EA871 – LAB. DE PROGRAMAÇÃO BÁSICA DE SISTEMAS DIGITAIS

EXPERIMENTO 8 – TPM, DMA e DMAMUX

Profa. Wu Shin-Ting

OBJETIVO: Apresentação das funcionalidades PWM do módulo TPMx (*Timer/PWM*).

ASSUNTOS: Geração de sinais PWM (*Pulse Width Modulation*), programação do MKL25Z128 para processamento destes sinais via módulos TPMx.

O que você deve ser capaz ao final deste experimento?

Entender o princípio de funcionamento de TPMx.

Saber programar a base de tempo do contador de um módulo TPMx.

Saber configurar os canais de TPM para as funções *Input Capture* (IC), *Output Compare* (OC) e *Pulse Width Modulation* (PWM).

Programar KL25Z para capturar eventos de entrada com "marcadores de tempo".

Programar KL25Z para gerar sinais digitais específicos de saída condicionados a "instantes de tempo"

Programar KL25Z para gerar sinais PWM de ciclos de trabalho de interesse.

Entender o princípio de funcionamento dos circuitos *Direct Memory Access* (DMA) e Multiplexador de DMA (DMAMUX).

Saber configurar os módulos DMA, DMAMUX, a fonte e o destino para transferências via DMA.

Ter uma ideia do projeto de um filtro passa-baixo para converter sinais modulados por largura de pulso em sinais modulados por níveis de tensão.

Saber aplicar os contadores dos temporizadores para gerar números aleatórios.

Saber usar o tipo `struct` para organizar dados estruturados e parametrizá-los.

Saber diferenciar a aritmética de inteiros e de pontos flutuantes em C.

Ter uma noção sobre os problemas envolvidos no processamento de pontos flutuantes.

Ter uma noção da conversão de pontos flutuantes para inteiros em C.

Saber aplicar máquina de estados e mecanismo de interrupções na proteção de ações indevidas dos usuários.

Saber implementar os exemplos de aplicação dos módulos do microcontrolador apresentados em [14].

INTRODUÇÃO

Além do *timer* integrado ao núcleo, *SysTick* (Seção B.3.3/página 277 em [1]), e dos módulos PIT (*Periodic Interrupt Timer,* Cap. 32/página 573 em [2]), LPTMR (*Low-Power Timer*, Cap. 33, página 587, em [2]), RTC (*Real Time Clock*, Cap. 34/página 597 em [2]), o microcontrolador KL25Z dispõe ainda de três módulos `TPM` (**Timer/PWM**, Cap. 31/página 547 em [2]), 1 com 6 canais e 2 com 2 canais. Diferente de outros temporizadores, são integrados nos módulos TPM os circuitos dedicados de captura na entrada (*input capture*), saída por comparação (*compare output*) e modulação de largura de pulso (*pulse width modulation*). Esses circuitos conseguem gerar sinais mais complexos, além de uma sequência de eventos regulares de um temporizador típico, e suportam pinos de comunicação com o

mundo externo do microcontrolador, oferecendo um melhor suporte à implementação de aplicações mais complexas.

Neste experimento vamos apresentar as funções configuráveis nos módulos TPMx através de três exemplos de projeto e aplicar o aprendizado no desenvolvimento de um projeto de medidor de tempos de reação, `tempo_reacao`. Entende-se como o **tempo de reação** o intervalo de tempo entre a geração de um estímulo e uma ação motora [16]. Os estímulos a serem aplicados neste projeto são audíveis (som de um *buzzer*). Veremos também uma possível forma de coletar esses dados de forma mais eficiente, com mínimas intervenções do processador, fazendo uso do módulo ***Direct Memory Access*** (DMA).

Módulo TPM

Em cada módulo TPMx há um contador LPTPM de 16 *bits*, mapeado no registrador `TPMx_CNT`, que conta ciclicamente de forma crescente (*up,* `TPMx_SC_CPWMS`=0) ou crecente-decrescente (*up-down,* `TPMx_SC_CPWMS`=1) (Figura 31-1/página 549 em [2]). Qualquer acesso de escrita em `TPMx_CNT` reseta o contador em zero. LPTPMs são pulsados por um sinal de relógio assíncrono `TPM clock` (Seção 5.7.5/página 124 em [2]), independente do sinal do barramento (*bus clock*) que coordena o restante dos circuitos do módulo. Assim, o contador pode manter a sua contagem mesmo em modos de baixo consumo energético. A fonte de `TPM clock` é selecionável pelos campos `SIM_SOPT2_TPMSRC` e `SIM_SOPT2_PLLFLLSEL` a partir dos diferentes tipos de sinais de relógio gerados pelo módulo MCG, `MCGFLLCLK`, `MCGPLLCLK/2`, `OSCERCLK` e `MCGIRCLK` (Figura 24-1/página 370 em [2]). Os sinais de relógio de LPTPM podem ser também externos, vindos dos pinos devidamente multiplexados em `TPM_CLKIN0` ou `TPM_CLKIN1` (Seção 10.3.1/página 162 em [2]). Além da seleção da fonte, é necessário habilitá-la para os módulos TPM0, TPM1 e TPM2 pelos respectivos *bits* `SIM_SCGC6_TPM0`, `SIM_SCGC6_TPM1` e `SIM_SCGC6_TPM2` (Seção 12.2.10/página 207 em [2]). E, para ativar efetivamente a contagem em LPTPM, é necessário setar o *bit* `TPMx_SC_CMOD` em '1'. O estouro na contagem, ou seja quando `TPMx_CNT` atinge o valor setado em `TPMx_MOD`, seta a *flag* (de estado) `TPMx_SC_TOF`.

Um módulo TPMx contém múltiplos canais programáveis para uma das três funções: *input capture*, *output compare* ou *pulse width modulation*. Esses canais compartilham a mesma base de tempo definida por LPTPM, mas possuem seus próprios registradores de configuração `TPMx_CnSC`, registradores de dados `TPMx_CnV`, comparadores, pinos de comunicação com o mundo externo, e circuitos de controle dos pinos e de interrupções. Isso permite que um canal seja programado com uma função independente, porém sincronizada com as funções executadas em outros canais do mesmo módulo, e a sua *flag* `TPMx_CnSC_CHF` seja setada em função da ocorrência de um evento relacionado com a função programada nele.

É possível ainda configurar através do registrador de configuração `TPMx_CONF` uma base de tempo global/comum para os três módulos TPMx e o comportamento dos módulos nos modos de operação *Debug* e Espera (Seção 31.3.7/página 559 em [2] e Seção 12.3.8/página 126 em [14]).

Configuração de um Período em TPM

De acordo com as Seções 31.4.2/página 562 e 31.4.3/página 562 em [2], o período (contagem máxima) de LPTPM depende, além da frequência da fonte f_{clock} (`TPM clock`) selecionada, dos valores setados em `TPMx_MOD` (valor de referência para contagem máxima, Seção 31.3.3/página 554 em [2]) e em `TPMx_SC_PS` (divisor *prescaler*, Seção 31.3.1/página 552 em [2])

$$Periodo = TPMx\_MOD \; x \frac{2^{TPMx\_SC\_PS}}{f_{clock}} x \left(1 + TPMx\_SC\_CPWMS\right) \qquad (1)$$

Funções Programáveis nos Canais de TPMx

A principal característica do módulo TPM é que cada um dos seus canais pode ser configurado para operar num dos três modos através dos *bits* `TPMx_CnSC_MSnB` e `TPMx_CnSC_MSnA`, do registrador de controle do canal `TPMx_CnSC`: *Input Capture* (Seção 31.4.4/página 564 em [2]), *Output Compare* (Seção 31.4.5/página 565 em [2]) e *PWM* (Seções 31.4.6 e 31.4.7/página 566 em [2]). Os níveis ou as bordas de interesse em cada um dos três modos de operação são configurados pleos bits `TPMx_CnSC_ELSnB` e `TPMx_CnSC_ELSnA`, do mesmo registrador. Usando como base de tempo o contador `TPMx_CNT` do módulo TPMx, um canal configurado com a função:

Input Capture (`MSnB:MSnA==00`) associa precisamente ao evento de interrupção detectado (borda de subida – `ELSnB:ELSnA==01`, borda de descida – `ELSnB:ELSnA==10`, ou ambas as bordas – `ELSnB:ELSnA==11`) o valor de contagem registrado em `TPMx_CNT`. O circuito captura o valor de `TPMx_CNT` no registrador `TPMx_CnV` no momento em que um evento pré-especificado ocorre. Em paralelo à captura, é setada a *flag* `TPMx_CnSC_CHF` em '1' (Seção 31.4.4/página 564, em [2]). Podemos dizer que é um circuito que rotula os eventos de interrupção com os "dados" que nos permitem inferir instantes de tempo de forma acurada. Para que o sinal de entrada seja corretamente amostrado, a sua frequência (mais alta) deve ser 2 vezes menor que a frequência de contagem do relógio do módulo TPMx (teorema de amostragem Nyquist-Shannon [17]).

Output Compare (`MsnB:MsnA==01` ou 11) gera um sinal de saída pré-configurado (alterna – `ELSnB:ELSnA==01`, reseta – `ELSnB:ELSnA==10`, seta – `ELSnB:ELSnA==11`, pulso positivo – `ELSnB:ELSnA==X1` ou pulso negativo – `ELSnB:ELSnA==10`) quando o valor de contagem no contador `TPMx_CNT` se iguale ao valor em `TPMx_CnV`. A periodicidade de atualização dessa saída é alinhada com o valor `TPMx_CnV` (Figuras 31-82 a 31-84/página 566 em [2]). No momento em que `TPMx_CNT==TPMx_CnV`, é setada a *flag* `TPMx_CnSC_CHF` em '1' (Seção 31.4.4/página 564, em [2]). Podemos dizer que é um circuito que faz contagens cíclicas a partir de um valor de contagem e permite pré-programar, de forma acurada, uma saída condicionada a um valor de contagem que pode corresponder a um instante específico de tempo.

Pulse Width Modulation (`MsnB:MsnA==10`) gera um sinal de saída de largura de pulso e de polaridade controláveis pelos *bits* de configuração `TPMx_SC_CPWMS` e pelo registrador de dados `TPMx_CnV`. A periodicidade dessa saída é alinhada com `TPMx_CNT==0`. Quando `TPMx_SC_CPWMS==0`, o alinhamento é com uma borda do pulso (*edge-aligned* PWM ou EPWM, Figura 31-87/página 568 em [2]). O nível lógico de saída assume '1' no instante em que `TPMx_CNT==0` se `ELSnB:ELSnA==10`, e assume '0' se `ELSnB:ELSnA==X1`. Esse nível é alternado quando `TPMx_CNT==TPMx_CnV`. E quando `TPMx_SC_CPWMS==1`, o alinhamento se dá com o centro do pulso (*center-aligned* PWM ou CPWM, Figura 31-88/página 569 em [2]). O nível lógico de saída assume '1' no instante em que `TPMx_CNT== TPMx_CnV` na contagem decrescente se `ELSnB:ELSnA==10`, e assume '0' se `ELSnB:ELSnA==X1`. O sinal é alternado quando `TPMx_CNT== TPMx_CnV` na contagem crescente. Sempre que `TPMx_CNT==TPMx_CnV`, é setada a *flag* `TPMx_CnSC_CHF` em '1' (Seção 31.4.4/página 564, em [2]). Podemos dizer que é um circuito que gera uma forma de onda de período fixo, definido pelo `TPMx_MOD`, com larguras configuráveis pelo registrador `TPMx_CnV`. A razão cíclica das larguras dos pulsos em relação ao

período do sinal é conhecida como **ciclo de trabalho** (*duty cycle*, Figura 1). Note que o incremento para m unidades sempre ocorre na transição de `m-1` para `m`. **Para que tenhamos um ciclo de trabalho 100% no modo EPWM, é necessário que o valor setado em `TPMx_CnV` seja uma unidade maior do que o valor setado em `TPMx_MOD`** (Seção 31.4.6/página 567 em [2]).

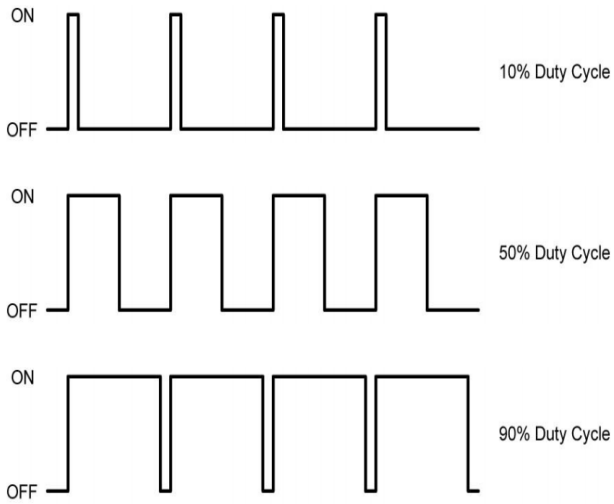

Figura 1: Modulação de largura de pulso [3]

Alocação de Pinos para TPMx

Diferentes dos temporizadores básicos, os módulos TPMx podem capturar sinais externos como gerar sinais com características específicas através dos seus canais. Para trocas desses sinais com o mundo externo, devemos alocar um pino físico para cada canal TPMx_CHn. Essa alocação deve ser baseada na consulta da tabela de multiplexação dos pinos na Seção 10.3.1/página 162 em [2], onde são listados os pinos que podem ser multiplexados para a função TPMx_CHn. Por exemplo, os pinos PTA4 e PTB18 poderão servir, respectivamente, o canal TPM0_CH1 e o canal TPM2_CH0 se os pinos estiverem devidamente multiplexados respectivamente para a função de TPM0_CH1 e TPM2_CH0. Ou seja, se `PORTA_PCR4_MUX`==0x03 e `PORTB_PCR18_MUX`==0x03. No entanto, TPMx podem ser usados como simples temporizadores básicos, somente para contagem de tempos por períodos, como os temporizadores básicos SysTick (Cap. B3.3/página 275 em [1]) e PIT (Cap. 32/página 573 em [2]). Neste caso, como todos os temporizadores básicos que vimos, não é necessário alocar um pino para o canal.

Disparos Externos para TPMx_CNT

O TPMx permite que eventos externos, denominados **disparos** (***triggers***), forcem a reinicialização da contagem de LPTPM em 0 assincronamente. Os identificadores das fontes válidas de disparos estão listados na Tabela 3-38/página 86 em [2]. Uma vez setado um desses identificadores no campo `TPMx_CONF_TRGSEL`, ele passa a controlar os instantes em que LPTPM muda o modo de operação. Por exemplo, se for setado 0b1000 nesse campo, o conteúdo de `TPMx_CNT` é alterado quando a *flag* de `TPM0_SC_TOF` fique em '1'. A forma de modificação depende dos valores setados nos *bits* de configuração `TPMx_CONF_CROT` (`TPMx_CNT` é resetado em '0'), `TPMx_CONF_CSOO` (`TPM_CNT` pára a contagem na ocorrência de um estouro e só retoma a contagem quando rehabilitado ou quando `TPMx_CONF_CSOT`==1 e recebe um disparo) e `TPMx_CONF_CSOT` (só inicia a contagem quando recebe um disparo).

O projeto `rot8_example1` ilustra a aplicação desses disparos para delimitar num período de tempo a ocorrência das bordas de subida RE e de descida FE de pulsos PWM de largura variáveis ciclicamente a fim de simplificar o cômputo dessas larguras. Reduziu-se o cálculo na diferença de dois valores de contagem capturados pela função *Input Capture* configurada nos canais TPM1_CH0 e TPM1_CH1. Se multiplicarmos essa diferença pelo período dos pulsos do contador `TPM1_CNT`, teremos o tempo que transcorreu entre as duas bordas. Figura 2 ilustra os intervalos, entre os instantes CSOT e CSOO, em que a contagem de TPM1 é efetivamente ativada. Vale destacar aqui que a configuração dos disparos periódicos só foi feita uma única vez na inicialização do módulo TPM1. Toda vez que um disparo é ativado, é reiniciada automaticamente, por *hardware*, a contagem de `TPM1_CNT`. O controle da parada dos incrementos quando `TPM1_CNT` atinge a contagem máxima é também por *hardware*. Porisso, não há nenhuma outra instrução além das instruções de configuração iniciais para processar a forma de onda TPM1 (na terceira linha).

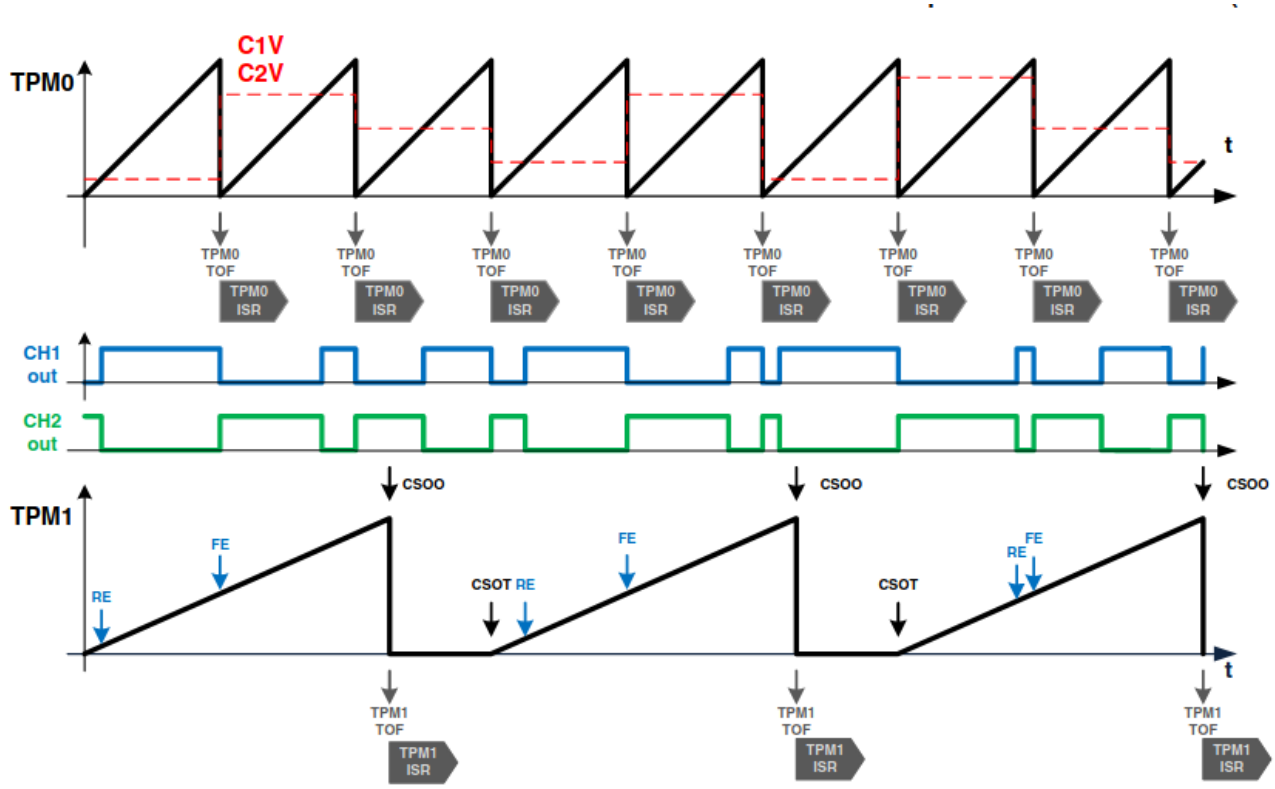

Figura 2: Disparos para controlar a contagem de `TPMx_CNT` no tempo (cópia da Fig. 12-4 em [14]).

Processamento de Interrupções em TPMx

Os contadores nos módulos TPMx são de corrida livre (*free running*), isto é eles passam por todos os estados num ciclo completo de contagem, de 0 até o valor de contagem máxima `TPMx_MOD`. Como todos os temporizadores, TPMx gera um evento de *Overflow* quando o valor do contador `TPMx_CNT` se iguala a `TPMx_MOD`. Além do estado de estouro do contador LPTPM (*overflow*) mostrado no *bit* `TPMx_SC_TOF`, cada canal tem um *bit* de estado `TPMx_CnSC_CHF` associado para indicar a ocorrência de um evento de interrupção configurado.

Quando os *bits* de habilitação de interrupção, `TPMx_SC_TOIE` e `TPMx_CnSC_CHIE`, correspondentes a cada um desses *bits* de estado estão em '1', é gerada uma requisição de interrupção IRQ com o número de vetor associado ao módulo assim que o *bit* de estado fiquem em '1'. Esse número de vetor é igual a 33 para TPM0 (IRQ=17), a 34 para TPM1 (IRQ=18) e a 35 para TPM2 (IRQ=19) (Tabela 3-7/página 52 em [2]). E se IRQ17, IRQ18 e IRQ19 estiverem habilitadas no lado do controlador NVIC (Seção B3.4/página 281 em [1]), o fluxo de controle é, então, automaticamente

desviado para a rotina de serviço pré-declaradas no arquivo `Project_Settings/Startup_Code/kinetis_sysinit.c` gerado pelo IDE CodeWarrior. Os nomes declarados para as rotinas que tratam os eventos de TPM0, TPM1 e TPM2 são, respectivamente, `FTM0_IRQHandler`, `FTM1_IRQHandler` e `FTM2_IRQHandler`.

Note que só há uma linha de requisição associada a cada TPMx. As solicitações originadas de diferentes fontes são combinadas por uma lógica OU dentro do módulo. Para identificar a origem de uma interrupção nas rotinas de serviço, deve-se avaliar os *bits* de estado individualmente. Como uma forma de otimizar acessos a esses *bits* espalhados em diferentes registradores de controle/estado, há um espelho desses *bits* num único registrador de estado `TPMx_STATUS`. Para evitar "reentrâncias", todos os *bits* de estado devem ser resetados em '0' com um acesso de escrita de '1' (*write-1-to-clear*) após o atendimento .

Como muitas funções suportadas por TPMx são executadas integralmente por *hardware* e ele consegue gerar uma série de eventos de interrupção funcionalmente distintos, a programação das tarefas a serem executadas pode ser reduzida em pequenas intervenções por *software* dentro das rotinas de serviço como ilustra `rot8_example1`. A tarefa desse projeto é fazer medições dos intervalos de tempo entre pares de eventos de interrupção gerados pelas capturas das bordas RE e FE (Figura 2). A forma mais precisa para ler as contagens capturadas é através da rotina de serviço `FTM1_IRQHandler`. Como o processamento dos dois valores capturados é muito simples, a diferença é calculada na mesma rotina e o laço principal vazio da função `main` só mantém o processador no aguardo das solicitações de interrupções.

Módulo DMA

O circuito **Acesso Direto à Memória** (***Direct Memory Access***, DMA) é um controlador que permite a transferência de dados entre dispositivos sem a intervenção do processador, aumentando a eficiência e liberando-o para outras tarefas [13]. Em KL25Z todos os registradores dos módulos-periférico são mapeados no espaço de endereços de 32 *bits* do processador.

O módulo DMA contém 4 canais independentes para transferência de 8-, 16- e 32-*bits*, cuja prioridade de atendimento segue a ordem canal 0 > canal 1 > canal 2 > canal 3. Cada canal `n` tem

- um registrador de dados `DMA_SARn` (Seção 23.3.1/página 353 em [2]) para armazenar o endereço do rementente/fonte (*source*),
- um registrador `DMA_DARn` (Seção 23.3.2/página 354 em [2]) para o endereço do destinatário,
- um registrador de estado `DMA_DSRn` (Seção 23.3.3/página 355 em [2]) que indica o estado de uma transferência (erro de configuração – `DMA_DSR_BCRn_CE`, erros no barramento – `DMA_DSR_BCRn_BES`, `DMA_DSR_BCRn_BED`), e do estado do canal (estado de pendência – `DMA_DSR_BCRn_REQ`, estado ocupado – `DMA_DSR_BCRn_BSY` e estado concluído – `DMA_DSR_BCRn_DONE`),
- um contador de 24 *bits* `DMA_DSR_BCRn_BCR` (Seção 23.3.3/página 355 em [2]), que contém a quantidade de *bytes* a serem transferidos e é decrementado da quantidade de *bytes* transferidos após cada transferência bem sucedida, e
- um registrador de controle/configuração `DMA_DCRn` (Seção 23.3.4/página 357 em [2]), que habilita a interrupção quando completa uma transferência (`DMA_DCRn_EINT`), a requisição de um periférico para uso do canal (`DMA_DCRn_ERQ`), o mecanismo de roubos de ciclos (`DMA_DCRn_CS`), o auto-alinhamento com base nos endereços e no tamanho de dados

(DMA_DCRn_AA), e requisições assíncronas (DMA_DCRn_EADREQ). Através deste registrador, configura-se os tamanhos dos *buffers* circulares (DMA_DCRn_SMOD e DMA_DCRn_DMOD no remetente e no destinatário, respectivamente), o tamanho de dados por transferência (DMA_DCRn_SSIZE e DMA_DCRn_DSIZE no remetente e no destinatário, respectivamente) e modo de atualização dos endereços para a próxima transferência (DMA_DCRn_SINC e DMA_DCRn_DINC, respectivamente).

Antes de iniciar uma transação, é necessário carregar os endereços iniciais dos blocos de dados do remetente e do destinatário respectivamente em DMA_SARn e DMA_DARn, os tamanhos dos dados, o tamanho dos *buffers* circulares a serem alocados, o modo de atualização dos endereços para a próxima transferência no registrador de configuração DMA_DCRn, e a quantidade total de *bytes* a serem transferidos na transação em DMA_DSR_BCRn_BCR (Seção 23.4.2.2/página 362 em [2]). Uma transação pode ser iniciada por *software*, setando o *bit* DMA_DCRn_START, ou por requisição de um módulo periférico se o *bit* DMA_DCRn_ERQ estiver setado em '1'. É necessário que a requisição à transferência por DMA seja habilitada individualmente no módulo que solicitará a requisição. Por exemplo, para os módulos PORTx/GPIOx a habilitação é pelos *bits* PORTx_PCRn_IRQC, para o módulo UART0 é pelos *bits* UART0_C5_TDMAE e UART0_C5_RDMAE e para um módulo TPMx, pelo *bit* de configuração TPMx_CnSC_DMA.

Processamento de Interrupções em DMA

Quando os *bits* DMA_DSR_BCRn_DONE e DMA_DCRn_EINT estiverem em '1' é gerado um evento de interrupção IRQ com o número de vetor associado ao canal. Esse número de vetor é igual a 16 para o canal 0 (IRQ=0), a 17 para o canal 1 (IRQ=1) e a 18 para o canal 2 (IRQ=2) e a 19 para o canal 3 (IRQ=3) (Tabela 3-7/página 52 em [2]). E se IRQ0, IRQ1, IRQ2 e/ou IRQ3 estiverem habilitadas no lado do controlador NVIC (Seção B3.4/página 281 em [1]), o fluxo de controle é, então, automaticamente desviado para a rotina de serviço pré-declaradas no arquivo Project_Settings/Startup_Code/kinetis_sysinit.c gerado pelo IDE CodeWarrior. Os nomes declarados para as rotinas que tratam os eventos dos canais 0, 1, 2 e 3 são, respectivamente, DMA0_IRQHandler, DMA1_IRQHandler, DMA2_IRQHandler e DMA3_IRQHandler. Note que só há uma linha de requisição associada a cada canal. A conclusão de uma transação de dados ou a ocorrência de algum erro faz o *bit* DMA_DSR_BCRn_DONE ficar em '1'. Para identificar a origem de uma interrupção nas rotinas de serviço, deve-se avaliar os *bits* de estado individualmente. O *bit* DMA_DSR_BCRn_DONE precisa ser resetado com um acesso de escrita (*write-1-to-clear*) para remover o evento de interrupção dentro da rotina de serviço. Esse acesso de escrita reseta todos os *bits* de estado do DMA.

Módulo DMAMUX

O circuito DMAMUX é um multiplexador que permite a seleção de várias fontes/*slots* de dados para serem transferidas pelo controlador DMA. Juntos, eles permitem que múltiplos dispositivos compartilhem o uso do DMA para transferir seus dados. É importante frisar que a função de DMAMUX é só de multiplexagem de uma rota, todos os controles de transferência pela rota selecionada são de responsabilidade do módulo DMA. Em KL25Z o módulo DMAMUX consegue rotear 63 *slots* habilitáveis nos respectivos módulos e 6 *slots* sempre-habilitáveis para os 4 canais disponíveis. São sempre habilitáveis as transferências por DMA entre os módulos GPIO e nas unidades de memória. No modo de roteamento normal, os dados (remetentes) endereçados são roteados

diretamente para o canal pré-determinado em cada requisição pré-especificada. Os códigos de todas as possíveis requisições a uma transferência DMA estão listados na Tabela 3-20/página 64 em [2]. Para melhorar a **previsibilidade**, os dois primeiros canais, 0 e 1, podem ter as suas requisições às transferências sobrepostas pelos disparos periódicos automáticos gerados pelo temporizador PIT (Seção 22.4.1/página 341 em [2]). A seção 22.5.2/página 344 em [2] apresenta os exemplos de configuração e habilitação dos *slots* habilitáveis, mais especificamente do *slot* #5 (Transmissor de UART1), para o canal 2 do DMA com e sem disparos automáticos de requisições (Seção 22.4.1/página 341 em [2]).

No projeto `rot8_example2` é ilustrada a transferência, via o canal 0 do módulo DMA, dos dados de SRAM (sempre-habilitável) para o registrador de dados de 16 *bits* `TPM1_C1V` do canal `TPM1_CH1` que é configurado com a função EPWM. É necessário habilitar esse canal setando em '1' o *bit* `TPM1_C1SC_DMA` e especificar os instantes em que os dados devem ser roteados para o canl 0 do DMA setando o código da fonte de requisição em `DMAMUX0_CHCFG0_SOURCE`. No caso, o código é #55 que corresponde ao evento *TPM0 Overflow*. Os valores pré-carregados em SRAM definem diferentes larguras de pulso no sinal de saída de TPM1_CH1, que, ao passarem por um filtro RC, resultam num sinal analógico com variações em níveis de tensão.

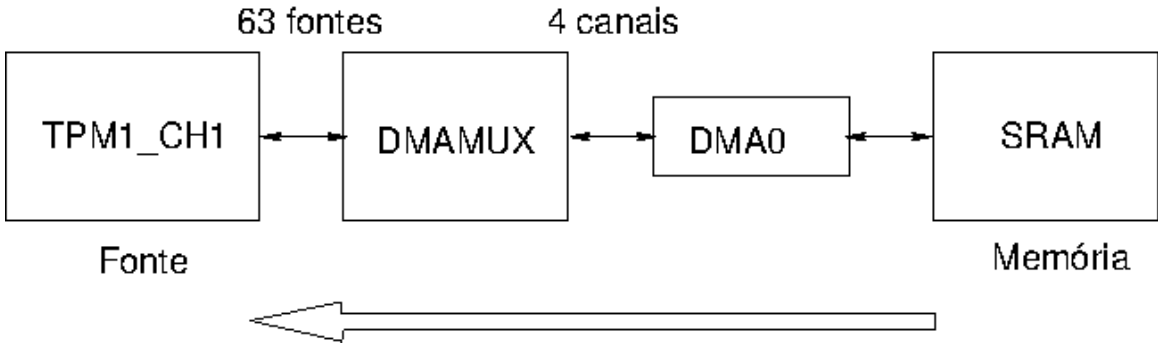

Figura 3: Projeto `rot8_example2`: transferência, via o canal 0 de DMA, dos dados de SRAM (sempre-habilitável) para TPM1_CH1 (habilitável por `TPM1_C1SC_DMA`) através do sinal de requisição 55 (TPM1 *Overflow*).

Conversão de Sinal Modulado por Largura de Pulso em Sinal Modulado por Nível de Tensão

Para converter um sinal modulado por largura de pulso (PWM) em um sinal modulado por nível de tensão, podemos usar um circuito RC, com um resistor e um capacitor em série.

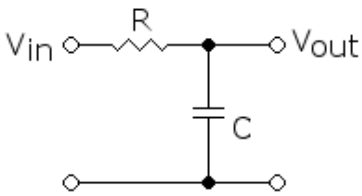

Figura 4: Filtro passivo passa-baixas.

O valor da resistência R, em Ohms, e da capacitância C, em Farads, é determinado em função da constante de tempo do circuito. A **contante de tempo** RC é o tempo, em segundos, necessário para carregar um capacitor em série com um resistor até atingir 63.2% do valor da tensão (de alimentação contínua) aplicada sobre ele. Para um sinal PWM de período T em que a menor largura seja T/N, podemos usar um filtro RC tal que

$$RC \geq N \times T.$$

Por exemplo se T = 0.001s (frequência de amostragem em 1kHz) e as larguras dos pulsos são múltiplos de T/64, então podemos escolher para R=100Ω um capacitor de capacitância

$$C \geq \frac{(64 \times 0.001)}{100} = 640uF$$

para atenuar as ondulações entre os pulsos gerados pelo circuito PWM.

Cômputo de Intervalo de Tempo entre dois Eventos de Interrupção de *Input Capture* por *Overflows*

No projeto `rot8_example1` (Figura 2), se não forçarmos a reinicialização do contador `TPMx_CNT` e setarmos um período de TPMx que cubra o intervalo de tempo entre as ocorrências RE e FE, as duas ocorrências podem acontecer em períodos diferentes. Precisamos então levar em conta as contagens dos períodos completos que transcorreram entre as duas ocorrências por meio de detecção dos eventos de estouros (o*verflows)*. Tendo a contagem de `N` estouros, pode-se computar o intervalo de tempo `t` entre as duas contagens C_{T1} e C_{T2} nos instantes T1 e T2 com as seguintes equações

T1 > T2:

$$t = \left(\frac{(TPMx\_MOD + C_{T2} - C_{T1})}{TPMx\_MOD} + (N-1) \right) \times Período \quad (2)$$

T1 ≤ T2:

$$t = \left(\frac{(C_{T2} - C_{T1})}{TPMx\_MOD} + N \right) \times Período \quad , \qquad (3)$$

onde `TPMx_MOD` é o valor máximo de um ciclo completo de contagem e o *Período*, o intervalo de tempo deste ciclo completo (Figura 12-1 em [14]). O *Período*, por sua vez, pode ser calculado com a Eq. (1) com f_{clock}=20,97152MHz nos nossos projetos em que usamos o sinal de relógio MCGFLLCLK.
A captura por *hardware* dos eventos de *overflows* e dos eventos de entrada, junto com os seus "marcadores de tempo" detectados de "forma imediata", aumenta a precisão do cômputo do intervalo de tempo. Se resetarmos sincronamente o valor do contador LPTPM em 0 no instante T1, C_{T1} na Eq. (3) assumitá o valor 0. Isso simplificará ainda mais o cômputo.

Segue-se um pseudo-código do cálculo de `t` na rotina de serviço que trata as capturas de um canal TPMx_CHn

```
Se (flag de interrupção de TPMx_CHn  estiver em 1) então
  Se for a primeira captura então
     C_T1 ← TPMx_CnV;
     N ← 0;
     Habilitar a interrupção de Overflow de TPMx;
  Se for a segunda captura então
     C_T2 ← TPMx_CnV;
     Computar Eq. (2) ou (3);
Se (flag de overflow de TPMx estiver em 1) então
  N← N+1;
```

Cômputo de Intervalo de Tempo entre dois Eventos de Interrupção de *Input Capture* por *Output Compare*

A função *Output Compare* permite gerar, por *hardware*, um evento de interrupção quando o valor do contador `TPMx_CNT` se iguale ao valor setado no registrador `TPMx_CnV`. Isso aumenta a precisão e a velocidade da resposta do sistema em relação a uma referência pré-especificada. Em conjunto com a função *Input Capture*, ela permite simplificar o cômputo do intervalo de tempo entre os dois instantes capturados, T1 e T2, se ambas as funções compartilharem a mesma base de tempo. Análogo à contagem da distância percorrida em volta de um circuito fechado, o procedimento consiste em setar no registrador `TPMx_CnV` do canal de função *Output Compare* o valor C_{T1} capturado pelo registrador `TPMx_CnV` do canal de função *Input Capture* e contar a quantidade de ciclos de contagem do contador em relação a C_{T1}. Essa contagem pode ser implementada habilitando a interrupção do canal *Output Compare* para que ele gere um evento de interrupção cada vez que o contador `TPMx_CNT` passe por C_{T1}, fazendo uma contagem de ciclos de forma análoga à contagem de *overflows*. Com a contagem por *hardware* da quantidade *M* de ciclos junto com as contagens C_{T1} no instante inicial T1 e C_{T2} no instante final T2, podemos estimar com precisão o intervalo de tempo t entre as duas contagens C_{T1} e C_{T2} através das expressões:

T1 ≥ T2:

$$t = \left(M + \frac{\left(TPMx\_MOD - C_{T1}\right) + C_{T2}}{TPMx\_MOD}\right) \times Periodo \qquad (4)$$

T1 < T2

$$t = \left(M + \frac{\left(C_{T2} - C_{T1}\right)}{TPMx\_MOD}\right) \times Periodo \quad . \qquad (5)$$

Em relação ao cômputo de intervalos de tempo por *overflows*, requer-se nessa alternativa a alocação de um canal configurado com a função *Output Compare* para contar M, como mostra o seguinte pseudo-código de contagem de quantidade de ciclos completos de TPMx na rotina de serviço que trata as capturas de um canal TPMx_CHn

```
Se (flag de interrupção de TPMx_CHn  estiver em 1) então
  Se for a primeira captura então
     C_T1 ← TPMx_CnV;
     M ← 0;
     Ativar e habilitar um canal de Output Compare de TPMx;
  Se for a segunda captura então
     C_T2 ← TPMx_CnV;
     Computar Eq. (4) ou (5);
Se (flag de interrupção do canal de Output Compare estiver em 1) então
  M← M+1;
```

PWM em Geração de Sinais Audíveis

As frequências de modulação para as notas musicais variam de acordo com a escala musical e a temperatura utilizada. Para a escala de 440Hz, temos as seguintes frequências:

- Dó (C) : 261.63 Hz (T=0,0038s)
- Ré (D) : 293.66 Hz (T=0,0034s)
- Mi (E) : 329.63 Hz (T=0,003s)

- Fa (F) : 349.23 Hz (T=0,0029s)
- Sol (G) : 392.00 Hz (T=0.0026s)
- Lá (A) : 440.00 Hz (T=0.0023s)
- Si (B) : 493.88 Hz (T=0.002s)
- Dó (C) : 523.25 Hz (T=0.0019s)

Podemos gerar sinais de áudio de notas musicais usando um módulo a função PWM de TPM. **Para cada nota musical, configuramos como o período do TPM o período da nota** e podemos setar no registrador de dados `TPMx_CnV` um valor que resulte numa forma de onda retangular de frequência da nota musical. Por exemplo, se configurarmos MCGFLLCLK (20.971.520 Hz) como a fonte de sinais de relógio e 32 como divisor *prescaler* do contador LPTPM, podemos gerar um som audível da nota Lá num *buzzer* conectado no pino PTE21, se

- multiplexarmos o pino PTE21 para o canal TPM1_CH1,
- configurarmos o a contagem máxima `TPM1_MOD` em ((20971520)/(32*440),
- configurarmos a função EPWM para o canal TPM1_CH1 via o registrador de configuração TPM1_C1SC, e
- setamos no seu registrador de dados `TPM1_C1V` uma percentagem menor que 100% da contagem máxima.

Parametrização de Blocos de Dados

KL25Z dispõe de 3 módulos de TPM cujos registradores são mapeados no espaço de memória 0x40038000-0x003A088:

```
#define TPM0_BASE_PTR                 ((TPM_MemMapPtr)0x40038000u)
/** Peripheral TPM1 base pointer */
#define TPM1_BASE_PTR                 ((TPM_MemMapPtr)0x40039000u)
/** Peripheral TPM2 base pointer */
#define TPM2_BASE_PTR                ((TPM_MemMapPtr)0x4003A000u)
```

No ambiente `CodeWarrior` este espaço é abstraído em três blocos do tipo de dado `struct TPM_MemMap` definido no arquivo `MKL25Z.h` [11]. Através desse tipo de dados são definidas as macros que nos permitem acessar os registradores pelos mesmos nomes usados no manual de referência [2].

```
typedef struct TPM_MemMap {
  uint32_t SC;               /**< Status and Control, offset: 0x0 */
  uint32_t CNT;              /**< Counter, offset: 0x4 */
  uint32_t MOD;              /**< Modulo, offset: 0x8 */
    struct {                  /* offset: 0xC, array step: 0x8 */
      uint32_t CnSC;  /**< Channel (n) Status and Control, array offset: 0xC, array step: 0x8 */
      uint32_t CnV;           /**< Channel (n) Value, array offset: 0x10, array step: 0x8 */
   } CONTROLS[6];
    uint8_t RESERVED_0[20];
    uint32_t STATUS;          /**< Capture and Compare Status, offset: 0x50 */
   uint8_t RESERVED_1[48];
    uint32_t CONF;            /**< Configuration, offset: 0x84 */
```

```
} volatile *TPM_MemMapPtr;
```

Além disso, há uma série de macros pré-definidas que nos permite "parametrizar" os blocos de dados referentes aos três módulos TPMx Se definirmos um vetor de ponteiros ao tipo de dado `TPM_MemMapPtr` no nosso código, como

```
TPM_MemMapPtr moduloTPM[] = TPM_BASE_PTRS,
```

podemos usar `TPM[x]→SC`, com `x` variando de 0 a 2, para acessarmos o registrador `SC` de cada módulo `TPMx` ao invés de usarmos separadamente as macros `TPM0_SC`, `TPM1_SC` e `TPM2_SC`. Os dois registradores `CnSC` e `CnV` referentes a cada um dos 6 canais de um módulo são agrupados, por sua vez, numa outra `struct` de cujo tipo foi declarada um vetor CONTROLS de 6 elementos.

Com o uso das seguintes macros, também definidas no arquivo `MKL25Z.h`,

```
#define TPM_CnSC_REG(base,index)   ((base)->CONTROLS[index].CnSC)
#define TPM_CnV_REG(base,index)    ((base)->CONTROLS[index].CnV),
```

os acessos individuais aos registradores de cada canal `n` podem também ser parametrizados. No lugar de `TPM1_C0SC`, `TPM1_C0V`, `TPM1_C1SC` e `TPM1_C1V`, temos, respectivamente, as alternativas `TPM_CnSC_REG (moduloTPM[1], 0)`, `TPM_CnV_REG (moduloTPM[1], 0)`, `TPM_CnSC_REG (moduloTPM[1], 1)` e `TPM_CnV_REG (moduloTPM[1], 1)`.

Podemos ainda usar os membros das estruturas definidas para acessar os mesmos registradores:
`moduloTPM[1]->CONTROLS[0].CnSC`
`moduloTPM[1]->CONTROLS[0].CnV`
`moduloTPM[1]->CONTROLS[1].CnSC`
`moduloTPM[1]->CONTROLS[1].CnV`.
Observe que nas duas últimas alternativas, fazemos referências aos mesmos membros de dois blocos de dados distintos por parametrização. Isso facilita a parametrização das funções como na implementação das funções `TPM_config_especifica` e `TPM_CH_config_especifica` nos projetos `rot8_example1`, `rot8_example2` e `rot8_aula`.

Aritmética de Pontos Flutuantes

Na Figura 1 os ciclos de trabalho são expressos em percentagens do tempo em que a carga entra em atividade. Uma outra forma é representá-la como uma razão no intervalo [0,1.] usando valores em ponto flutuante. As representações em ponto flutuante, seguindo o padrão IEEE754 [5], são as mais difundidas para representar **as aproximações dos números reais**. Essas representações permitem descrever, com uma acurácia maior, a parte fracionária dos valores de diferentes ordens de grandeza usando uma quantidade fixa de *bytes* (4 para precisão simples e 8 para precisão dupla). A figura 5 ilustra o padrão IEEE754 da representação de pontos flutuantes. Ao invés de espaçamentos equidistantes dos valores do tipo inteiro, a parte inteira dos valores do tipo `float` é espaçada de forma não uniforme ao longo da reta real, de forma que quanto menores são os valores menor é o espaçamento entre eles. Porém, a quantidade de pontos entre dois valores subsequentes, representando a parte fracionária entre eles, é a mesma. Assim, a resolução da parte fracionária varia conforme o valor da parte inteira. Quanto menor o valor da parte inteira, maior é a resolução da parte fracionária.

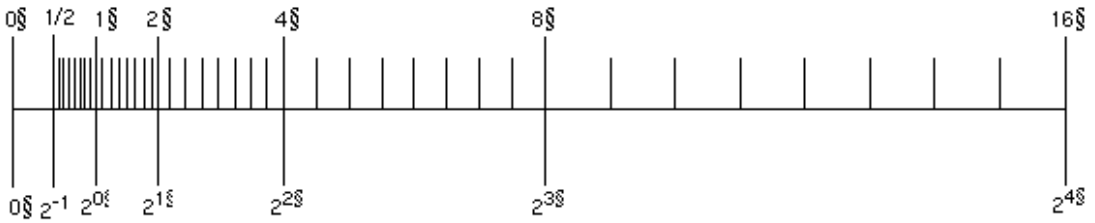

Figura 5: Densidade dos pontos representáveis pelo padrão IEEE754 na reta real (Fonte: **[24]**).

Em termos de *hardware*, o processamento da aritmética dos pontos flutuantes é distinto do processamento da aritmética dos inteiros [6]. Essas diferenças são consideradas em linguagem de programação C. Dois tipos de dados nativos, **float** e **double**, são reservados em C para declarar variáveis de valores fracionários, e o ponto '.' entre os dígitos para separar as casas inteiras das casas decimais.

O compilador C distingue as operações inteiras e de pontos flutuantes pelos tipos de operandos envolvidos. Quando se tratam de dois operandos inteiros, a aritmética de inteiros é aplicada e o resultado é truncado para um valor inteiro. Por exemplo, o resultado da divisão de duas constantes (1/2) é 0 em ponto fixo, e 0.5 em ponto flutuante. Para usar a aritmética de pontos flutuantes, um dos operandos envolvidos na operação deve ser ponto flutuante. Por exemplo, representar 1 pela convenção de ponto flutuante 1. (1.0) ou fazer uma **conversão explítica** ((float)1). Automaticamente, outros operandos são "promovidos" implicitamente para pontos flutuantes e a aritmética de pontos flutuantes é aplicada pelo compilador [7].

Em KL25Z só há circuitos dedicados para processamento de inteiros. As operações em pontos flutuantes são emuladas. Para muitos projetos de sistemas embarcados, a aritmética dos inteiros é suficiente. Antes de decidir pela aritmética de pontos flutuantes, deve-se ponderar cuidadosamente o compromisso entre o desempenho, a precisão, a propagação de erros de arredondamento e truncamentos implícitos, e os cuidados adicionais na codificação [8]. Em alguns casos, o uso de aritmética de pontos flutuantes deve ser evitado para não gerar efeitos bizarros na interface com usuários. Se aplicarmos a aritmética de pontos flutuantes para extrair os dígitos antes e depois da vírgula ou do ponto, podemos obter resultados inesperados pelos erros de truncamento acumulados. Por exemplo, mostrar 4.99 ao invés de 5.0 esperado. Há técnicas para converter os pontos flutuantes para inteiros, como mostra em [9].

Conversão de pontos flutuantes para *strings*

Neste roteiro estamos interessados em converter valores em ponto flutuante para *strings* e renderizá-las no LCD ou no Terminal. Um algoritmo, que reduz o problema de conversão de um valor em ponto flutuante à conversão de dois valores inteiros, teria os seguintes passos:

1. Verifique se o número é negativo. Se for, registre um sinal negativo e altere o valor para positivo.
2. Converta a parte inteira do número para *string* usando a função `itoa`.
3. Multiplique o valor fracionário por uma potência de 10 igual à quantidade de casas decimais desejadas.
4. Arredonde o resultado do produto para um valor inteiro.
5. Converta o resultado arredondado para *string* usando a função `itoa` ou o equivalente.
6. Adicione um ponto decimal ao final da *string* da parte inteira.
7. Concatene as duas strings da parte inteira e fracionária para formar a *string* final.
8. Se o número original era negativo, adicione um sinal negativo à *string* final.

Em [12] é apresentada uma implementação do algoritmo em C. Dois casos não foram considerados no algoritmo: (1) o número ser negativo e (2) arredondamento das casas decimais descartadas.

Geração de Números Aleatórios

Um gerador de números aleatórios gera uma sequência de números que não seguem uma ordem determinística. O procedimento pode ser implementado por *hardware* ou por *software*. A imprevisibilidade dos valores gerados o torna bastante atraente para aplicações em que comportamentos imprevisíveis e aleatórios são altamente desejáveis, como em jogos de azar, simulações, criptografia e geração de dados para teste de *software* e treinamento de máquina.

A função `rand` disponível na biblioteca-padrão de C, `stdlib.h`, consegue gerar **números inteiros pseudoaleatórios** em um intervalo especificado. São denominados pseudoaleatórios porque são gerados por um procedimento determinístico a partir de uma semente inicial. Para tornar a sequência de números gerados mais imprevisível, é comum usar a função `srand` para gerar diferentes sementes à função `rand` [18].

Podemos também explorar a imprevisibilidade dos números lidos dos contadores integrados nos temporizadores dos microcontroladores para gerar números aleatórios. Essa imprevisibilidade decorre de vários fatores externos, como interrupções, variações de frequências, variações de temperatura, variações de tensão, e até execução de instruções que acessam o contador. Um algoritmo simples de geração de um valor aleatório de n *bits* é extrair os n *bits* menos significativos do valor lido de um contador pulsado por um sinal de relógio de alta frequência. Por exemplo, podemos usar o contador `TPM1_CNT` para gerar um valor aleatório entre 0x3E8 e 0xFFF amostrando **repetidamente** o conteúdo do contador com os seguintes comandos até encontrar um valor maior que 0x3E8:

```
aleatorio = (TPM1_CNT & 0xFFF);
while (aleatorio < 0x3E8) aleatorio = (TPM1_CNT & 0xFFF);
```

Proteção de Ações Indevidas dos Usuários usando Máquinas de Estado

Vimos no roteiro 7 [10] que, ao modelarmos o sistema em projeto como uma máquina de estados e definirmos adequadamente as ações permitidas em cada estado, podemos proteger as regiões críticas. Podemos adotar estratégia similar para proteger o nosso sistema de ações indevidas dos usuários. Ao implementarmos as regras de restrição nas ações dos usuários como também nas transições entre os estados de um sistema, podemos evitar que ações potencialmente danosas levem o sistema a um estado não previsto e causem estragos inesperados. O projeto `medidor` exige que as reações das pessoas, por meio de pressionamento na botoeira NMI, sejam sincronizadas com os estímulos gerados pelo microcontrolador. Para proteger o sistema do processamento das ações não-sincronizadas sobre a botoeira, podemos desabilitar as interrupções geradas por tais entradas e, quando não for possível, definir as ações permitidas para cada estado e implementar nas rotinas de serviço que tratam tais entradas as regras de validação do estado do sistema. Isso permite que o processamento de uma entrada indevida seja interrompido. No entanto, é importante que todos os tratamentos sejam cuidadosamente ponderados para evitar descartes equivocados dos eventos de interrupção e para garantir a transparência e a fluidez na operação do sistema.

EXPERIMENTO

Neste experimento vamos desenvolver o projeto de tempos de reação audível, `tempo_reacao`, que segue a seguinte dinâmica:

a) Geram-se intervalos de tempo de espera aleatórios e renderiza-se no visor do LCD “Pressione

IRQA12" (estado `PREPARA_INICIO`).
b) Aguarde o acionamento de IRQA12 (estado `INICIO`). Ao pressionar o botão IRQA12, vai mostrar no visor "Teste Auditivo" (estado `PREPARA_AUDITIVO`) e entra no estado de espera por um intervalo de tempo aleatório (estado `ESPERA_ESTIMULO_AUDITIVO`).
c) Após um tempo aleatório, **gera-se um estímulo auditivo** e armazena-se o valor do contador C_{T1} do instante em que iniciou o estímulo. O *buzzer* (Figura 6) soa e fica aguardando a reação via o acionamento da botoeira NMI (estado `ESPERA_REACAO_AUDITIVA`).

Figura 6: *Buzzer* piezoelétrico.

d) Ao pressionar NMI, captura a contagem C_{T2}. Mostra-se no visor do LCD o tempo de reação medido (estado `RESULTADO`) e entra no estado de espera de 3s para que os valores possam ser lidos (estado `LEITURA`).
e) Volta para (a).

Figura 7 sintetiza os 7 estados do sistema que foram usados para modelar a dinâmica do sistema.

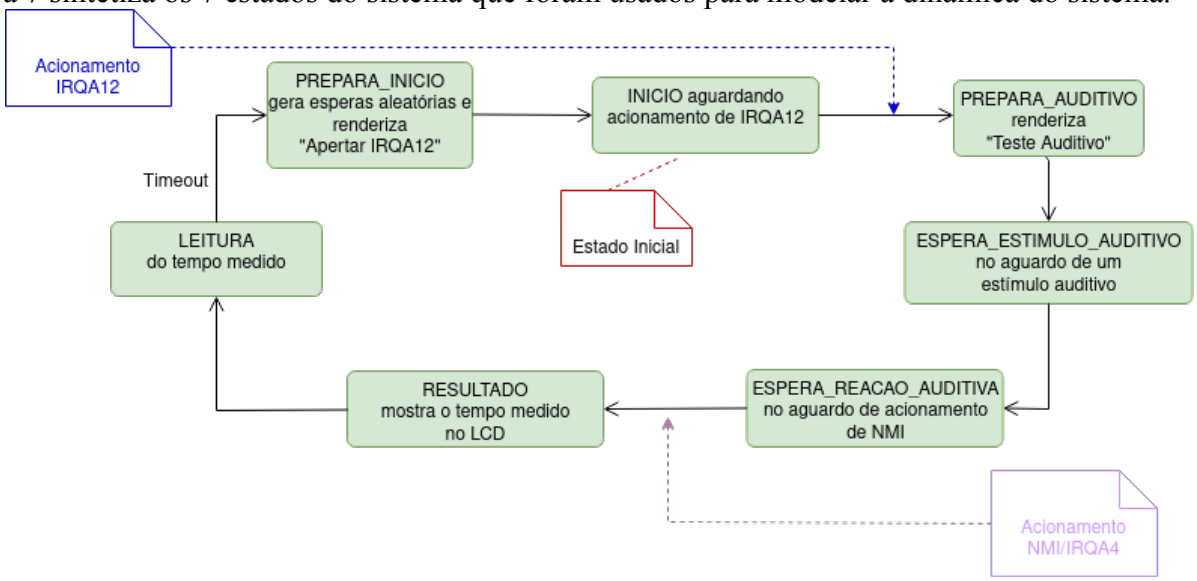

Figura 7: Diagrama de máquina de estados do projeto `temp_reacao` (editado em [19]).

A figura 8 mostra os componentes em *hardware* (vermelho) e *software* (preto) necessários para a implementação do projeto no *kit* disponível no nosso laboratório.

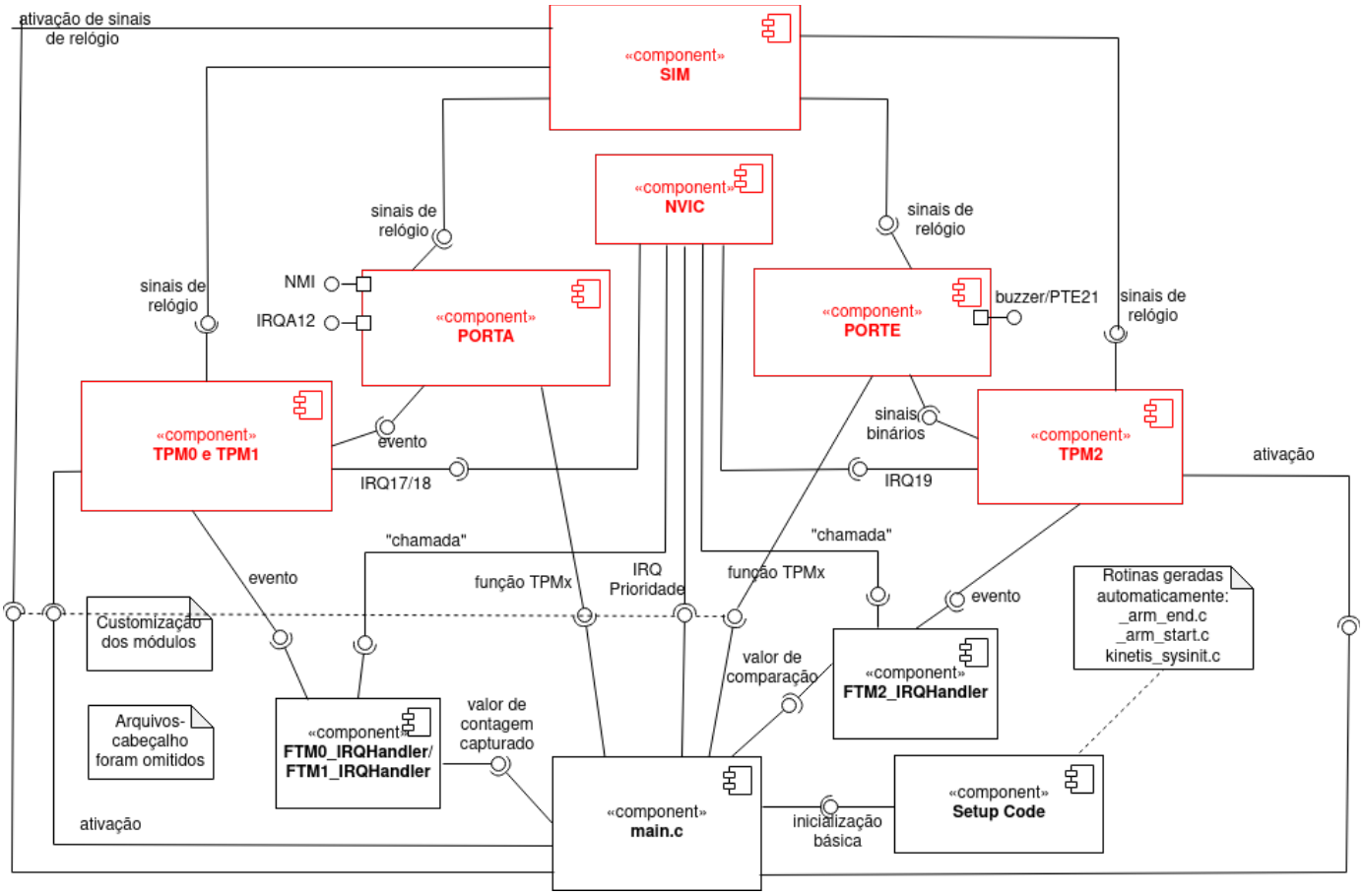

Figura 8: Diagrama de componentes do projeto `temp_reacao` (editado em [19]).

A fonte de sinais de relógio é MCGFLLCLK 20.971.520Hz. O período de TPM0, com o divisor *prescaler* setado em 128, é configurado em 0.25s para facilitar a contagem de tempo. A frequência de TPM1, com o divisor *prescaler* setado em 32, é configurado em 440Hz para gerar um sinal audível correspondente à nota LÁ da escala de 440Hz.

Segue-se um roteiro para o desenvolvimento do projeto. Usa-se na implementação do projeto as macros do arquivo-cabeçalho `derivative.h`.

1 **Aprender com os Exemplos dos Manuais**: Na seção 12.4/página 127 em [14] são apresentados dois exemplos de configuração de TPMx. O Exemplo 1 [20] é uma aplicação em que os canais TPM1_CH0 e TPM1_CH1 são configurados com função *Input Capture* (IC) e os canais TPM0_CH1 e TPM0_CH2, configurados com a função *edge-aligned* PWM (EPWM). Além disso, os eventos de estouro (*overflow*) de TPM0 são configurados como disparos para reinicialização periódica do contador `TPM1_CNT`, de maneira que o início de cada ciclo de contagem é sincronizado com um evento de estouro de TPM0. E o Exemplo 2 [21] é uma aplicação em que o canal TPM0_CH2 é configurado com a função *center-aligned* PWM (CPWM) e a transferência dos dados de 3 formas de onda distintas na memória para seu registrador `TPM0_C2V` via DMA no modo de baixo consumo energético (*Very Low Power Stop*, VLPS). A botoeira (PTA4) é configurada para "acordar" o sistema do modo de baixo consumo e alterar as formas de onda a serem transferidas para `TPM0_CH2`. Por problemas de acessos físicos aos pinos alocados para TPM0 no *shield* FEEC871, **foi usado o módulo TPM2 (CH0 e CH1) e TPM1 no lugar de TPM0 em `rot8_example1` e `rot8_example2`**. Executar os projetos no modo *Debug* do IDE *CodeWarrior* pode ajudar na análise.

1.1 **Módulo TPM**: Quais são as fontes de relógio e frequências configuradas para os sinais de barramento (*bus clock*) e os sinais de relógio `TPM clock` em exemplos apresentados em [14], em `rot8_example1` e `rot8_example2`?

1.2 **Configuração de um Período em TPM**: Quais são os valores configurados em `TPMx_MOD` e `TPMx_SC_PS` os módulos TPMx utilizados em [14], em `rot8_example1` e em `rot8_example2`? São condizentes com os *Períodos*/as frequências especificadas? A diferença na

configuração de `TPM1_MOD` em `rot8_example2` em relação ao Exemplo 2 em [14] teria impactos diferentes nas formas de onda geradas? Justifique.

1.3 **Funções Programáveis nos Canais de TPMx**: Quais são os valores configurados nos registradores `TPMx_CnSC` dos canais utilizados em [14], em `rot8_example1` e em `rot8_example2`? Estão condizentes com as funções especificadas paraa os canais utilizados nos dois exemplos?

1.4 **Alocação de Pinos para TPMx**: Quais pinos são alocados as canais utilizados nos exemplos e `rot8_*` para que eles se comuniquem com o mundo externo? Como eles são configurados? Observe que todos os pinos de saída estão com o *bit* `POTRx_PCRn_DSE` (*drive strength enable*) setado em '1' para aumentar a corrente de saída.

1.5 **Disparos Externos para TPMx_CNT**: O Exemplo 1 em [14], implementado em `rot8_example1`, demonstra o controle na operação do contador LPTPM por disparos externos, permitindo que o período da sua contagem seja sincronizado com algum evento externo. Para isso, basta configurar adequadamente o registrador `TPMx_CONF`. Qual foi a configuração feita nos campos `TPM1_CONF_TRGSEL`, `TPM1_CONF_CROT`, `TPM1_CONF_CSOO` e `TPM1_CONF_CSOT` do Exemplo 1? Consulte na Tabela 3-38/página 86 em [2] a fonte de disparos setada? A configuração está condizente com o esboço de ondas mostradas na Figura 2? Há outras insttruções, além das de configuração, para garantir que o comportamento se repita periodicamente? Justifique.

1.6 **Processamento de Interrupções em TPMx**: A implementação `rot8_example1` incluei duas rotinas de serviço, `FTM1_IRQHandler` e `FTM2_IRQHandler`.

1.6.1 A quais eventos de interrupção correspondem os tratamentos dados nas duas rotinas de serviço? Qual é a prioridade de atendimento setado aos eventos?

1.6.2 As variações das larguras de pulsos mostradas pelas linhas vermelhas pontilhadas na Figura 2 são controladas dentro da rotina de serviço `FTM2_IRQHander`. O que está sendo processado dentro dessa rotina de serviço que resulta na variação das larguras dos pulsos.

1.7 **Módulo DMA:** O Exemplo 2 em [14], implementado em `rot8_example2`, demonstra a transferência das amostras de uma forma de onda quandrada, triangular e senoidal para o canal TPM1_CH1 configurado com a função CPWM onde são gerados pulsos de larguras definidas pelos valores das amostras. As configurações para transferências estão implementadas em `DMA0_MemoTPM1CH1_config_especifica` no `rot8_example2`. Vale destacar as instruções adicionais inseridas em `rot8_example2` para que o programa seja operacional.

1.7.1 **Módulo DMAMUX:** Ao setar no código

```
DMAMUX0_CHCFG0 |= DMAMUX_CHCFG_SOURCE(55);
```

qual é a fonte habilitável para transferências via DMA? E qual dos 4 canais do DMA é selecionado para transferência?

1.7.2 Quais são os endereços iniciais dos blocos de dados configurados em `DMA_SAR0`, `DMA_DAR0`? Qual é o tamanho dos dados em *bytes* configurados em `DMA_DCR0_SSIZE` e `DMA_DCR0_DSIZE` para cada transferência?

1.7.3 Por quê o *bit* `DMA_DCR_SINC_MASK` está configurado em '1' e o *bit* `DMA_DCR_DINC_MASK` está setado em '0'?

1.7.4 Onde foram usados os *buffers* circulares em `rot8_example2`? E em `Example 2`?

1.8 **Processamento de Interrupções em DMA**: A implementação `rot8_example2` incluei duas rotinas de serviço, `PORTA_IRQHandler` e `DMA0_IRQHandler`. Ao acionar a botoeira NMI/PTA4 ou ao finalizar uma transferência, são gerados eventos de interrupção que fazem o microcontrolador acordar do seu estado VLPS para atendê-los.

1.8.1 Sob quais condições as formas de onda geradas na saída são alteradas entre senóide, triangular e quadrada?

1.8.2 Ao mudar a forma de onda, é nessário carregar no bloco cujo endereço está setado no canal 0 do DMA as amostras da nova forma de onda. Como a cópia pode ser "longa", o bloco de instruções de cópia foi deslocado para o laço principal da função `main`. Identifique este bloco de instruções.

1.8.3 Identifique o bloco de instruções que coloca o processador no modo VLPS enquanto o DMA transfere os dados da memória para TPM1_CH1. Como são somente 3 instruções, podemos

executá-las dentro das rotinas de serviço, ou seja, a ordem da execução das instruções altera o comportamento do sistema?

1.9 **Conversão de Sinal Modulado por Largura de Pulso em Sinal Modulado por Nível de Tensão**: Os sinais gerados no pino PTE21/TPM1_CH1 em `rot8_example2` são modulados por larguras dos pulsos. Para recurperá-los no formato de modulação por nível de tensão, devemos colocar um circuito RC, como ilustra a Figura 9. Registre as formas de onda dos sinais filtrados num osciloscópio usando R = 100Ω e variando C entre 22nF, 10uF, 22uF e 68uF para as 3 ondas. Ao conectar um capacitor eletrolítico, fique atento à sua polaridade. Qual é a relação entre os valores dos capacitores e o grau de suavização dos sinais filtrados?

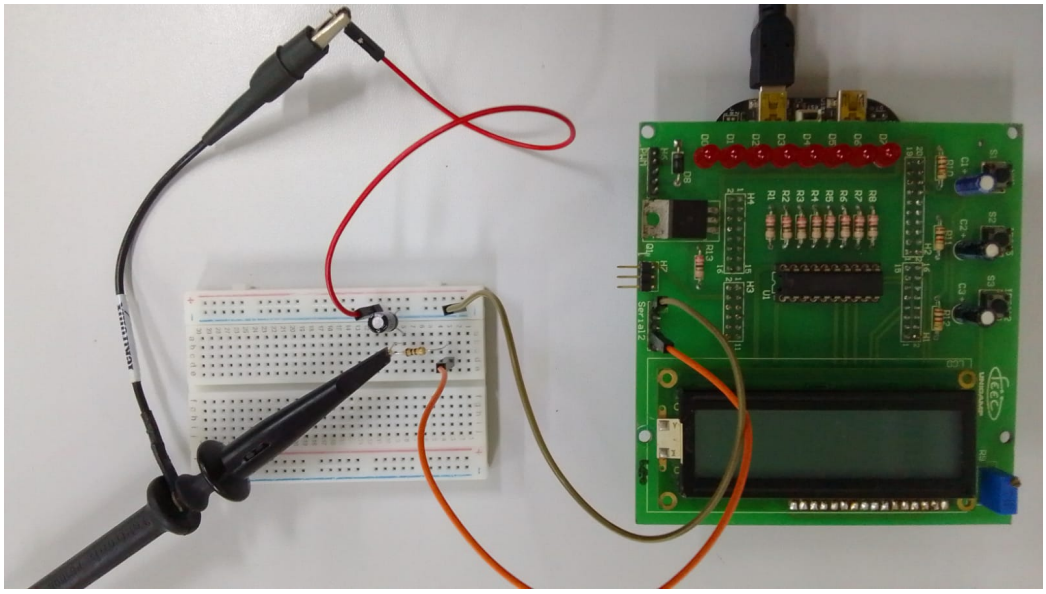

Figura 9: Filtro RC (a linha laranja deve ser conectada com o pino 3, ao invés do pino 2, do *header* H5 do shielder FEEC-EA871).

1.10 **Parametrização de Blocos de Dados**: Analise a implementação das funções `TPM_config_especifica` e `TPM_CH_config_especifica` que permitem configurar qualquer módulo TPMx e qualquer canal TPMx_CHn.

1.10.1 Como são acessados os registradores específicos nos canais e nos módulos nessas duas funções?

1.10.2 ~~Ao flexibilizarmos as funções, tivemos que abrir mão de algumas macros disponíveis e setar os *bits* diretamente. Como são setados os 4 *bits* TPMx_CnSC_[MsnB:MsnA:ELSnB:ELSnA]?~~

2 O projeto `rot8_aula` [22] demonstra o uso do par *Input Capture* (`TPM0_CH1`) e *Output Compare* (`TPM0_CH2`) para computar com precisão um intervalo de tempo e o uso do modo *Output Compare* (`TPM1_CH1`) para controlar com uma precisão maior o instante de ativação e desativação de um sinal de saída. O pino que serve o canal `TPM0_CH1` é o pino `PTA4`, onde está conectada a botoeira `NMI`. O pino que serve o canal `TPM1_CH1` é o pino 3 do *header* H5. Esse canal é ativado no primeiro pressionamento de NMI e desativado no segundo pressionamento. Entre os dois pressionamentos o canal coloca '1' na sua saída; do contrário, a saída é '0'.Como não temos acesso aos pinos que servem `TPM0_CH2` (Seção 10.3.1/página 161, em [2]), espelhamos o seu sinal de saída no pino PTE22 (pino 2 do *header* H5 do *shield* FEEC871). A fonte de sinais de relógio é a padrão MCGFLLCLK 20,971520MHz. Execute o programa e capture os sinais com o analisador lógico por um intervalo de 10s enquanto a botoeira `NMI` for acionada 6 vezes.

2.1 **Configuração de um Período em TPM:** Use um par de marcadores para mostrar o período do `TPM0`. Analise se o período registrado é condizente com a configuração feita.

2.2 **Cômputo de Intervalo de Tempo entre dois Eventos de Interrupção de *Input Capture* por *Output Compare***: Use 1 par de marcadores para mostrar, em unidade de tempo, a largura de um dos 3 pulsos mostrados no pino 3 do *header* H5. Compare o valor medido com o tempo correspondente à quantidade de períodos do `TPM0` contabilizados no seu canal 2, espelhados no pino 2 do *header* H5. Qual é a diferença esperada entre os dois tempos? A diferença observada está condizente com a esperada.

Dica: Coloque um *breakpoint* no final do bloco de código de tratamento do segundo pressionamento da botoeria para acessar os valores computados das variáveis `counter` e `valor1`, e o valor do registrador `TPM0_C2V`.

3 Desenvolva o projeto `tempo_reacao` em que a botoeira IRQA12 e o *buzzer* são conectados com os pinos PTA12 e PTE21 multiplexados, respectivamente, para TPM1_CH0 (IC) e TPM1_CH1 (EPWM) e o pino em que a botoeira NMI está conectada é multiplexado para TPM0_CH1. Para **proteção de ações indevidas dos usuários**, os canais são normalmente desativados (`MsnB:MsnA:ELSnB:ELSnA` = 0b0000). Somente nos estados `PREPARA_INICIO` e `INICIO`, é ativado o modo *IC* (0b00), sensível a borda de descida (0b10), para a botoeira IRQA12. Os canais da botoeira NMI e do *buzzer* só são ativados nos estados `ESPERA_ESTIMULO_AUDITIVO` e `ESPERA_REACAO_AUDITIVA`. E para **aumentar a aleatoriedade** dos momentos em que são produzidos os estímulos, é gerado um número aleatório `m` no estado `PREPARA_INICIO`.

Recomenda-se os seguintes passos que procuram reusar os códigos dos projetos `rot8_aula`.

3.1 Crie um novo projeto `tempo_reacao` (Seção 2.1/página 4 em [23]).

3.2 Adicione as últimas versões de `GPIO_latch_lcd.*`, `SIM.*`, `TPM.*`, `util.*` para reuso (Seção 2.2.3/página 14 em [23]). Crie novos arquivos `ISR.*` (Seção 2.2.2/página 14 em [23]).

3.3 Defina os estados e as regras de transições válidas para cada par de estados mostrados no diagrama de máquina de estados da figura 4. Especifique as ações permitidas em cada estado. Adicione o tipo de dado `enum estado_tag` em `ISR.h`, redefinido como `tipo_estado`, que nomeia os valores constantes associados aos diferentes estados com os nomes intuitivos dos estados, `PREPARA_INICIO`, `INICIO`, `ESPERA_ESTIMULO_AUDITIVO`, `ESPERA_REACAO_AUDITIVA`, `RESULTADO` e `LEITURA`. São esperadas as seguintes interações entre os eventos que ocorrem em diferentes estados e geram transições para outros estados :

a) Quando a **botoeira IRQA12** é acionada no estado `INICIO`, dispara-se a contagem de um intervalo aleatório de tempo no estado `ESPERA_ESTIMULO_AUDITIVO`. Para contar o tempo, pode-se habilitar a interrupção TOF do canal do *buzzer* (TPM1_CH1) com `TPM1_C1V`==0 (sem som) para contar um tempo (`m`**Período*$_{TPM1}$ + *resíduo*). O resíduo corresponde ao intervalo de tempo que o contador precisa para gerar o primeiro evento de estouro.

b) Transcorridas as **`m` ocorrências de TOF** em TPM1, é produzido um estímulo sonoro (*buzzer* deve soar). Para isso basta atribuir um valor, diferente de 0 e de `TPM1_MOD+1`, em `TPM1_C1V`. Para evitar interferências indevidas, desabilita-se o canal TPM1_CH0 (botoeira IRQA12). Por outro lado, é necessário habilitar a interrupção do `TPM0_CH1` (botoeira NMI) e iniciar a contagem do tempo de reação, resetando o contador `M` e habilitando o canal `TPM0_CH4` configurado no modo OC e com o valor corrente de `TPM0_CNT` setado em `TPM0_C4V`. Passa-se para o estado `ESPERA_REACAO_AUDITIVA` em que `M` é incrementado a cada evento de interrupção gerado por TPM0_CH4.

c) Quando a **botoeira NMI é acionada**, o valor de `TPM0_C1V` é copiado em C_{T2} e o valor de `TPM0_C4V` em C_{T1}. O **cômputo do intervalo de tempo entre as duas capturas por *Input Compare*** pode ser efetuado com as equações 3 e 4. Os canais de *buzzer* e NMI podem ser desabilitados (*buzzer* pára de soar).

d) **Aritmética de Pontos Flutuantes**: O intervalo de tempo em segundos é renderizado no visor do LCD no estado `RESULTADO`. Para isso, o valor numérico em **ponto flutuante deve ser convertido para uma *string***. Além disso, o valor renderizado deve se manter no visor por um intervalo de tempo, compatível com a velocidade de leitura de uma pessoa normal, no estado `LEITURA`.

e) Neste projeto, fixamos em 3s o tempo de leitura. Esse tempo de espera pode ser contado com uso de qualquer contador. Se escolhermos o canal TPM0_CH4 configurado em OC com período em 0.25s, deveremos aguardar por 12 eventos de interrupção para passar para o estado `PREPARA_INICIO`.

f) No estado `PREPARA_INICIO` é apagado o resultado no LCD e um novo número aleatório é gerado. O sistema é reinicializado para a próxima sequência de teste passando para o estado `INICIO`.

3.4 Inicialize o sistema.

3.4.1 Inicialize a conexão do LCD e KL25Z com a função `GPIO_ativaConLCD` e inicialize o LCD com as instruções recoomendadas pelo fabricante com `GPIO_initLCD`.

3.4.2 Configure a fonte de sinais de relógio para os contadores de TPM via `SIM_setaTPMSRC` e `SIM_setaFLLPLL`.

3.4.3 Implemente a função `void TPM0TPM1_PTA4PTA12PTE21_config_basica ()` em que são habilitados os sinais de relógio dos módulos TPM0 e TPM1 e alocados os pinos para eles.

3.4.4 Inicialize os módulos TPM0 e TPM1 com `TPM_config_especifica` e os canais que os pinos servem com `TPM_CH_config_especifica`.

3.4.5 **Alocação de Pinos para TPMx**: Inicialize o canal TPM0_CH4 como suporte às contagens de tempo no modo OC sem pino alocado.

3.5 Implemente os estados e as transições dos estados.

O comportamento do sistema é orientado aos eventos externos (itens (a)-(c), e) cujos tratamentos pelas rotinas de serviço consistem essencialmente em reconfiguração de alguns registradores específicos para alterar o modo de operação do *hardware* antes da transição para um novo estado. Outros 2 estados (d, f) estão relacionados com realimentações visuais, envolvendo processamentos lentos do LCD que são deslocados para o fluxo principal `main`. Portanto, as tarefas relacionadas a um estado e a sua transição a um outro estado estão distribuídas entre as rotinas de serviço e `main`.

3.5.1 `FTM1_IRQHandler`: trata a transição de `INICIO`→`ESPERA_ESTIMULO_AUDITIVO` na ocorrência do evento IRQA12 e a transição de `ESPERA_ESTIMULO_AUDITIVO`→ `ESPERA_REACAO_AUDITIVA` na ocorrência do evento TOF. Antes da transição são executadas as tarefas listadas nos itens 3.3.a e 3.3.b. Por exemplo, para o tratamento de TOF

```
if (estado == ESPERA_ESTIMULO_AUDITIVO) {
  tempo_aleatorio--;
  if (tempo_aleatorio == 0) {
     //limpa flag
     //(1) habilitar o buzzer no modo EPWM;
     //(2) habilitar a botoeira NMI no modo IC com interrupção;
     //(3) desabilitar evento de interrupção TOF de TPM1;
     //(4) resetar tempo_reacao;
     estado = ESPERA_REACAO_AUDITIVA;
  }
}
```

3.5.2 `FTM0_IRQHandler`: trata a transição de `ESPERA_REACAO_AUDITIVA`→ `RESULTADO` na ocorrência do evento NMI

```
} else if (estado == ESPERA_REACAO_AUDITIVA) {
  //limpa flag
  //(1) computar a contagem total;
  //(2) desabilitar o buzzer;
  //(3) desabilitar NMI;
  //(4) desabilitar o contador TPM0_CH4
  estado = RESULTADO;
}
```

e a transição de `LEITURA`→`PREPARA_INICIO` quando é habilitado o modo OC do canal `TPM0_CH4` com o contador `tempo_leitura` inicializado em 12

```
} else if (estado == LEITURA) {
  tempo_leitura--;
  if (tempo_leitura == 0) estado
  estado = PREPARA_INICIO;
}
```

3.5.3 `main`: trata a transição `RESULTADO`→`LEITURA` quando são atualizadas as mensagens mostradas no LCD e a transição `PREPARA_INICIO`→`INICIO` quando o visor do LCD é resetado para a próxima sessão de teste.

Sendo as tarefas executadas por *hardware* devidamente configurado, os testes consistem essencialmente na verificação do *hardware* configurado. Faça **testes de unidade** da atualização dos estados, colocando os pontos de parada no início de cada rotina de serviço na sequência esperada e execute os código, trecho por trecho.

3.6 Implemente as funções auxiliares:

3.6.1 **Geração de Números Aleatórios**: Uma função `uint32_t GeraNumeroAleatorio()` que gera um número aleatório para determinação do tempo de espera de um estímulo. O algoritmo que aproveita do contador requer que um contador esteja ativado. Talvez seja mais interessante inserí-la em `ISR.*`.

3.6.2 **Conversão de pontos flutuantes para *strings***: Implemente em `util.*` a função `void ftoa(float n, char* res, int afterpoint)` disponível em [12].

3.6.3 Comunicação entre arquivos: Foi optado o agrupamento de todas as rotinas de serviço em arquivos `ISR.*`, separadas da função `main` (`main.c`). Por modularidade, a visibilidade das variáveis declaradas em `ISR.*` são restritas a `ISR.*`. Para acessar o estado do sistema, copie em ISR.* as funções `ISR_LeEstado` e `ISR_EscreveEstado` implementadas nos projetos anteriores.

3.7 Implemente uma interface com usuário via LCD. São esperadas as seguintes mensagens:

- no estado `INICIO`: "Aperte IRQA12" a ser enviado no estado `PREPARA_INICIO`.
- no estado `LEITURA`: "Reação em xxx.x segundos", a ser enviada no estado `RESULTADO`, após o cômputo do tempo de reação em segundos com uma casa decimal e conversão para uma *string*.

Faça **testes de unidade** da operação do LCD, colocando um ponto de parada após o envio de uma mensagem.

3.8 Implmente a Máquina de Estados. Revise se todos os estados na Figura 5 são implementados. Com base nas regras definidas em 3.3, revise as transições implementadas para cada estado e as restrições implementadas que evitam traansições inválidas. Faça **testes funcionais** do projeto.

3.9 Refinamento. "Agrupar" os estados sem instruções na função `main` como o caso `default`. Fazer ajustes nas mensagens de interação com usuários e na documentação das funções implementadas.

3.10 Habilite *Print Size* para uma simples análise do tamanho de memória ocupado. Gere um executável e refaça os **testes funcionais** do projeto para diferentes situações para ver se a resposta está condizente com a especificação.

3.11 Gere uma documentação do projeto com Doxygen [15].

RELATÓRIO

O relatório deve ser devidamente identificado, contendo a identificação do instituto e da disciplina, o experimento realizado, o nome e RA do aluno. O prazo para execução deste experimento é duas semanas. O relatório é dividido em duas partes. Para a primeira semana, responda num arquivo em pdf, as perguntas dos itens 1 e 2 e suba o arquivo no sistema *Moodle*. Para a segunda semana, faça uma descrição sucinta dos testes conduzidos ao longo do desenvolvimento do projeto `tempo_reacao`, junto com algumas imagens ilustrativas, num arquivo em pdf. Exporte o projeto `tempo_reacao` **devidamente documentado** num arquivo comprimido no IDE CodeWarrior. Suba os dois arquivos no

sistema *Moodle*. **Não se esqueça de limpar o projeto (*Clean* …) e apagar as pastas html e latex geradas pelo Doxygen antes**.

REFERÊNCIAS

[1] ARMv6-M Architecture Reference Manual – ARM Limited.
https://www.dca.fee.unicamp.br/cursos/EA871/references/ARM/ARMv6-M.pdf
[2] KL25 Sub-Family Reference Manual – Freescale Semiconductors (doc. Number KL25P80M48SF0RM), Setembro 2012.
https://www.dca.fee.unicamp.br/cursos/EA871/references/ARM/KL25P80M48SF0RM.pdf
[3] PWM – Modulação por Largura do Pulso
http://www.mecaweb.com.br/eletronica/content/e_pwm
[4] Nova versão do esquemático do shield FEEC
https://www.dca.fee.unicamp.br/cursos/EA871/references/complementos_ea871/Esquematico_EA871-Rev3.pdf
[5] IEEE754 Converter
http://www.h-schmidt.net/FloatConverter/IEEE754.html
[6] Fixed-point vs. Floating-Point Digital Signal Processing
https://www.analog.com/en/technical-articles/fixed-point-vs-floating-point-dsp.html
[7] Type conversion in C
https://www.geeksforgeeks.org/type-conversion-c/
[8] Floating-point data in embedded software
https://www.embedded.com/floating-point-data-in-embedded-software/
[9] Simple Fixed-Point Conversion in C
https://embeddedartistry.com/blog/2018/07/12/simple-fixed-point-conversion-in-c/
[10] Roteiro 7
http://www.dca.fee.unicamp.br/cursos/EA871/1s2023/roteiros/roteiro7.pdf
[11] Understanding embedded C: What Are Structures?
https://www.allaboutcircuits.com/technical-articles/understanding-embedded-C-what-are-structures/
[12] Acervo Lima. Converta um número de ponto flutuante em string em C
https://acervolima.com/converta-um-numero-de-ponto-flutuante-em-string-em-c/
[13] Embedded Staff. Direct Memory Access (DMA)
https://www.embedded.com/introduction-to-direct-memory-access/
[14] Kinetis L Peripheral Module Quick Reference (Rev. 0.09/2012)
https://www.dca.fee.unicamp.br/cursos/EA871/references/ARM/KLQRUG.pdf
[15] Doxygen
https://www.doxygen.nl/manual/docblocks.html
[16] Tempo de Reação
https://mundoeducacao.uol.com.br/fisica/tempo-reacao.htm
[17] Elettroamici, Teorema de Nyquist-Shannon
https://www.elettroamici.org/pt/teorema-di-nyquist-shannon/
[18] Intellectuale. Valores aleatórios em C com a função rand
http://linguagemc.com.br/valores-aleatorios-em-c-com-a-funcao-rand/
[19] Diagrams.net
https://www.diagrams.net/
[20] rot8_example1.zip
http://www.dca.fee.unicamp.br/cursos/EA871/1s2023/codes/rot8_example1.zip
[21] rot8_example2.zip
http://www.dca.fee.unicamp.br/cursos/EA871/1s2023/codes/rot8_example2.zip
[22] rot8_aula.zip
http://www.dca.fee.unicamp.br/cursos/EA871/1s2023/codes/rot8_aula.zip
[23] Wu, S.T. Ambiente de Desenvolvimento de Software
https://www.dca.fee.unicamp.br/cursos/EA871/references/apostila_C/AmbienteDesenvolvimentoSoftware_V1.pdf
[24] IEEE Arithmetic
https://docs.oracle.com/cd/E19957-01/806-3568/ncg_math.html

Revisado em Fevereiro de 2023
Revisado em Março de 2022
Revisado em Maio e Julho de 2021
Revisado em Novembro de 2020