EXPERIMENTO 9 - ADC e LPTMR

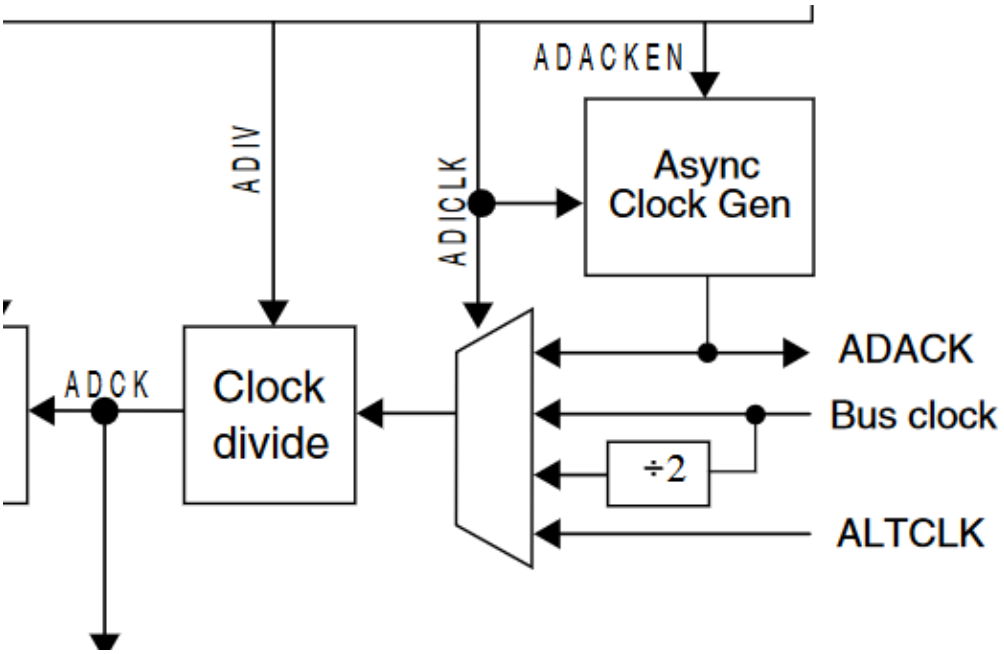
FEEC | EA871

Thiago Maximo Pavão - 247381
Vinícius Esperança Mantovani - 247395

Primeira parte

1.

1) A frequência de ADCK é configurada conforme o diagrama da Figura 28-1 do manual de referência



Pelo campo ADC0_CFG1_ADICLK, vemos que a fonte de clock selecionada é Bus Clock, como já visto, este último é MCGOUTCLK dividido por OUTDIV1 e OUTDIV4, no caso temos MCGOUTCLK dividido por 2. Esta frequência é posteriormente dividida por um divisor configurado pelo campo ADC0_CFG1_ADIV, como é possível observar na janela do debugger, causa divisão por 4. Conclui-se que a frequência é

$$ADCK = MCGOUTCLK / (2 \times 4) = 20.971.520 / 8 = 2,62144\ MHz$$

2) Pelo campo ADC0_CFG1_MODE, percebemos que a resolução em quantidade de bits é igual a 16, uma vez que o campo assume o valor de 0b11, o que implica que o valor selecionado é de 16 bits, como afirmado anteriormente. Destaquemos, ainda, que como o MODE é 0b11, o campo ADC0_SC1n_DIFF não influencia no tamanho da resolução em bits, pois neste caso de MODE, o valor de DIFF apenas determina se a conversão é single-ended ou diferencial.

11 When DIFF=0:It is single-ended 16-bit conversion; when DIFF=1, it is differential 16-bit conversion with 2's complement output.

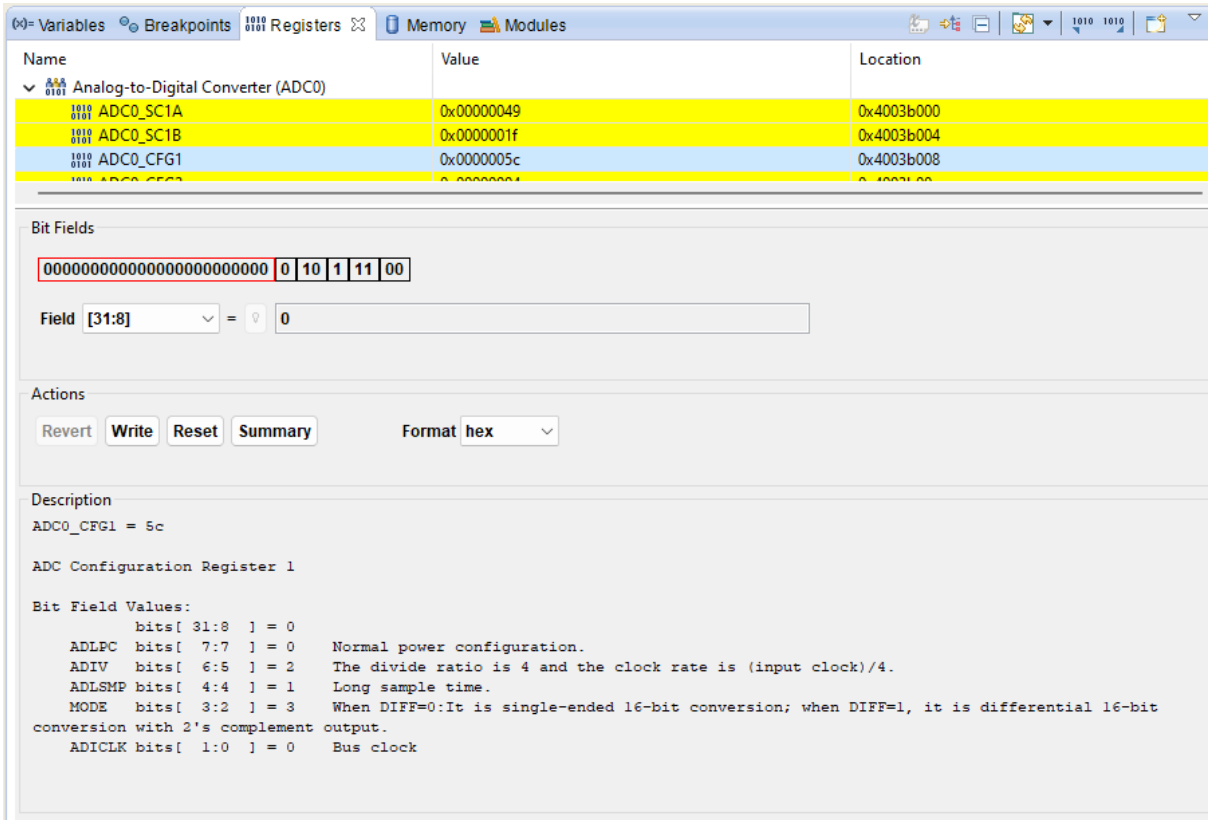


Figura 1: Campos do registrador ADC0_CFG1 após a inicialização

3) A conversão é selecionada em modo de alta velocidade, dado que o campo ADC0_CFGE = 1.

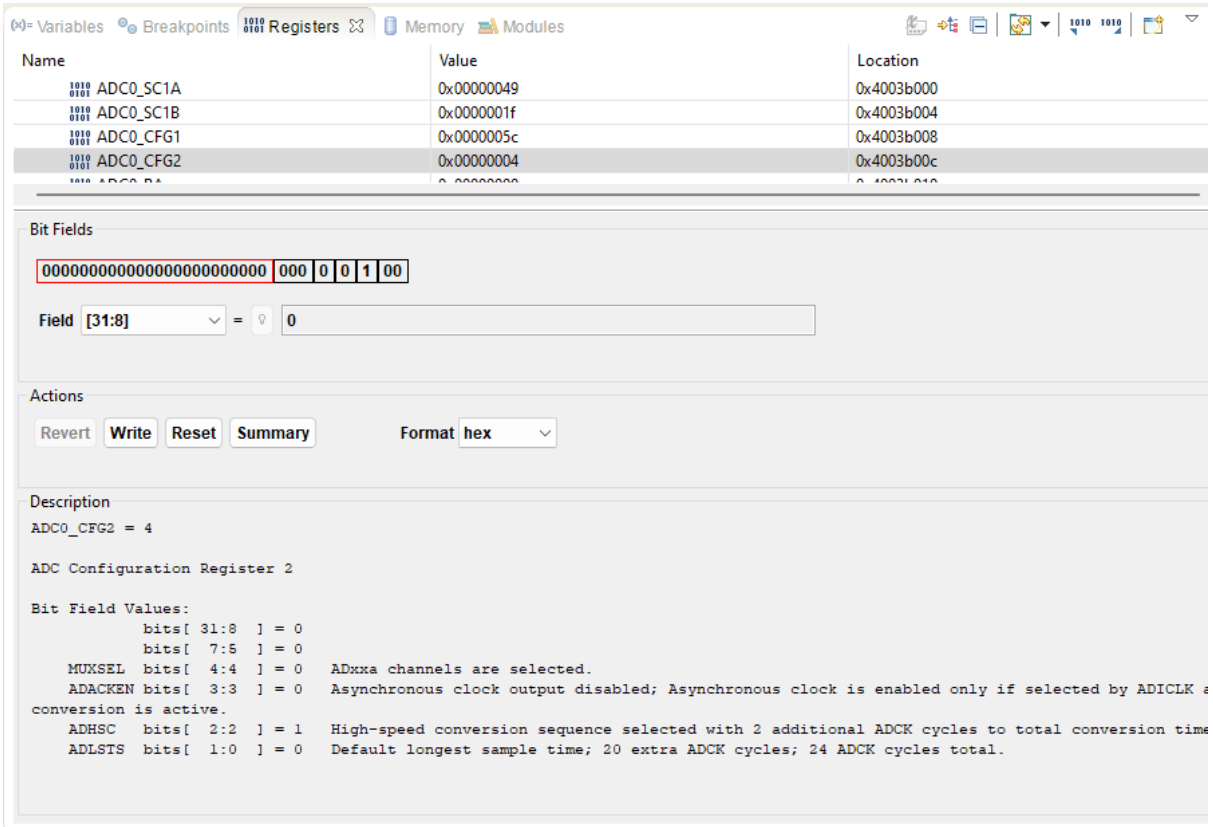


Figura 2: Campos do registrador ADC0_CFG2 após inicialização

4) Como o valor de ADC0_SC3_ADCO é setado em 0b0 e o campo ADC0_SC3_AVGE é setado em 0b1, então o fluxo de conversão não é contínuo, ou seja, ocorre uma conversão ou um conjunto de conversões, mas não mais que uma ou mais que um conjunto.

5) A quantidade de amostras configurada é 32, pois ADC0_SC3_AVGE foi setado, habilitando a função de média por hardware e ADC0_SC3_AVGS = 0b11, que de acordo com o manual de referência configura a média por 32 amostras.

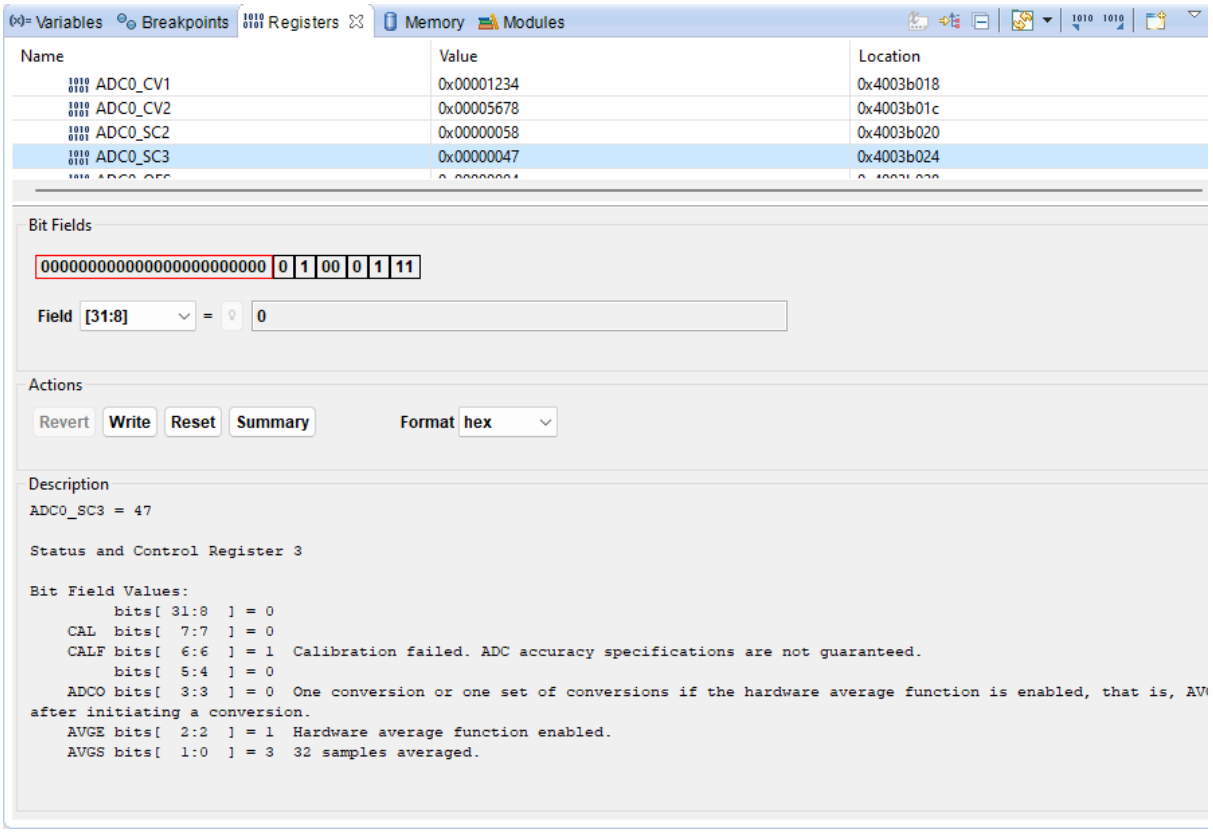


Figura 3: Campos do registrador ADC0_SC3 após inicialização

6) Nota-se que *ADC0_CFG2_MUXSEL = 0b0* (Figura 2), então canais do tipo A são selecionados. Além disto, *ADC0_SC1A_ADCH = 0b01001* (Figura 4), portanto o canal selecionado é AD9, para o qual o pino PTB1 foi multiplexado.

4 MUXSEL	ADC Mux Select Changes the ADC mux setting to select between alternate sets of ADC channels. 0 ADxxa channels are selected. 1 ADxxb channels are selected.

A seleção da entrada é feita pela função *ADC_selecionaCanal*, enquando a multiplexação do pino é feita por *ADC_PTB1_config_basica*.

```
void ADC_selecionaCanal(uint8_t canal) {
    ADC0_SC1A &= ~ADC_SC1_ADCH(0b11111);
    ADC0_SC1A |= ADC_SC1_ADCH(canal);
}
```

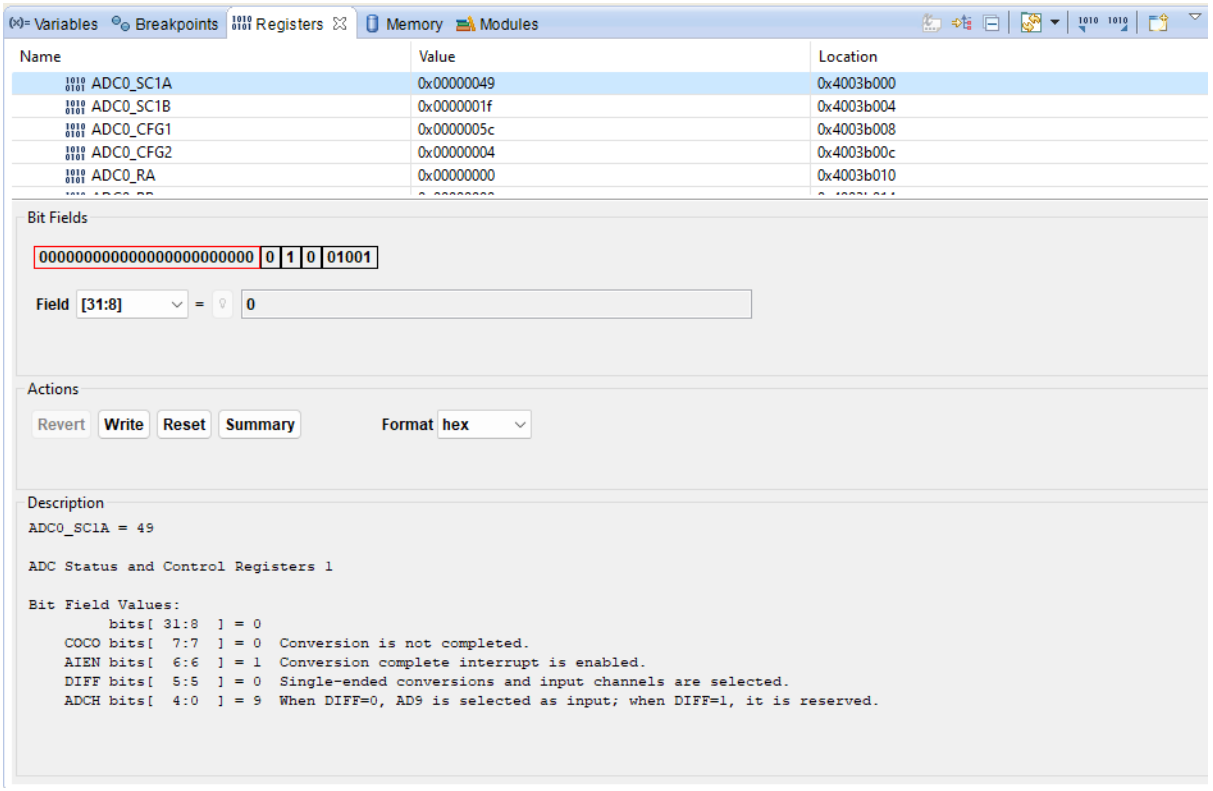


Figura 4: Campos do registrador ADC0_SC1A após inicialização

2.

Na figura 1, vemos que ADC0_CFG1_ADLSMP = 0b1, que configura o longo tempo de amostragem, além disso, da Figura 2 temos ADC0_CFG2_ADLSTS = 0b00, que de acordo com o manual de referência escolhe o maior tempo disponível para amostragem, que passa a levar o tempo para 24 oscilações completas em ADCK, portanto, o tempo de amostragem é

$$Tempo\ de\ Amostragem\ = 24 \times ADCK^{-1}\ = 24\ /\ 2,62144\ MHz\ = 9,155\ us$$

1–0 ADLSTS	Long Sample Time Select Selects between the extended sample times when long sample time is selected, that is, when CFG1[ADLSMP]=1. This allows higher impedance inputs to be accurately sampled or to maximize conversion speed for lower impedance inputs. Longer sample times can also be used to lower overall power consumption when continuous conversions are enabled if high conversion rates are not required. 00 Default longest sample time; 20 extra ADCK cycles; 24 ADCK cycles total. 01 12 extra ADCK cycles; 16 ADCK cycles total sample time. 10 6 extra ADCK cycles; 10 ADCK cycles total sample time. 11 2 extra ADCK cycles; 6 ADCK cycles total sample time.

Supondo que a impedância de entrada no pino PTB1 seja muito baixa, é mais proveitoso utilizar um tempo de amostragem menor, diminuindo o tempo de conversão. Para isto, poderia ser utilizado o tempo de amostragem curto (ADC0_CFG1_ADLSMP = 0b0) ou utilizar o longo mas com a menor adição de tempo possível (ADC0_CFG2_ADLSTS = 0b11).

3.

1) Como o campo ADC0_SC2_ACFE tem valor igual a 0, percebemos que a função de comparação está desabilitada. Como se percebe na Figura 5, em que se vê os valores dos bits do registrador ADC0_SC2 com o código parado pelo debugger após a inicialização do sistema. No entanto, notamos na mesma figura, que os campos ADC0_SC2_ACREN e ADC0_SC2_ACFGT estão setados em 1, o que implica que a função de comparação ocorre com relação a algum dos quatro intervalos formados fora ou entre os valores de CV1 e CV2, incluido-os ou não. Notamos ainda, pelas figuras 6 e 7, que o valor CV1 < CV2. Portanto, temos que a comparação do resultado de conversão é feita com relação ao intervalo de [CV1, CV2].

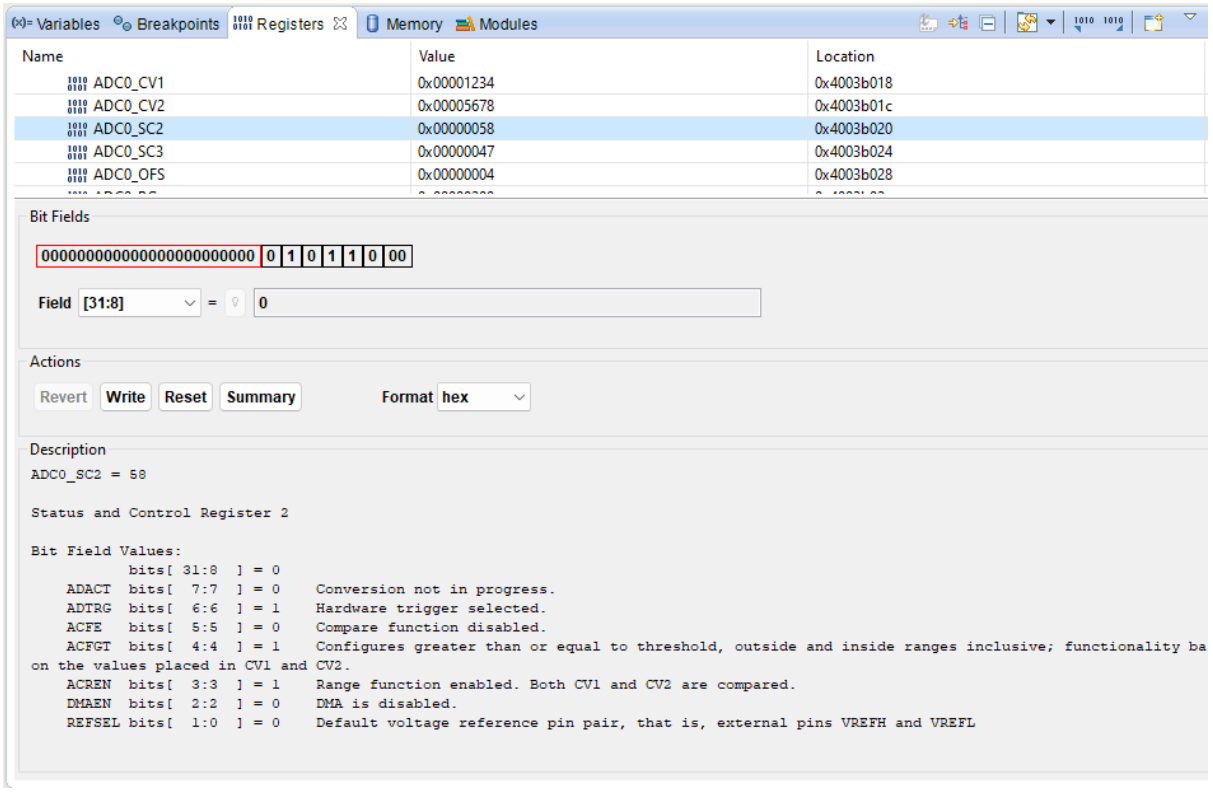


Figura 5: Campos do registrador ADC0_SC2 após inicialização

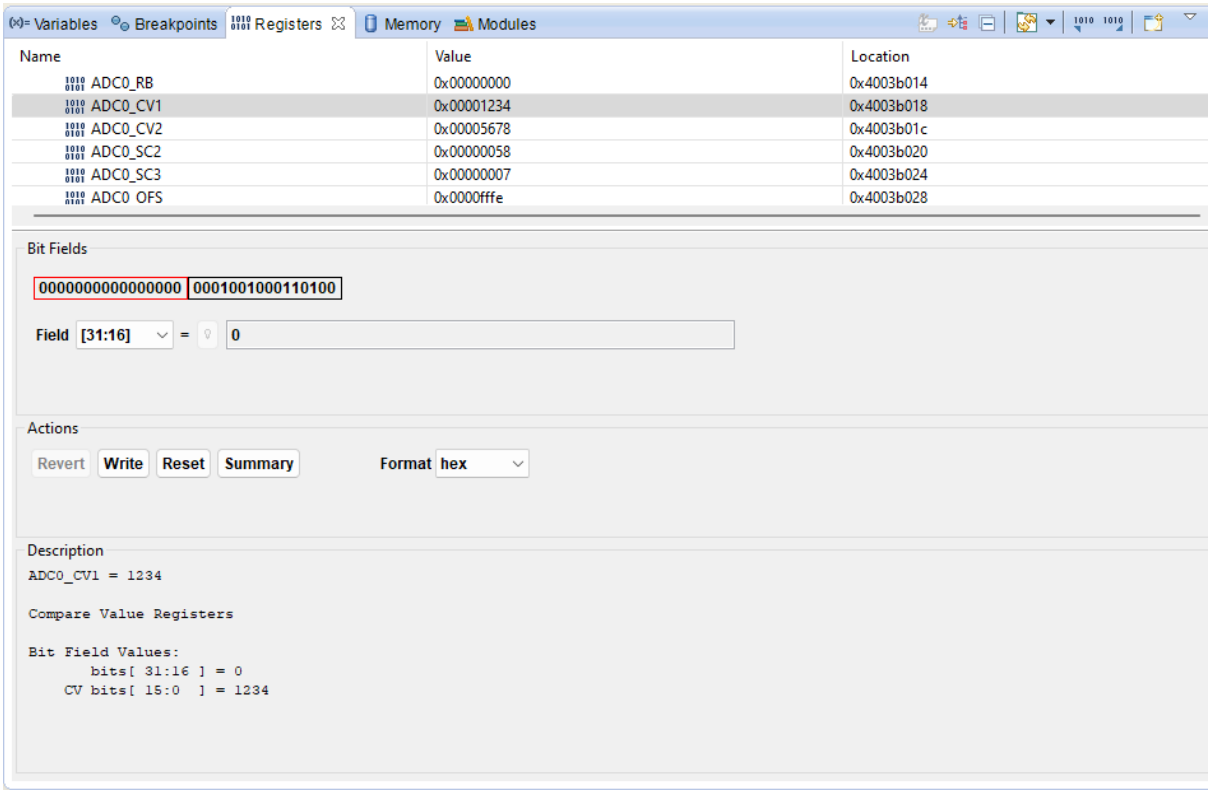


Figura 6: Campos do registrador ADC0_CV1 após inicialização

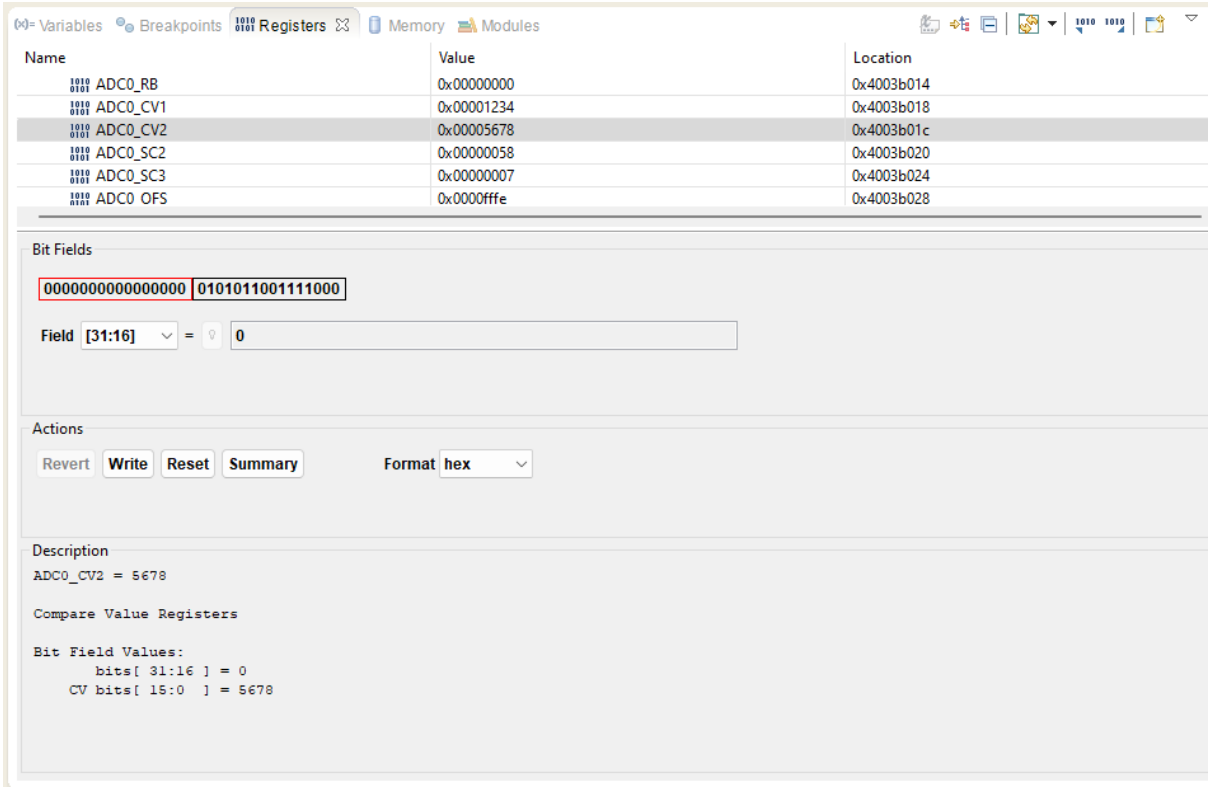


Figura 7: Campos do registrador ADC0_CV2 após inicialização

2) Após habilitar a função na linha `| ACFE ENABLED` de main.c, temos que ao mover o potenciômetro, existindo um breakpoint em ADC0_IRQHandler, o valor de ADC0_RA é setado para *0x5095*, isto ocorreu pois a posição inicial do potênciometro estava gerando uma tensão de saída de cerca de 3 V (medido com osciloscópio). A rotação fez com que o valor variasse até entrar no intervalo configurado, gerando um valor próximo do máximo (0x5678).

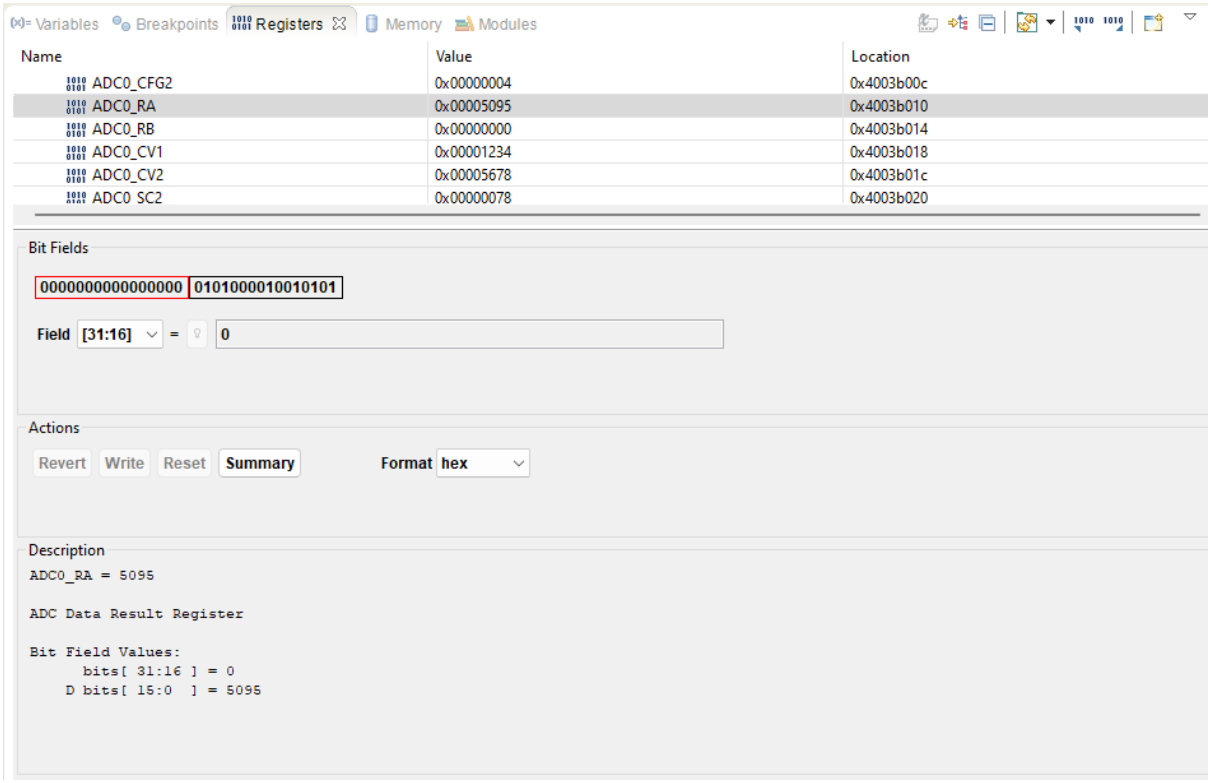


Figura 8: Campos do registrador ADC0_RA após inicialização

4.

A conversão é feita de forma única, em contraposição à conversão contínua, porque o registrador ADC0_SC3 tem seu campo ADCO configurado em 0. Isto é feito pela função *ADC_Config_Alt* com os valores definidos na *struct Master_Adc_Config*. Além disso, estes disparos são realizados pelo LPTMR0, esta configuração é feita pela função *ADC_PTB1_config_basica*, pelo bit ADC0TRGSEL do registrador SIM_SOPT7, esta explicação está feita com mais detalhes na questão 1.13.1.

5.

Pela tabela do manual de referência, nota-se que o PTB1 é multiplexado por padrão para ADC0_SE9.

80 LQFP	64 LQFP	48 QFN	32 QFN	Pin Name	Default	ALT0	ALT1	ALT2	ALT3
44	36	28	21	PTB1	ADC0_SE9/ TSI0_CH6	ADC0_SE9/ TSI0_CH6	PTB1	I2C0_SDA	TPM1_CH1

Isto é também garantido pela execução da função *ADC_PTB1_config_basica*, que zera o campo de multiplexação com a seguinte linha

```
PORTB_PCR1 &= ~PORT_PCR_MUX(0b111);
```

Não seria possível utilizar o filtro passivo do PORT (PORTx_PCRn_PFE) para filtrar ruídos do sinal de entrada porque pinos analógicos tem essa função desabilitada, isto é especificado no manual de referência na seção 11.6.1, no trecho

> When the Pin Muxing mode is configured for analog or is disabled, all the digital functions on that pin are disabled. This includes the pullup and pulldown enables, and passive filter enable.

6.

Nota-se primeiramente que no manual a flag AIEN do registrador ADC0_SC1A nunca é configurada em 1. Isto faz com que o canal não gere interrupções no fim de uma conversão, um erro por parte do manual. Isso foi corrigido em *rot9_example1*: *ADC_habilitaInterrupCOCO* realiza a configuração.
Outra diferença percebida é a ativação do número de interrupção de LPTMR0 pelo manual, isto é desnecessário pois o evento de overflow dispara a amostragem da tensão por ADC sem que a interrupção seja habilitada, sem a necessidade de intervenção do processador. Isto também foi corrigido no *rot9_example*.

7.

A estratégia utilizada é a de polling, como se percebe na figura a seguir:

segue a figura

```
while (end->SC3 & ADC_SC3_CAL_MASK)   {} ///< aguarda a calibracao
```

linha na qual se aguarda a conclusão da calibração, conferindo se o campo ADC_SC3_CAL vale 1, ou seja, se a calibração está em progresso, até que o while é finalizado no caso de o campo valer 0 (a calibração ter sido concluída).

7 CAL	Calibration Begins the calibration sequence when set. This field stays set while the calibration is in progress and is cleared when the calibration sequence is completed. CALF must be checked to determine the result of the calibration sequence. Once started, the calibration routine cannot be interrupted by writes to the ADC registers or the results will be invalid and CALF will set. Setting CAL will abort any current conversion.

(É setado em 1 até a conclusão, quando é resetado para 0).

8.

Foi implementado um filtro exponencial por software com $\alpha = 0.5$, isto é feito pelas instruções

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

9.

Os valores iniciais dos membros da variável Master_Adc_Config são configurados por meio de atribuição em main.c. conforme a imagem a seguir.

```
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=ADLPC_NORMAL
        | ADC_CFG1_ADIV(ADIV_4)
        | ADLSMP_LONG
        | ADC_CFG1_MODE(MODE_16)
        | ADC_CFG1_ADICLK(ADICLK_BUS),
        .CFG2=MUXSEL_ADCA                    //select ADC0_SE9 (PTB1 potenciometro externo)
        | ADACKEN_DISABLED
        | ADHSC_HISPEED
        | ADC_CFG2_ADLSTS(ADLSTS_20),
        .CV1=0x1234u,
        .CV2=0x5678u,
        .SC2=ADTRG_HW //Hardware trigger
        | ACFE_ENABLED
        | ACFGT_GREATER
        | ACREN_ENABLED
        | DMAEN_DISABLED
        | ADC_SC2_REFSEL(REFSEL_EXT),
        .SC3=CAL_OFF
        | ADCO_SINGLE
        | AVGE_ENABLED
        | ADC_SC3_AVGS(AVGS_32),
};
```

Na função ADC_Config_Alt, ocorre que os valores atribuídos anteriormente (na main) para a variável Master_Adc_Config são atribuídos para os membros de ADC0_BASE_PTR, ponteiro da base de ADC0. Conforme a imagem a seguir:

```
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;
}
```

10.

LPTMRx_CSR_TMS é configurado em zero, então o modo de operação do módulo LPTMR0 é de **contador de tempo.**

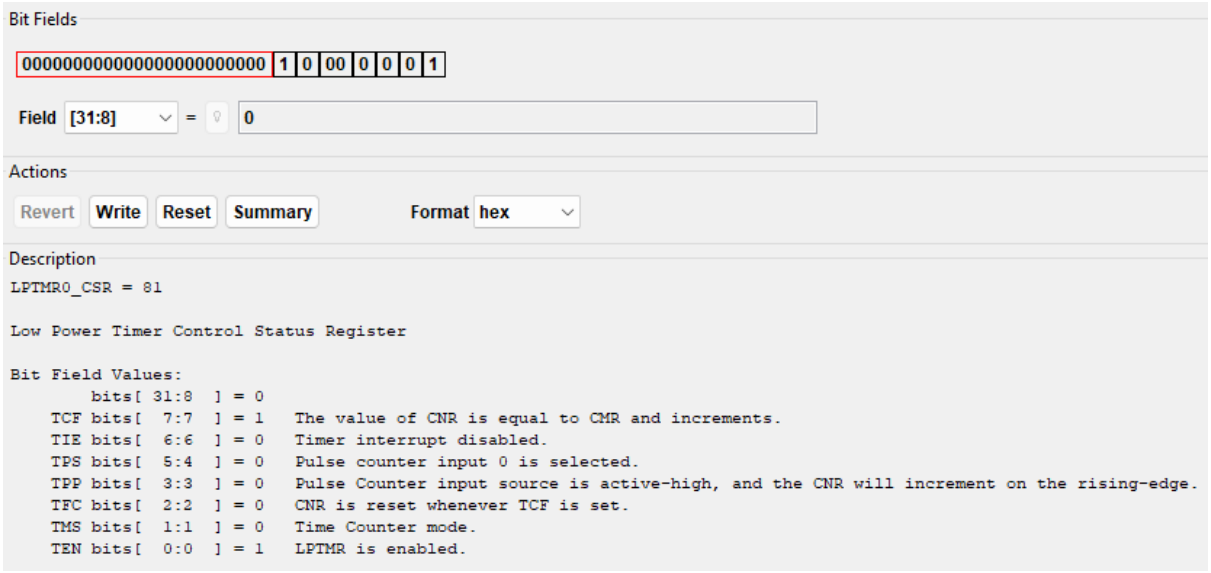


Figura 9: Campos do registrador LPTMR0_SCR após inicialização

11.

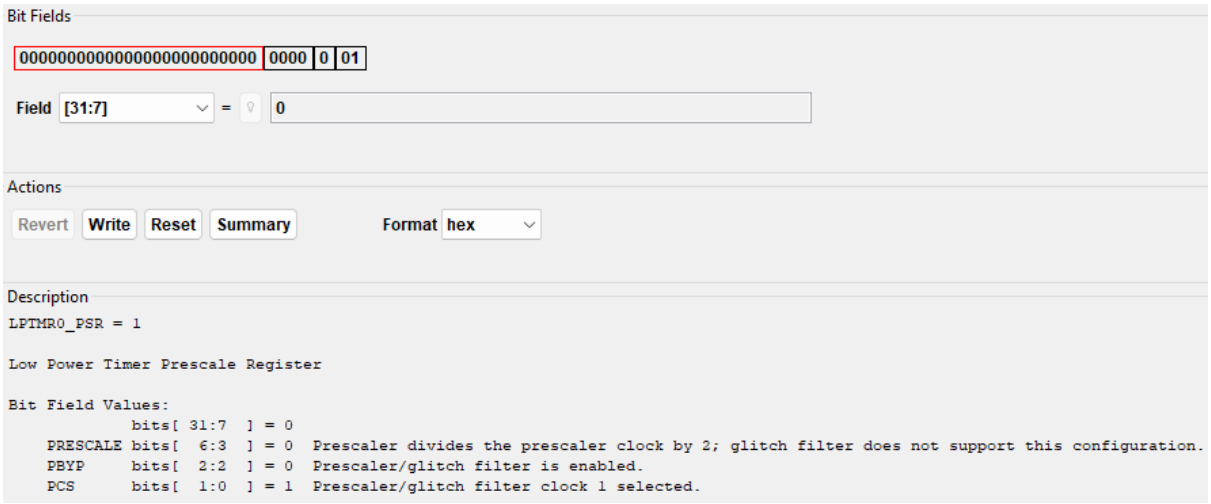


Figura 10: Campos do registrador LPTMR0_PSR após inicialização

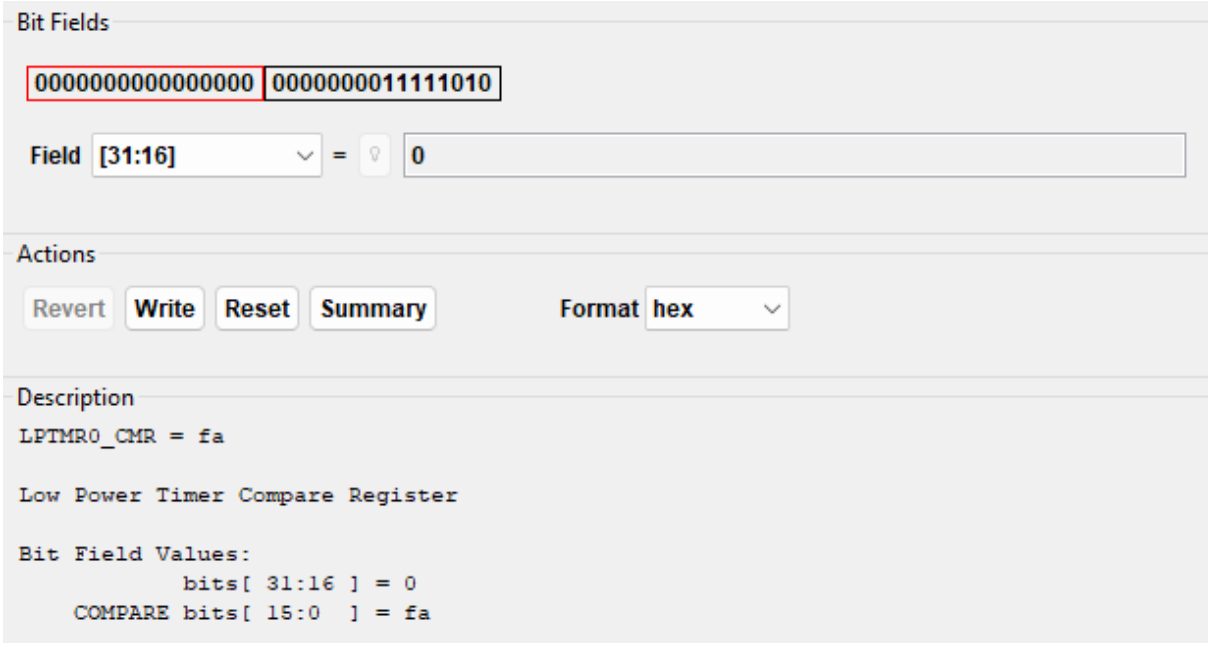


Figura 11: Campos do registrador LPTMR0_CMR após inicialização

LPTMR0_PSR_PBYP = 0 (Figura 10) e LPTMR0_PSR_TCF = 1 (Figura 9) , então a equação que define o período do timer é

$$Período = LPTMR0\_CMR \times \frac{2\text{^}(LPTMR0\_PSR\_PRESCALE + 1)}{f_{LPTMR}}$$

No programa *rot9_example1*, a fonte de clock configurada é LPO, que tem a 1 kHz de frequência, o prescaler é obtido da Figura 10, sendo igual a 0 e LPTMR0_CMR = *0xfa = 250*. Portanto o período é

$$Período = 250 \times 2 / 1000 = 0,5\ s$$

Confirmando o resultado no analisador lógico após descomentar as linhas de configuração de interrupção por LPTMR0 e da respectiva rotina de tratamento de interrupção, vemos que o valor experimentalmente medido concorda com o previsto

12.

A seguir está uma comparação entre a função implementada e o exemplo:

`LPTMR0_PSR |= LPTMR_PSR_PCS(clock_source);` → IMPLEMENTAÇÃO: configura a fonte do clock por meio do parâmetro clock_source, como a fonte LPO de 1kHz;

```
PORTC_PCR1 |= PORT_PCR_MUX(1);//select RTC_CLKIN function
```
→ EXEMPLO: configura a fonte do clock como RTC.

```
LPTMR0_PSR |= LPTMR_PSR_PRESCALE(prescaler); // 0000 is div 2
LPTMR0_PSR &= ~LPTMR_PSR_PBYP_MASK;  //habilitar PSR
```
→ IMPLEMENTAÇÃO: configura prescaler para divisão por dois e, habilita PSR para habilitar a divisão (prescaler).

```
LPTMR0_PSR = (LPTMR_PSR_PRESCALE(0) // 0000 is div 2
              | LPTMR_PSR_PBYP_MASK
              | LPTMR_PSR_PCS(clock_source)) ;
```
→ EXEMPLO: seta prescaler para divisão por 2. Porém, existe um erro no exemplo, pois o campo LPTMR_PSR_BPYP como 1, logo o prescaler é ignorado.

```
LPTMR0_CMR = LPTMR_CMR_COMPARE(count); //Set compare value
```
→ IMPLEMENTAÇÃO: seta o valor de comparação, por meio do argumento *count*.

```
LPTMR0_CMR = LPTMR_CMR_COMPARE(count);  //Set compare value
```
→ EXEMPLO: seta o valor de comparação por meio do argumento *count.*

```
LPTMR0_CSR &= ~(LPTMR_CSR_TPP_MASK //TMR Pin polarity
        | LPTMR_CSR_TFC_MASK // Timer Free running counter is reset whenever TMR
        //counter equals compare
        | LPTMR_CSR_TMS_MASK //LPTMR0 as Timer
        | LPTMR_CSR_TEN_MASK //!!! LPTMR0 disabled (para que os disparos HW acontecam
        //depois da conclusao de configuracao do sistema)
);
```
→ IMPLEMENTAÇÃO: configura a polaridade da entrada, configura para que sempre que o contador TMR seja igual a compare, o contador do timer seja resetado. Além disso, na penúltima linha, seta o LPTMR como timer e na última, desabilita o LPTMR0.

```
LPTMR0_CSR = (LPTMR_CSR_TCF_MASK    // Clear any pending interrupt
              | LPTMR_CSR_TIE_MASK    // LPT interrupt enabled
              | LPTMR_CSR_TPS(0)      //TMR pin select
              |!LPTMR_CSR_TPP_MASK    //TMR Pin polarity
              |!LPTMR_CSR_TFC_MASK    // Timer Free running cour
```
→ EXEMPLO: além daquilo que foi citado no caso da implementação acima, notamos que, no manual, é selecionado o pin 0, em TPS e, é habilitada interrupção de LPT. No entanto, ao contrário da implementação, aqui não ocorre a configuração de TMS nem TEN, sendo a primeira feita em seguida no manual e a segunda ignorada.

```
LPTMR0_CSR |= (LPTMR_CSR_TCF_MASK // Clear any pending interrupt
        //    | LPTMR_CSR_TIE_MASK //!!! nao eh necessario se nao precisarmos que evento tenha
        //tratamento especifico
        | LPTMR_CSR_TPS(0) // CMP0_OUT pin select (Secao 3.8.3.2/p. 89 em Manual)
);
```
→ IMPLEMENTAÇÃO: limpa interrupções pendentes e seleciona o pin 0 de entrada.

```
|!LPTMR_CSR_TMS_MASK    //LPTMR0 as Timer
```
→ EXEMPLO: seta o LPTMR como timer.

Não é alocado um pino fixo para LPTMR0, uma vez que, conforme se percebe na imagem acima, é selecionado como input o output de CMP0 (comparador) no lugar de um possível pino físico.

13.

1) Em *ADC_PTB1_config_basica* configura-se o campo ADC0TRGSEL do registrador SIM_SOPT7 com o valor passado como parâmetro para função: *trigger*. Na chamada da função em *main.c* o valor passado é o da macro LPTMR0_TRG que é igual a 0b1110. Por fim, consultado o manual, nota-se que isto faz com que a fonte de disparos para conversão de ADC0 seja LPTMR0.

```
SIM_SOPT7 |= (SIM_SOPT7_ADC0ALTTRGEN_MASK
 | SIM_SOPT7_ADC0TRGSEL(trigger)) ;
```

3–0 ADC0TRGSEL	ADC0 trigger select Selects the ADC0 trigger source when alternative triggers are functional in stop and VLPS modes. . 0000 External trigger pin input (EXTRG_IN) 0001 CMP0 output 0010 Reserved 0011 Reserved 0100 PIT trigger 0 0101 PIT trigger 1 0110 Reserved 0111 Reserved 1000 TPM0 overflow 1001 TPM1 overflow 1010 TPM2 overflow 1011 Reserved 1100 RTC alarm 1101 RTC seconds 1110 LPTMR0 trigger 1111 Reserved

2) Percebe-se a ativação do número de interrupção de LPTMR0 pelo manual, isto é desnecessário pois o evento de overflow dispara a amostragem da tensão por ADC sem que a interrupção seja habilitada, sem a necessidade de intervenção do processador. Isto foi corrigido no *rot9_example*, que não habilita a interrupção.

No manual:

11.2.1.7 Enable the LPTMR and ADC interrupt

```
enable_irq(ADC0_irq_no);
enable_irq(LPTMR0_irq_no);
```

Em *rot9_example*:

```
//Habilita interrupcoes no NVIC
ADC_habilitaNVICIRQ(0);
```

3) A ativação dos disparos periódicos ocorre com a ativação do timer, isto é feito pela função *LPTMR0_ativaCNR*, que seta o campo TEN de LPTMR0_CSR em 1. Essa função é chamada após toda a inicialização do sistema, logo antes da entrada no loop *for* da main. Após a execução desta linha, a contagem se inicia e o primeiro disparo é gerado em meio segundo.

```
void LPTMR0_ativaCNR () {
    LPTMR0_CSR |= LPTMR_CSR_TEN_MASK;
}
```

Segunda parte

1.

A conversão no canal 9 é iniciada por *hardware*, isso é configurado na inicialização do sistema, é configurado que TPM0 seja trigger de ADC pela chamada da função *ADC_PTB1_config_basica*, e o canal AD9 é selecionado, pela chamada de *ADC_selecionaCanal*. Já a conversão do canal 26 ocorre por software, pois assim que a conversão do canal 9 termina, é gerada uma interrupção que salva o valor convertido, altera o modo de disparo de conversão para por software: setando em zero o campo ADC_SC2_ADTRG e seleciona o canal 26 pela chamada da mesma função (*ADC_selecionaCanal*). Esta última chamada realiza uma escrita em ADC0_SC1A, que inicia a conversão.

```
void ADC0_IRQHandler(void) {
    if( ADC0_SC1A & ADC_SC1_COCO_MASK )
    {
        if (flag == 1) {
            GPIOE_PCOR = GPIO_PIN(21);
            valor[0] = ADC0_RA;
            ADC0_SC2 &= ~ADC_SC2_ADTRG_MASK;
            ADC_selecionaCanal (0b11010);
            GPIOE_PSOR = GPIO_PIN(21);
            flag = 2;
        } else if (flag == 2) {
            GPIOE_PCOR = GPIO_PIN(21);
            valor[1] = ADC0_RA;
            ADC0_SC2 |= ADC_SC2_ADTRG_MASK;
            ADC_selecionaCanal (0b01001);
            flag = 3;
        }
    }
}
```

Vale notar também que ADC_SC2_ADTRG é inicialmente setado em 1 (interrupção por hardware) pela função *ADC_Config_Alt*, que recebe como parâmetro a struct *Master_Adc_Config*, nesta struct, vemos que o campo é setado para interrupção por hardware.

2.

Conforme notamos no código, o campo PIT_LDVAL0 tem valor igual ao argumento *parametro*, cujo valor na chamada na main é de 20971520 (`PIT_initTimer0(20971520, 1);`). Além disso,

percebemos, também, que OUTDIV2 tem valor igual a 0b001, conforme a imagem a seguir

```
SIM_CLKDIV1 &= ~SIM_CLKDIV1_OUTDIV4(0b111); /
SIM_CLKDIV1 |= SIM_CLKDIV1_OUTDIV4(OUTDIV4);
```

Portanto, colocando na fórmula:

$$T_{PIT} = \frac{PIT_{LDVAL}}{\frac{MCGOUTCLK}{OUTDIV1 \times OUTDIV2}} = \frac{PIT_{LDVAL} \times OUTDIV2}{MCGOUTCLK} = \frac{PIT_{LDVAL} \times OUTDIV2}{20.971.520}$$

$$T_{PIT} = 20971520 * 2/20971520 = 2s$$

Assim, temos que o período do pit estimado é condizente com o que se vê por meio do analisador, o primeiro é igual a 2 e o segundo é aproximadamente igual a dois (1,998395688 s).

3.

O tempo de conversão é dado pela expressão abaixo, os valores de cada variável são determinados por tabelas, consultadas utilizando os valores dos campos dos registradores nelas listados. Executamos o programa no debugger, parando a execução com um breakpoint após toda a inicialização, os valores dos campos foram então lidos com a aba de registradores da IDE. Fazendo o processo temos

$$Tempo\ de\ conversão = SFCAdder + AverageNum * (BCT + LSTAdder + HSCAdder)$$

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

ADC0_CFG1_ADLSMP = 0b1 → tempo de amostragem longo
ADC0_CFG1_ADICLK = 0b01

$$SFCAdder = 3\ ADCK\ cycles + 5\ bus\ clock\ cycles$$

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

ADC0_SC3_AVGE = 0b1
ADC0_SC3_AVGS = 0b01

$$AverageNum = 8$$

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

ADC0_CFG1_MODE = 0b00
ADC0_SC1A_DIFF = 0b0
então é configurada conversão single-ended 8-bit conversion

$$BCT = 17\ ADCK\ cycles$$

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

ADC0_CFG1_ADLSMP = 0b1
ADC0_CFG2_ADLSTS = 0b00

$$LSTAdder = 20\ ADCK\ cycles$$

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

ADC0_CFG2_ADHSC = 0b1

$$HSCAdder = 2\ ADCK\ cycles$$

Temos, por fim, que o tempo de conversão é

$$Tempo\ de\ conversão = 3\ ADCK\ cycles + 5\ bus\ clock\ cycles + 8 * ((17 + 20 + 2)\ ADCK\ cycles)$$

$$Tempo\ de\ conversão = 5\ bus\ clock\ cycles + 315\ ADCK\ cycles$$

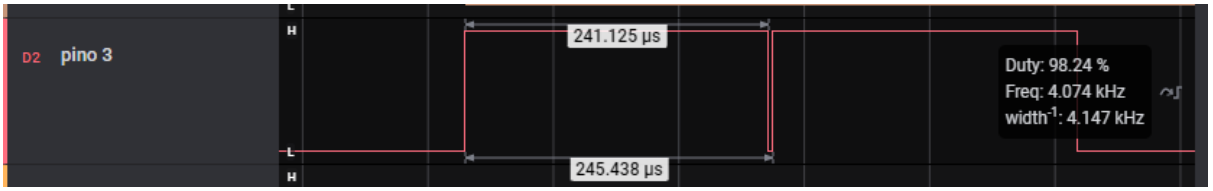
ADC0_CFG1_ADICLK = 0b1 → $input\ clock = bus\ clock / 2$
ADC0_CFG1_ADIV = 0b10 → $ADCK = input\ clock / 4 = bus\ clock / 8$

$bus\ clock = 20.971.520 / 2 = 10.485.760\ Hz$ $\quad ADCK = bus\ clock / 8 = 1.310.720\ Hz$

Conclui-se que

$$Tempo\ de\ conversão = 5 \times (10.485.760\ Hz)^{-1} + 315 \times (1.310.720\ Hz)^{-1} = 240,8\ us$$

No analisador, medimos um tempo de cerca de *241 us*, que é condizente com o valor esperado



4.

Nesta questão, vale destacarmos que o valor inicial da função no programa é de 20971520 (ao contrário do roteiro). Logo, alterando esse valor para 10485760, notamos que o resultado esperado (pela fórmula de estimação) seria de 1s de período de TPM, o que condiz com o que o analisador lógico nos mostra, 999,04575ms, aproximadamente 1s.

5.

Para que não haja sobreposição de conversões, o período do PIT deve ser maior que duas vezes o tempo de conversão mais o tempo entre o fim de uma e início de outra conversão, que apesar de ser muito menor, influencia no resultado. Uma vez que, caso essa condição não seja atendida, ocorrerão inícios de conversões durante processos de conversão em progresso. No caso presente, o período de PIT tem que ser maior que $T_{PIT} > 241 * 2 = 482\ us$.

6.

Seguindo as tabelas e a equação do item 3 desta parte, destacamos inicialmente os valores dos registradores:

ADC0_CFG1_ADLSMP = 0b1 → tempo de amostragem longo
ADC0_CFG2_ADICLK = 0b00

$$SFCAdder = 3\ ADCK\ cycles + 5\ bus\ clock\ cycles$$

ADC0_SC3_AVGE = 0b1
ADC0_SC3_AVGS = 0b11

$$AverageNum = 32$$

ADC0_CFG1_MODE = 0b11
ADC0_SC1A_DIFF = 0b0
então é configurada conversão single-ended 16-bit conversion

$$BCT = 25\ ADCK\ cycles$$

ADC0_CFG1_ADLSMP = 0b1
ADC0_CFG2_ADLSTS = 0b00

$$LSTAdder = 20\ ADCK\ cycles$$

ADC0_CFG2_ADHSC = 0b1

$$HSCAdder = 2\ ADCK\ cycles$$

Temos, por fim, que o tempo de conversão é

$$Tempo\ de\ conversão = SFCAdder + AverageNum * (BCT + LSTAdder + HSCAdder)$$

$$Tempo\ de\ conversão = 3\ ADCK\ cycles + 5\ bus\ clock\ cycles + 32 * (25\ ADCK\ cycles + 20\ ADCK\ cycles + 2\ ADCK\ cycles)$$

$$Tempo\ de\ conversão = 5\ bus\ clock\ cycles + 1507\ ADCK\ cycles$$

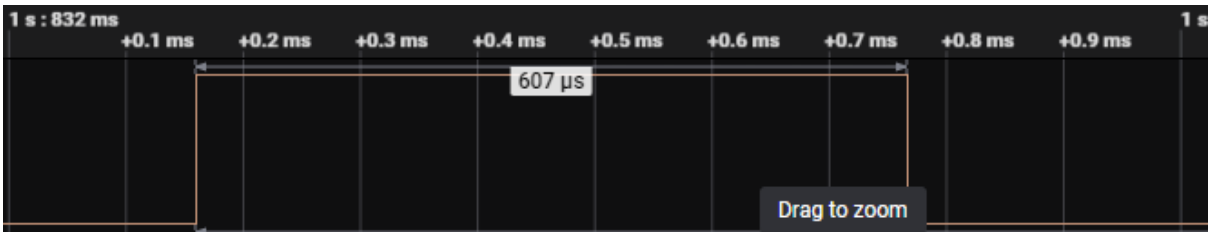
ADC0_CFG1_ADICLK = 0b00 → $input\ clock = bus\ clock$
ADC0_CFG1_ADIV = 0b10 → $ADCK = 4$

$bus\ clock = 20971520/2 = 10.485.760Hz$ $\quad$ $ADCK = bus\ clock/4 = 2.621.440\ Hz$

Conclui-se que

$$Tempo\ de\ conversão = 5 \times (10.485.760\ Hz)^{-1} + 1.507 \times (2.621.440\ Hz)^{-1} = 575,35\ us$$

No analisador, medimos um tempo de cerca de *607us*, que é condizente com o valor esperado



7.

Seguindo as tabelas e a equação do item 3 desta parte, destacamos inicialmente os valores dos registradores:

Como a maioria dos valores foi obtida por meio das especificações, os valores dos registradores listados abaixo são baseados nas especificações, de forma a atender aquilo que foi pedido. Portanto, nem todos os valores escritos abaixo são necessários para a estimativa do tempo de conversão, mas são necessários para configurarmos o rot9_aula com as especificações do projeto controle_cooler.

ADC0_CFG1_ADLSMP = 0b0 → tempo de amostragem curto
ADC0_CFG2_ADICLK = 0b01 → pois, de acordo com o valor de ADCK especificado no roteiro, temos que o bus clock deve ser dividido por 16. Logo, o ADIV deve ser 0b11 para dividir por 8.

$$SFCAdder = 5\ ADCK\ cycles + 5\ bus\ clock\ cycles$$

ADC0_SC3_AVGE = 0b1

ADC0_SC3_AVGS = 0b10

$$AverageNum = 16$$

Foi especificado que é conversão single-ended 16-bit conversion:

ADC0_CFG1_MODE = 0b11
ADC0_SC1A_DIFF = 0b1

$$BCT = 25\ ADCK\ cycles$$

Como foi especificado tempo de amostragem curto, o valor é:

ADC0_CFG1_ADLSMP = 0b0
ADC0_CFG2_ADLSTS = 0b00

$$LSTAdder = 0\ ADCK\ cycles$$

ADC0_CFG2_ADHSC = 0b0

$$HSCAdder = 0\ ADCK\ cycles$$

Temos, por fim, que o tempo de conversão é

$$Tempo\ de\ conversão = SFCAdder + AverageNum * (BCT + LSTAdder + HSCAdder)$$

$$Tempo\ de\ conversão = 5\ ADCK\ cycles + 5\ bus\ clock\ cycles + 16 * (25\ ADCK\ cycles + 0\ ADCK\ cycles + 0\ ADCK\ cycles)$$

$$Tempo\ de\ conversão = 5\ bus\ clock\ cycles + 400\ ADCK\ cycles$$

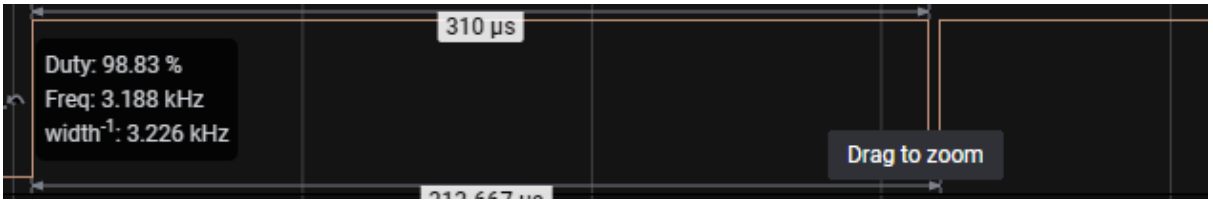
ADC0_CFG1_ADICLK = 0b01 → $input\ clock = bus\ clock/2$
ADC0_CFG1_ADIV = 0b11 → $ADCK = input\ clock/8$

$$bus\ clock = 20971520/2 = 10.485.760Hz \qquad ADCK = 1.310.720\ Hz$$

Conclui-se que

$$Tempo\ de\ conversão = 5 \times (10.485.760\ Hz)^{-1} + 400 \times (1.310.720\ Hz)^{-1} = 305,65\ us$$

No analisador, medimos um tempo de cerca de *310us*, que é condizente com o valor esperado



Terceira parte

1.

O período configurado é de aproximadamente 12,5ms, conforme se nota a seguir:

```
SIM_setaTPMSRC (0b01);
```
→ configura a fonte de relógio de TPM para FLL ou PLL/2;

```
SIM_setaFLLPLL (0);
```
→ configura a fonte de relógio de TPM para FLL;

```
TPM_config_especifica (1, 4095, 0b1111, 0, 0, 0, 0, 0, 0b110);
```
→ entra na função;

```
TPM[x]->MOD = TPM_MOD_MOD(mod);
```
→ configura o campo MOD com o valor do parâmetro mod, que vale 4095;

```
TPM[x]->SC &= ~TPM_SC_PS(0b111);
TPM[x]->SC |= TPM_SC_PS(ps);
```
→ configura o campo PS com o valor do parâmetro ps, que vale 0b110, que se refere ao valor 64 de postscaler;

```
if (cpwms) TPM[x]->SC |= TPM_SC_CPWMS_MASK;
```
→ Seta como 0 o campo CPWMS;

Assim, usando a fórmula a seguir, chegamos ao valor aproximado citado anteriormente. Vale destacar ainda que a contagem está setada como progressiva apenas, pois o valor do campo CPWMS é 0.

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

$$Periodo = 4095 \times \frac{2^{6}}{20.971.520} \times (1 + 0) = 12,497\,ms$$

Verificando com o analisador lógico, vemos que o período foi configurado corretamente.

2.

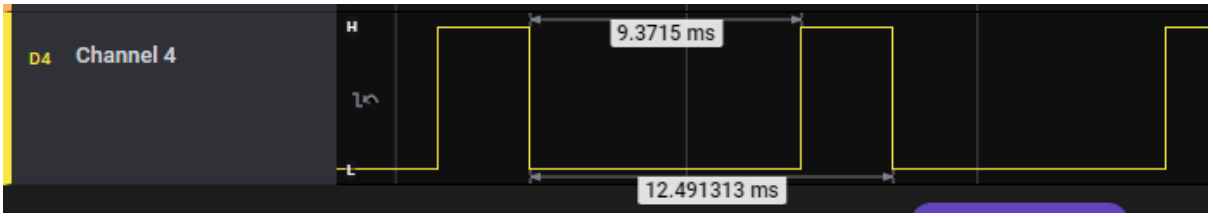
O valor configura a razão de trabalho do PWM, o valor original é ¾ to total, portanto em 75% do período a saída do canal é 1 e no resto ela é zero, isto é visto na figura acima. Alterar este valor muda a proporção de trabalho do pulso PWM, em cada caso, para:

- 0 Sinal sempre em zero, a saída permanece nula e o cooler não gira.
- ¼ MOD 25% do pulso em 1, cooler gira devagar.

- ½ MOD 50% do pulso em 1, cooler gira mais rapidamente
- MOD + 1 Sinal sempre em 1, o cooler recebe toda a tensão da fonte continuamente.

Neste último, a potência máxima é transmitida, e o cooler gira em sua velocidade máxima.

No analisador, vimos os sinais sempre em zero ou sempre em um, nos casos extremos. Nos outros, foi possível observar o ciclo de trabalho desejado. Em ¼ MOD:



Em ½ MOD: