EXPERIMENTO 8 – TPM, DMA e DMAMUX

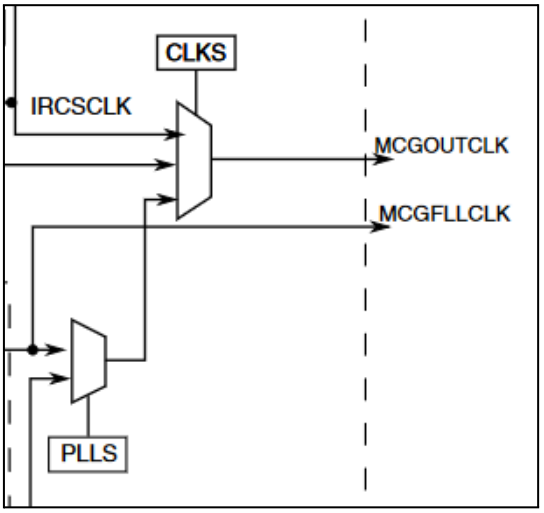
FEEC | EA871

Thiago Maximo Pavão - 247381
Vinícius Esperança Mantovani - 247395

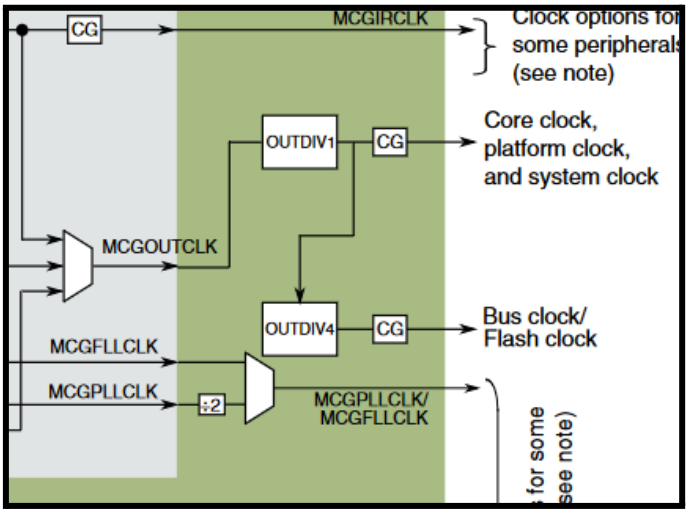
Primeira parte

1.

Inicialmente, notamos que o MCGOUTCLK é igual ao MCGFLLCLK pois o campo CLKS de MCG_C1 é configurado com o valor 0b00 no reset, o que implica que poderia ser FLL ou PLL, mas, como o campo PLLS do MCG_C6 está configurado, por padrão, com o valor 0 também no reset, selecionando o FLL.



Deve-se perceber, ainda, que em ambos os exemplos, o bus clock é o MCGOUTCLK dividido por OUTDIV1 e OUTDIV4, sendo o primeiro configurado, por padrão na memória da placa, para dividir por um. Enquanto que o segundo, é definido no reset para dividir por dois.



Isso implica que teremos o valor de MCGOUTCLK igual a FLL e, por fim o bus clock com relógio igual ao MCGOUTCLK divido por 2. Cabe ainda, ressaltar que a frequência de FLL é configurada, por meio dos campos DRST_DRS e DMX32 do registrador MCG_C4. No *rot8_example1* a frequência de FLL é definida em 48 Mhz, enquanto que, em *rot8_example2*, a frequência de FLL assume o valor de 20,971520 Mhz. Seguindo a frequência de DCO da tabela abaixo

DRST_DRS	DMX32	Reference Range	FLL Factor	DCO Range
00	0	31.25–39.0625 kHz	640	20–25 MHz
	1	32.768 kHz	732	24 MHz
01	0	31.25–39.0625 kHz	1280	40–50 MHz
	1	32.768 kHz	1464	48 MHz
10	0	31.25–39.0625 kHz	1920	60–75 MHz
	1	32.768 kHz	2197	72 MHz
11	0	31.25–39.0625 kHz	2560	80–100 MHz
	1	32.768 kHz	2929	96 MHz

Fazendo a conta, vamos que no exemplo 1 bus clock tem a frequência 24 Mhz, e no exemplo 2 a frequência 10,485760 Mhz.

Nos exemplos do manual, novamente a fonte de MCGOUTCLK é selecionada como FLL. Pois, como já visto, o campo CLKS de MCG_C1 é resetado em *0b00* por padrão e MCG_C6_PLLS em 0. OUTDIV1 e OUTDIV4 não são mencionados, então assume-se que eles receberão o valor padrão que causa a divisão por dois, portanto o bus clock tem novamente a frequência de FLL dividida por dois.

No exemplo 1 do manual, a frequência de FLL foi setada em 48Mhz, uma vez que, apesar de não se afirmar isso explicitamente no manual, diz-se em determinado trecho que “assuming that MCGFLLOUT is set to 48Mhz”, portanto em algum momento anterior da programação o valor foi configurado.

Já no exemplo 2 DRST_DRS e DMX32 não são configurados, mas no manual é possível ver que os valores são também inicializados em 0, que coloca o FLL em low range (vide tabela acima). Portanto, temos a frequência de FLL configurada em 20,971520 Mhz que nos dá novamente a frequência de bus clock 10,485760 Mhz.

A fonte de clock do TPM de maneiras equivalentes nas implementações dadas em rot8_exemplo_1 e 2 com seus respectivos exemplos do manual. Ela é configurada pelo campo SIM_SOPT2_TPMSRC, no exemplo 1 o valor é configurado em 1, que escolhe a fonte como MGCFLLCLK ou MCGPLLCLK/2.

25–24 TPMSRC	TPM clock source select Selects the clock source for the TPM counter clock 00 Clock disabled 01 MCGFLLCLK clock or MCGPLLCLK/2 10 OSCERCLK clock 11 MCGIRCLK clock

Posteriormente a frequência é selecionada pelo campo PLLFLLSEL deste mesmo registrador, que seleciona MGCFLLCLK, como já visto, neste exemplo a frequência deste clock é 48 Mhz.

Já no exemplo 2, o valor setado para esse campo SIM_SOPT2_TPMSRC é 3 (0b11), o que implica essa seleção de MCGIRCLK como fonte de TPM, cuja frequência assume o valor de 4 Mhz. Isso porque, o campo IRCS de MCG_C2 é setado em 1, selecionando, portanto, o fast internal reference clock (4 MHz) em detrimento do slow (32 kHz).

0 IRCS	Internal Reference Clock Select Selects between the fast or slow internal reference clock source.

```
MCG_C2 |= MCG_C2_IRCS_MASK;                // sinal de referencia interna rapida
```

2.

O valor de máximo (TPMx_MOD) e de *prescaler* (TPM_SC_PS) são configurados pela função TPM_config_especifica, que recebe os valores como parâmetro e utiliza os ponteiros de *struct* definidos por macro e armazenada no vetor TPM do arquivo TPM.c

```
static TPM_MemMapPtr TPM[] = TPM_BASE_PTRS;

void TPM_config_especifica (uint8_t x, uint16_t mod, uint8_t trigger, uint8_t crot,
                            uint8_t csoo, uint8_t csot, uint8_t dma, uint8_t cpwms, uint8_t ps)
```

Atribuição no registrador utilizando o vetor de endereços

```
/**
 * Configurar a contagem maxima
 */
TPM[x]->MOD = TPM_MOD_MOD(mod);
```

Configuração do prescaler, limpando o campo e escrevendo o valor passado em *ps*

```
/*
 * Configurar periodo do contador T = PS*MOD/freq.
 */
TPM[x]->SC &= ~TPM_SC_PS(0b111);
TPM[x]->SC |= TPM_SC_PS(ps);
```

No exemplo 1, TPMx_MOD é configurado em *0x12C0 = 4800* para o módulo TPM2 e *0x2BC0 = 11200* para TPM1, já no exemplo 2 utiliza-se apenas o TPM1, com valor máximo configurado em *0x07D0 = 2000*. Em ambos os exemplos e em ambos os módulos, no caso do exemplo 1, temos TPMx_SC_PS = *0b000* que divide o clock por um, mantendo o clock fonte para o contador sem alteração. A fórmula do período do módulo TPM é dada pela primeira equação do Roteiro 8, copiada aqui

$$Periodo = TPMx\_MOD \times \frac{2^{TPMx\_SC\_PS}}{f_{clock}} \times (1 + TPM\_SC\_CPWMS)$$

Para o exemplo 1, no módulo TPM1, a frequência desejada é 10 kHz, que dá um período de 10 ms. O campo TPM_SC_CPWMS configura se a contagem é progressiva ou progressiva-regressiva, em nenhum dos dois exemplos este valor é configurado, então ele assume o valor padrão do reset, zero.

$$Periodo = 4800 \times \frac{2^0}{48\times10^6} \times (1 + 0) = 10\,ms$$

Que concorda com o esperado. No caso de TPM2, a frequência desejada é 4285 Hz, que tem período igual a aproximadamente 0,2334 ms. Fazendo a conta com os valores configurados, temos

$$Periodo = 11200 \times \frac{2^0}{48\times10^6} \times (1 + 0) = 0,2334\,ms$$

Novamente, conforme o esperado. Para o exemplo 2, temos a contagem progressiva-regressiva, logo

$$Periodo = 2000 \times \frac{2^0}{4\times10^6} \times (1 + 1) = 1\,ms$$

No roteiro, a frequência é especificada em 1 Hz, o que é provavelmente um erro. Considerando o valor de 1 kHz, obtemos o período encontrado pela equação, de *1 ms*, então esta deve ser a frequência desejada.

Além deste erro, há aparentemente mais um: o texto coloca que o valor máximo é 2000, mas no código de exemplo insere o valor *0x0FA0 = 4000*, isto faria com que o período do TPM fosse dobrado, fazendo com que os pulsos fossem duas vezes mais longos mas ainda mantendo a proporção de ciclo de trabalho, já que o período com o sinal baixo também seria multiplicado por dois.

Então, as formas de onda emitidas pelo PWM seriam alteradas, mas a razão do ciclo de trabalho não, o que talvez não altere tanto a forma de onda após a passagem pelo filtro RC.

3.

Em ambos, rot8_example1 e rot8_example2, os valores em TPMx_CnSC são setados por meio de uma função de nome *TPM_CH_config_especifica*, que zera os bits de 2 a 5 e, em seguida, seta-os com o valor passado pelo parâmetro MS_ELS, conforme a imagem abaixo

```
TPM[x]->CONTROLS[n].CnSC &= ~(0b1111<<2);
TPM[x]->CONTROLS[n].CnSC |= (MS_ELS<<2);
```

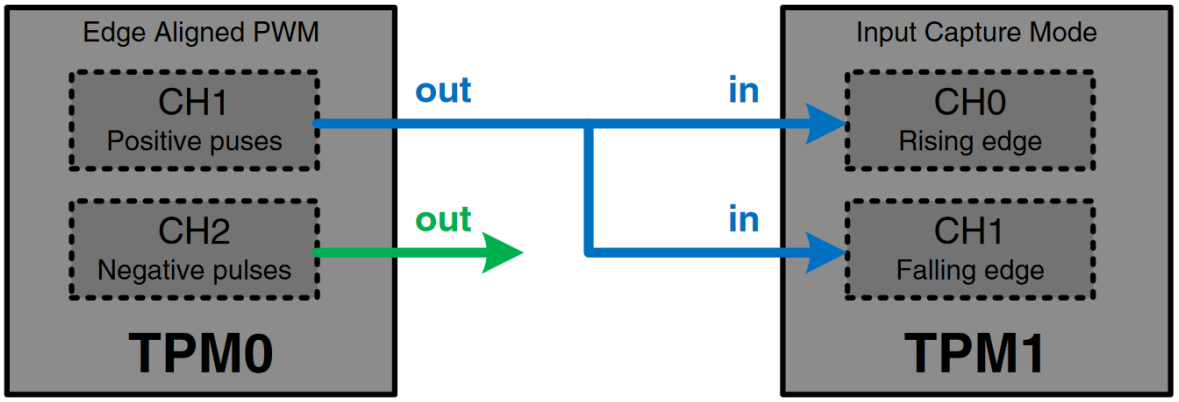
Dessa forma, em *rot8_example1*, os valores configurados em TPMx_CnSC são:
0b101000 para TPM2_C0SC; 0b100100 para TPM2_C1SC;
0b000100 para TPM1_C0SC; 0b001000 para TPM1_C1SC.

Assim, os campos preenchidos com 1 são:
TPM2_C0SC_ELSB e TPM2_C0SC_MSB; TPM2_C1SC_ELSA e TPM2_C1SC_MSB;
TPM1_C0SC_ELSA; TPM1_C1SC_ELSB.

Enquanto que, em *rot8_example2*, os valores configurados em TPMx_CnSC são:
0b101000 para TPM1_C1SC;

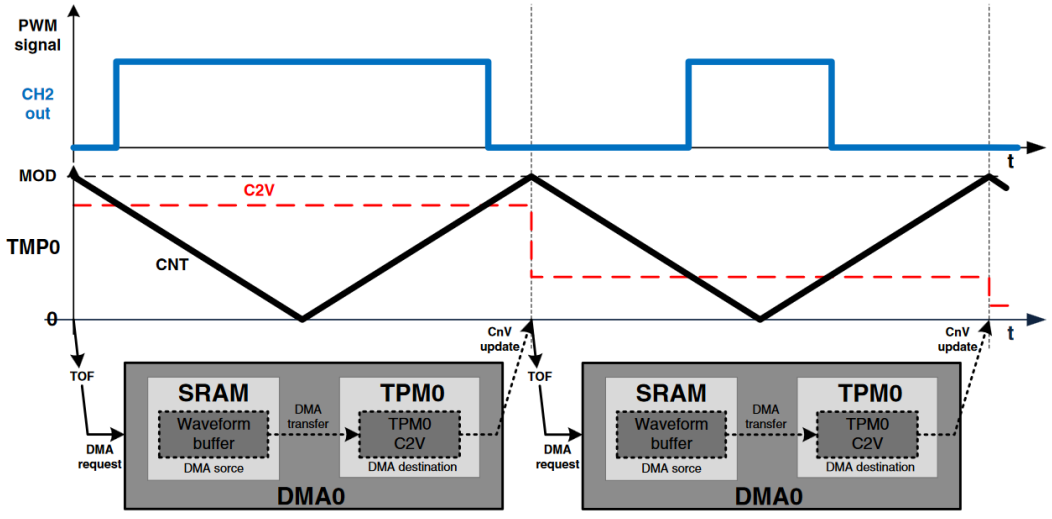
Assim, os campos preenchidos com 1 são:
TPM1_C1SC_MSB e TPM1_C1SC_ELSB;

Observando inicialmente a proposta do exemplo 1, notamos, de acordo com a imagem que segue que os canais 1 e 2 foram substituídos respectivamente por 0 e 1 na implementação e, o TPM0 é substituído por TPM2 na implementação. Primeiramente, constatamos que TPM2_CPWMS e TPM1_CPWMS assumem o valor 0, configurando a contagem para progressiva apenas. Assim, o TPM2_C0SC é configurado de maneira condizente com o que se deseja no exemplo, pois TPM2_C0SC_MSB é 1, enquanto que TPM2_C0SC_MSA é 0, o que implica que o modo é setado para PWM e, os campos TPM2_C0SC_ELSB e TPM2_C0SC_ELSA são respectivamente 1 e 0, implicando que o pulso é positivo se TPMx_CNT == 0, de maneira igual ao que se nota na imagem de exemplo 1. Quanto ao canal 1, percebemos que o campo TPM2_C1SC_MSB é 1, enquanto que MSA é 0, o que implica que o modo do canal 1 é, também, PWM. Em seguida, percebemos que TPM2_C1SC_ELSB e TPM2_C1SC_ELSA são respectivamente 0 e 1, implicando que o pulso é negativo quando TPMx_CNT == 0. Dessa maneira, vê-se que ambos os canais descritos até o momento têm funções condizentes com as explicitadas no manual.



Ainda sobre o exemplo 1, descrevamos o TPM1. Na imagem, os canais 0 e 1 de TPM1 devem ser configurados para o modo INPUT CAPTURE, respectivamente, com rising edge e falling edge. analisando, portanto, a implementação, é possível reparar que está condizente com o descrito no manual, uma vez que, ambos os campos MSA e MSB, de ambos os canais de TPM1, são 0, logo, a função desses canais é setada para INPUT CAPTURE. Além disso, o primeiro canal, canal 0, tem o valor 1 em ELSA e o valor 0 em ELSB, apresentando, portanto, borda de subida, enquanto que o canal 1, tem valor 0 em ELSA e 1 em ELSB, apresentando, portanto, borda de descida.

Agora, sobre o exemplo 2, da imagem a seguir,



vale destacar que, na implementação, o TPM0 é substituído por TPM1 e, o canal 2 foi substituído pelo canal 1 na implementação. Inicialmente, vimos que TPM1_CPWMS assume o valor 1, setando a contagem para progressiva-regressiva. Analisemos, então, a configuração do canal 1, o campo TPM1_C1SC_MSB tem valor 1, enquanto que, o campo TPM1_C1SC_MSA tem valor 0, então, o modo do canal é PWM. Além disso, os campos TPM1_C1SC_ELSB e TPM1_C1SC_ELSA têm valor, respectivamente, 1 e 0, lançando, por tanto, pulso positivo da primeira ocorrência de TPM1_CNT == TPM1_C1V até a segunda (contagens progressiva e regressiva, respectivamente). Logo, o TPM está configurado de maneira condizente com aquela descrita para o exemplo 2 no manual.

4.

Os pinos são configurados pelo campo MUX do campo PORTx_PCRn, multiplexando as portas para serem utilizadas pelos canais. No exemplo 1, estes valores são configurados na função, e no bloco de código a seguir

Nome da função: *TPM1TPM2_PTE20PTE21PTE22PTE23_config_basica*

```
PORTE_PCR20 |= (PORT_PCR_ISF_MASK |          //TPM1_CH0
        PORT_PCR_MUX(0x3));
PORTE_PCR21 |= (PORT_PCR_ISF_MASK |          //TPM1_CH1
        PORT_PCR_MUX(0x3));
PORTE_PCR22 |= (PORT_PCR_ISF_MASK |          //TPM2_CH0
        PORT_PCR_MUX(0x3)|
        PORT_PCR_DSE_MASK);                  //drive strength enable
                                             //e.g. 18 mA vs. 5 mA @ > 2.7 V,
PORTE_PCR23 = (PORT_PCR_ISF_MASK |           //TPM2_CH1
        PORT_PCR_MUX(0x3)|
        PORT_PCR_DSE_MASK);
```

É possível ver que todos os campos MUX são configurados em *3*, consultando a tabela de multiplexação dos pinos no manual de referência, vemos pela coluna *ALT3* que isso multiplexa os pinos para os canais dos módulos TPM:

- TPM1_CH0 em PTE20
- TPM1_CH1 em PTE21
- TPM2_CH0 em PTE22
- TPM2_CH1 em PTE23

No exemplo 2, *TPM1DMA_PTE21_config_basica* configura o único canal ativo: TPM1_CH1, para saída no pino PTE21.

```
PORTE_PCR21 = (PORT_PCR_ISF_MASK |           //TPM1_CH1
        PORT_PCR_MUX(0x3)|
        PORT_PCR_DSE_MASK);                  //acionamento maior na corrente de saida
```

E *GPIO_initSwitchNMI* configura a botoeira NMI

```
    // Configura modo de interrupcao
    PORTA_PCR4 |= PORT_PCR_ISF_MASK |
//            PORT_PCR_PS_MASK|                         //pullup
//            PORT_PCR_PE_MASK|                         //habilitar pull
              PORT_PCR_PFE_MASK|                        //habilitar filtro passivo
              PORT_PCR_IRQC(IRQC);              // modo de interrupcao
```

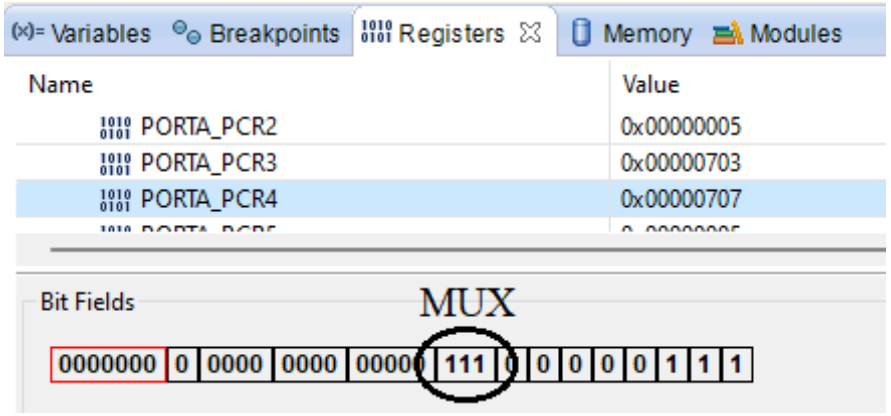
As máscaras comentadas foram recomendadas pelo manual, mas não são necessárias porque a chave NMI usada já tem um resistor pullup externo conectado.

Por fim, em rot8_aula a função *TPM0TPM1_PTA4PTE21_config_basica* configura a multiplexação de dois pinos

```
PORTA_PCR4 &= ~PORT_PCR_MUX(0x4);          // PTA4 em TPM0_CH1
PORTE_PCR21 |= PORT_PCR_MUX(0x3);          // PTE21 em TPM1_CH1
```

- TPM0_CH1 em PTA4
- TPM1_CH1 em PTE21

Neste caso, para PTA4, nota-se que ao invés de escrever 1 nos dois bits menos significativos de MUX, é escrito um 0 no terceiro bit, o que tem o mesmo efeito se este registrador tiver como valor inicial algo da forma 0bX11. Pois desta forma, após a configuração o valor será 3. No manual, o valor do campo após reset é indeterminado, mas executando o programa em modo debug e parando a execução antes da configuração do campo vemos que o valor inicial no caso deste registrador é realmente 0b111



Enquanto que para os outros vistos até agora o valor inicial era da forma 0b0XX, e a operação de OU com 0b11 já era suficiente.

Por fim, vale notar que alguns pinos, os dos canais configurados em PWM tem o campo POTRx_PCRn_DSE, que aumenta a corrente que pode ser consumida pelo pino. Isto faz sentido visto que é esperado que seja consumido certo nível de corrente já que é um pino de saída.

5.

No manual, TPM1_CONF é configurado pela linha

TPM1_CONF=TPM_CONF_TRGSEL(8)|TPM_CONF_CSOO_MASK|TPM_CONF_CSOT_MASK

Na implementação dada em *rot8_example1* a configuração é feita pela chamada da função *TPM_config_especifica*, e os valores dos campos de configuração são enviados por parâmetro, mas o efeito é o mesmo.

TRGSEL tem o valor *8 = 0b1000*, que configura o evento de overflow em TPM0 como disparo.

Table 3-38. TPM trigger options

TPMx_CONF[TRGSEL]	Selected source
0000	External trigger pin input (EXTRG_IN)
0001	CMP0 output
0010	Reserved
0011	Reserved
0100	PIT trigger 0
0101	PIT trigger 1
0110	Reserved
0111	Reserved
1000	TPM0 overflow
1001	TPM1 overflow
1010	TPM2 overflow
1011	Reserved
1100	RTC alarm
1101	RTC seconds
1110	LPTMR trigger
1111	Reserved

CSOO faz com que o contador pare de contar quando atinge o valor máximo, o valor do contador ainda é reiniciado em 0, esta flag faz apenas com que após o reinício a contagem não continue. CSOT configura para que o contador seja iniciado quando ocorre o disparo. Ademais, CROT não foi configurado, e seu valor na inicialização é especificado pelo manual em zero, que faz com que o contador não seja reiniciado em zero quando ocorre o evento configurado no disparo

Juntando todas estas configurações espera-se o seguinte comportamento: O contador é incrementado até alcançar seu valor máximo, o valor é reiniciado mas a contagem não. Quando ocorre o overflow em TPM0, a contagem se inicia novamente, reiniciando o ciclo. É isto que é observado nas formas de onda dadas pelo manual, conforme o esperado.

Por fim, não são necessárias mais instruções para garantir que o comportamento seja cíclico, isto porque o módulo é capaz de repetir o comportamento configurado em TPM1_CONF de forma interna.

6.

1) Comecemos por afirmar que FTM1_IRQHandler e FTM2_IRQHandler são, respectivamente, a rotina de tratamento de interrupções de TPM1 e de TPM2. FTM1_IRQHandler e FTM2_IRQHandler correspondem, respectivamente, ao overflow do contador TPM1 e do contador TPM2. A prioridade de FTM1_IRQHandler é setada em 1, enquanto que a prioridade de FTM2_IRQHandler é setada em 3. Esta configuração é feita pela chamada da função TPM_habilitaNVICIRQ, ativando as IRQs 18 e 19 de acordo com a prioridade passada como parâmetro.

2) Na rotina FTM2_IRQHandler, ocorre que TPM2_C0V e TPM2_C1V têm seus valores alterados de acordo com a quantidade de vezes que a interrupção aconteceu, ou seja, sempre que ocorre um overflow do contador de TPM2, a rotina é chamada e o valor desses campos é alterado de acordo com a quantia de vezes que já foi chamada a rotina, conforme ilustrado a seguir:

```
uint16_t u16PWMDuty[5] = {480, 4320, 2400, 1440, 4320};
```

→ vetor de números é declarado (usado para alterar os valores dos campos abaixo);

```
static uint8_t ind=0;
```

→ declaração do índice do vetor (varia de acordo com o número de vezes que foi chamada a rotina). Variável estática, então a atribuição (=0) é feita somente uma vez.

```
TPM2_C0V = u16PWMDuty[ind];
TPM2_C1V = u16PWMDuty[ind];
```
→ alteração dos campos;

```
ind=(ind+1)%5;
```

→ alteração do valor de ind (como é estático, na próxima chamada de rotina, permanece com o mesmo valor, logo, os campos citados assumirão um valor diferente do assumido na chamada atual);

```
TPM2_SC |= TPM_SC_TOF_MASK;
```
→ limpa a flag de overflow para terminar a rotina.

Desse modo, a cada chamada, o valor dos campos citados varia e, em consequência, as larguras dos pulsos representadas pelas linhas vermelhas tracejadas é alterada de acordo com o valor que os campos assumem, pois elas são a representação desses campos.

7.

1) Ao setar o DMAMUX0_CHCFG0 |= DMAMUX_CHCFG_SOURCE(55) no código, setamos o valor no campo MAMUX0_CHCFG0 é o referente ao 55 na tabela 3-20 da página 64 do manual, o que implica que temos como fonte para a transferência o evento de overflow no contador de TPM1. Além disso, o canal selecionado de DMA é o canal 0, conforme se nota na chamada da função DMA0_MemoTPM1CH1_config_especifica, de DMA.c, na main, que chama a função DMAMUX_setaReq com o valor do parâmetro x igual a 0, o que implica que o canal 0 é desabilitado, em seguida a fonte desse canal é determinada e, por fim o canal é novamente habilitado, conforme se vê na imagem:

```
void DMAMUX_setaReq (uint8_t x, uint8_t src) {

    DMAMUX->CHCFG[x] &= ~DMAMUX_CHCFG_ENBL_MASK;

    DMAMUX->CHCFG[x] |= DMAMUX_CHCFG_SOURCE(src);    // Tabela 3-20/pag. 64 do Manual

    DMAMUX->CHCFG[x] |= DMAMUX_CHCFG_ENBL_MASK;

    return;
}
```

Ainda vale notar que no manual o código inserido como fonte é 54, que corresponde ao evento de overflow do contador de TPM0, isto não foi mantido pois não há acesso à pinos para comunicação com canais deste módulo no *shield*, e a mudança para o módulo TPM1 exige esta diferença na configuração.

2) Inicialmente, os endereços de DMA_SAR0 e DMA_DAR0 são configurados pela função *DMA0_MemoTPM1CH1_config_especifica* de DMA.c, chamada na main. Esta função chama *DMA_SAR_DAR*, deste mesmo arquivo, com os parâmetros *src* e *dst* valendo respectivamente "end" e (uint32_t *)&TPM1_C1V. Notemos que "end" e, por consequência DMA_SAR0, vem do parâmetro passado para *DMA0_MemoTPM1CH1_config_especifica* na chamada feita na main, logo, vale (uint32_t *)au16PwmDuties, constante definida na main que contém as larguras das formas de onda que serão geradas. E, DMA_DAR0 assume o valor de *dst* que vem da definição por macro:

```
#define TPM1_C1V                                    TPM_CnV_REG(TPM1_BASE_PTR,1)
```

Temos também, que DMA_DCR0_DSIZE e DMA_DCR0_SSIZE são ambos configurados pela mesma função (*DMA_SAR_DAR*) com os valores passados à ela. Analisando na chamada, vemos que os valores passados são *ssize = 2 = 0b10* e *dsize = 2 = 0b10*. Pelo manual, vemos que isto faz com que os tamanhos dos dados da fonte e destino sejam configurados em *16 bits = 2 bytes*, que é coerente com o tipo dos dados do vetor que serão enviados: *uint16_t;*

3) O campo DMA_DCR_SINC é configurado para que os dados do vetor sejam transmitidos em ordem, fazendo com que inicialmente o primeiro valor do vetor seja transmitido, e então o endereço da próxima transmissão seja incrementado em 2 bytes (valor configurado do tamanho dos dados da fonte, DMA_DCR0_SSIZE) e então que o próximo dado seja transmitido quando houver a próxima transferência, e não o mesmo.

   Já o campo DMA_DCR_DINC é configurado em zero pois deseja-se que os valores sejam sempre escritos em TPM1_C1V, já que é este campo que configura a largura dos pulsos gerados pelo PWM. Se o bit fosse setado em 1, a próxima transferência de dados teria como destino um endereço potencialmente inválido, e a execução não seria como previsto.

4) No exemplo, existem três *buffers* armazenando as larguras para formar as formas de onda quadrada, triangular e senoidal, na implementação dada em *rot8_example2* estes vetores são au16PwmBuffSin, au16PwmBuffTrg e au16PwmBuffSqr. Há ainda mais um vetor, au16PwmDuties e este é o único *buffer* que é lido de forma circular pelo módulo DMA. Os dados nos outros vetores são copiados para este na inicialização e quando a botoeira é

pressionada. A implementação é feita da mesma forma no exemplo do manual, em relação aos *buffers* circulares: um na fonte dos dados e sem buffer no destino.

8.

1) A forma de onda amostrada é alterada quando o valor da variável u8BuffSelect estática de ISR.c tem seu valor alterado, pois, desse modo, na main, o valor da variável "atual" passa a ser diferente do valor da variável "anterior", o que ocasiona a entrada em `if (atual != anterior)` . Neste if, há um switch que define a forma de onda de acordo com o valor de atual, que contém o valor de u8BuffSelect

```
switch(atual)
{
case 1: p_aul6PwmDuties = (uint16_t *)aul6PwmBuffSin; break;
case 2: p_aul6PwmDuties = (uint16_t *)aul6PwmBuffTrg; break;
case 3: p_aul6PwmDuties = (uint16_t *)aul6PwmBuffSqr; break;
default: break;
}
```

sendo assim, cabe explicar que a variável p_aul16PwmDuties é um ponteiro que contém o endereço do vetor que será copiado para o vetor au16PwmDuties, setado como fonte do DMA usado, que é declarado anteriormente e é, inicialmente, preenchido pelos elementos do vetor correspondente à forma de onda quadrada. Assim, sendo alterado o valor de u8BuffSelect, o vetor fonte de DMA é alterado para conter os elementos do vetor correspondente ao valor de u8BuffSelect no switch exposto.
Assim, devemos destacar o que pode causar alteração no valor de u8BuffSelect. Para tanto, devemos analisar as rotinas de serviço DMA0_IRQHandler e PORTA_IRQHandler. A primeira rotina citada é chamada quando ocorre o fim de uma transferência de DMA e, altera o valor de u8BuffSelect na linha `u8BuffSelect--;` e no if

```
if (!u8BuffSelect)
{
    u8BuffSelect = 3;
}
```

caso seu valor seja diferente de todos os utilizados no switch da main, ou seja, se for 0. Sendo assim, notamos que, sempre, no fim de uma transferência, a forma de onda é alterada.
Além disso, a rotina PORTA_IRQHandler é chamada quando a botoeira NMI é pressionada e, altera o valor de u8BuffSelect na linha `u8BuffSelect--;` e no if (de maneira semelhante ao feito na outra rotina citada)

```
if (!u8BuffSelect)
{
    u8BuffSelect = 3;
}
```

Sendo assim, percebe-se que, sempre que a botoeira NMI é pressionada, ocorre a alteração da forma de onda.
Vale ressaltar que, por conta de sempre que o DMA encerrar uma transferência a forma de onda mudar, ocorre que a alteração causada pelo pressionamento da botoeira não ocasiona nenhuma mudança significativa, pois a forma é logo alterada novamente por conta de um fim de transferência.

2) O bloco de instruções referido é o seguinte

```
for (i = 0; i < 64; i++)
{
    au16PwmDuties[i] = p_au16PwmDuties[i]>>1;
}
```

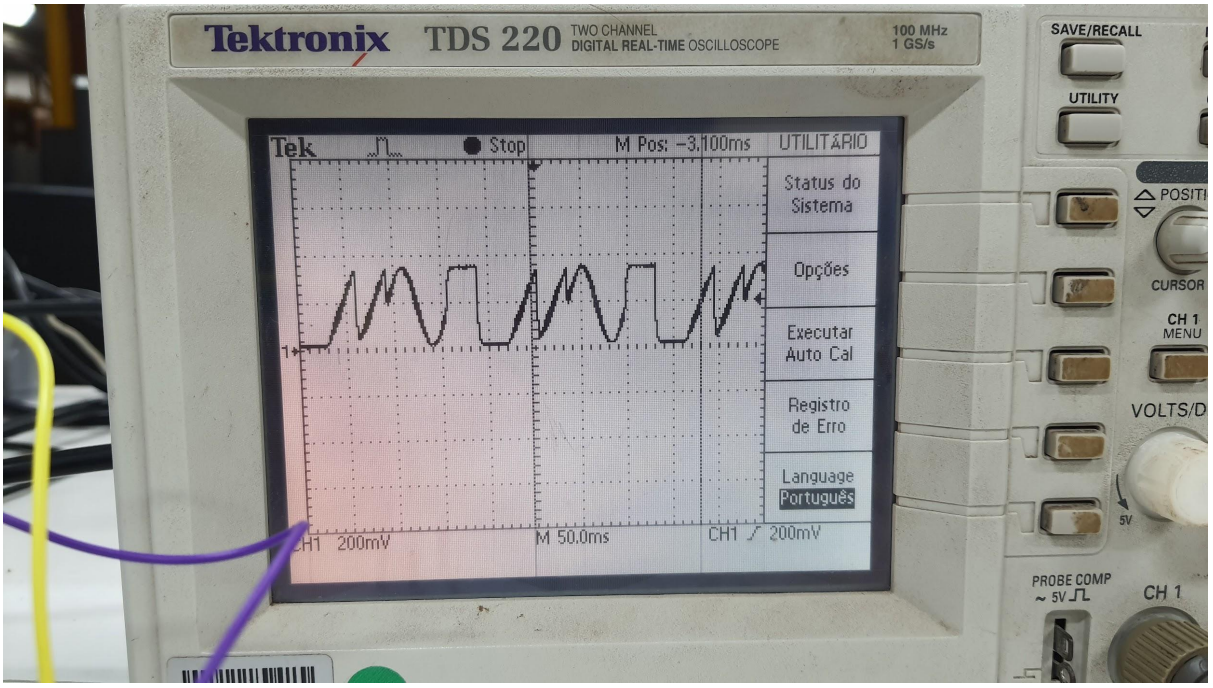
No qual são copiados os elementos de um dos três vetores representantes das formas de onda (au16PwmBuffSin, au16PwmBuffTrg, au16PwmBuffSqr) para o vetor fonte de DMA0, aul16PwmDuties, por meio do ponteiro de vetor p_au16PwmDuties que aponta para o vetor que contém a forma de onda desejada.

3) Inicialmente, os modos VLPx são ativados por meio do campo SMC_PMPROT_AVLP na linha `SMC_PMPROT |= SMC_PMPROT_AVLP_MASK;`. Em seguida, o modo VLPS é habilitado, por meio do campo SMC_PMCTRL_STOPM, na linha `SMC_PMCTRL |= SMC_PMCTRL_STOPM(2);  //Very-Low-Power Stop`. E, por fim, há a instrução para configurar o deepsleep em `SCB_SCR |= SCB_SCR_SLEEPDEEP_MASK;  //Habilitar deep sleep (System Control Block (B3.2.7/p. 270) em ARMv6)`

Ademais, percebe-se que a ordem das instruções altera a execução do programa, uma vez que não é possível configurar o modo VLPS sem ter habilitado os VLPx anteriormente, ou seja, a linha `SMC_PMCTRL |= SMC_PMCTRL_STOPM(2);  //Very-Low-Power Stop`, não pode vir antes da linha `SMC_PMPROT |= SMC_PMPROT_AVLP_MASK;`.

9.

Na implementação dada, a forma de onda selecionada é alterada sempre que ela é transmitida, o que faz com que o sinal fique alternando entre a onda quadrada, triangular e senoidal de forma estranha, gerando a seguinte relação de tensão pelo tempo

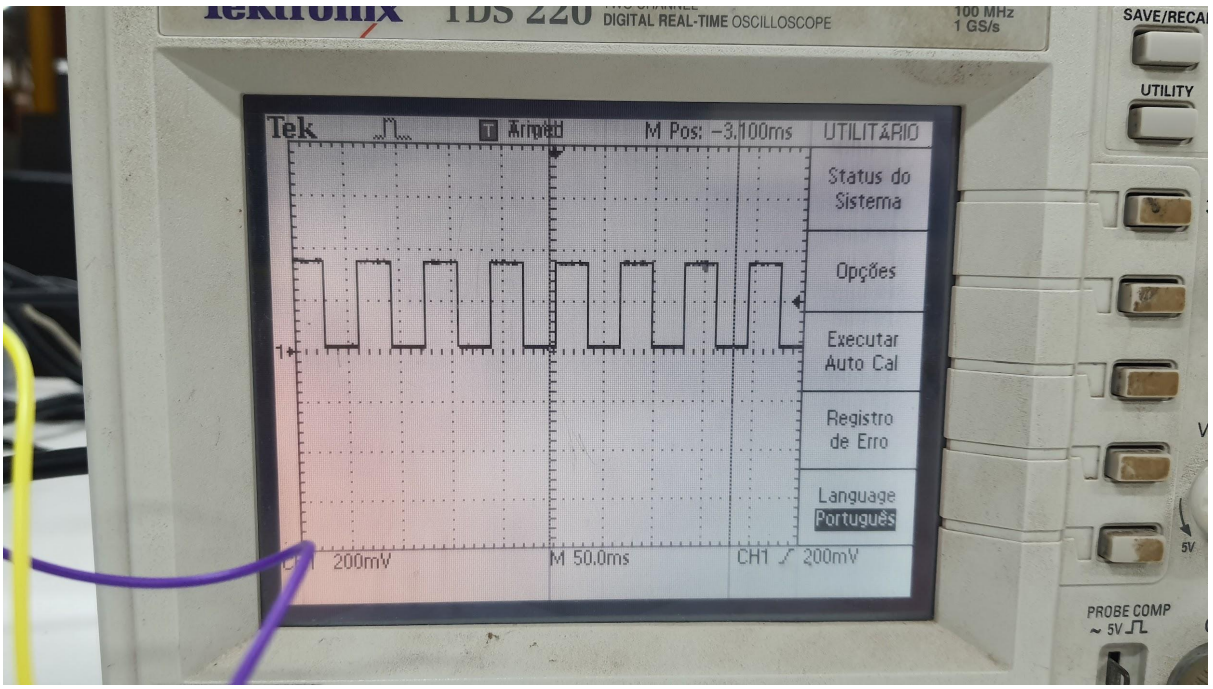


Para visualizar apenas uma mesma onda por vez, o seguinte bloco de código foi comentado, desta maneira, a forma de onda selecionada pode ser alternada entre as opções apenas pelos pressionamentos da botoeira NMI.

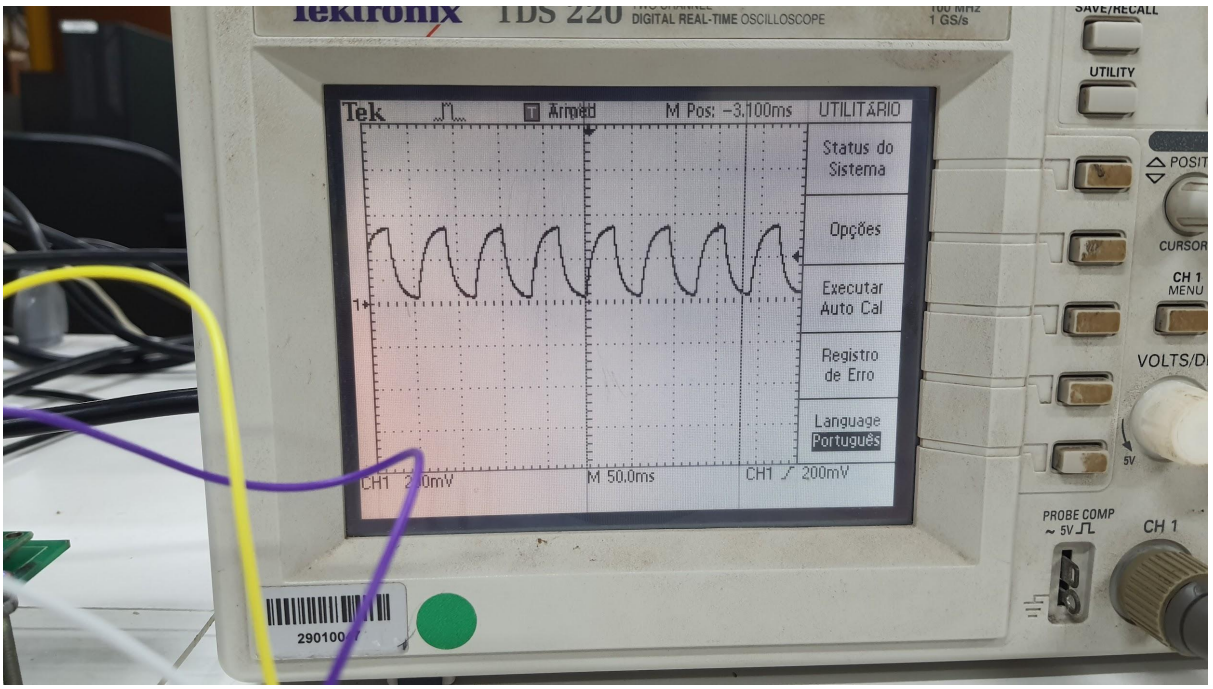
```
void DMA0_IRQHandler () {
    DMA_DSR_BCR0 |= DMA_DSR_BCR_DONE_MASK;

//  DMA_DSR_BCR0 |= DMA_DSR_BCR_BCR(64000); // tamanho do bloco a ser transferido em bytes ...
    DMA_DSR_BCR0 |= DMA_DSR_BCR_BCR(BlockSize);

    DMA_SAR0 = (uint32_t) end_SAR; //!!! eh necessario reiniciar o endereco inicial do bloco

//  u8BuffSelect--;
//
//  //!!! Manter os valores de u8BuffSelect validos
//  if (!u8BuffSelect)
//  {
//      u8BuffSelect = 3;
//  }

}
```

Vemos que a onda quadrada é formada melhor utilizando o capacitor de 22 nF, que forma a menor constante de tempo do circuito, quanto mais alta a capacitância mais o sinal é distorcido, deixando de ser uma onda quadrada ao passo que as trocas de nível demoram mais tempo.

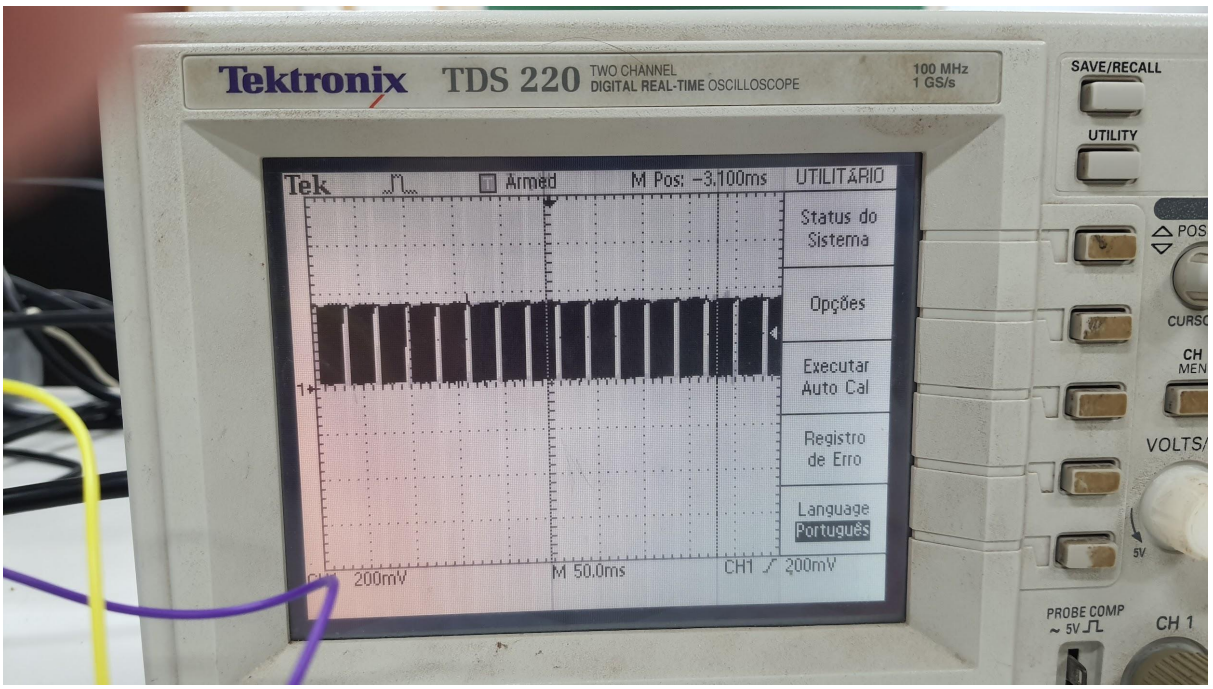
Para 22 nF:



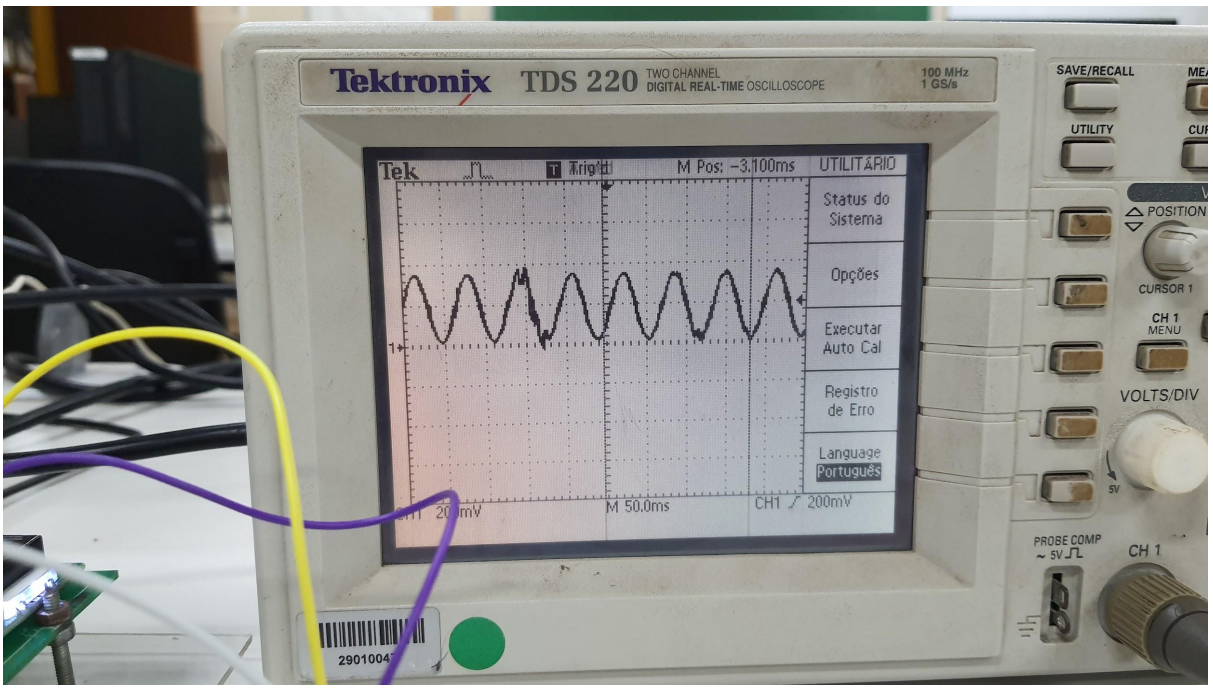
Para *68 uF:*

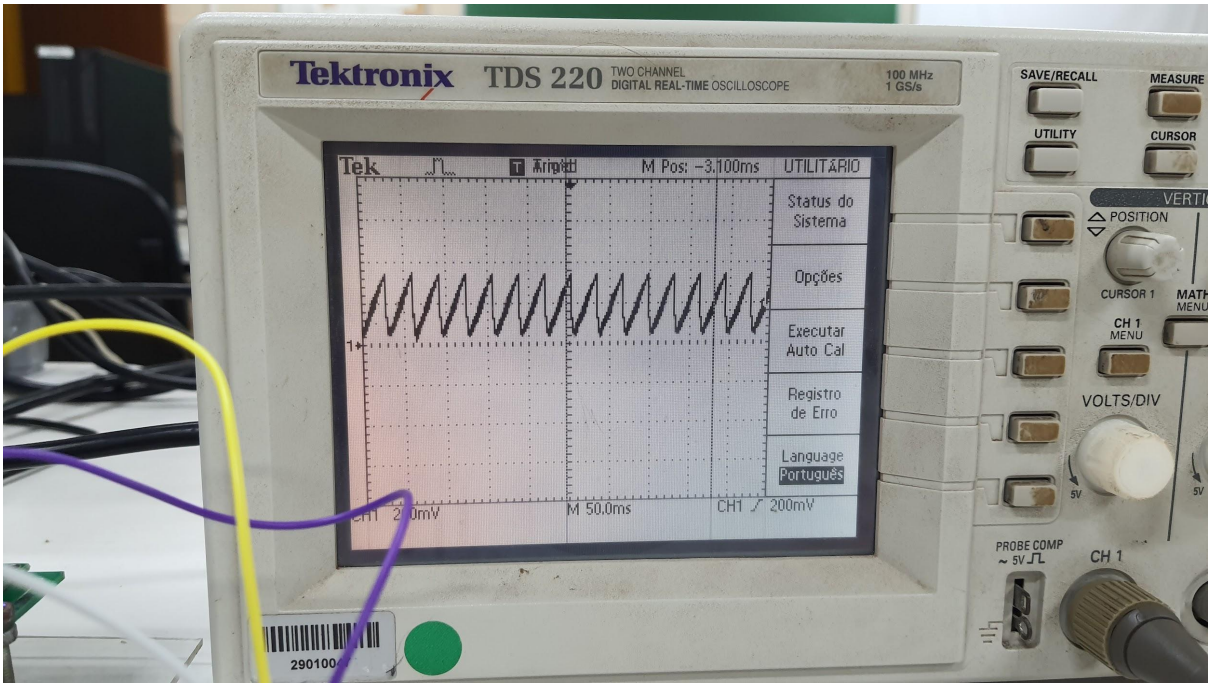


No entanto, esta vantagem da pequena capacitância só se mantém para esta forma de onda, as outras exigem a filtragem, pois o sinal de saída é formado apenas por nível digital alto e baixo, e a filtragem é responsável por fazer as formas de onda analógicas.
Por exemplo, o sinal triangular com o capacitor de menor capacitância resulta em uma forma de onda na saída mostrada na imagem abaixo. O sinal senoidal resulta em algo parecido, o que indica que é necessário aumentar a filtragem, o valor da capacitância.



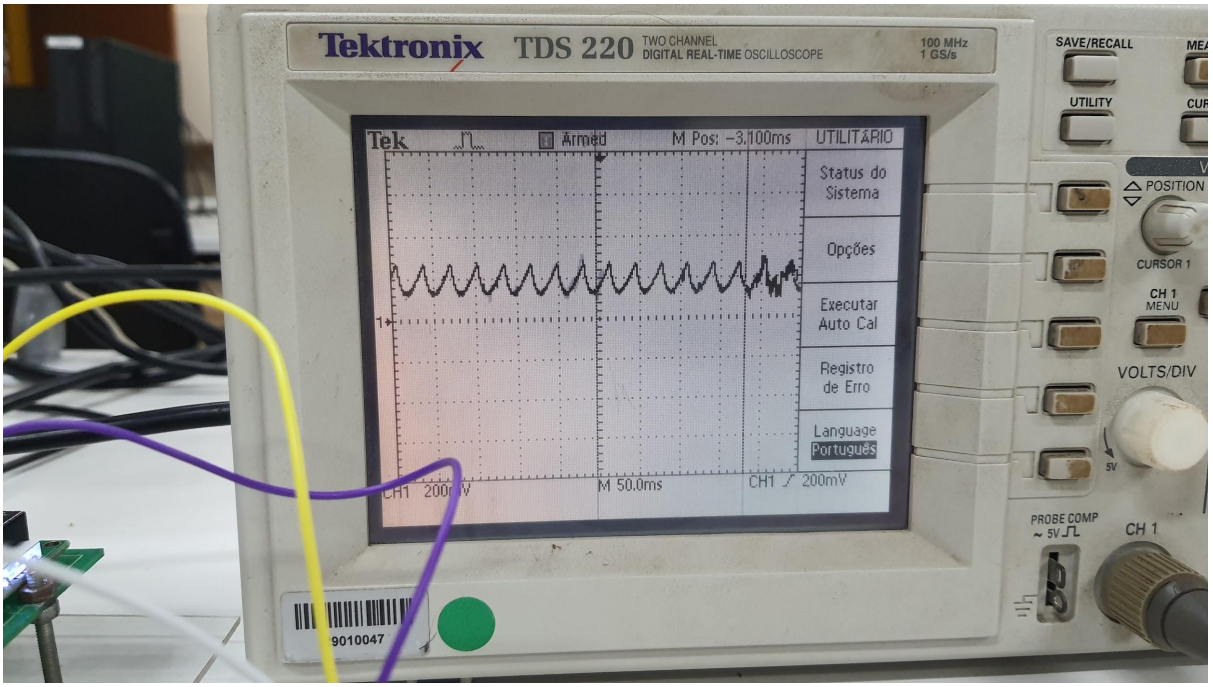
Aumentando para *10 uF*, temos a forma triangular e senoidal



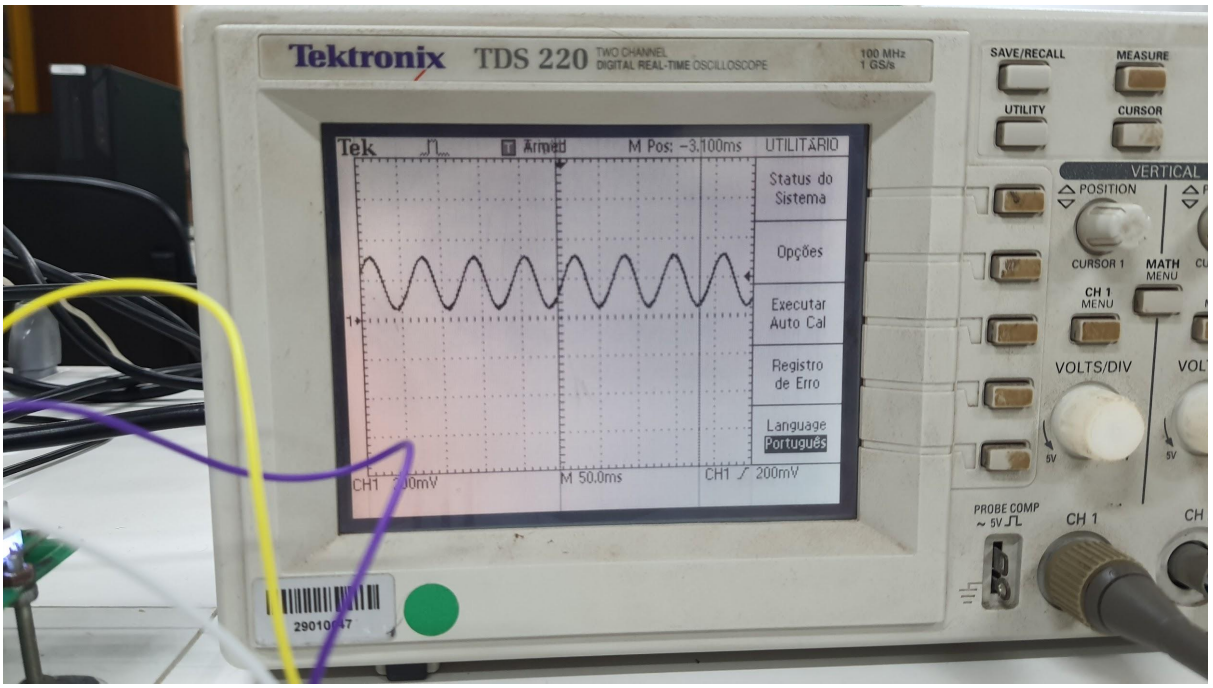


Também não é bom aumentar muito a capacitância, da mesma forma que a onda quadrada é afetada por ela, a onda triangular é distorcida ao se utilizar uma capacitância de valor muito alto, que é justificado pelas variações rápidas que ocorrem ao chegar no topo do triângulo e são progressivamente mais filtradas quando a capacitância é muito alta. Isto até diminui a amplitude do sinal.

Triangular com *68 uF*:



A onda senoidal é beneficiada pelo filtro alto, ficando com uma forma bem suave:



10.

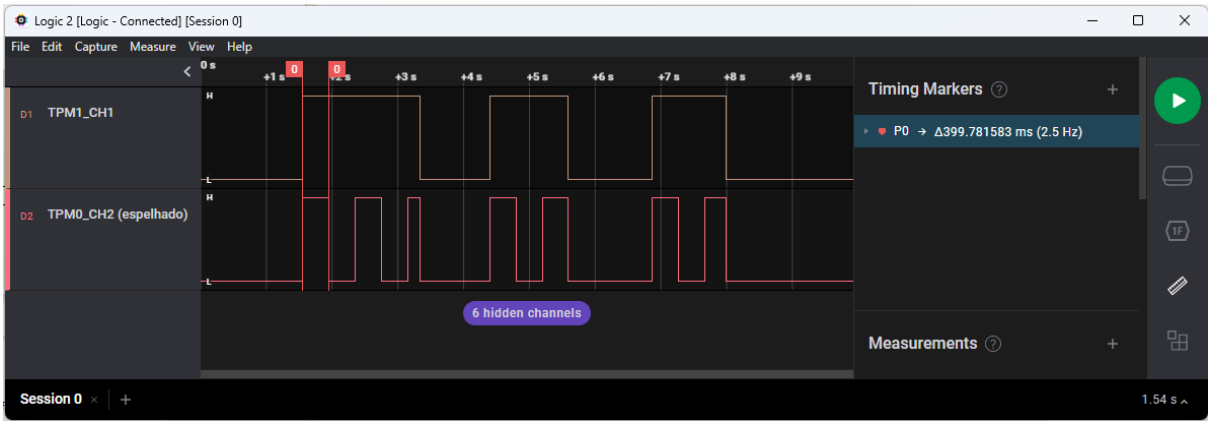
Nas rotinas TPM_config_especifica e TPM_CH_config_especifica, os registradores específicos são alterados por meio das structs que definem um módulo TPM e os canais do módulo. Isso é feito de maneira que, em MKL25Z4.h, é definida a struct TPM_MemMap que contém os registradores do módulo, facilitando seu acesso, uma vez que, basta que se use o nome do registrador como atributo da struct. Sendo assim, dentro dessa struct, existe uma struct que define um canal de TPM. Desse modo, nas rotinas citadas, são usadas as structs por meio de vetores do tipo da struct de TPM, cujo elemento

configurado na chamada da rotina depende do parâmetro x, que é usado de índice para o vetor. Nesse sentido, é usado o elemento do vetor para acessar seus atributos e alterar, portanto, os registradores dos módulos de TPM e dos canais.

Segunda parte

1.

Ao apertar a botoeira uma primeira vez, a interrupção do canal dois é habilitada pela execução de TPM0_C2SC |= TPM_CnSC_CHIE_MASK; E o sinal no pino PTE22 é invertido pela rotina de tratamento de interrupção na linha GPIOE_PTOR = (1 << 22); Portanto, basta medir a largura de um dos pulsos no pino:



Que nos dá um período de aproximadamente *0,3998 s*. Na chamada da função que configura o módulo, vemos que o valor máximo foi setado em 65535, e o *prescaler* em 7. Além disso, temos que o clock é *20,971520 MHz* e CPWMS é zero para que a contagem seja progressiva. Utilizando a equação do período do TPM temos

chamada da função:
TPM_config_especifica (0, 65535, 0b1111, 0, 0, 0, 0, 0, 0b111);

$$Periodo = 65535 \times \frac{2^7}{20971520} \times (1 + 0) = 0,399\,994\,s$$

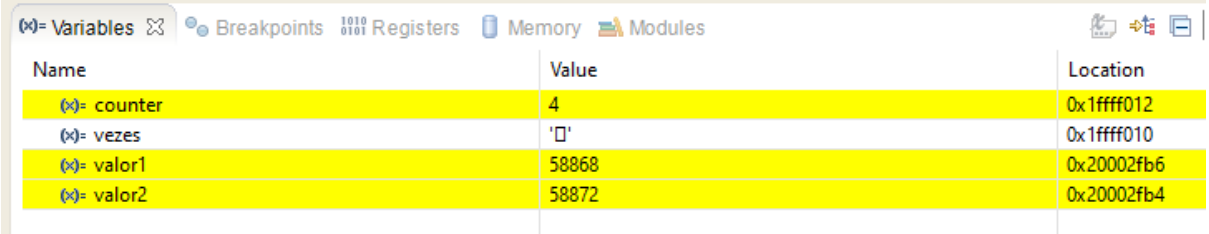
Que coincide com o valor medido experimentalmente pelo Logic.

2.

Para que o período fosse medido e comparado com o valor experimental, foi necessário executar o programa sem colocar um breakpoint na rotina de tratamento de interrupção. Isso porque parar a execução lá faz com que a forma de onda em H5 seja distorcida, então não seria possível medir o tempo entre pressionamentos da botoeira pelo analisador lógico.

O que foi feito: Primeiramente o programa foi executado com um breakpoint na rotina de serviço de TPM para que o endereço da variável estática counter fosse visto, na imagem abaixo vemos que o endereço é *0x1ffff012*. Isto é necessário pois não será possível ver o valor da variável sem o

breakpoint na rotina, então seu valor será visto diretamente na memória, que é possível pois a variável é declarada como estática.



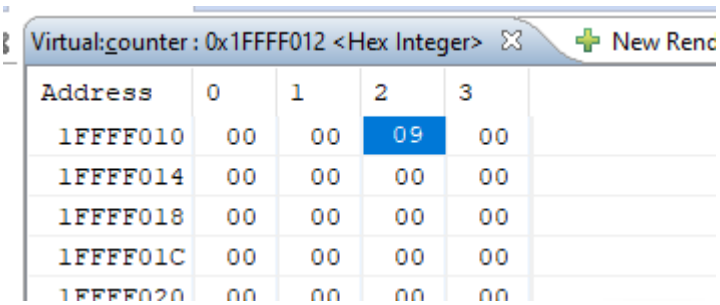
Então, o breakpoint foi removido, e o código foi posto em execução. O programa Logic foi acionado para medir os sinais no header H5, e a botoeira foi pressionada. Neste momento o valor de TPM0_CNT é copiado em TPM0_C1V instantaneamente, a interrupção é gerada e este valor é lido transferido para TPM0_C2V, que está configurado em OUTPUT COMPARE. Agora, até o próximo pressionamento, sempre que a contagem de TPM0 atingir este mesmo valor, será gerada uma interrupção pelo canal 2 que incrementa a variável *counter*.

Quando a botoeira é novamente pressionada, o canal 1 faz com que TPM0_CNT seja instantaneamente salvo em TPM0_C1V, o fluxo de código no tratamento de interrupção do canal 1 é diferente, que faz com que as interrupções do canal 2 sejam desativadas, estabilizando o valor de *counter*.

Conclusão: Temos neste momento, em TPM0_C2V o valor do contador no momento do primeiro pressionamento, TPM0_C1V tem o valor do contador quando a botoeira é pressionada pela segunda vez. *counter*, na posição de memória que já foi descoberta, armazena quantos períodos completos ocorreram entre os pressionamentos.

Agora, o fluxo de execução foi parado pelo botão *PAUSE* do debugger, e os valores foram analisados:

Da imagens abaixo, temos que o valor de *counter*, é 0x0009, pois tem 16 bits e está armazenado em *0x1ffff012* e *0x1ffff013*, com os bits menos significativos no menor endereço. Então *counter* = 9.

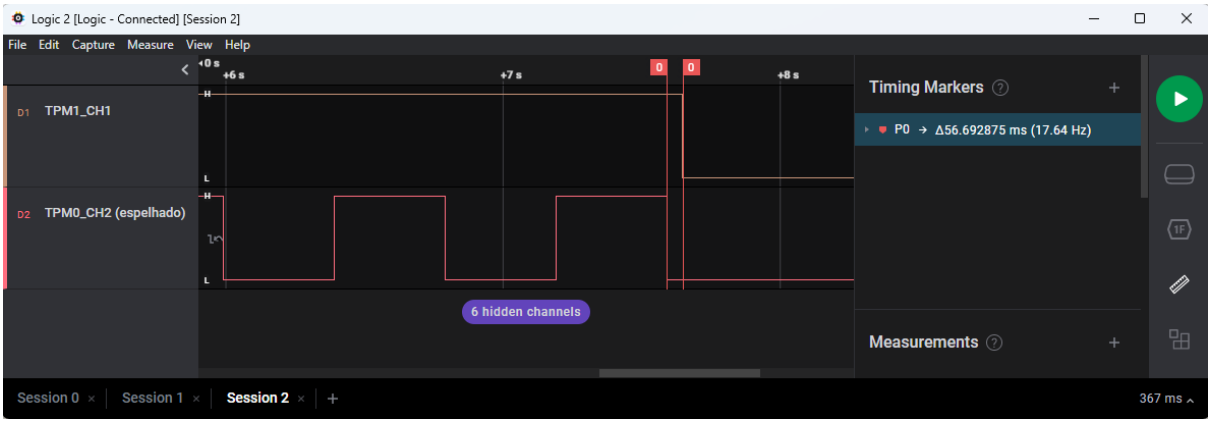


Agora na imagem abaixo, vemos que TPM0_C1V = 50467 e TPM0_C2V = 41181. Portanto, além das nove contagem completas, foram contados mais 50467 - 41181 = 9286. Em relação ao valor máximo, isto é 9286 / 65535 = 0,1417 de um período completo, que como já visto tem duração de aproximadamente 0,4 segundos. Portanto, espera-se que no sinal visto no analisador lógico a diferença de tempo entre o último fim de contagem antes do segundo pressionamento da botoeira seja de aproximadamente 9286 / 65535 * 0,399994 = 0,056 677 segundos.

Variables | Breakpoints | Registers | Memory | Modules

Name	Value	Location
TPM0_CNT	0x00009abd	0x40038004
TPM0_MOD	0x0000ffff	0x40038008
TPM0_C0SC	0x00000000	0x4003800c
TPM0_C0V	0x00000000	0x40038010
TPM0_C1SC	0x00000048	0x40038014
TPM0_C1V	50467	0x40038018
TPM0_C2SC	0x00000098	0x4003801c
TPM0_C2V	41181	0x40038020
TPM0_C3SC	0x00000000	0x40038024

Analisando esta diferença no analisador vemos que o tempo é de aproximadamente 56,97 ms = 0,05697 segundos, um valor muito próximo do calculado pelos valores no registradores, que mostra como a precisão do TPM é elevada.



Por fim, Podemos ver o período completo entre os pressionamentos, no analisador vemos que o tempo foi de aproximadamente 3,6548 segundos. E o valor teórico é 9 vezes o período somado com a diferença que foi calculada por último, temos então 9 * 0,399994 + 0,056677 = 3.6566, também muito próximo do medido experimentalmente.