EA871 – LAB. DE PROGRAMAÇÃO BÁSICA DE SISTEMAS DIGITAIS

EXPERIMENTO 9 – ADC e LPTMR

Profa. Wu Shin-Ting

OBJETIVO: Apresentação das funcionalidades do módulo ADC (Conversor Analógico-Digital).

ASSUNTOS: Programação do módulo ADC do MKL25Z128 para conversão de sinais analógico-digitais com iniciação controlada por *software* ou por *hardware*.

O que você deve ser capaz ao final deste experimento?

Saber a diferença entre amostragem e quantização.

Entender o princípio de funcionamento de um conversor ADC por registrador de aproximações sucessivas (SAR).

Entender os termos utilizados para caracterizar um conversor ADC.

Entender os erros inerentes a um algoritmos de quantização de um sinal analógico.

Entender os erros envolvidos no *hardware* de conversão e o uso de auto-calibração para mitigá-los.

Entender a importância da filtragem dos sinais analógicos e o impacto do filtro no tempo de amostragem.

Saber configurar o conversor ADC para operar em diferentes modos de operação.

Saber estimar o tempo de conversão de um circuito ADC.

Entender o princípio de funcionamento de um temporizador/contador LPTMR.

Saber configurar LPTMR para operar como um temporizador.

Saber programar em KL25Z conversões periódicas disparadas por *hardware*.

Entender a técnica de filtragem exponencial na aquisição de dados com "memória".

Saber recuperar uma grandeza física a partir do valor binário amostrado por um conversor ADC.

Saber implementar os exemplos de aplicação dos módulos do microcontrolador apresentados em [5].

INTRODUÇÃO

A maioria dos sensores e sistemas audiovisuais gera sinais analógicos. Para serem processados pelos processadores digitais, como o nosso MCU, estes sinais precisam ser digitalizados, ou convertidos em sinais digitais através de um **conversor Analógico-Digital** (ADC). Podemos dizer que um conversor ADC é um circuito que procura mapear, de forma mais linear possível, as amostras de tensões no intervalo `[VREFL,VREFH]` em sinais digitais específicos, correspondentes aos códigos binários 0 a 2^N-1 representáveis por `N` *bits* pré-definidos.

O bloco comum a todos os conversores analogico-digital (**ADC**) é um circuito de captura/**amostragem e retenção** (*sample and hold*) em que o circuito lê uma **amostra** Vi do sinal a

ser convertido e a **retem** num componente, como capacitor, de modo que, mesmo que o sinal varie, a amostra Vi é mantida até o próximo instante de captura. O valor analógico Vi é, então, convertido para um valor digital por um processo denominado **quantização**. Há diferentes técnicas de quantização, envolvendo distintas tecnologias [4]. Essencialmente, distinguem-se dois esquemas: com e sem uso de um conversor digital-analógico (**DAC**). Os mais utilizados são ainda os que usam conversores DAC. Dada uma saída digital de N *bits*, o procedimento consiste em varrer as possíveis combinações binárias de N *bits*, convertendo-as para valores analógicos e compará-los com o valor analógico de entrada, até chegar a um valor que se aproxime da entrada. Ele acontece em 5 passos (Figura 1):

1. disparo para inicialização (`Start`);
2. geração de um valor binário, por um contador (`Counter`) por exemplo;
3. conversão do valor binário em tensão analógica Va via DAC,
4. comparação de Va com a tensão de entrada Vi
5. ativação do fim de uma conversão (EOC, *end-of-conversion*), quando Va-Vi é suficientemente pequena; senão volta-se para o passo (2) para tentar com um próximo valor binário do `Counter`.

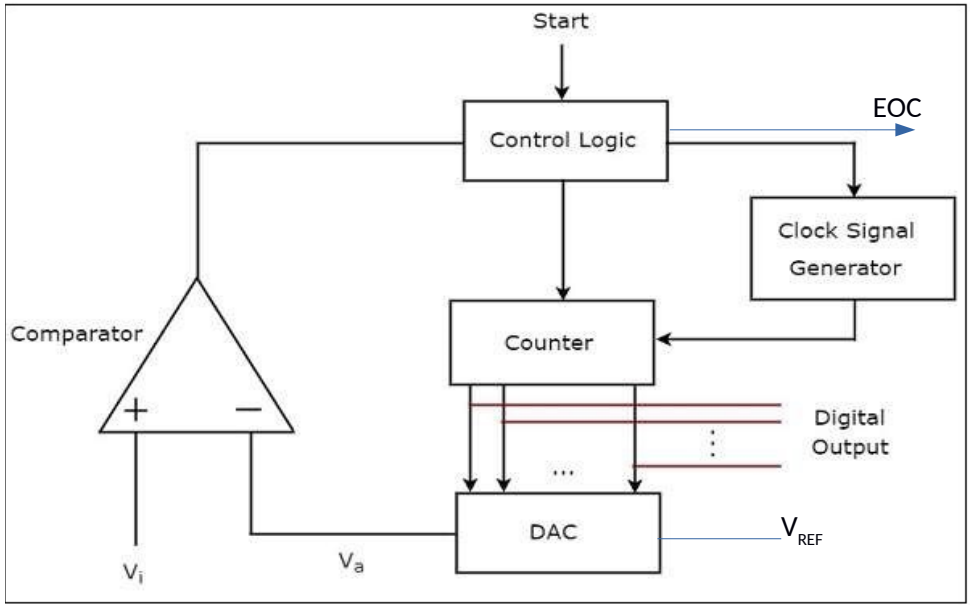


Figura 1: Diagrama de blocos de um conversor ADC (Fonte: [16]).

Neste experimento vamos apresentar as funções configuráveis no módulo ADC através de dois exemplos de projeto e aplicar o aprendizado no desenvolvimento de um projeto de um `controle_cooler` da velocidade de um *cooler* via um potenciômetro rotativo cujas tensões podem ser alteradas girando um botão conectado ao seu eixo de giro. As tensões na saída do potenciômetro são amostradas periodicamente através do pino PTB1 e digitalizadas em códigos binários pelo módulo ADC. Esses valores binários são usados para controlar o ciclo de trabalho de um sinal PWM gerado pelo canal TPM1_CH0 (PTB0) que alimenta a base de um transistor Darlingtong NPN TIP31, chaveando-o entre os estados de corte e de saturação num circuito em série com um *cooler* (Figura 10). Temporizadores (LPTMR e PIT) são configurados para gerar disparos periódicos que inicializam conversões em ADC, assegurando a periodicidade na amostragem dos sinais do potenciômetro.

Amostragem e Quantização

Segundo o t**eorema de amostragem de *Nyquist***, a frequência de amostragem (f) deve ser no

mínimo duas vezes a maior frequência do espectro do sinal. E o tempo de amostragem e de conversão terá que ser menor que 1/f a fim de permitir que o sinal seja perfeitamente recuperável a partir de uma sequência infinita de amostras.

Um sistema digital de N *bits* consegue representar até 2^N valores distintos. Considerando que as tensões de entrada sejam divididas igualmente entre 2^N **níveis de quantização**, a menor unidade representável seria o ***bit* menos significativo** (***least significant bit***, **LSB**). Uma variação (discreta) nesse *bit* corresponde a uma faixa de variações (contínuas) no sinal analógico de entrada. Dado o valor máximo VREF que um conversor consegue converter, o LSB é definido como a razão entre esse valor máximo e 2^N níveis (LSB = VREF/2^N). Essa razão é conhecida como a **resolução** do conversor. É comum expressá-la em termos de percentagem do valor máximo VREF, isto é,

$$Resolução = \frac{LSB}{VREF} \times 100\,\%. \qquad (1)$$

Na prática, essa resolução é especificada pelo número N de *bits* do conversor.

Chamamos de **erro de quantização** a diferença entre o valor analógico VA e o valor analógico correspondente ao nível de quantização de VA. Quando o valor analógico representante de um nível de quantização corresponde ao ponto médio do intervalo, o erro máximo de quantização é |0.5 LSB|. E quando o valor analógico representante é o extremo menor do intervalo, o erro máximo de quantização é |1 LSB|. A seção 28.6.2.4/página 505 em [1] mostra que no microcontrolador KL25Z o valor analógico representante é o ponto médio para as conversões de 8, 10 e 12 *bits* como mostra na figura 2. Note a assimetria na conversão: a largura do degrau do código binário 0 é 0.5LSB e a do último código binário 2^N-1 é 1.5LSB.

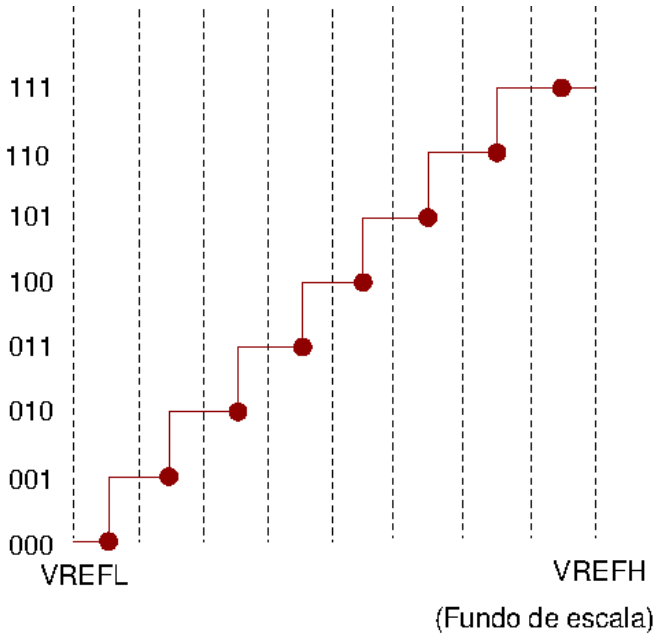


Figura 2: Níveis de quantização: pontos médios de intervalos analógicos.

E para a conversão de 16 *bits*, o valor analógico representante é o extremo menor do intervalo como mostra na figura 3. Neste caso, o erro de quantização varia entre -1LSB (maior "desvio" em relação à entrada amostrada) e 0LSB, e a largura do degrau é 1LSB para todos os níveis de quantização.

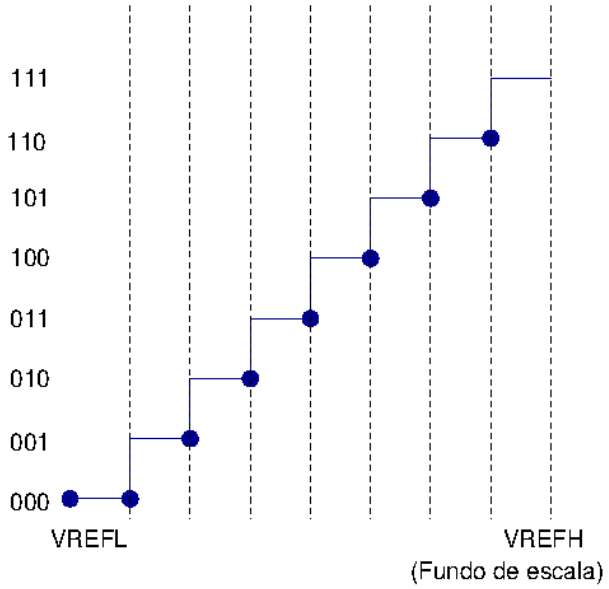


Figura 3: Níveis de quantização: extremo menor de intervalos analógicos.

Erros inerentes do *Hardware* de ADC

Além do erro de quantização, há ainda erros inerentes aos componentes de um conversor, como as tensões de referência, os resistores e os comparadores. Esses erros podem fazer com que o valor analógico digitalizado se afasta do valor analógico real. A **acurácia** de um conversor é uma forma de quantificação dessa diferença, usualmente dada em percentagem de VREF. Embora na prática a acurácia e a resolução sejam da mesma ordem de grandeza, a acurácia e a resolução são dois parâmetros independentes. Aumentar a quantidade de *bits* não implica no aumento da acurácia dos componentes de um conversor, .

Distinguem-se entre os erros inerentes aos componentes de um conversor erros estáticos e erros dinâmicos. Os **erros estáticos** são aqueles que afetam a acurácia do conversor quando este converte um sinal CC. Esses erros podem ser completamente descritos por quatro erros e são aplicados na análise do desempenho de um conversor ADC [15]:

- **erros de *offset***: é o valor de entrada no meio do degrau quando o nível de quantização é 0.
- **erro de ganho**: é a diferença entre o valor do meio do degrau ideal e o valor do meio do degrau real quando o nível de quantização assume o valor máximo.
- erro de linearidade diferencial: é a diferença entre a largura de 1LSB de um degrau ideal e a largura de um degrau real.
- erro de linearidade integral: é o desvio da inclinação da função real de conversão da função ideal.

Registrador por Aproximações Sucessivas

Na figura 1 é apresentada uma das versões mais simples do ADC em que se usa um contador (`Counter`) para varrer incrementalmente os possíveis códigos binários, até chegar num valor que esteja suficientemente próximo do valor amostrado. Pelo sinal de saída do `Counter` ter a forma de onda de uma rampa, o conversor é conhecido por ADC de **rampa digital**. Para melhorar o tempo de busca por um código binário mais próximo, reduzir o consumo de energia e simplificar o circuito, os ADCs integrados em microcontroladotes são de **registrador de aproximações**

sucessivas (***Successive Approximation Register,*** SAR) [2]. SAR é um circuito digital que usa uma **técnica de busca binária** para determinar o valor digital correspondente ao valor analógico de entrada. A técnica de busca binária consiste em dividir o intervalo de valores possíveis em duas metades a cada iteração, até que o valor que mais se aproxima do valor analógico de entrada seja encontrado. Isso reduz a quantidade de comparações para o pior caso, de 2^N num ADC de N *bits* de rampa digital para $\log_2 2^N = N$ num ADC de registrador de aproximações sucessivas.

Módulo ADCx

O microcontrolador KL25Z possui integrado um conversor analógico-digital de 16 *bits* (Capítulo 28/página 457, em [1]). A técnica implementada é a de **aproximações sucessivas** com um registrador de aproximações sucessivas (*successive approximation register SAR*) de 16 *bits*[2], ou seja, o `Counter` na Figura 1 é substituído por um circuito SAR, como ilustra a Figura 4. O comparador realimenta o circuito do registrador SAR com a diferença dos dois sinais e atualiza o conteúdo do SAR com base nesta diferença. E assim, sucessivamente, até que a tensão correspondente ao código binário em SAR se iguale a V_{IN} dentro de uma faixa de tolerância pré-estabelecida, e o *bit* de estado EOC (*end of conversion*)/COCO (*conversion complete*) fique em '1'.

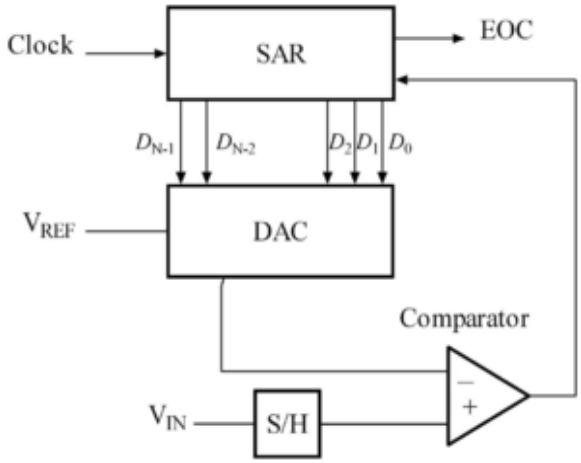


Figura 4: Diagrama de blocos de um ADC por SAR (Fonte: [18]).

De acordo com a tabela na Seção 3.7.1.3.1/página 79 em [1], o conversor ADC de KL25Z dispõe de 16 pinos físicos para entrada dos sinais analógicos. Estes pinos podem servir as 24 entradas simples para conversões unipolares (*singular*) ou as 4 entradas diferenciais para conversões bipolares (*differential*). As tensões de referência `VREFH` e `VREFL` utilizadas na conversão (Seção 28.4.2/página 484 em [1]) são configuráveis pelos *bits* `ADCx_SC2_REFSEL` (Seção 28.3.6/página 470 em [1]). O *kit* disponivel no laboratório foi projetado para operar no modo `0b00` (`VREFH~3V3` e `VREFL=0V`) (Seção 28.6.1.2/página 502 em [1]). Todas as amostras de tensão dentro desse intervalo são diretamente proporcionais aos códigos binários de 0 até 2^N-1, sendo `N` a quantidade de *bits* alocados para representação digital. Em KL25Z, a resolução de uma conversão pode ser de 8, 10, 12 ou 16 *bits*, configurável pelos *bits* `ADCx_CFG1_MODE` (Seção 28.3.2/página 465 em [1]). Amostras menores que `VREFL` e maiores que `VREFH` tem, respectivamente, os códigos binários truncados em 0 e 2^N-1, conforme explica a Seção 28.6.1.3/página 503 em [1].

O modo de operação, uni- ou bipolar, em cada entrada é controlado pelos *bits* de controle `ADCx_SC1n[DIFF]`. Como só há um circuito conversor, o sinal analógico processado em cada instante é o que está configurado nos 5 *bits* `ADCx_SC1n_ADCH` da entrada `SC1n` selecionada pelo

bit de configuração ADCx_CFG2_MUXSEL. Há um sensor de temperatura AN3031 [3] integrado em KL25Z (Seção 28.4.8/página 497 em [1]). Esse sensor pode ser usado para monitorar a temperatura do processador. Ele já se encontra associado ao canal 0b11010 do conversor. Portanto, para amostrarmos as tensões adquiridas por este sensor, basta setarmos este canal no campo ADCx_SC1A_ADCH. E para desabilitarmos a entrada SC1n do módulo ADC0, atribuímos o código 0b1111 a ADCx_SC1n_ADCH.

É necessário habilitar o sinal de relógio do módulo ADC através do *bit* de configuração SIM_SCGC6_ADC0 para que o módulo inicie processamentos (Seção 12.2.10/página 207 em [1]). O ADCx é alimentado com duas fontes de sinais de relógio: sinal de relógio para comunicação com o barramento (*bus clock*) e sinal de relógio ADCK para o circuito de conversão (Tabela 5-2/página 121 em [1]). A fonte de ADCK é configurável pelos *bits* de controle ADC0_CFG1_ADICLK (Seção 28.3.3/página 466 em [1]). Dentro do módulo ADC, há ainda um divisor de frequência ADC0_CFG1_ADIV para reduzir a frequência do sinal de relógio (Seção 28.4.1/página 483 em [1]).

A forma como se inicia/dispara uma conversão é configurável através do campo ADCx_SC2_ADTRG (Seção 28.4.3/página 484 em [1]). O disparo pode ser por *software*, através de um acesso de escrita ao registrador ADCx_SC1A, ou por *hardware*, através da seleção da fonte de disparo pelos *bits* SIM_SOPT7_ADCxTRGSEL (Seção 12.2.6/página 200 em [1]). No caso de *hardware*, a fonte selecionada deve estar devidamente configurada para produzir os eventos geradores de disparos. Os disparos acontecem automaticamente, sem intervenção de *software*. KL25Z tem 2 entradas abstraídas em dois registradores SC1A e SC1B. Elas são configuráveis separadamente e operáveis no modo "*ping-pong*". Somente a entrada SC1A suporta os dois tipos de disparos, por *software* e por *hardware*. A entrada SC1B só suporta disparos por *hardware*.

Enquanto uma conversão estiver em progresso, o *bit* ADCx_SC2_ADACT se mantém setado em '1'. O modo de conversão pode ser único (uma só vez) ou contínuo (sucessivamente após um único disparo inicial), configurável pelo *bit* ADCx_SC3_ADCO (Seção 28.3.7/página 472 em [1]). Quando se completa uma conversão (Seção 28.4.4.2/página 486 em [1]) e o resultado satisfaz as funções comparadoras configuradas, o *bit* de estado ADCx_SC1n_COCO é automaticamente setado em '1' e o resultado da conversão é guardado no registrador de dados ADCx_Rn da entrada n selecionado pelo *bit* de configuração ADCx_CFG2_MUXSEL. A tabela na Seção 3.7.1.3.1/página 79 em [1] mostra os sinais de entrada permitidos para ambas as entradas A e B e os nomes dos pinos usados na tabela da Seção 10.3.1/página 161 em [1].

O resultado transferido para ADCx_Rn é, de fato, a média de um conjunto de amostras. O número de amostras por resultado é configurável através dos campos ADCx_SC3_AVGE e ADCx_SC3_AVGS (Seção 28.4.4.7/página 492 em [1]). Quando devidamente habilitada a função comparadora pelo *bit* de configuração ADCx_SC2_ACFE, a média gerada pela conversão é automaticamente comparada com os limites do intervalo pré-setado nos registradores de dados ADCx_Cvn cujos valores relativos devam atender as condições especificados pelo fabricante, apresentadas na Tabela 28-78/página 493 em [1]. O tipo de comparação a ser feito é configurável pelos *bits* de controle ADCx_SC2_ACFGT e ADCx_SC2_ACREN.

Filtragem de Dados Analógicos

Os sinais analógicos estão sujeitos a ruídos e distorções, como interferências eletromagnéticas, flutuações de tensão e outros. Esses ruídos podem causar distorções no processo de conversão. Um pequeno circuito RC de baixo custo é usualmente incluído nos pinos de entrada analógica. O valor

R é tipicamente 100 Ohms e o valor C é escolhido de tal forma que remova, antes da amostragem de um sinal, as frequências acima da frequência de Nyquist, ou seja a metade da frequência de amostragem, para evitar distorções (Seção 11.2.2/página 120 em [5])

$$f_{amostragem} = 2 \times f_{Nyquist} = \frac{2}{2 \times \pi \times R \times C} = \frac{1}{\pi \times R \times C} \quad (2)$$

Por exemplo, para uma frequência de amostragem de 30Hz, o valor C deve ser aproximadamente 100uF. É importante lembrar que **a resistência R pode afetar o tempo de conversão** de um conversor ADC de várias maneiras, incluindo atraso no tempo de resposta e erros de conversão (Seção 28.6.2.2/página 504 em [1]). O módulo ADC0 permite que sejam feitos alguns ajustes no tempo de amostragem de acordo com a impedância no pino de entrada através dos *bits* `ADCx_CFG1_ADLSMP` (Seção 28.3.2/página 466 em [1]) e `ADCx_CFG2_ADLSTS` (Seção 28.3.3/página 467 em [1]). Dados empíricos mostram que capacitores de 0,01 µF com boas características de alta frequência são muitas vezes suficientes. Esses capacitores não são necessários em todos os casos, mas quando usados, devem ser colocados o mais próximo possível dos pinos e ter o mesmo terra analógico do microcontrolador (Seção 28.6.1.2/página 502 em [1]).

Funções de Comparação em ADCx

Em KL25Z, ACFE, ACREN e ACFGT são termos usados para descrever as seis funções de comparação nos seus conversores ADC que determinam a transferência de um resultado de conversão para o registrador de saída `ADCx_Rn`. As 6 funções são configuradas através dos seguintes *bits* (Seção 28.4.5/página 493 em [1]):

1. `ADCx_SC2_ACFE` (habilitação da função de comparação): quando setado, é aplicada a função de comparação configurada sobre os resultados de conversão.
2. `ADCx_SC2_ACFGT` (comparação do resultado em relação a `ADCx_CV1`): quando setado em '1', a função comparadora aplicada no resultado de uma conversão em relação ao valor `ADCx_CV1` é "maior que". Senão, a função é "menor ou igual".
3. `ADCx_SC2_ACREN` (comparação do resultado em relação aos intervalos definidos por `ADCx_CV1` e `ADCx_CV2`): quando é resetado em '0', a função comparadora configurada em `ADCx_SC2_ACFGT` é aplicada no resultado da conversão em relação a `ADCx_CV1`. Se o *bit* é setado em '1', a comparação do resultado de conversão é feita em relação a um dos quatro intervalos de valores:
   - (-∞,`ADCx_CV1`) ou (`ADCx_CV2`,∞), se `ADCx_SC2_ACFGT==0` e `ADCx_CV1` ⩽ `ADCx_CV2`,
   - (`ADCx_CV2`, `ADCx_CV1`), se `ADCx_SC2_ACFGT==0` e `ADCx_CV1` > `ADCx_CV2`,
   - [`ADCx_CV1`, `ADCx_CV2`], se `ADCx_SC2_ACFGT==1` e `ADCx_CV1` ⩽ `ADCx_CV2`, e
   - (-∞,`ADCx_CV2`] ou [`ADCx_CV1`,∞), se `ADCx_SC2_ACFGT==1` e `ADCx_CV1` > `ADCx_CV2`.

Configuração de Disparos de Conversão por *Hardware*

Vimos que uma conversão pode ser disparada por *software* ou por *hardware*. Em KL25Z, todas as entradas suportam disparos por *hardware*. Somente a entrada ADCx_SC1A suporta adicionalmente uma conversão disparada por *software*. Quando se opta por *hardware*, é necessário ainda

configurar e habilitar o módulo que gera sinais de disparo. Na Tabela 3-1/página 45 em [1] é apresentada uma lista de módulos interconectados em KL25Z, incluindo os módulos cujos eventos gerados possam servir de disparos para o módulo ADC (*ADC Trigger*). Na sexta coluna dessa tabela são mostrados os campos do registrador `SIM_SOPT7` que configuram as fontes de disparo para as entradas `A` e `B` do ADC0. Os códigos válidos para serem setados estão descritos na Seção 12.2.6/página 201 em [1]. O projeto `rot9_example1` [17], uma implementação do exemplo de configuração apresentado no Capítulo 11/página 117 em [5], ilustra o uso do temporizador LPTMR0 para gerar disparos periódicos de amostragem e conversão de um sinal analógico no pino PTB1 (canal 9 do ADC0) (Seção 3.7.1.3.1/página 79 e Seção 10.3.1/página 162 em [1]). O projeto `rot9_aula` [11], por sua vez, aplica o temporizador PIT0 para gerar disparos periódicos para conversões no mesmo canal.

Alocação de Pinos para ADCx

Deve-se alocar pinos físicos aos canais do módulo ADCx para a entrada dos sinais analógicos (tensões). Esses pinos devem ser multiplexados para a função de "entradas do conversor analógico-digital". A tabela na Seção 10.3.1/página 162 em [1] contém todas as funções multiplexáveis para os pinos físicos. Cada pino serve apenas um canal de uma entrada, A ou B. Por exemplo, de acordo com a tabela, o pino PTB3 poderia servir a entrada 13 do ADCx se `PORTB_PCR3_MUX`==0x00. Destaca-se aqui que todas as funções configuráveis para os pinos digitais, através dos *bits* de configuração `PORTx_PCRn_DSE`, `PORTx_PCRn_PFE`, `PORTx_PCRn_PRE` e `PORTx_PCRn_PE` (Seção 11.5.1/página 183 em [1]), não são aplicáveis para os sinais analógicos.

Processamento de Interrupções em ADCx

O conversor ADC opera em conjunto com o controlador NVIC, ou seja, quando o seu *bit* de controle `ADCx_SC1n_AIEN` estiver setado, assim que o *bit* de estado `ADCx_SC1n_COCO` estiver em '1', indicando que o resultado está disponível no registrador de dados `ADCx_Rn` (Seção 28.4.4.2/página 486 em [1]), gera-se uma solicitação de interrupção IRQ15 (Tabela 3-7/página 53 em [1]). Se essa linha linha de requisição estiver devidamente habilitada, o fluxo de controle é automaticamente desviado para a rotina de serviço. Consultando o arquivo `Project_Settings/Startup_Code/kinetis_sysinit.c` gerado pelo IDE *CodeWarrior*, o nome da rotina de serviço declarado para IRQ15 é `ADC0_IRQHandler`. O estado do *bit* `ADCx_SC1n_COCO` em '0' é resetado automaticamente quando se faz um acesso de leitura a `ADCx_Rn` ou um acesso de escrita em `ADCx_SC1n`.

Calibração de ADCx

Para aumentar a acurácia dos valores convertidos, há uma função de auto-calibração disparável por *software*. Os *bits* de estado `ADC0_SC3_CAL` e `ADC0_SC3_CALF` mostram, respectivamente, o progresso e o resultado de um processo da calibração. O fim do processo de calibração é também indicado pelo *bit* `ADCx_SC1n_COCO` (Seção 28.4.6/página 494, em [1]). A função de calibração gera os valores de compensação para os erros de *offset* e de ganho, aplicados automaticamente pelo circuito de conversão. Na seção 28.4.6/página 494 em [1] encontra-se um procedimento de calibração recomendado pelo fabricante. Nos projetos `rot9_example1` [17] e `rot9_aula` [11] foi implementado este procedimento de calibração.

Estimativa do Tempo de Conversão em ADCx

Os tempos gastos numa conversão, que envolve a amostragem e a quantização, são medidos em termos de número de ciclos de `ADCK` e do relógio do barramento (*`bus clock`*). Eles dependem

do modo de amostragem setado nos campos ADC0_CFG1_ADLSMP e ADC0_CFG2_ADLSTS, a velocidade de conversão configurada no campo ADC0_CFG2_ADHSC e da resolução do resultado digital (Seção 28.4.4.5/página 487 em [1]), conforme a expressão dada na Figura 28-62/página 489 em [1]. Os tempos de cada termo são fornecidos pelo fabricante (Figuras 5 a 7). **A estimativa do tempo gasto numa conversão é importante na especificação da frequência máxima a ser aplicada na amostragem dos sinais.**

Vamos ilustrar como se estima o tempo de conversão com base nos dados apresentados nas Figuras 5 a 7. Dada uma configuração do módulo ADC:

- frequência ADCK: 10485760Hz, sendo a frequência do sinal de barramento 20971520Hz. (ADC0_CFG1_ADIV = 0b01 ou ADC0_CFG1_ADICLK = 0b01),
- resolução de 16 *bits* (ADC0_CFG1_MODE = 0b11),
- tempo de amostragem longo, com 12 ADCK ciclos extra, ativado (ADC0_CFG1_ADLSMP = 1; ADC0_CFG2_ADLSTS = 0b01),
- alta velocidade de conversão ativada (ADC0_ADHSC = 1),
- média ativada para 8 amostras por conversão (ADC0_SC3_AVGE = 1; ADV0_SC3_AVGS = 0b01).

Segundo a tabela na Figura 5, **SFCAdder = 3 ciclos de ADCK + 5 ciclos de *bus clock***. Pelas tabelas da Figura 6, AverageNum = 8 e BCT = 25 ciclos de ADCK. Das tabelas da Figura 7, temos LSTAdder = 12 ciclos de ADCK e HSCAdder = 2 ciclos de ADCK. Substituindo esses valores na equação acima, temos

Tempo de conversão total = SFCAdder + AverageNum*(BCT+LSTAdder+HSCAdder)
= (3 ciclos de ADCK + 5 ciclos de bus clock) + 8 *(25+12+2) ciclos de ADCK
= (3 ciclos de ADCK + ~~10~~5/2 ciclos de ADCK) + 8 *(25+12+2) ciclos de ADCK
= ~~325~~ 317,5 ciclos de ADCK
= ~~325~~317.5*(1/10485760) = 0,000030~~994~~279s = 30,~~994~~279us

O fabricante fornece também tempos gastos na amostragem dos sinais em ciclos do sinal de relógio ADCK (Figura 8). O~~u seja, levando em conta o~~ tempo de amostragem mostrado na Figura 8 (18 ciclos de ADCK ~1,7us)~~,~~ e o intervalo de tempo de amostragem e conversão de um~~a~~ valor no registrador ADCx_RA ~~amostra~~ não deve ser menor que ~(30,~~994~~279us~~+1,7us~~) para o módulo ADC com a configuração acima.

$$\text{ConversionTime} = \text{SFCAdder} + \text{AverageNum} \times (\text{BCT} + \text{LSTAdder} + \text{HSCAdder})$$

Figure 28-62. Conversion time equation

Table 28-70. Single or first continuous time adder (SFCAdder)

CFG1[ADLSMP]	CFG2[ADACKEN]	CFG1[ADICLK]	Single or first continuous time adder (SFCAdder)
1	x	0x, 10	3 ADCK cycles + 5 bus clock cycles
1	1	11	3 ADCK cycles + 5 bus clock cycles[1]
1	0	11	5 µs + 3 ADCK cycles + 5 bus clock cycles
0	x	0x, 10	5 ADCK cycles + 5 bus clock cycles
0	1	11	5 ADCK cycles + 5 bus clock cycles[1]
0	0	11	5 µs + 5 ADCK cycles + 5 bus clock cycles

Figura 5: Diferentes tempos para o termo SFCAdder.

Table 28-71. Average number factor (AverageNum)

SC3[AVGE]	SC3[AVGS]	Average number factor (AverageNum)
0	xx	1
1	00	4
1	01	8
1	10	16
1	11	32

Table 28-72. Base conversion time (BCT)

Mode	Base conversion time (BCT)
8b single-ended	17 ADCK cycles
9b differential	27 ADCK cycles
10b single-ended	20 ADCK cycles
11b differential	30 ADCK cycles
12b single-ended	20 ADCK cycles
13b differential	30 ADCK cycles
16b single-ended	25 ADCK cycles
16b differential	34 ADCK cycles

Figura 6: Diferentes quantidade de amostras por resultado digital e diferente tempos de BCT.

Table 28-73. Long sample time adder (LSTAdder)

CFG1[ADLSMP]	CFG2[ADLSTS]	Long sample time adder (LSTAdder)
0	xx	0 ADCK cycles
1	00	20 ADCK cycles
1	01	12 ADCK cycles
1	10	6 ADCK cycles
1	11	2 ADCK cycles

Table 28-74. High-speed conversion time adder (HSCAdder)

CFG2[ADHSC]	High-speed conversion time adder (HSCAdder)
0	0 ADCK cycles
1	2 ADCK cycles

Figura 7: Diferentes tempos de LSTAdder e HDCAdder.

ADC configuration			Sample time (ADCK cycles)	
CFG1[ADLSMP]	CFG2[ADLSTS]	CFG2[ADHSC]	First or Single	Subsequent
0	X	0	6	4
1	00	0	24	
1	01	0	16	
1	10	0	10	
1	11	0	6	
0	X	1	8	6
1	00	1	26	
1	01	1	18	
1	10	1	12	
1	11	1	8	

Figura 8: Diferentes tempos de amostragem.

Módulo LPTMRx

O módulo LPTMR (*Low Power Timer*) é um circuito que pode ser configurado para funcionar como um contador de tempo ou como um contador de pulsos de 16 *bits* em todos os modos de energia, incluindo no modo de baixo vazamento. O módulo tem dois domínios de sinais de relógio. O sinal de relógio de barramento (*bus clock*) para os circuitos de comunicação com o barramento e o sinal `LPTMRx clock` para o contador (Tabela 5-2/página 121 em [1]). A habilitação do *bus clock* é pelo *bit* `SIM_SCGC5_LPTMR` (Seção 12.2.8/página 206 em [1]), enquanto o sinal `LPTMRx clock` é selecionável dentre as 4 fontes através dos *bits* de configuração `LPTMRx_PSR_PCS`: sinal de referência interno `MCGIRCLK`, `LPO` (*`Low Power Oscillator`*) 1kHz, `ERCLK32K`, e sinal de referência externo `OSCERCLK` (Seção 3.8.3.3/página 90 em [1]). É possível dividir a frequência do sinal `LPTMRx clock` resetando o *bit* de configuração `LPTMRx_PSR_PBYP` em '0' e setar o divisor no campo `LPTMRx_PSR_PRESCALE` (Seção 33.3.2/página 591 em [1]) para chegarmos a uma frequência de operação do contador f_{LPTMR}. Somente após a configuração e habilitação dos sinais de relógio dos dois domínios, deve-se habilitar o módulo LPTMR setando o *bit* de controle `LPTMRx_CSR_TEN`

(Seção 33.3.1/página 589 em [1]) em '1'.

No modo de operação como **contador de tempo** (o *bit* de configuração `LPTMRx_CSR_TMS` resetado em '0'), o contador `LPTMRx_CNR` (Seção 33.3.4/página 592 em [1]) conta até o valor configurado no registrador de dados `LPTMRx_CMR` (Seção 33.3.3/página 592 em [1]). Quando `LPTMRx_CNR` se iguala a `LPTMRx_CMR`, o *bit* de estado `LPTMRx_CSR_TCF` é setado automaticamente em '1' (Seção 33.3.1/página 589 em [1]). E o valor do contador `LPTMRx_CNR` é resetado em 0, se o *bit* de configuração `LPTMRx_CSR_TFC` (Seção 33.3.1/página 589 em [1]) estiver em '0', senão `LPTMRx_CNR` manterá a sua contagem até estouro (0xFFFF) para então resetar em 0.

No modo de operação como **contador de pulsos** (o *bit* de configuração `LPTMRx_CSR_TMS` setado em '1'), os sinais que pulsam o contador LPTMRx_CNR são os sinais externos configuráveis pelos *bits* de configuração `LPTMRx_CSR_TPS`. Na Seção 3.8.3.2/página 89 em [1] são relacionados os códigos binários configuráveis nesses *bits* com as funções multiplexáveis dos pinos: 0b00 aos pinos multiplexáveis para a função `CMP0_OUT`, 0b01 aos `LPTMR_ALT1`, 0b10 aos `LPTMR_ALT2`, e 0b11 aos `LPTMR_ALT3`. É ainda configurável a polaridade dos pulsos que efetivamente incrementam `LPTMRx_CNR`: ativo-alto ou na borda de subida (se o *bit* de configuração `LPTMRx_CSR_TPP` estiver resetado em '0') e ativo-baixo ou na borda de descida (se o *bit* de configuração `LPTMRx_CSR_TPP` estiver setado em '1'). Operando como contador de pulsos, os *bits* de configuração `LPTMRx_PSR_PBYP` e `LPTMRx_PSR_PRESCALE` assumem a função de "registradores configuradores" de filtros de transientes (glitches) antes de iniciar contagem. Vale destacar ainda que a frequência desses sinais externos não pode ser mais alta do que f_{LPTMR}=24MHz especificada na folha de dados técnicos [7].

Configuração de um Período em LPTMRx

Quando LPTMRx é configurado no modo de contador de tempo, o período (contagem máxima) de `LPTMR_CNR` depende, além da frequência da fonte f_{LPTMR} (`LPTMR clock`) selecionada, dos valores setados em `LPTMRx_CMR` (valor de referência para contagem máxima, Seção 33.3.3/página 592 em [1]) e em `LPTMRx_PSR_PRESCALE` (divisor *prescaler*, Seção 33.3.2/página 590 em [1]). Para LPTMRx temos ainda que diferenciar os seguintes casos:

1) `LPTMRx_PSR_PBYP==0 e LPTMRx_CSR_TFC == 0`

$$Periodo = LPTMRx\_CMR \, x \frac{2^{LPTMRx\_PSR\_PRESCALE+1}}{f_{LPTMR}} \quad (3.a)$$

2) `LPTMRx_PSR_PBYP==1 e LPTMRx_CSR_TFC == 0`

$$Periodo = LPTMRx\_CMR \, x \frac{1}{f_{LPTMR}} \quad (3.b)$$

3) `LPTMRx_PSR_PBYP==0 e LPTMRx_CSR_TFC == 1`

$$Periodo = 65535 \, x \frac{2^{LPTMRx\_PSR\_PRESCALE+1}}{f_{LPTMR}} \quad (3.c)$$

4) `LPTMRx_PSR_PBYP==1 e LPTMRx_CSR_TFC == 1`

$$Periodo = 65535 \frac{1}{f_{LPTMR}} \qquad (3.d)$$

Alocação de Pinos para LPTMRx

Para captura dos sinais externos no modo de operação de contador, é necessário alocar um pino físico multiplexado para a função configurada em `LPTMRx_CSR_TPS`. Para isso, podemos fazer uso da tabela na Seção 10.3.1/página 162 em [1] que nos dá as setuintes alternativas: `CMP0_OUT (PTC0, PTC5, PTE0)`, `LPTMR0_ALT1 (PTA19)` e `LPTMR0_ALT2 (PTC5)`.

Processamento de Interrupções em LPTMRx

Quando o *bit* de estado `LPTMRx_CSR_TCF` e o *bit* de controle `LPTMRx_CSR_TIE` estiverem setados em '1', é gerado um evento de interrupção IRQ 28 para o controlador NVIC (Tabela 3-7/página 54 em [1]). Se essa linha linha de requisição estiver devidamente habilitada, o fluxo de controle é automaticamente desviado para a rotina de serviço. Consultando o arquivo `Project_Settings/Startup_Code/kinetis_sysinit.c` gerado pelo IDE *CodeWarrior*, o nome da rotina de serviço declarado para IRQ28 é `LPTimer_IRQHandler`. O estado do *bit* `LPTMRx_CSR_TCF` pode ser resetado em '0' se fizermos um acesso de escrita no *bit* (*write-1-to-clear*) ou desabilitarmos o módulo (Seção 33.4.7/página 596 em [1]).

Disparos de LPTMRx para outros Módulos

Os eventos detectados pelo *bit* de estado `LPTMRx_CSR_TCF` servem de fontes de disparos para outros módulos sem nenhuma interferência de *software*. Se `LPTMRx_CMR` estiver configurado com um valor diferente de 0 e `LPTMRx_CSR_TCF` estiver setado em '1', é gerado sempre um disparo em cada estoutro até o próximo incremento no contador `LPTMRx_CNR` (Seção 33.4.6/página 596 em [1]).

Reuso de Estruturas Pré-Definidas

Em vista da quantidade de parâmetros (campos) distribuídos em 5 registradores para configurar a operação de um módulo ADCx, podemos definir em C um novo tipo de dado que proporcione uma configuração "mais centralizada" de ADCx. No projeto `rot7_aula` [12] é definido o tipo de dado `struct _UART0Configuration_tag` cujos membros são os campos dos registraddores de configuração/controle especificados pelo fabricante. A seção 11.2.1/página 117 em [5] sugere, no entanto, uma alternativa que permite reusar as estruturas e as macros já disponíveis no IDE *CodeWarrior*. Em `Project_Headers/MKL25Z.h` é definida para cada módulo uma nova estrutura que acessa os endereços físicos de memória por meio de nomes dos registradores usados nos manuais. Para o módulo ADCx, é definidio o tipo de dado `struct ADC_MemMap` e o respectivo ponteiro `ADC_MemMapPtr` cujos membros são registradores ao invés dos campos dos registradores. Os nomes dos membros são os msmos nomes dos registradores adotados pelo fabricante (Seção 28.3/página 461 em [1]), exceto os dois registradores de controle de entrada `ADC0_SC1A` e `ADC0_SC1B`, e os dois registradores de dados de saída, `ADC0_RA` e `ADC0_RB`. No lugar de quatro membros, foram declarados dois vetores de 2 elementos, `SC1[2]` e `R[2]`, tal que `SC1[0]`, `SC1[1]`, `R[0]` e `R[1]` correspondem, respectivamente, a `ADC0_SC1A`, `ADC0_SC1B`, `ADC0_RA` e `ADC0_RB`:

```
typedef struct ADC_MemMap {
  uint32_t SC1[2];
  uint32_t CFG1;
  uint32_t CFG2;
  uint32_t R[2];
  uint32_t CV1;
  uint32_t CV2;
  uint32_t SC2;
  uint32_t SC3;
  uint32_t OFS, PG, MG;
  uint32_t CLPD, CLPS;
  uint32_t CLP4, CLP3, CLP2, CLP1, CLP0;
  uint8_t RESERVED_0[4];
  uint32_t CLMD, CLMS;
  uint32_t CLM4, CLM3, CLM2, CLM1, CLM0;
} volatile *ADC_MemMapPtr;
```

Para setar o conteúdo de cada campo do registrador, fica a cargo do desenvolvedor aplicar operações *bit*-a-*bit*. Por exemplo, no projeto `rot9_example1` [17] foi declarada em `main.c` a variável `Master_Adc_Config` do tipo `struct ADC_MemMap`. Para inicializar os valores dos registradores `ADC0_SC1A`, `ADC0_SC1B`, `ADC0_CFG1`, `ADC0_CFG2`, `ADC0_CV1`, `ADC0_CV2`, `ADC0_SC2` e `ADC0_SC3`, compôs-se com o operador lógico *bit*-a-*bit* OU (|) os valores especificados separadamente aos campos de um registrador antes de atribuí-los ao registrador. Os valores nos campos dos registradores são, por sua vez, definidos com uso das macros `ADC_*` em `Project_Headers/MKL25Z.h`. Por exemplo, as macros usadas para definir os valores dos campos do registrador CFG1

`(0<<7)|ADC_CFG1_ADIV(0b10)|ADC_CFG1_ADLSMP_MASK| ADC_CFG1_MODE(0b11)| ADC_CFG1_ADICLK(0b00)`

equivalem a

`(0<<7)|(0b10<<5)|(1<<4)|(0b11<<2)|(0b00)`

que corrresponde à seguinte palavra a ser setada no registrador CFG1:

01011100

Fazendo atribuições análogas a outros registradores, podemos inicilizar todos os registradores de configuração do módulo ADC com o seguinte comando:

```
struct ADC_MemMap Master_Adc_Config = {
          .SC1[0]=(0<<6)         //AIEN
           | (0<<5)              //DIFF
           | ADC_SC1_ADCH(31),
          .SC1[1]=(0<<6)         //AIEN
           | (0<<5)              //DIFF
           | ADC_SC1_ADCH(31),
          .CFG1=(0<<7)           //ADLPC
           | ADC_CFG1_ADIV(0b10)
           | ADC_CFG1_ADLSMP_MASK
```

```
              | ADC_CFG1_MODE(0b11)
              | ADC_CFG1_ADICLK(0b00),
             .CFG2=(0<<4)          //MUXSEL
              | (0<<3)             //ADACKEN
              | ADC_CFG2_ADHSC_MASK
              | ADC_CFG2_ADLSTS(0b00),
             .CV1=0x1234u,
             .CV2=0x5678u,
             .SC2=ADC_SC2_ADTRG_MASK
              | (0<<5)               //ACFE
              | ADC_SC2_ACFGT_MASK
              | ADC_SC2_ACREN_MASK
              | (0<<2)               //DMAEN
              | ADC_SC2_REFSEL(0b00),
             .SC3= (0<<7)            // CAL
              | (0<<3)               // ADCO
              | ADC_SC3_AVGE_MASK
              | ADC_SC3_AVGS(0b11),
};
```

Essas operações *bit*-a-*bit* na inicialização dos valores dos registradores permitem que os seus resultados sejam atribuídos diretamente aos endereços da memória onde estão mapeados os registradores físicos do módulo ADC. No lugar de operações a nível de *bits* em `UART0_configure`, comandos de atribuição dos valores dos registradores setados em `Master_Adc_Config` ao bloco de memória a partir do endereço (`(ADC_MemMapPtr)0x4003B000u`) (Seção 28.3/página 461 em [1]), são suficientes para transferir todos os dados inicializados numa estrutura temporária (Master_Adc_Config) aos endereços onde estão mapeados os registradores do módulo ADC, como demonstra a seguinte função:

```
void ADC_Config_Alt (ADC_MemMapPtr end, ADC_MemMapPtr dados) {
        end->SC1[0] = dados->SC1[0];
        end->SC1[1] = dados->SC1[1];
        end->CFG1 = dados->CFG1;
        end->CFG2 = dados->CFG2;
        end->CV1 = dados->CV1;
        end->CV2 = dados->CV2;
        end->SC2 = dados->SC2;
        end->SC3 = dados->SC3;
        return;
}
```

Usando a função, pode-se fazer a transferência dos dados com a chamada

```
ADC_Config_Alt ((ADC_MemMapPtr)0x4003B000u), &Master_Adc_Config);
```

Tendo ainda uma macro definida para o endereço inicial do bloco da memória em `Project_Headers/MKL25Z.h`

```
#define   ADC0_BASE_PTR                ((ADC_MemMapPtr)0x4003B000u)
```

podemos aumentar a legibilidade do código usando esta macro em `main.c` na chamada da função para transferência do conteúdo de `Master_Adc_Config` aos registradores de ADC:

```
ADC_Config_Alt (ADC0_BASE_PTR, &Master_Adc_Config).
```

Definição de Macros das Macros

Para aumentar a legibilidade dos nossos códigos, podemos criar novas macros que refletem melhor o significado de cada código binário, como as definidas em `ADC.h` no projeto `rot9_example1`[17]:

```
#define ADLSTS_20 0b00
#define DMAEN_ENABLED ADC_SC2_DMAEN_MASK
```

Muitas novas macros em `ADC.h` são praticamente uma espécie de redenominação das macros existentes. Na fase de pré-processamento C elas são substituídas recursivamente até os comandos compiláveis de C. Com uso das novas macros, o código de inicialização da variável `Master_Adc_Config` tende a ser mais auto_explicativa:

```
struct ADC_MemMap Master_Adc_Config = {
          .SC1[0]=AIEN_OFF
           | DIFF_SINGLE
           | ADC_SC1_ADCH(31),
          .SC1[1]=AIEN_OFF
           | DIFF_SINGLE
           | ADC_SC1_ADCH(31),
          .CFG1=0x00
           | ADC_CFG1_ADIV(ADIV_4)
           | ADLSMP_LONG
           | ADC_CFG1_MODE(MODE_16)
           | ADC_CFG1_ADICLK(ADICLK_BUS),
          .CFG2=MUXSEL_ADCA
           | ADACKEN_DISABLED
           | ADHSC_HISPEED
           | ADC_CFG2_ADLSTS(ADLSTS_20),
          .CV1=0x1234u,
          .CV2=0x5678u,
          .SC2=ADTRG_HW
           | ACFE_DISABLED
           | ACFGT_GREATER
           | ACREN_ENABLED
           | DMAEN_DISABLED
           | ADC_SC2_REFSEL(REFSEL_EXT),
          .SC3=CAL_OFF
           | ADCO_SINGLE
           | AVGE_ENABLED
           | ADC_SC3_AVGS(AVGS_32),
};
```

Filtragem Exponencial para Suavização de Dados Digitalizados

A filtragem exponencial é um método de suavização de séries temporais digitais que combina uma média ponderada do valor medido x_i no instante i e dos resultados anteriores x_{i-1} … x_0 para produzir um resultado y_i suavizado no instante i

$$y_i = \alpha x_i + (1-\alpha) y_{i-1} = \alpha x_i + (1-\alpha)\alpha x_{i-1} + \ldots + (1-\alpha)^{i-1}\alpha x_1 + (1-\alpha)^i \alpha x_0 \quad (4)$$

O peso α dado a cada valor anterior decai ao longo do tempo, permitindo que a resposta do filtro a mudanças recentes seja mais rápida do que a resposta a mudanças antigas. Isso resulta em suavização dos dados, minimizando o ruído e destacando tendências subjacentes.

No projeto `rot9_example1`[17] é aplicada uma filtragem exponencial com α=0.5, pois os comandos em `ADC0_IRQHandler` (`ISR.c`):

```
exponentially_filtered_result += result0A;
exponentially_filtered_result /= 2;
```

somam os resultados anteriores acumulados na variável estática `exponentially_filtered_result` com o valor amostrado `result0A` e divide a soma por 2:

$$exponentially\_filtered\_result = \frac{exponentially\_filtered\_result + result0A}{2}$$

$$0.5 \times result0A + (1-0.5) \times exponentially\_filtered\_result$$

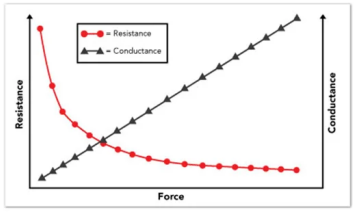
Interpretação dos Valores Amostrados

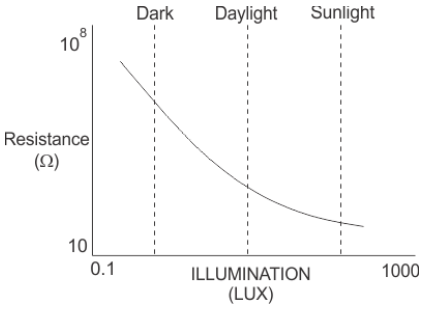
É importante ressaltar que o resultado de uma conversão, acessível pelo registrador `ADCx_Rn`, é uma representação binária em N *bits* do valor de tensão amostrado. Quando devidamente calibrado, podemos considerar que o valor de tensão no intervalo `[VREFL,VREFH]` seja diretamente proporcional ao código binário entre 0 a 2^N-1, de forma que o valor da amostra de tensão, `Tensão_amostrada`, possa ser obtido a partir do código binário armazenado em `ADCx_Rn` por uma regra de três simples usando **operações em ponto flutuante**:

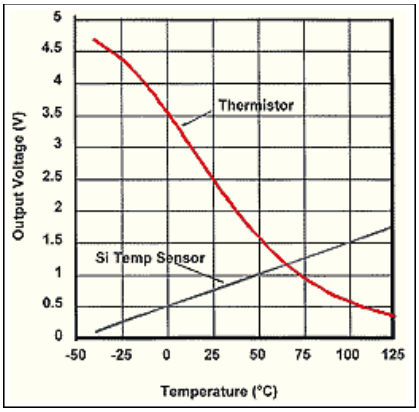
(`Tensão amostrada-VREFL`) → [`ADCx_Rn`]

(`VREFH – VREFL`) → 2^N-1

Caso sejam de interesse os **valores em grandezas físicas originais** dos sinais amostrados, **é necessário pós-processar as amostras de tensão** recuperadas, `Tensão_amostrada`, convertendo-as para valores em grandezas físicas originais. Para isso, precisamos recorrer aos *datasheets* dos fabricantes dos sensores, como ilustra a Figura 9.







Sensor de força [8] Sensor de luz [9] Sensor de temperatura [10]

Figura 9: Relações entre as grandezas físicas e elétricas geradas por diferentes sensores.

Para o sensor de temperatura AN3031 integrado em KL25Z, disponível no canal `0b11010` do módulo ADC, é especificada pelo fabricante a relação entre a temperatura medida e a tensão gerada através da expressão (Seção 2.1/página 3 em [3]):

$$Temperatura = 25 - \left(\frac{V_{Temperatura} - V_{25}}{m}\right) \qquad (5)\text{, onde}$$

$$m = 1.646\ \text{V/}^{o}\text{C},\ \text{se}\ V_{Temperatura} \geqslant V_{25}\ (\ Temperatura \leqslant 25^{o}\text{C})$$

$$m = 1.769\ \text{V/}^{o}\text{C},\ \text{se}\ V_{Temperatura} < V_{25}\ (\ Temperatura > 25^{o}\text{C})$$

$$V_{25} \sim 0.703125\text{V}$$

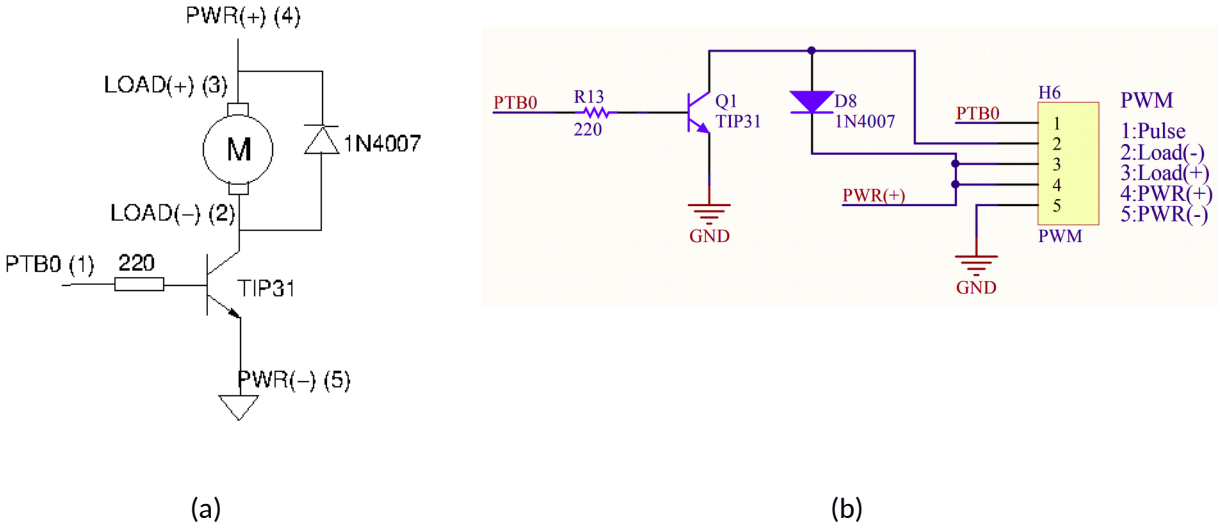
Ou seja, a partir de um valor de tensão amostrado por um ADC podemos estimar a temperatura que resultou aquele valor de tensão.

PWM em Controle de Transferência de Potência

O projeto `controle_cooler` gera sinais que atuam sobre um *cooler*, variando a sua velocidade de rotação de acordo com o ângulo de giro do eixo do potenciômetro. A velocidade de rotação do *cooler* é diretamente relacionada com a potência aplicada nele. Um sinal de largura de pulso modulável como PWM é uma forma simples e elegante para controlar a potência de uma carga [4]. Porém, a corrente suprida por microcontroladores é muito baixa para a maioria das cargas.

Uma solução muito utilizada é injetar o sinal PWM na base de um transistor de potência (Figura 10(a)) que opera como uma chave eletrônica de um circuito de alimentação de corrente maior. Ele fecha o circuito no estado de saturação (pulso no nível 1) e abre o circuito no estado de corte (pulso no nível 0), de maneira que a carga passe a receber uma porcentagem da potência máxima que ela receberia se a alimentação fosse contínua. Na Figura 10(a), o diodo 1N4007 paralelo com o motor serve para proteger o motor de tensões reversas (que aparecem quando interrompemos subitamente a corrente através de uma indutância) e a resistência R13 é para limitar a corrente na base do transistor. Está integrada no *shield* FEEC871 uma parte desse circuito, como mostra um recorte do esquemático do *shield* [6] na Figura 10(b). Se ligarmos uma carga, como um *cooler*, entre os pinos

2 e 3 do *header* H6 e uma fonte de alimentação de 12V DC entre os pinos 4 e 5 do mesmo *header*, podemos controlar a velocidade do *cooler* por um sinal PWM gerado no pino PTB0 do KL25Z.



(a) (b)

Figura 10: Controle da velocidade de um motor por PWM.

Os pinos marcados com as letras A, B, C, D e E na figura 11, correspondem aos pinos 5, 4, 3, 2 e 1 na figura 10(b).

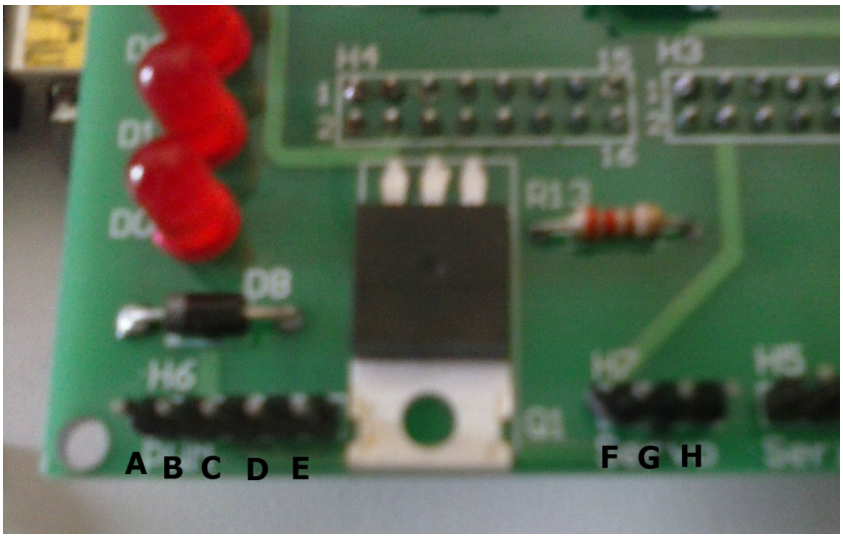


Figura 11: Pinagens em *Shield* FEEC871.

EXPERIMENTO

Neste experimento vamos desenvolver o projeto `controle_cooler` que controla a velocidade do *cooler* por meio de um potenciômetro conectado no pino `PTB1` do *header* H7 do *shield* FEEC871 (Figura 12), multiplexável ao canal `0b01001` do módulo ADC. O valor do potenciômetro é amostrado periodicamente numa resolução de 16 *bits* para atualizar a largura de pulso do sinal gerada no canal TPM1_CH0 e transferido ao pino PTB0 (Figura 10). O ciclo de

trabalho desse canal, no formato “DUTY: YY.YY”, é mostrado no meio da segunda linha do visor do LCD. Além disso, é monitorada na mesma periodicidade a temperatura do microcontrolador pelo sensor de temperatura AN3031 [3] integrado em KL25Z. O valor amostrado é convertido para a unidade em graus Celsius e mostrado no formato “TEMP: XX.XX C” no meio da primeira linha do LCD. As conversões periódicas do sinal de potenciômetro são iniciadas por *hardware*, pelos eventos de estouro do módulo TPM2. Em seguida, por *software*, é disparada a conversão do sinal amostrado pelo sensor de temperatura. O *led* RGB acenderá em vermelho (conectado em TPM2_CH0) se a temperatura do microcontrolador estiver superior a 25ºC e em verde (conectado em TPM2_CH1) se estiver abaixo. A intensidade do vermelho aumenta de 25º (apagada) a 50º (máxima) e a de verde aumenta de 25º (apagada) a 0º (máxima).

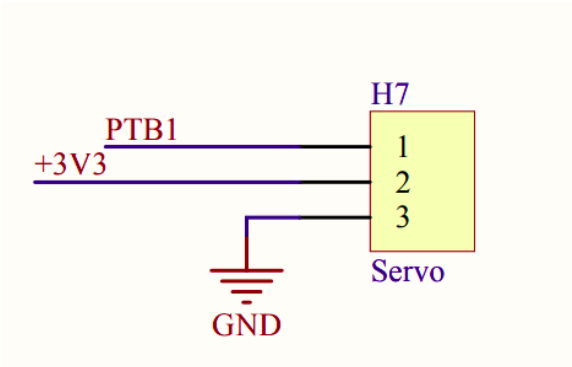


Figura 12: Pinagem do *header* H7 do shield FEEC871.

Na Figura 13 são mostrados os 3 estados do sistema: `AMOSTRA_VOLT`, `AMOSTRA_TEMP` e `ATUALIZAÇÃO`. Por trás desses estados, o temporizador TPM2 fica ativado e faz disparos periódicos de aproximadamente 645*tempo de conversão do módulo de ADC (detalhado no item 2.7). Esses disparos fazem que seja amostrado e convertido o sinal do potenciômetro no estado `AMOSTRA_VOLT`. Após a coleta do valor amostrado do potenciômetro, passa-se para o estado `AMOSTRA_TEMP` em que o sinal analógico do sensor de temperatura AN3031 é amostrado. Após a amostragem dos dois sinais analógicos, muda-se para o estado de `ATUALIZAÇÃO` em que o estado do *led* vermelho, a temperatura e o valor do potenciâometro amostrados são atualizados no LCD e a velocidade do *cooler* atualizado. Ao fim das atualizações, o sistema volta ao estado `AMOSTRA_VOLT` e fica aguardando o disparo de TPM2 para fazer uma nova amostragem do sinal de potenciômetro.

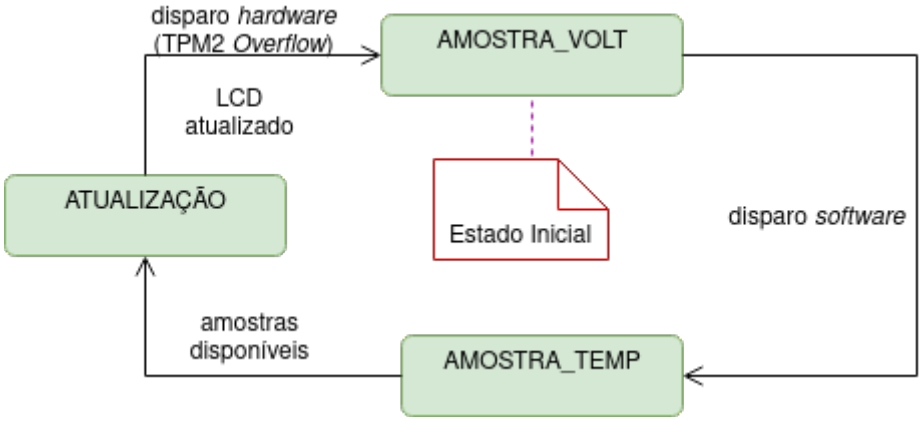


Figura 13: Diagrama de máquina de estados do `controle_cooler` (editado em [14]).

Na Figura 14 é apresentado um diagrama de componentes proposto para este projeto.

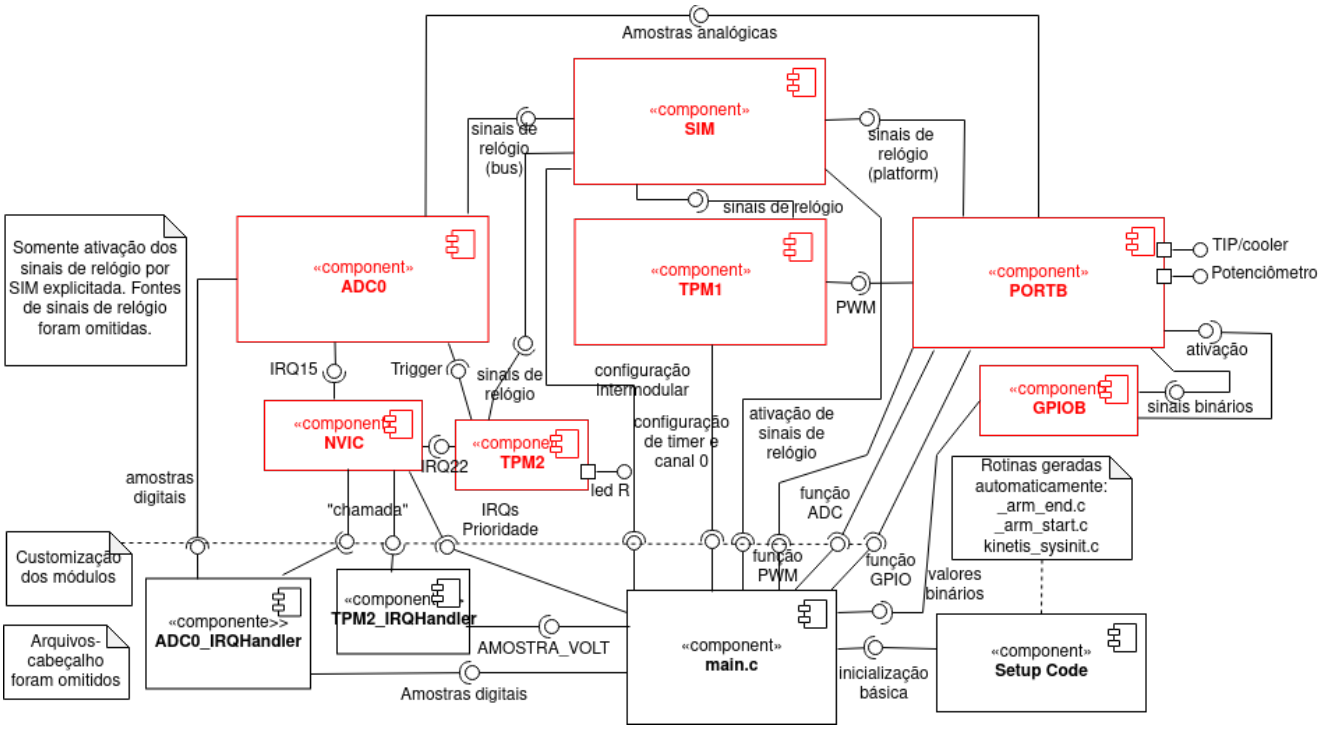


Figura 14: Diagrama de componentes do `controle_cooler` (por legibilidade, foram omitidos os módulos PORTC e GPIOC que controlam LCD, editado em [14]).

O **modo de operação do módulo ADC** especificado é

> **disparo por *hardware* usando o evento TPM2 *Overflow*** (amostragem da tensão do potenciômetro) e por *software* (amostragem do sensor de temperatura)
>
> **frequência ADCK**: 1310720Hz, sendo a frequência do sinal de barramento 20971520Hz e a frequência da fonte de ADICK a metade dessa frequência.
>
> **resolução de 16 *bits***
>
> tempo de amostragem **curto (sem circuito RC na entrada)**
>
> velocidade **normal** de conversão habilitada
>
> média habilitada para 16 **amostras por conversão**

Para facilitar o mapeamento dos valores amostrados do potenciômetro em "ciclos de trabalho", TPM2_MOD é 65535 (0xFFFF) em 16 *bits*.

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 11.2/página 117 em [5] é apresentado um exemplo de configuração do módulo ADC para amostrar o sinal analógico de um potenciômetro integrado ao *kit* dos autores. Este potenciômetro está conectado ao canal 4 da entrada B de ADC0. Os eventos de estouro do temporizador LPTMR0 são usados como disparos periódicos de conversões únicas, sem intervenções de *software*. Escolheu-se como a fonte de sinais de relógio do contador de LPTMR0 o sinal externo RTC_CLKIN (Seção 5.7.4/página 124 em [1]) cuja entrada é o pino PTC1 (Seção 10.3.1/página 162 em [1]). Para adequar aos *kits* de desenvolvimento do LE30, foram feitos três ajustes na implementação do exemplo dado em `rot9_example1` [17]:

- Como em FRDMKL25Z não há um potenciômetro integrado, foi usado um potenciômetro externo conectado no pino PTB1, multiplexável ao canal 9 de ADC0 (Seção 10.3.1/página 162 em [1]),
- Sendo PTC1 ocupado por um pino de LCD/*Latch*, usou-se LPO como a fonte de sinais de relógio do contador do LPTMR0 (Seção 5.7.4/página 124 em [1]), e
- Não dispondo de um *led* laranja, usou-se um *led* verde conectado no pino PTB19.

Diferentes dos outros projetos que analisamos nos roteiros anteriores, os autores apresentaram um esboço da ideia de configuração neste projeto. Foi necessário preencher as lacunas, como a implementação das funções `ADC_Config_Alt`, `ADC_Cal` e `GPIO_initLedG`, em `rot9_example1`. Vale destacar que as instruções de configuração do módulo LPTMR foram alteradas ligeiramente para que elas sejam operações por *bits*. Execute o projeto no modo *Debug* do IDE *CodeWarrior*.

1.1 **Módulo ADC**: São configuráveis a frequência do sinal de relógio ADCK, a resolução, a velocidade da sequência de conversão, fluxo de conversão, e a quantidade de amostras por conversão.

1.1.1 Qual é a frequência do sinal ADCK configurada? Justifique com base na fonte de sinais de relógio configurada para ADC0 e nos valores configurados em `ADC0_CFG1_ADIV` e `ADC0_CFG1_ADICLK`.

1.1.2 Qual é a resolução configurada, em quantidade de *bits*? Justifique com base no valor setado em `ADC0_CFG1_MODE`.

1.1.3 Qual é a velocidade configurada para uma sequência de conversão? Justifique com base no valor setado em `ADC0_CFG2_ADHSC`.

1.1.4 Qual é o fluxo de conversão configurado? Justifique com base no valor setado em `ADC0_SC3_ADCO` e `ADC0_SC3_AVGE`.

1.1.5 Qual é a quantidade de amostras configurada para cada conversão? Justifique com base no valor setado em `ADC0_SC3_AVGE` e `ADC0_SC3_AVGS`.

1.1.6 Qual canal e qual entrada do ADC0 são habilitados para amostragem? Justifique com base nos valores setados em `ADCx_CFG2_MUXSEL` e `ADCx_SC1A_ADCH` depois da chamada da função `ADC_selecionaCanal`. Note que a instrução de seleção do canal SE4B não foi incluída na Seção11.2/página 117 em [5]. Veja a instrução de seleção do canal SE9 em `rot9_example1`.

1.2 **Filtragem de Dados Analógicos:** Qual é o tempo de amostragem configurado? Justifique com base nos valores setados em `ADC0_CFG1_ADLSMP` e `ADC0_CFG1_ADLSTS`. Supondo que a impedância de entrada no pino PTB1 seja ~~muito~~ baixa, dentro da faixa de impedâncias de entrada recomendada, qual configuração de tempo de amostragem você selecionaria? Dica: Leia a recomendação do fabricante sobre a configuração de tempo de amostragem na Seção 28.3.2, página 466, em [1].

1.3 **Funções de Comparação em ADC**: É possível remover os resultados de conversão que não sejam de interesse ao configurarmos os intervalos de valores de interesse.

1.3.1 Qual é a função de comparação configurada para o resultado de uma conversão? Justifique com base nos valores setados no registrador `ADC0_SC2`, `ADC0_CV1` e `ADC0_CV2`.

1.3.2 A função de comparação não está habilitada. Habilite a função, regere o executável e execute o programa com um ponto de parada em `ADC0_IRQHandler`. Analise os diferentes resultados de conversão em `ADC0_RA` ao dar um giro completo o eixo do potenciômetro. Dica: Para verificar se o valor amostrado é convertido e transferido para ADC0_RA, pode-se setar um ponto de parada na linha result0A=ADC0_RA da rotina de serviço ADC0_IRQHandler.

1.4 **Configuração de Disparos de Conversão por *Hardware***: Identifique em `main.c` e na função `ADC_PTB1_config_basica` (ADC.c) os blocos de instruções responsáveis pela configuração de conversões únicas disparadas periodicamente pelos eventos de estouro do LPTMR0.

1.5 **Alocação de pinos para ADCx:** O pino PTB1 é usado para entrada do sinal analógico ao canal 9 de ADC0. Quais configurações foram feitas para habilitar este pino para esta função? Para filtrar ruídos nos sinais, você setaria em '1' o *bit* `PORTB_PCR1_PFE` no lugar de usar um circuito externo RC? Justifique. Dica: Leia a função do *bit* PORTx_PCRn_PFE na Seção 11.5.1, página 184, em [1].

1.6 **Processamento de Interrupções em ADCx:** Compare as instruções relacionadas com a configuração do processamento ~~do fim~~ de uma conversão por interrupção na Seção11.2/página 117 em [5] e em `rot9_example1`. Quais instruções ~~faltam~~ desencessárias usadas no exemplo do Manual para que seja gerado um evento de interrupção quando há um resultado válido em `ADC0_RA`? Dica: Analise a configuração e o processamento das interrupções geradas pelos eventos do fim de uma conversão no exemplo de Manual e no rot9_example1.

1.7 **Calibração de ADCx**: A calibração requer um tempo de processamento. A sua conclusão pode ser detectada por *polling* ou por interrupção. Qual estratégia foi selecionada na implementação da função `ADC_Cal`?

1.8 **Filtragem Exponencial para Suavização de Dados Digitalizados**: Destaque as instruções que implementam essa filtragem exponencial dos resultados de conversões.

1.9 **Reuso de Estruturas Pré-Definidas**: Como são setados os valores iniciais nos membros da variável `Master_Adc_Config`? Como são copiados os valores desses membros nos registradores do módulo ADC em `ADC_Config_Alt`?

1.10 **Módulo LPTMR**: Qual é o modo de operação configurado para LPTMR0? Justifique com base no valor setado em `LPTMR0_CSR_TMS`.

1.11 **Configuração de um Período em LPTMR:** Qual é o período configurado em LPTMR0? Justifique com base na fonte do sinais de relógio selecionada, nos valores setados em no valor setado em `LPTMR0_PSR_PRESCALE`, `LPTMR0_PSR_PBYP`, `LPTMR0_PSR_PCS` e `LPTMR0_CMR`.

1.12 **Alocação de pinos para LPTMRx**: Analise a função `LPTMR_config_especifica` e as instruções equivalentes no exemplo da Seção11.2/página 117 em [5]. É alocado um pino físico a LPTMR0? Qual?

1.13 **Processamento de interrupções em LPTMRx/Disparos para outros Módulos**: Eventos de interrupções podem ser usados como disparos de alguma tarefa em outros módulos através das comunicações intermodulares.

1.13.1 Destaque as instruções em `rot9_example1` que habilitam os eventos de estouro de LPTMR0 para dispararem conversões em ADC0.

1.13.2 Para gerar disparos periódicos ao módulo ADC0, é necessário habilitar o mecanismo de interrupção de LPTMRx? Justifique com base na comparação das instruções de configuração das interrupções entre o exemplo da Seção11.2/página 117 em [5] e o projeto `rot9_example1`.

1.13.3 Quando se iniciam os disparos periódicos de LPTMR0 para ADC0 em `rot9_example1`?

2 **Estimativa de Tempo de Conversão:** O projeto `rot9_aula` [11] demonstra configurações do módulo ADC para amostrar periodicamente dois sinais analógicos, um do potenciômetro (canal 9) conectado no *header* H7 (Figura 12) e outro do sensor AN3031 (canal 26). O módulo PIT gera eventos de estouro que são configurados para iniciar a conversão de um sinal amostrado do potenciômetro e em seguida, do sensor AN3031. Os códigos binários dos resultados, em 8 *bits*, são mostrados nos 8 *leds* vermelhos. O tempo de amostragem e conversão do módulo ADC e o período configurado para PIT são espelhados como pulsos nos pinos 3 e 2 do *header* H5 do *shield* FEEC871, respectivamente. Execute-o e capture os sinais com o analisador lógico. Formas de onda similares às mostradas na Figura 15 devem ser renderizadas na tela

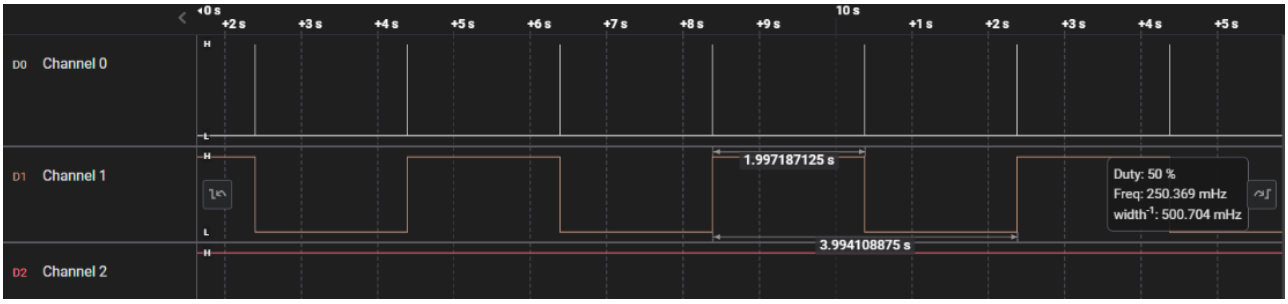


Figura 15: Formas de onda no pino 3 (*Channel* 0) e 2 (*Channel* 1).

Se passarmos da escala de segundos para milisegundos, veremos que os "palitos" brancos no *Channel* 0 são de fato dois pulsos de largura de centenas de micro-segundos que correpondem aos tempos de amostragem e de conversão de uma amostra, primeiro do canal 9 e depois do canal 26.

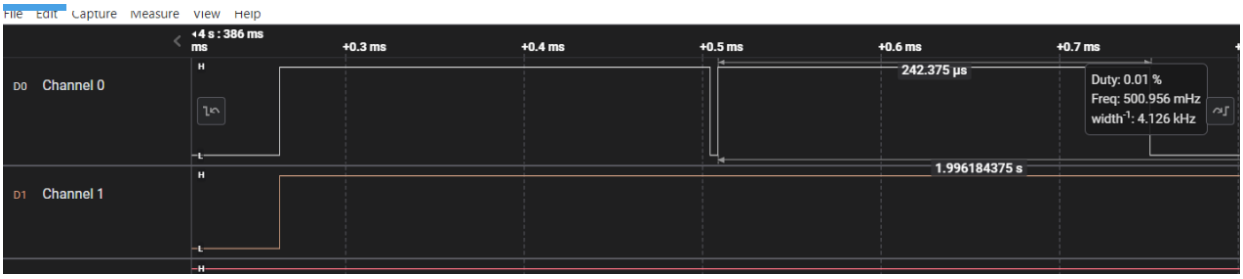


Figura 16: Tempo de amostragem e de conversão de duas amostras no ADC.

2.1 Como é iniciada uma conversão no canal 9? Por *hardware* ou por *software*? E no canal 26? Justifique.

2.2 Estime o período programado para o PIT. É condizente com os valores medidos pelo analisador?

2.3 Estime o tempo de conversão para a configuração do ADC0 programada em `rot9_aula`. É condizente com os valores medidos pelo analisador?

2.4 Substitua o argumento de entrada da função `PIT_init` de 10485760 por 20971520. Analise se as alterações nas formas de onda são condizentes com as esperadas.

2.5 Qual é o menor período configurável para o PIT para o qual não ocorram sobreposições das conversões?

2.6 Estime o tempo de conversão para a configuração do ADC0 programada em `rot9_example1` [17], descomentando as linhas de instruções segundo as orientações dadas no código ~~Reconfigure o módulo ADC0 de `rot9_aula` com esses valores~~. Gere o executável, execute-o e analise os tempos medidos pelo analisador. São condizentes com a configuração programada?

2.7 Estime o tempo de conversão especificado para ~~a configuração d~~o ADC0 no projeto `controle_cooler`. Reconfigure o módulo ADC0 de `rot9_aula` com esses valores de configuração, inclusive o valor de `SIM_CLOCKDIV_OUTDIV4`, para verificar o tempo estimado com um analisador lógico. São condizentes com a configuração especificada?

3 **PWM em Controle de Transferência de Potência.** Conecte o *cooler* nos pinos 2 e 3 do *header* H6 do *shield* FEEC871 e a fonte de alimentação nos pinos 4 e 5, conforme ilustra a Figura 10. Ligue a fonte numa tomada da bancada. Conecte o pino 1 do *header* H6 num canal do analisador. Ele espelha o sinal que alimenta a base do transistor TIP31. Execute o projeto `rot9_cooler` [20].

3.1 Qual é o período configurado para TPM1? Está condizente com os valores medidos pelo analisador?

3.2 No programa foi setado um valor igual a (3.*TPM1_MOD)/4 em `TPM1_CH0`. Se reduzirmos para 0, TPM1_MOD/4 e TPM1_MOD/2, o que acontecerá com as formas de onda e a rotação do *cooler*? E se aumentarmos para (TPM1_MOD+1)?

4 Desenvolva o projeto `controle_cooler` em que o sinal analógico do potenciômetro e o sinal analógico do sensor de temperatura são amostrados periodicamente como no projeto `rot9_aula`.

4.1 Criação de um novo projeto `controle_cooler` (Seção 2.1/página 4 em [19]).

4.2 Inclusão das últimas versões de `ADC.*`, `GPIO_latch_lcd.*`, `SIM.*`, `TPM.*`, `util.*` para reuso (Seção 2.2.3/página 14 em [19]). Crie novos arquivos `ISR.*` (Seção 2.2.2/página 14 em [19]).

4.3 Definição dos estados e das regras de transições válidas para cada par de estados mostrados no diagrama de máquina de estados da figura 13. Distinguem-se os seguintes estados e as transições entre eles:

a) `AMOSTRA_VOLT`: o sistema aguarda por um **evento de estouro** do contador `TPM2_CNT` para disparar uma amostragem e conversão do sinal do potenciômetro (canal 9/PTB1). Assim que é gerada uma interrupção pelo **evento de conclusão de uma conversão**, o fluxo de controle é desviado para a rotina `ADC0_IRQHandler` onde o resultado da conversão é armazenado. Reconfigura-se o sistema para amostragem e conversão do sinal do sensor de temperatura (canal 26) por *software*. Inicia a conversão por software e passa para o estado `AMOSTRA_TEMP`.

a) `AMOSTRA_TEMP`: o sistema aguarda por um novo **evento de conclusão de uma conversão**. Assim que é gerada a segunda interrupção, o fluxo de controle é desviado novamente para a rotina `ADC0_IRQHandler` e o resultado da conversão é armazenado. É reconfigurado o disparo por *hardware* para a amostragem e conversão do sinal do potenciômetro antes de passar para o estado `ATUALIZACAO`.

a) `ATUALIZACAO`: são atualizados na função `main` o visor do LCD e(PTC) as larguras dos pulsos dos sinais EPWM enviados para o *cooler* (TPM1_CH0/PTB0) e para os *leds* indicadores (vermelho e verde) de temperatura (TPM2_CH0/PTB18 e TPM2_CH1/PTB19). Passa-se para o estado `AMOSTRA_VOLT`.

Por legibilidade, 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, `AMOSTRA_VOLT`, `AMOSTRA_TEMP` e `ATUALIZACAO`.

4.4 Inicialização do sistema.

4.4.1 Inicialize o LCD usando `GPIO_ativaConLCD` (habilita conexão entre LCD e KL25Z via PORTC) e `GPIO_initLCD` (inicializa LCD com as instruções recomendadas pelo fabricante).

4.4.2 Selecione a fonte dos sinais de relógio para os contadores de TPM (Seção 5.7.5/página 124 em [1]) e a frequência do sinal de barramento (Seção 12.2.12/página 210 em [1]).

4.4.3 Implemente a função `void TPM1TPM2_PTB0PTB18PTB19_config_basica()` em que são habilitados os sinais de relógio dos módulos TPM1 e TPM2 e alocados os pinos para os canais TPM1_CH0, TPM2_CH0 e TPM2_CH1.

4.4.4 Determine o divisor *prescaler* para TPM2 de forma que o seu período seja aproximadamente 645*tempo de conversão do módulo de ADC. Configure TPM1, TPM2 e seus canais com uso de `TPM_config_especifica` e `TPM_CH_config_especifica`.

4.4.5 Adicione a macro em `ADC.h` (Seção 12.2.6/página 201 em [1]) :

```
#define TPM2_TRG    0b1010
```

para usá-la na inicialização básica do módulo ADC, com o pino PTB1 alocado ao canal 9 e a fonte dos disparos por *hardware*, através da função `ADC_PTB1_config_basica`.

4.4.6 Configure o modo de operação do ADC com a função `ADC_Config_Alt`.

4.4.7 Auto-calibre ADC com a rotina `ADC_Cal`.

Faça **testes de unidades** de operação de cada módulo (muitos destes testes já foram feitos).

4.5 Programação dos estados e das suas transições.

Todas as tarefas relacionadas com amostragens e gerações de sinais em formas de onda específicas podem ser realizadas em *hardware*. A tarefa do programador se reduz em configurar/habilitar/ativar os circuitos apropriados em cada estado e atualizar as informações no LCD. Pelas considerações no item 3.3, as tarefas relacionadas com os estados `AMOSTRA_VOLT` e `AMOSTRA_TEMP` podem ser executadas em `ADC0_IRQHandler` antes de passar, respectivamente, para os estados `AMOSTRA_TEMP` e `ATUALIZACAO`. Segue-se um esboço:

```
void ADC0_IRQHandler(void) {
   if( ADC0_SC1A & ADC_SC1_COCO_MASK ) {
      valor = ADC0_RA;
      if (estado == AMOSTRA_VOLT) {
        //tarefas a serem executadas
        estado = AMOSTRA_TEMP;
      } elss if (estado == AMOSTRA_TEMP) {
        //tarefas a serem executadas
        estado = ATUALIZACAO;
      }
   }
}
```

Como as tarefas no estado `ATUALIZACAO` envolvem o LCD, elas devem ser executadas fora das rotinas de serviço. Sugere-se que sejam feitas em `main`

```
if (estado == ATUALIZACAO) {
   //tarefas de atualização do LCD e dos sinais de saída para
   //cooler e leds.
   estado = AMOSTRA_VOLT;
}
```

Para **testes de unidade de transições dos estados**, habilite a contagem do TPM2 e a interrupção do ADC. Execute o programa setando pontos de parada nos diferentes blocos de estados.

4.6 Implementação de funções auxiliares:

4.6.1 Interpretação dos Valores Amostrados: para mostrar no LCD a temperatura em graus Celsius, é necessário converter o código binário `valor` gerado pelo conversor ADC em temperatura. Implemente em `util.*` a função `float AN3031_Celsius(uint16_t valor)` que faz essa conversão. Faça **testes de unidade** de conversão.

4.6.2 Atualização de `TPM1_C0V` no controle de velocidade: uma forma simples é copiar o valor amostrado do sinal do potenciômetro em `TPM1_C0V`, já que ambos os valores são de 16 *bits* e tem uma relação direta (maior tensão, maior valor amostrado; maior valor em `TPM1_C0V`, maior potência transferida para o *cooler*).

4.6.3 Atualização das intensidades dos *leds*: atualize o conteúdo de `TPM2_C0V` (PTB18)/TPM2_C1V(PTB19) com |T-25ºC|/(25ºC-0ºC)*(TPM2_MOD) ou |T-25ºC|/(50ºC-25ºC)*(TPM2_MOD), onde T é a temperatura amostrada.

4.6.4 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 e os dados amostrados pelo ADC são implementadas as funções `tipo_estado ISR_LeEstado()`, `void ISR_EscreveEstado (tipo_estado valor)` e `uint16_t *ISR_leValoresAmostrados(uint16_t *valores)`.

4.7 Implementação da Máquina de Estados

Complete os estados e as transições para a máquina de estados projetada. Faça uma revisão geral do código implementado com base nas regras definidas no item 3.3. Verifique se todas as restrições em transições entre estados são contempladas. Se possível, execute o programa com pontos de parada para certificar se o fluxo condiz com o esperado. Faça **testes funcionais** girando o eixo do potenciômetro nos dois sentidos.

4.8 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.

4.9 Gere uma documentação do projeto com Doxygen [13].

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, 2 e 3 e suba o arquivo no sistema *Moodle*. Para a segunda semana, faça uma descrição sucinta dos testes conduzidos ao longo do desenvolvimento do projeto `controle_cooler`, junto com algumas imagens ilustrativas, num arquivo em pdf. Exporte o projeto `controle_cooler` **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] Freescale. *KL25 Sub-Family Reference Manual*.
https://www.dca.fee.unicamp.br/cursos/EA871/references/ARM/KL25P80M48SF0RM.pdf
[2] Elliot Smith. Understanding the Successive Approximation Register ADC.
https://www.allaboutcircuits.com/technical-articles/understanding-analog-to-digital-converters-the-successive-approximation-reg/
[3] Temperature Sensor for the HCS08 Microcontroller Family
https://www.nxp.com/docs/en/application-note/AN3031.pdf
[4] Instituto Newton C. Braga. Como funcionam os conversores A/D?
http://www.newtoncbraga.com.br/index.php/como-funciona/1508-conversores-ad
(parte 1) e http://www.newtoncbraga.com.br/index.php/como-funciona/1509-conversores-ad-2
(parte 2)
[5] Freescale. *Kinetis L Peripheral Module Quick Reference (Rev. 0.09/2012)*.
https://www.dca.fee.unicamp.br/cursos/EA871/references/ARM/KLQRUG.pdf
[6] Nova versão do esquemático do shield FEEC
https://www.dca.fee.unicamp.br/cursos/EA871/references/complementos_ea871/Esquematico_EA871-Rev3.pdf
[7] Folha de dados técnicos - Kinetis KL25 Sub-Family 48 MHz Cortex-M0+ Based Microcontroller with USB.
https://www.dca.fee.unicamp.br/cursos/EA871/references/ARM/KL25P80M48SF0.pdf
[8] Azo Sensors. How Force Sensing Resistors Measure Force
https://www.azosensors.com/article.aspx?ArticleID=1718
[9] Resistance vs. Illumination curve
https://www.researchgate.net/figure/Resistance-vs-illumination-curve-LDRs-are-light-dependent-devices-whose-resistance_fig3_318029856
[10] Temperature sensor ICs simplify Design
https://www.maximintegrated.com/en/design/technical-documents/app-notes/6/694.html
[11] rot9_aula.zip
http://www.dca.fee.unicamp.br/cursos/EA871/1s2023/codes/rot9_aula.zip
[12] rot7_aula.zip
http://www.dca.fee.unicamp.br/cursos/EA871/1s2023/codes/rot7_aula.zip
[13] Doxygen
https://www.doxygen.nl/manual/docblocks.html

[14] Diagrams.net
https://www.diagrams.net/
[15] Analog Devices. The ABCs of Analog to Digital Converters: How ADC Errors Affect System Performance
https://www.analog.com/en/technical-articles/the-abcs-of-analog-to-digital-converters-how-adc-errors-affect-system-performance.html
[16] Direct Type ADCs
https://www.tutorialspoint.com/linear_integrated_circuits_applications/linear_integrated_circuits_applications_direct_type_adcs.htm
[17] rot9_example1.zip
http://www.dca.fee.unicamp.br/cursos/EA871/1s2023/codes/rot9_example1.zip
[18] Successive-approximation ADC
https://en.wikipedia.org/wiki/Successive-approximation_ADC
[19] Wu, S.T. Ambiente de Desenvolvimento de Software
https://www.dca.fee.unicamp.br/cursos/EA871/references/apostila_C/AmbienteDesenvolvimentoSoftware_V1.pdf
[20] rot9_cooler.zip
http://www.dca.fee.unicamp.br/cursos/EA871/1s2023/codes/rot9_cooler.zip

Revisado em Fevereiro de 2023
Revisado em Março e Outubro de 2022
Revisado em Maio e Julho de 2021
Revisado em Novembro de 2020
Revisado em Agosto de 2017
Elaborado com base no roteiro do Experimento 13 no Segundo Semestre de 2015.