 0x8afe, 0x8bc6,
21     0x8c8e, 0x8d57, 0x8e1f, 0x8ee7, 0x8fae, 0x9076, 0x913e, 0x9205,
22     0x92cc, 0x9393, 0x945a, 0x9521, 0x95e7, 0x96ad, 0x9773, 0x9839,
23     0x98fe, 0x99c4, 0x9a89, 0x9b4d, 0x9c12, 0x9cd6, 0x9d9a, 0x9e5e,
24     0x9f21, 0x9fe4, 0xa0a7, 0xa169, 0xa22b, 0xa2ed, 0xa3af, 0xa470,
25     0xa530, 0xa5f1, 0xa6b1, 0xa770, 0xa830, 0xa8ef, 0xa9ad, 0xaa6b,
26     0xab29, 0xabe6, 0xaca3, 0xad5f, 0xae1b, 0xaed7, 0xaf92, 0xb04d,
27     0xb107, 0xb1c0, 0xb27a, 0xb332, 0xb3ea, 0xb4a2, 0xb559, 0xb610,
28     0xb6c6, 0xb77c, 0xb831, 0xb8e5, 0xb999, 0xba4d, 0xbb00, 0xbbb2,
29     0xbc64, 0xbd15, 0xbdc6, 0xbe76, 0xbf25, 0xbfd4, 0xc082, 0xc12f,
30     0xc1dc, 0xc288, 0xc334, 0xc3df, 0xc489, 0xc533, 0xc5dc, 0xc684,
31     0xc72c, 0xc7d3, 0xc879, 0xc91f, 0xc9c3, 0xca67, 0xcb0b, 0xcbae,
32     0xcc4f, 0xccf1, 0xcd91, 0xce31, 0xcd0, 0xcf6e, 0xd00b, 0xd0a8,
33     0xd144, 0xd1df, 0xd279, 0xd313, 0xd3ac, 0xd443, 0xd4db, 0xd571,
34     0xd606, 0xd69b, 0xd72f, 0xd7c2, 0xd854, 0xd8e5, 0xd975, 0xda05,
35     0xda93, 0xdb21, 0xdbae, 0xdc3a, 0xdcc5, 0xdd4f, 0xddd9, 0xde61,
36     0xdee9, 0xdf6f, 0xdf5, 0xe07a, 0xe0fd, 0xe180, 0xe202, 0xe283,
37     0xe303, 0xe382, 0xe400, 0xe47d, 0xe4fa, 0xe575, 0xe5ef, 0xe668,
38     0xe6e0, 0xe758, 0xe7ce, 0xe843, 0xe8b7, 0xe92b, 0xe99d, 0xea0e,
39     0xea7e, 0xeaed, 0xeb5b, 0xebc8, 0xec34, 0xec9f, 0xed09, 0xed72,
40     0xedda, 0xee41, 0xeea7, 0xef0b, 0xef6f, 0xefd1, 0xf033, 0xf093,
41     0xf0f2, 0xf150, 0xf1ad, 0xf209, 0xf264, 0xf2be, 0xf316, 0xf36e,
42     0xf3c4, 0xf41a, 0xf46e, 0xf4c1, 0xf513, 0xf564, 0xf5b3, 0xf602,
43     0xf64f, 0xf69b, 0xf6e6, 0xf730, 0xf779, 0xf7c1, 0xf807, 0xf84d,
44     0xf891, 0xf8d4, 0xf916, 0xf956, 0xf996, 0xf9d4, 0xfa11, 0xfa4d,
45     0xfa88, 0xfac1, 0xfafa, 0xfb31, 0xfb67, 0xfb9c, 0xfbd0, 0xfc02,
46     0xfc33, 0xfc63, 0xfc92, 0xfcc0, 0xfcec, 0xfd17, 0xfd42, 0xfd6a,
47     0xfd92, 0xfdb8, 0xfdde, 0xfe01, 0xfe24, 0xfe46, 0xfe66, 0xfe85,
48     0xfea3, 0xfec0, 0xfedb, 0xfef5, 0xff0e, 0xff26, 0xff3c, 0xff52,
49     0xff66, 0xff79, 0xff8a, 0xff9b, 0xffaa, 0xffb8, 0xffc4, 0xffd0,
50     0xffda, 0xffe3, 0xffeb, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
51     0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
52     0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
53     0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
54     0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
55     0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff
```

Figura 19 - LUT da senóide abaixo da dos FTWs.

Assim, podemos começar a desenvolver a engine do DDS. Vamos definir um header com a interface básica para a nossa aplicação:

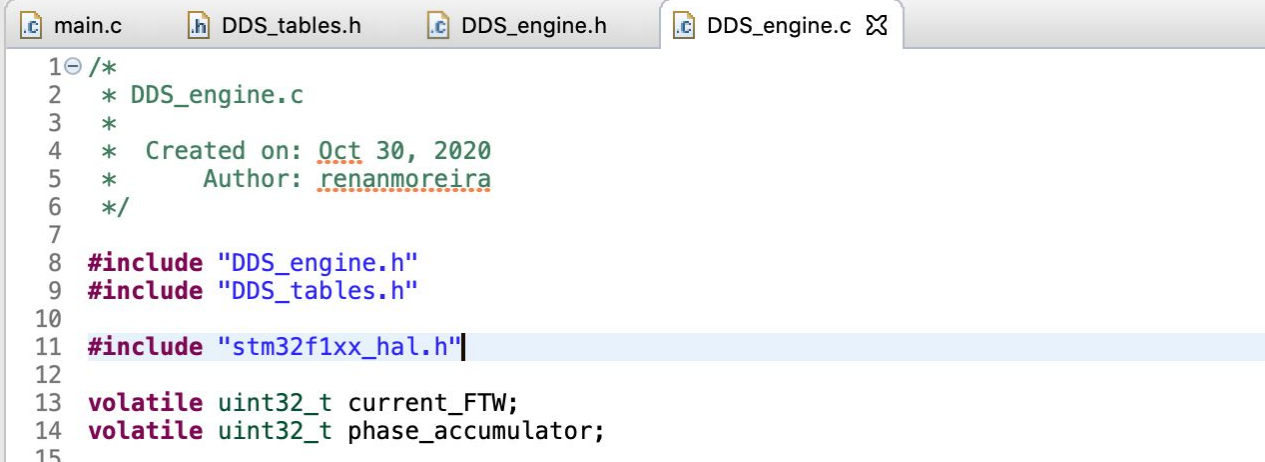


```
1 /*
2  * DDS_engine.h
3  *
4  * Created on: Oct 30, 2020
5  * Author: renanmoreira
6  */
7
8 #ifndef INC_DDS_ENGINE_H_
9 #define INC_DDS_ENGINE_H_
10
11 #include "stdint.h"
12
13 void DDS_init();
14 void DDS_start();
15 void DDS_stop();
16 void DDS_set_frequency_index(uint8_t index);
17
18 #endif /* INC_DDS_ENGINE_H_ */
```

Figura 20 - Interface da engine de DDS.

AN01000 - Modulação de Frequência através de PWM

Assim, podemos dar sequência desenvolvendo o arquivo source dessa interface para implementação da nossa lógica. Seguindo a síntese da perspectiva teórica, precisamos criar duas variáveis: a que guarda o FTW atual e o Phase Accumulator. Segue a definição delas em código:



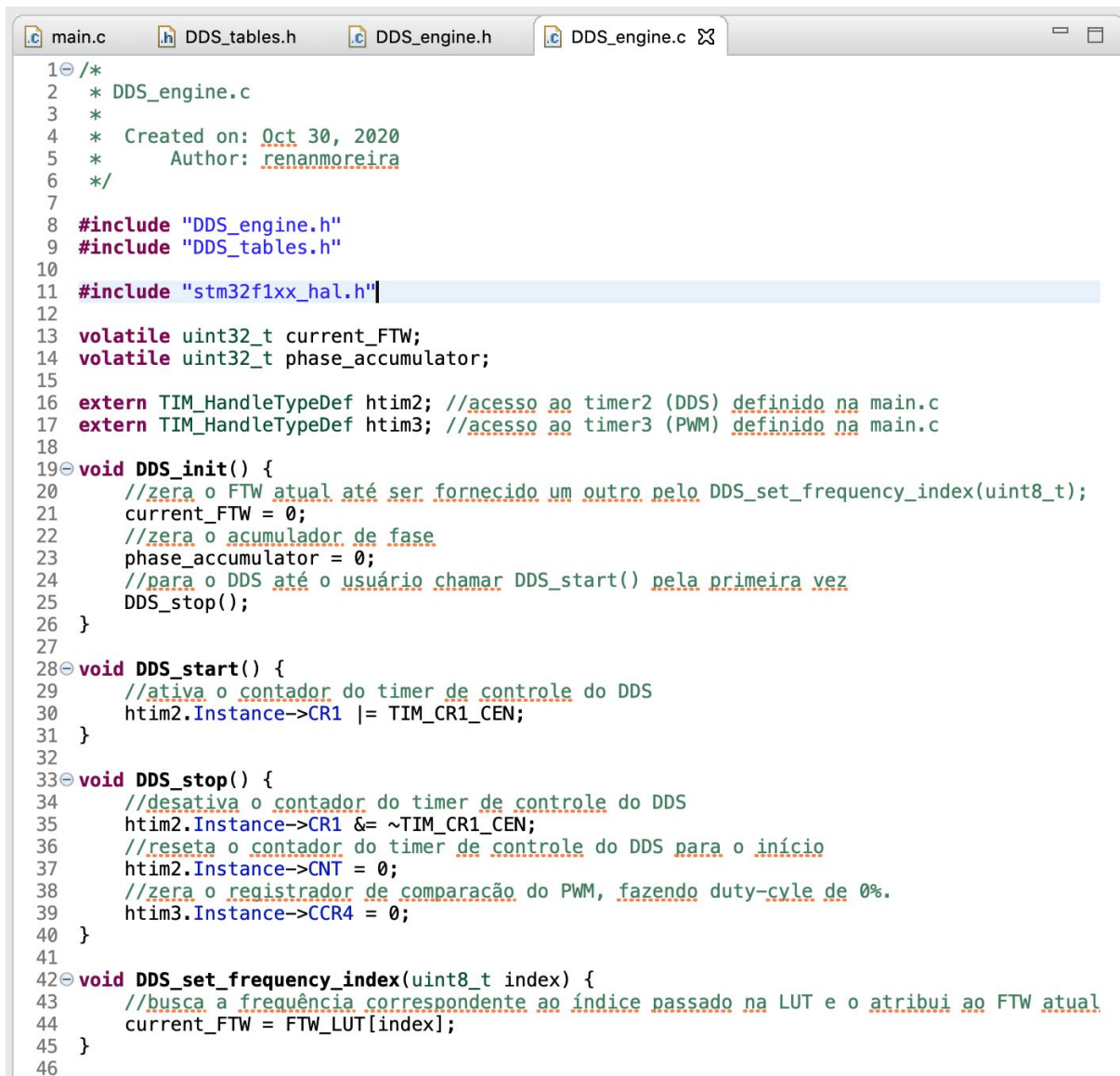
```
1  /*
2   * DDS_engine.c
3   *
4   * Created on: Oct 30, 2020
5   * Author: renanmoreira
6   */
7
8  #include "DDS_engine.h"
9  #include "DDS_tables.h"
10
11 #include "stm32f1xx_hal.h"
12
13 volatile uint32_t current_FTW;
14 volatile uint32_t phase_accumulator;
15
```

Figura 21 - Definição das variáveis.

É importante definir essas variáveis com o identificador *volatile* pois elas serão lidas e escritas durante interrupções. Dessa forma, impedimos que o compilador faça possíveis otimizações que comprometam a lógica.

Assim, pode-se seguir com a implementação das funções da interface, já comentadas para melhor entendimento:

AN01000 - Modulação de Frequência através de PWM



```
1 /*
2  * DDS_engine.c
3  *
4  * Created on: Oct 30, 2020
5  * Author: renanmoreira
6  */
7
8 #include "DDS_engine.h"
9 #include "DDS_tables.h"
10
11 #include "stm32f1xx_hal.h"
12
13 volatile uint32_t current_FTW;
14 volatile uint32_t phase_accumulator;
15
16 extern TIM_HandleTypeDef htim2; //acesso ao timer2 (DDS) definido na main.c
17 extern TIM_HandleTypeDef htim3; //acesso ao timer3 (PWM) definido na main.c
18
19 void DDS_init() {
20     //zera o FTW atual até ser fornecido um outro pelo DDS_set_frequency_index(uint8_t);
21     current_FTW = 0;
22     //zera o acumulador de fase
23     phase_accumulator = 0;
24     //para o DDS até o usuário chamar DDS_start() pela primeira vez
25     DDS_stop();
26 }
27
28 void DDS_start() {
29     //ativa o contador do timer de controle do DDS
30     htim2.Instance->CR1 |= TIM_CR1_CEN;
31 }
32
33 void DDS_stop() {
34     //desativa o contador do timer de controle do DDS
35     htim2.Instance->CR1 &= ~TIM_CR1_CEN;
36     //reseta o contador do timer de controle do DDS para o início
37     htim2.Instance->CNT = 0;
38     //zera o registrador de comparação do PWM, fazendo duty-cycle de 0%.
39     htim3.Instance->CCR4 = 0;
40 }
41
42 void DDS_set_frequency_index(uint8_t index) {
43     //busca a frequência correspondente ao índice passado na LUT e o atribui ao FTW atual
44     current_FTW = FTW_LUT[index];
45 }
46
```

Figura 22 - Implementação das funções da interface.

Nesse caso, acessou-se as instâncias do TIM2 e TIM3 que foram inicializadas no arquivo main.c por meio de um extern. Caso outros timers forem utilizados, é necessário substituir essas variáveis com os nomes das instâncias adequadas. Também é possível passar essas instâncias por referência através do init, por exemplo. Dessa forma, pode-se definir qual timer usar sem reescrever o código. No entanto, optou-se por simplificar a lógica e torná-la mais direta.

O próximo passo é, então, seguindo a síntese da perspectiva teórica, implementar a função de callback do timer de controle. Deve-se atentar à implementação utilizada: como definimos o TIM2 como timer de controle, implementa-se o callback desse timer. Segue:


```
47 void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {  
48     //verifica se a origem da interrupção é o timer de controle do DDS  
49     if (htim != &htim2) return;  
50  
51  
52 }
```

Figura 23 - Estrutura do callback para implementação do DDS.

Então, seguindo a síntese da perspectiva teórica, podemos, finalmente, implementar a lógica de DDS:

```
47 void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {  
48     //verifica se a origem da interrupção é o timer de controle do DDS  
49     if (htim != &htim2) return;  
50  
51     //incrementa o phase accumulator com o valor de FTW atual.  
52     phase_accumulator += current_FTW;  
53  
54     //pega apenas os 10 bits mais significantes do phase accumulator.  
55     //isso porque nossa LUT da forma de onda tem 2^10 posições  
56     uint16_t LUT_index = (phase_accumulator & 0xFFC00000) >> 22;  
57  
58     //busca essa posicao na lookup table da nossa forma de onda  
59     uint16_t output_value = SINE_LUT[LUT_index];  
60  
61     //como a resolucao do nosso PWM foi calculada como 2^12, e a saída  
62     //da LUT de forma de onda é de 16 bits, truncamos esse valor  
63     //novamente para caber no registrador de comparação do PWM.  
64     output_value = output_value >> 4;  
65  
66     //seta esse valor calculado no registrador de comparação  
67     htim3->Instance->CCR1 = output_value;  
68  
69     /*  
70     * de maneira alternativa, pode-se fazer essa mesma lógica em apenas uma linha:  
71     * htim3->Instance->CCR1 = SINE_LUT[((phase_accumulator+=current_FTW)&0xFFC00000)>>22]  
72     */  
73 }
```

Figura 24 - Algoritmo de DDS implementado no callback do TIM2

A lógica foi separada por estágios para facilitar a compreensão.

Agora, com a engine do DDS implementada, nos falta apenas rodá-la na main. Deve-se atentar ao fato de que é estritamente necessário chamar a função `DDS_init()`, pois ela inicializa o sistema antes de poder ser utilizado. Também é necessário chamar a função `DDS_Start()` para que o sistema comece a funcionar. Podemos observar com mais detalhes nos excertos de código abaixo:

```
19 ⊖ /* USER CODE END Header */
20 /* Includes -----
21 #include "main.h"
22
23 ⊖ /* Private includes -----
24 /* USER CODE BEGIN Includes */
25 #include "DDS_engine.h"
26 /* USER CODE END Includes */
27
```

Figura 25 - Inclusão da interface da engine de DDS criada

```
76 /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
77 HAL_Init();
78
79 /* USER CODE BEGIN Init */
80 DDS_init();
81 /* USER CODE END Init */
82
83 /* Configure the system clock */
84 SystemClock_Config();
85
```

Figura 26 - Inicialização da engine (linha 80)

```
104 while (1)
105 {
106     /* USER CODE END WHILE */
107
108     /* USER CODE BEGIN 3 */
109     DDS_start();
110     for (int i = 0; i < 10; ++i) {
111         DDS_set_frequency_index(i);
112         HAL_Delay(1000);
113     }
114     DDS_stop();
115     HAL_Delay(1000);
116 }
117 /* USER CODE END 3 */
118 }
119
```

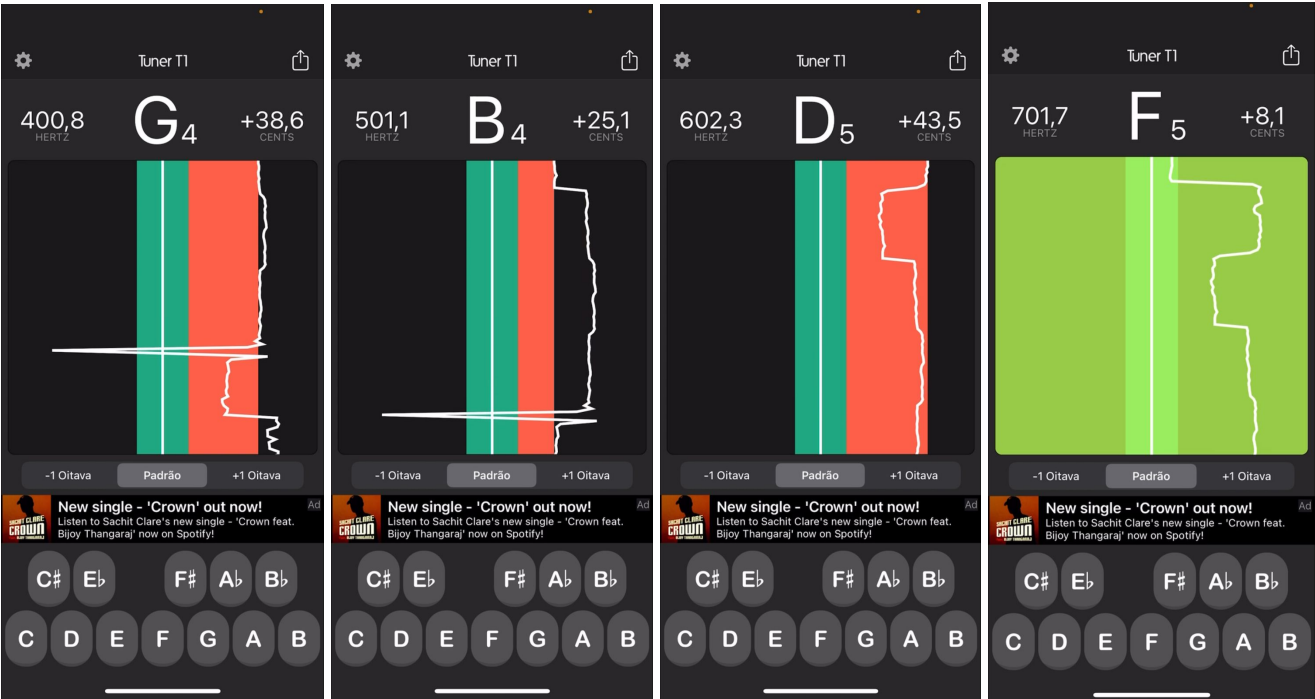
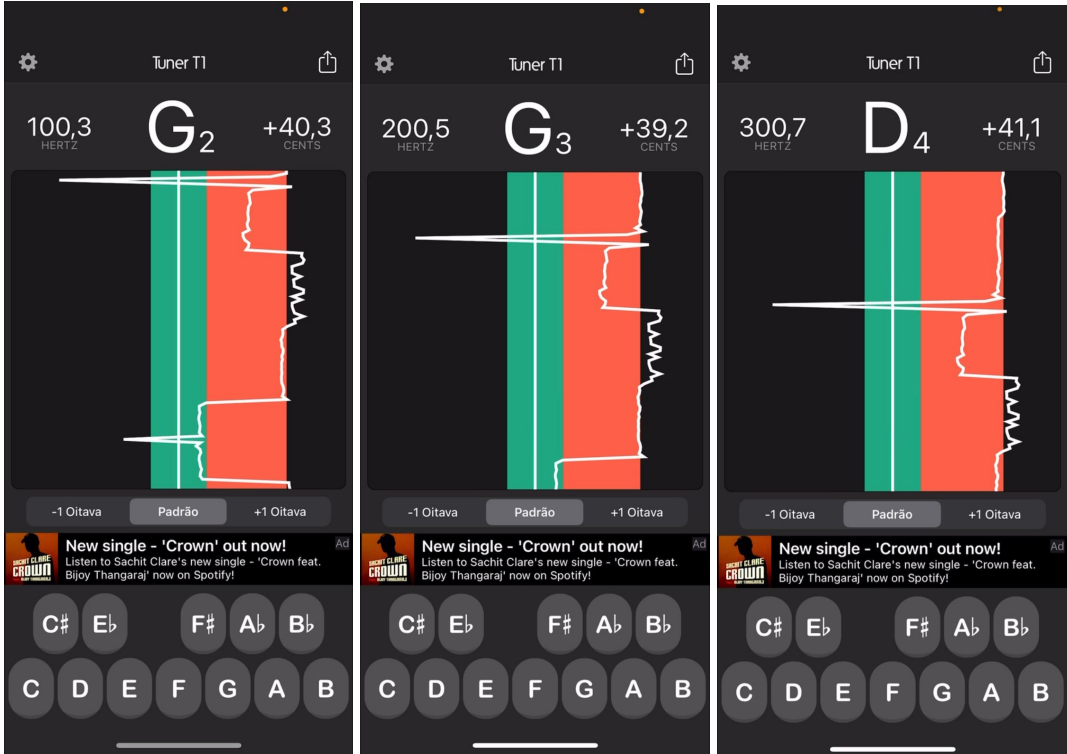
Figura 27 - Lógica implementada para averiguar funcionamento do sistema.

Pode-se notar, nessa última imagem, que a lógica feita para o teste consistiu em colocar o sistema para gerar todas as frequências projetadas em um intervalo de um segundo cada.

Para averiguar a corretude do sistema, de forma prática, a saída do PWM foi ligada a uma caixa de som e foi observado, através de um aplicativo de smartphone para afinação de instrumentos, a frequência produzida. Esse método foi utilizado devido a falta de

AN01000 - Modulação de Frequência através de PWM

equipamentos mais robustos como osciloscópios ou sistemas capazes de realizar Transformada Rápida de Fourier em tempo real. Abaixo, segue o resultado produzido para cada frequência gerada:



AN01000 - Modulação de Frequência através de PWM

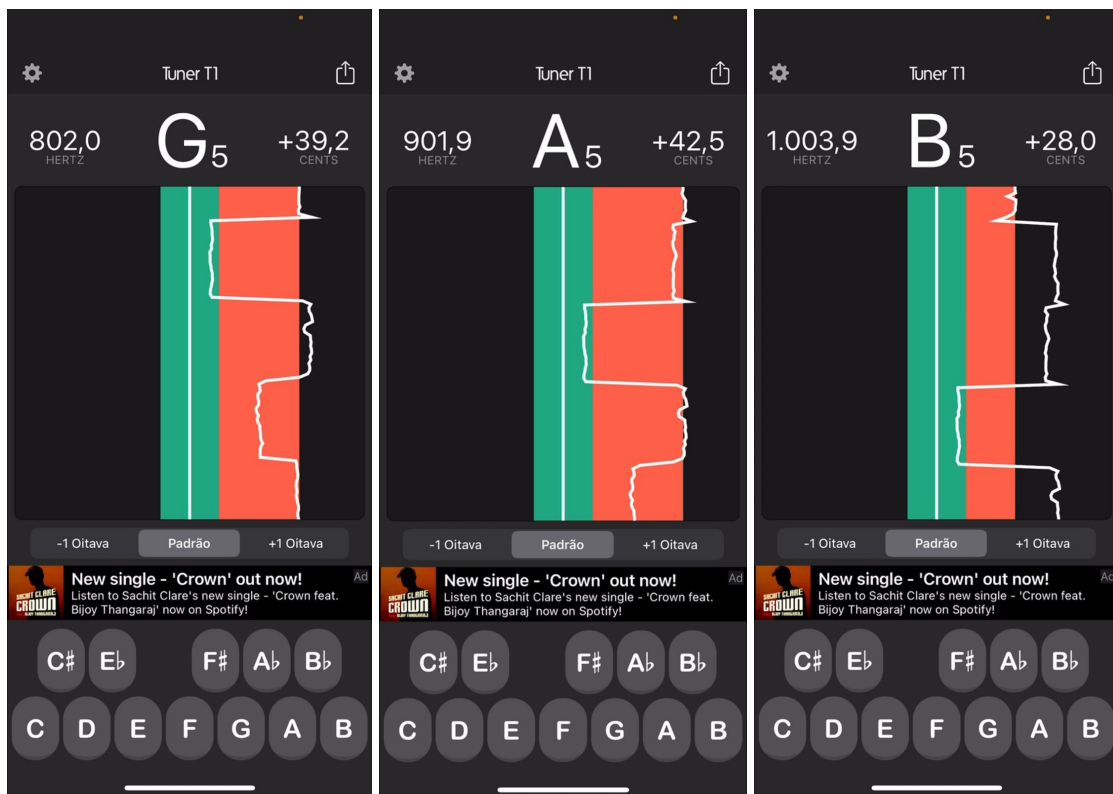


Figura 28 - Frequências geradas de 100 a 1000Hz.

Além disso, pode-se observar uma demonstração desse sistema em um vídeo no YouTube a partir do link: <https://www.youtube.com/watch?v=---L5pIXanU>. Também, vale conferir a seguinte API de geração de música polifônica a partir de um STM32, feita utilizando a mesma técnica de DDS:

<https://github.com/renanmoreira17/stm32-polyphonic-tunes>.

Licença

Copyright (C) 2020 Renan Moreira, Rodolfo de Albuquerque Lessa Villa Verde.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Referências

- TSAI, Charles. Application Report SPNA217. **Sine Wave Generation Using PWM With Hercules™ N2HET and HTU**, [s. l.], Maio 2015. Disponível em: https://www.ti.com/lit/an/spna217/spna217.pdf?ts=1603894398721&ref_url=https%253A%252F%252Fwww.google.com%252F. Acesso em: 30 out. 2020.
- MT-085 TUTORIAL. **Fundamentals of Direct Digital Synthesis (DDS)**, [s. l.], Agosto 2008. Disponível em: <https://www.analog.com/media/en/training-seminars/tutorials/MT-085.pdf>. Acesso em: 30 out. 2020.
- SINE Look Up Table Generator Calculator. [S. l.], 2019. Disponível em: <https://www.daycounter.com/Calculators/Sine-Generator-Calculator.phtml>. Acesso em: 30 out. 2020.
- PINI, Art. **The Basics of Direct Digital Synthesizers (DDSs) and How to Select and Use Them**, Digi-Key, 20 mar. 2019. Disponível em: <https://www.digikey.com/en/articles/the-basics-of-direct-digital-synthesizers-ddss>. Acesso em: 30 out. 2020.