DISCIPLINA EA701
Introdução aos Sistemas Embarcados

ROTEIRO 9: Entradas/Saídas Analógicas
Conversores D/A e A/D, DMA, *Buffer* Circular, Potenciômetro, Sensor de Temperatura LM61, *Joystick*

Profs. Antonio A. F. Quevedo e Wu Shin-Ting

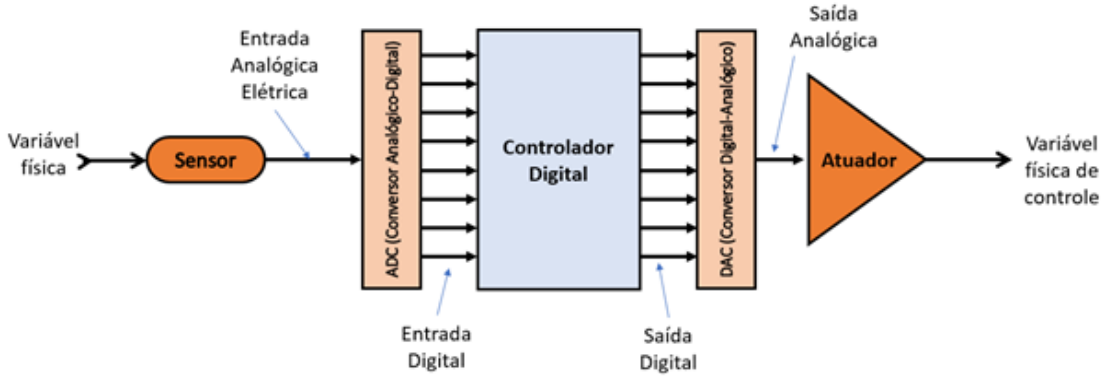
FEEC / UNICAMP

Revisado em outubro de 2024

INTRODUÇÃO

A maioria das grandezas físicas é, por sua natureza, analógica em termos de tempo e valor. Como os microprocessadores operam com sinais digitais, é necessário converter os sinais analógicos do mundo físico em formatos digitais que possam ser interpretados pelos sistemas.

Para aproveitar a capacidade de processamento dos sistemas digitais no controle de dispositivos com base nos sinais captados pelos sensores, é essencial desenvolver circuitos conversores que realizem a transição entre os sinais analógicos e digitais. O circuito responsável por converter sinais digitais em analógicos é chamado de **Conversor Digital-Analógico** (DAC, do inglês *Digital to Analog Converter*). Por sua vez, a conversão de sinais analógicos para digitais é feita pelos **Conversores Analógico-Digitais** (ADC, do inglês *Analog to Digital Converter*). Esses circuitos são fundamentais para integrar informações do mundo físico aos sistemas computacionais, permitindo o controle e processamento digital desses dados. Geralmente, esses conversores estão integrados em uma interface analógica.



Antigamente, os conversores analógico-digitais (ADC) e digitais-analógicos (DAC) eram frequentemente implementados como circuitos integrados individuais encapsulados, contendo todos os componentes necessários para a conversão entre sinais analógicos e digitais. Os DACs geravam sinais analógicos a partir de dados digitais, enquanto os ADCs convertiam sinais analógicos em formato digital para processamento em microcontroladores. Hoje em dia, esses conversores estão cada vez mais integrados diretamente em microcontroladores, sensores e atuadores, otimizando o *design* e a eficiência dos sistemas. Essa evolução permite maior compactação e eficiência nos dispositivos eletrônicos contemporâneos.

Além disso, a implementação de DMA (do inglês *Direct Memory Access*) nos sistemas digitais desempenha um papel crucial na eficiência geral. O DMA permite a transferência de dados entre os conversores e a memória sem a intervenção contínua da CPU, liberando recursos do processador para outras tarefas. Isso resulta em um processamento mais rápido e eficiente, reduzindo a latência e melhorando o desempenho em aplicações que requerem a coleta e o controle de grandes volumes de dados. Ao combinar conversores analógico-digitais e digitais-analógicos com tecnologias de DMA, os sistemas se tornam mais responsivos e capazes de lidar com a complexidade dos sinais do mundo físico de forma eficaz.

PROJETOS-EXEMPLO

Vamos desenvolver quatro projetos neste roteiro. No primeiro, sintetizaremos formas de onda utilizando os DACs do microcontrolador, aplicando duas técnicas diferentes. Durante esse

processo, exploraremos também o funcionamento do DMA. Nos três projetos seguintes, focaremos na leitura de sinais analógicos através do ADC, empregando três abordagens distintas: conversão simples controlada por software, conversão controlada por timer e transferência de dados por interrupção, além de conversão de dois canais sequenciais controlada por timer com transferência de dados via DMA.

Projeto de Síntese de Ondas

Muitas vezes é necessário sintetizar sinais analógicos variantes no tempo. Um exemplo são os geradores de função usados em bancadas de eletrônica. Para que possamos gerar estes sinais, é preciso converter as amostras digitais em sinais analógicos, e uma das maneiras de se fazer isto é usando o Conversor Digital-Analógico (DAC). Mas como podemos viabilizar essa transformação? Vamos aqui explorar duas formas de transferência de amostras digitais para o DAC: direta (com a taxa de transferência controlada por interrupção de *timer*) e via DMA (do inglês, *Direct Memory Access*, com a taxa de transferência controlada pelo *timer*, porém sem sua interrupção). Vamos usar os dois DACs do microcontrolador, cada um usando um dos métodos. No primeiro método, vamos gerar uma onda triangular, com as amostras sendo calculadas a cada interrupção. No segundo método, vamos usar um vetor previamente definido com amostras de uma senóide.

1. Crie um novo projeto usando o *Cube*, com o nome "Sintese_Ondas", **sem inicializar os periféricos**. Ative o *Debug* e gere o código, mantendo o *clock* padrão de 64MHz.

2. Para que se possa gerar a senóide, precisamos usar funções matemáticas que não estão nativas no C, mas estão disponíveis em uma biblioteca padrão. No escopo de `/* USER CODE BEGIN Includes */`, inclua a biblioteca:

```
#include <math.h>
```

3. Defina as constantes que vamos usar. No escopo de `/* USER CODE BEGIN PD */`, crie as macros:

```
#define PI 3.14159265358979
#define SAMPLES 100 // Numero de amortras do vetor
#define DAC_MAX_VALUE 4095 // Valor maximo do vetor (DAC de 12 bits)
#define OFFSET 2048 // "Offset" para somar em todas as amostras (todos os valores positivos)
```

4. Agora vamos definir o vetor que irá guardar as amostras da senóide como variável global, pois ele deve ser acessível a partir da função "main" e também pela função de configuração do DMA. No escopo de `/* USER CODE BEGIN PV */`, declare o vetor:

```
uint32_t sine_wave[SAMPLES]; // Amostras da senoide
```

5. Ainda nesta etapa de preparação, vamos criar os protótipos das funções de configuração dos periféricos que vamos utilizar (DAC, *Timers* e DMA). No escopo de `/* USER CODE BEGIN PFP */`, declare os protótipos das funções:

```
void Config_DAC(void);
void Config_DMA(void);
void Config_Timers(void);
```

6. Agora dentro da função "main", vamos declarar uma variável auxiliar para "varrer' o vetor e definir as amostras da senóide. No escopo de `/* USER CODE BEGIN 1 */`, declare a variável:

```
uint8_t i;
```

9. Antes de configurar os periféricos, vamos calcular as amostras do vetor. No escopo de `/* USER CODE BEGIN 2 */`, escreva o código:

```
// Calcula as amostras da senoide
for(i = 0; i < SAMPLES; i++) {
	sine_wave[i] = (uint32_t)(OFFSET + (OFFSET * sin(2 * PI * i /
SAMPLES)));
	if(sine_wave[i] > DAC_MAX_VALUE) {
		sine_wave[i] = DAC_MAX_VALUE;
	}
}
```

10. Na sequência (logo abaixo da geração das amostras da senóide), chame as funções de configuração dos periféricos. **A ordem de ativação é importante!**

```
Config_DAC();
Config_Timers();
Config_DMA();
```

11. Nada será feito no loop infinito, pois um *timer* irá periodicamente carregar uma nova amostra da onda triangular no primeiro DAC, e o DMA, com a cadência dada por outro *timer*, irá automaticamente "varrer" o vetor, carregando cada amostra no segundo DAC. O que precisamos é definir as funções de configuração dos periféricos, iniciando pelo DAC. No escopo de `/* USER CODE BEGIN 4 */`, escreva o código:

```
void Config_DAC(void) {
	// Habilitar o clock do DAC1 e DAC2
	RCC->APB1LENR |= RCC_APB1LENR_DAC12EN; // DAC1 (2 canais)
	RCC->APB4ENR |= RCC_APB4ENR_DAC2EN; // DAC2
	// Configurando DAC2, canal 1 (sem trigger)
	DAC2->CR = 0; // Inicia com os bits zerados
	DAC2->CR |= DAC_CR_EN1; // Habilitar o canal 1 do DAC2
	// Configurar o DAC1, canal 2 (sem trigger)
	DAC1->CR = 0; // Inicia com os bits zerados-
	DAC1->CR |= DAC_CR_EN2;    // Habilitar o canal 2 do DAC1
	// Configurar PA5 e PA6 para DAC_Out (modo analogico)
```

```
	RCC->AHB4ENR |= RCC_AHB4ENR_GPIOAEN; // Permite configurar o GPIOA
	GPIOA->MODER |= (GPIO_MODER_MODE5_Msk | GPIO_MODER_MODE6_Msk); // MODER
= 11 em PA5 e PA6
}
```

Nessa função apenas habilitamos os canais dos DACs e garantimos que PA5 e PA6 estejam no modo analógico. Note que quando um pino do microcontrolador STM32H7A3 é configurado como analógico, ele automaticamente é atribuído à função analógica definida para aquele pino (DAC ou ADC), sem necessidade de configuração de função alternativa.

O valor padrão de calibração é o valor de ajuste de fábrica, e ele é carregado uma vez que a interface digital do DAC é resetada. Quando as condições de operação diferem das condições de ajuste de fábrica nominais, pode ser feita re-calibração a qualquer momento durante a aplicação por meio de *software*.

12. Ainda no mesmo escopo, vamos definir a função de configuração do DMA, seguindo o procedimento fornecido pelo fabricante:

```
void Config_DMA(void) {
	uint32_t endreg, endvetor;
	endreg = (uint32_t)&(DAC2->DHR12R1); // Endereco do registrador de DAC2
onde se carrega o valor de 12 bits para o canal 1
	endvetor = (uint32_t)sine_wave; // Endereco inicial do vetor com a
senoide
	// TIM6_UP está na entrada de requisição 69 de DMAMUX1 (Tabela 101 do
RM)
	// Saida DMAMUX1 canal 1 ligada a request de DMA1 canal 1 (secao 17.3.2
do RM)
	RCC->AHB1ENR |= RCC_AHB1ENR_DMA1EN; // Habilitar clock do DMA1
	// RM secao 17.4.3 mostra a sequencia para se congifurar o DMA e DMAMUX
	DMA1_Stream1->CR = 0; // Inicia CR com bits zerados
	DMA1_Stream1->M0AR = endvetor; // Endereco da origem
	DMA1_Stream1->PAR = endreg; // Endereco de destino
	DMA1_Stream1->NDTR = SAMPLES; // Numero de transferencias
	DMA1_Stream1->CR |= DMA_SxCR_PL | // Prioridade maxima
			DMA_SxCR_MSIZE_1 | // Elemento da memoria de 32 bits
			DMA_SxCR_PSIZE_1 | // Registrador de periferico de 32 bits
			DMA_SxCR_MINC | // Incrementa memoria
			DMA_SxCR_CIRC | // Buffer circular
			DMA_SxCR_DIR_0; // Da memoria para o periferico
	DMAMUX1_Channel1->CCR = 69; // Associa a request 69 (TIM6 UP) ao canal
1 do DMAMUX1
	DMA1_Stream1->CR |= DMA_SxCR_EN; // Habilita Canal 1 do DMA1
}
```

Aqui configuramos a *stream* (ou canal) 1 do DMA1 com o endereço de origem dos dados (no caso a memória) sendo o endereço da primeira amostra do vetor da senóide. O endereço de destino dos dados é o registrador DHR12R1, que recebe o valor digital a ser convertido. “R1” indica que o canal usado é o 1, e que os dados estão alinhados à direita (“R”). Também definimos o número de transferências a ser executado (no caso são 100 amostras), e depois

fazemos a configuração geral: Prioridade máxima, transferências em 32 *bits* (apesar de usar 12 *bits*, os registradores de dados do DAC são de 32 *bits*), com incremento da posição na memória após cada transferência (para "varrer" o vetor), sem incrementar o endereço do periférico (o registrador de dados não "avança" o endereço), transferências da memória para o periférico e *buffer* circular (ao terminar a transferência, o endereço de origem volta ao valor inicial). Por fim, configuramos o MUX do DMA para associar a fonte de número 69 (*update* do *timer* 6) ao canal 1 do DMA1, e habilitamos o canal correspondente. Assim, a cada *update* do *timer* 6, o DMA transfere o valor da amostra no vetor para o DAC e avança para a próxima amostra. Caso chegue ao final (100 amostras), ele aponta novamente para a primeira amostra, repetindo o ciclo.

13. Agora vamos definir a função de configuração dos *timers*. Vamos usar o *timer* 6 para a cadência da senóide e o *timer* 7 para a cadência da onda triangular. Abaixo da função anteriormente criada, escreva essa nova função:

```
void Config_Timers(void) {
      // Enable clock for TIM6 e TIM7
      RCC->APB1LENR |= RCC_APB1LENR_TIM6EN | RCC_APB1LENR_TIM7EN;
      // TIM6: Faz DMA Request
      TIM6->PSC = 64 - 1; // Dividir clock por 64 -> 1MHz.
      TIM6->ARR = 10 - 1; // Contar até 10 -> 100kHz.
      TIM6->CR1 = 0; // Registradores de controle inicialmente zerados
      TIM6->CR2 = 0;
      TIM6->CR1 |= TIM_CR1_ARPE;  // Habilitar o auto-reload
      TIM6->DIER |= TIM_DIER_UDE;  // Habilitar requisição de DMA em evento
de atualização
      TIM6->CR1 |= TIM_CR1_CEN; // Start timer
      // TIM7: Faz interrupcao periodica
      TIM7->CR1 = 0;
      TIM7->PSC = 64 - 1; // Dividir clock por 64 -> 1MHz
      TIM7->ARR = 20 - 1; // Contar até 20 -> 50kHz
      TIM7->CR1 = 0; // Registradores de controle inicialmente zerados
      TIM7->CR2 = 0;
      TIM7->DIER |= TIM_DIER_UIE;  // Habilitar interrupção de update (UIE)
      TIM7->CR1 |= TIM_CR1_CEN; // Start timer
      // Configurar prioridade da interrupção de TIM7 e habilitar no NVIC
   NVIC_SetPriority(TIM7_IRQn, 1);
   NVIC_EnableIRQ(TIM7_IRQn);
}
```

No *timer* 6, usamos uma frequência de eventos de *autoreload* (*updates*) de 100kHz. Assim, a atualização do valor do DAC correspondente (via DMA) ocorre 100.000 vezes por segundo. Como a senóide possui 100 amostras, o vetor completo (e o ciclo da senóide) será repetido 1.000 vezes por segundo, gerando assim uma senóide de 1kHz. Note que também é necessátio habilitar o *timer* para gerar requisições de DMA.

No *timer* 7. foi realizado o procedimento padrão para interrupção periódica, com um período de 50kHz. Como vamos também gerar a onda triangular com 100 amostras por ciclo, teremos 500 ciclos por segundo, ou seja, uma frequência de 500Hz.

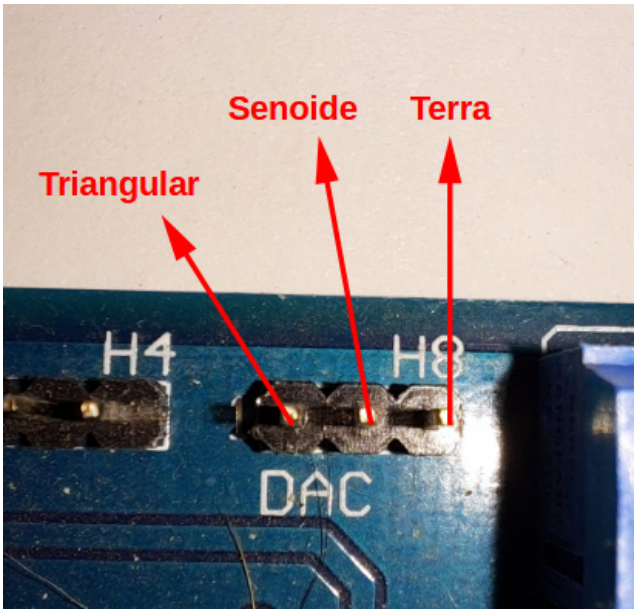
14. Por fim, é necessário definir a ISR para gerar a onda triangular. Abra o arquivo “stm32h7xx_it.c”. Vamos inicialmente criar 2 variáveis dentro do escopo deste arquivo para o cálculo das amostras da onda triangular. No escopo de `/* USER CODE BEGIN 1 */`, implemente a ISR:

```
void TIM7_IRQHandler(void) {
   static int8_t step = 0;
   static int8_t direction = 1;  // 1 para subir, -1 para descer

   // Verificar se a interrupcao e de update
   if (TIM7->SR & TIM_SR_UIF) {
       TIM7->SR &= ~TIM_SR_UIF;  // Limpar o flag de interrupcao
       // Calcular o proximo valor da onda triangular
       step += direction;
       // Verificar limites da onda triangular (0 a 4095)
       if (step >= 50) {
           direction = -1;  // Começar a descer
           step = 50;
       } else if (step <= 0) {
           direction = 1;  // Começar a subir
           step = 0;
       }
       // Ajustar o valor para escala de 12 bits (4096 níveis)
       uint32_t dac_value = (step * 4095) / 50;
       // Atualizar o valor do DAC no canal 2
       DAC1->DHR12R2 = dac_value;
   }
}
```

A ISR cria uma rampa que é ascendente nos 50 primeiros passos e descendente nos outros 50.

15. Faça um *Build*. Conecte as pontas de prova do osciloscópio nos pinos 1 e 2 do conector H8 (DAC). Ligue o terra do osciloscópio no pino 3 do mesmo conector, ou em outro ponto de terra, como por exemplo o pino 7 do conector H9 (ADC). Veja a figura a seguir.



16. Dê um *Debug* e execute o programa. Ajuste o osciloscópio para visualizar as formas de onda. Meça as frequências das mesmas, comparando com o previsto em função do número de amostras e da frequência dos *timers*.

17. Dê um *zoom* nos sinais e note a discretização dos mesmos (pequenos "degraus"). Vamos diminuir o tamanho dos "degraus". Para isso, podemos aumentar a resolução temporal, aumentando o número de amostras. Com isso, como o intervalo de tempo entre amostras é menor, o "salto" no valor de tensão entre amostras também será menor. Para a senoide, a geração de amostras e controle da transferência de dados estão parametrizadas com a macro SAMPLES. Assim, basta mudar o valor de SAMPLES na definição para todo o programa funcionar com o novo valor. Mude o valor de SAMPLES de 100 para 200.

Para fazer o mesmo com a onda triangular, todos os valores "50" dentro da ISR devem ser mudados para "100". Estes valores ocorrem em 3 das linhas de código da ISR, sendo de fácil localização. Verifique se mudou as 3 ocorrências do valor 50 para 100.

18. Recompile e re-execute o programa. Veja agora os "degraus" nas formas de onda. O que mudou? Você consegue explicar essa mudança?

19. Se conectarmos um *buzzer* entre o pino 1 e o terra ou entre o pino 2 e o terra, o que acontecerá? Você consegue explicar o que observou?

Projeto de conversão A/D básica

Você já usou um voltímetro? Sabe como ele funciona? Imagine a possibilidade de criar o seu próprio voltímetro digital usando um conversor ADC! Neste projeto, vamos explorar juntos como transformar uma simples leitura de tensão em um *display* digital. Vamos aprender não apenas sobre o funcionamento do ADC, mas também sobre a conversão de sinais analógicos

em dados que podemos visualizar e interpretar através de um potenciômetro. Pense nas potenciais aplicações: medir a tensão da bateria do seu celular, monitorar circuitos eletrônicos ou até mesmo experimentar com diferentes componentes! A implementação do seu próprio voltímetro digital será uma excelente oportunidade para aplicar conceitos teóricos na prática, desenvolvendo suas habilidades e ampliando seu conhecimento.

1. Crie um novo projeto usando o *Cube*, com o nome "ADC_Basico", **sem inicializar os periféricos**. Ative o *Debug* e gere o código, mantendo o *clock* padrão de 64MHz.

2. Vamos criar duas funções, uma para configurar o ADC e outra para realizar uma leitura no ADC em `main.c`. No escopo de `/* USER CODE BEGIN PFP */`, crie os protótipos das funções:

```
void Config_ADC(void);
uint16_t Le_ADC(void);
```

3. No escopo de `/* USER CODE BEGIN 4 */`, vamos implementar as funções, começando com a de configuração:

```
void Config_ADC(void) {
	//Configurar PC4 como analog
	RCC->AHB4ENR |= RCC_AHB4ENR_GPIOCEN;
	GPIOC->MODER |= GPIO_MODER_MODE4;
	// Habilitar o clock do ADC1
	RCC->AHB1ENR |= RCC_AHB1ENR_ADC12EN;
	// Resetar o ADC1 (garantir que o ADC esteja desabilitado antes de
configurar)
	if (ADC1->CR & ADC_CR_ADEN) {
		ADC1->CR |= ADC_CR_ADDIS;  // Desabilitar o ADC se já estiver
habilitado
		while (ADC1->CR & ADC_CR_ADEN);  // Aguardar até o ADC ser
desabilitado
	}
	ADC1->CR = 0;
	// Desabilitar o deep power down
	ADC1->CR &= ~ADC_CR_DEEPPWD;
	// Habilitar o regulador de tensão do ADC (modo intermediário)
	ADC1->CR |= ADC_CR_ADVREGEN;
	// Aguardar estabilização do regulador de tensão do ADC
	while(!(ADC1->ISR & ADC_ISR_LDORDY));
	// Definir a fonte de ADC clock: clock do sistema/2 (64MHz/2)
	// O registrador eh comum para os 2 modulos
	ADC12_COMMON->CCR &= ~(ADC_CCR_CKMODE);
	ADC12_COMMON->CCR |= ADC_CCR_CKMODE_1;
	// Calibrar o ADC1 (modo de entrada única)
	ADC1->CR &= ~ADC_CR_ADCALDIF;  // Garantir que a calibração seja no
modo single-ended
	ADC1->CR |= ADC_CR_ADCAL;      // Iniciar calibração
	while (ADC1->CR & ADC_CR_ADCAL);  // Aguardar fim da calibração
	// Após a calibração, aguardar a estabilização do ADC
	for(int i = 0; i < 10000; i++);
```

```
	// Configurar o ADC1 para conversão no canal 4
	ADC1->SQR1 = 0;
	ADC1->SQR1 &= ~ADC_SQR1_L;		// Configuração para conversão de 1
canal
	ADC1->SQR1 |= (4 << ADC_SQR1_SQ1_Pos);  // Selecionar canal 4 na
sequência regular
	// Configurar o tempo de amostragem do canal 4 (adequado para precisão)
	ADC1->SMPR1 &= ~ADC_SMPR1_SMP4;    // Limpar configurações anteriores
	ADC1->SMPR1 |= (4 << ADC_SMPR1_SMP4_Pos);  // Amostragem de 32.5 ciclos
de ADC
	ADC1->PCSEL |= (1UL << 4); // Pre-seleciona canal 4
	// Configurar a resolucao (16 bits)
	ADC1->CFGR &= ~ADC_CFGR_RES;
	// Habilitar o ADC1
	ADC1->ISR |= ADC_ISR_ADRDY;		// Limpar flag de prontidão
	ADC1->CR |= ADC_CR_ADEN;		// Habilitar ADC1
	while (!(ADC1->ISR & ADC_ISR_ADRDY));  // Aguardar até o ADC estar
pronto
}
```

Inicialmente, a função configura PC4 como pino analógico, mapeado no canal 4 do ADC1. Depois, aciona o *clock gating* do ADC1. Na sequência, configura o *clock* do sistema (64MHz)/2 como fonte de *clock* para a máquina de estados do ADC. Em seguida, ativa o regulador de tensão do ADC e realiza a autocalibração. Por fim, seleciona o canal 4 para aquisição e habilita o ADC1 para conversões.

4. Agora vamos implementar a função que realiza a conversão A/D e retorna o valor obtido. Após a função anterior, implemente a de leitura:

```
uint16_t Le_ADC(void) {
	// Verificar se o ADC está pronto
	while (!(ADC1->ISR & ADC_ISR_ADRDY)) {}
	// Iniciar conversão de canal único por software
	ADC1->CR |= ADC_CR_ADSTART;
	// Aguardar até a conversão estar completa
	while (!(ADC1->ISR & ADC_ISR_EOC)) {}
	// Verificar se ocorreu um overrun
	if (ADC1->ISR & ADC_ISR_OVR) {
		ADC1->ISR |= ADC_ISR_OVR;  // Limpar a flag de overrun, por
precaucao
	}
	// Ler o valor da conversão
	return (uint16_t)ADC1->DR;
}
```

Ao ser chamada, a função inicialmente verifica se o ADC está disponível para realizar uma conversão (*flag* ADRDY). Depois, dispara uma conversão A/D (setando o *bit* ADSTART) e entra em um *loop* testando o *flag* EOC (*End of Conversion*). Por fim, limpa o *flag* de *overrun* e lê o registrador DR, que armazena o valor da conversão em 16 *bits*, retornando o valor lido. Note que o registrador DR é de 32 *bits*, sendo os 16 menos significativos o resultado da conversão, e por isso é feito um *cast* para 16 *bits* sem sinal no valor a ser retornado.

5. Agora vamos à função “main()”. Inicialmente, vamos declarar variáveis locais. No escopo de `/* USER CODE BEGIN 1 */` , declare as variáveis:

```
uint16_t adc;
float tensao;
```

6. Antes do *loop* infinito vamos configurar o ADC. No escopo de `/* USER CODE BEGIN 2 */`, chame a função de configuração:

```
Config_ADC();
```

e abaixo da linha `/* USER CODE BEGIN 3 */`, escreva o código:

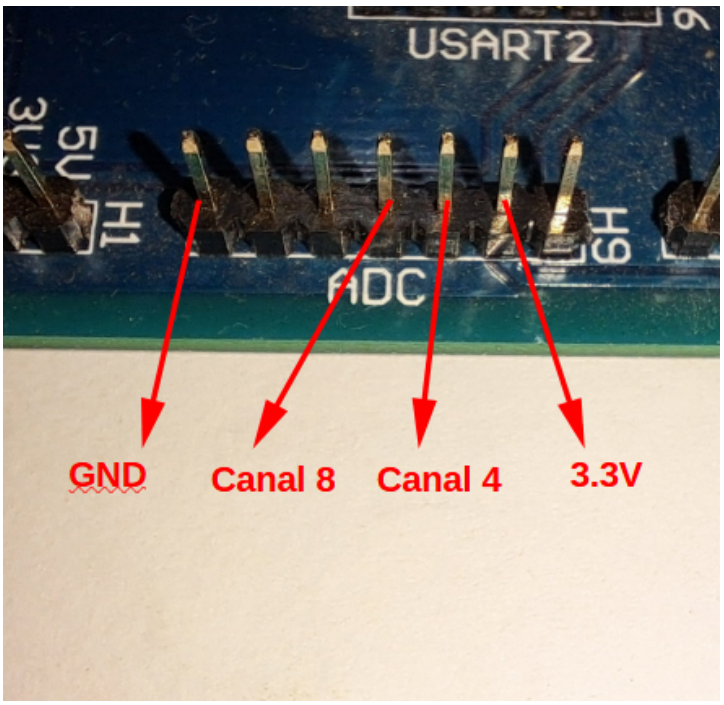
```
adc = Le_ADC();
tensao = (adc * 3.3) / 65535.0;
HAL_Delay(100);
```

A primeira linha realiza a conversão A/D, guardando o resultado na variável “adc”. A segunda linha calcula o valor de tensão correspondente ao valor lido no ADC, sendo o valor resultante expresso em Volts, em uma variável de ponto flutuante. A terceira linha realiza uma espera de 100ms antes de uma nova aquisição de valor analógico.

A equação para cálculo da tensão leva em conta que o valor de “Vrefl” é 0 e o valor de “Vrefh” é a tensão de alimentação, ou seja, 3.3V. Além disso, a resolução da conversão é em 16 *bits*, ou seja, o valor máximo de tensão corresponde ao valor máximo da variável de 16 *bits* (escala cheia).

7. Adicione um *breakpoint* na linha “HAL_Delay(100);”. Faça o *Build* do programa. Depois, conecte o terminal central do potenciômetro no canal 4 (pino 3) no conector H9 (ADC). Ligue um dos terminais laterais do potenciômetro no pino 2 do conector (3.3V) e o outro terminal lateral no pino 7 do conector (GND). Como referência, use a figura abaixo.

OBS: Cuidado para não usar o primeiro pino do conector, pois ele está ligado à fonte de 5V da placa. Se o potenciômetro for ligado neste pino, ele fornecerá tensões entre 0 e 5V, e não de 0 a 3.3V. O conversor A/D do STM32H7A3 aceita um valor máximo de 3.3V em suas entradas analógicas, e valores maiores podem causar danos ao componente.



8. Realize o *Debug*. Abra a aba *Variable* para monitorar o valor das variáveis “tensao” e “adc”, e execute o programa. Quando ele for interrompido (o que vai levar vários segundos), meça a tensão entre o terminal central e o GND do potenciômetro usando o multímetro. Analise o valor das variáveis “adc” e “tensao”, comparando esta última com o valor medido. Mova o potenciômetro para outra posição e dê um *Resume*, experimentando vários valores, sempre comparando a tensão medida no potenciômetro com o valor calculado no programa. Como você explica a fórmula usada para converter o valor “bruto” do ADC em um valor de tensão correspondente?

9. Quais modificações você faria no projeto se mudarmos a resolução de conversão de 16 *bits* para 10 *bits*?

Projeto de conversor A/D disparado por timer, como interrupção de “fime de conversão

Você já pensou na importância de monitorar dados meteorológicos, como a temperatura, de forma precisa e eficiente? Imagine um projeto onde você pode coletar essas informações automaticamente, transformando um simples sinal analógico em dados valiosos. Vamos explorar como obter amostras periódicas de um sensor de temperatura. Inicialmente, você pode fazer uma conversão A/D única, controlando o momento de aquisição pelo *software*, como vimos no projeto anterior. Mas e se quisermos uma coleta contínua, com uma taxa de amostragem consistente? Nesse caso, podemos aprimorar nosso projeto.

Utilizando interrupções de um temporizador, podemos configurar o sistema para que a conversão A/D ocorra automaticamente, reduzindo a latência e garantindo uma taxa de

amostragem estável. Assim, o programa principal se torna mais leve, focando apenas no tratamento dos resultados gerados pela conversão, que serão processados em uma ISR. Agora, imagine que esse sinal analógico vem de um sensor que gera uma tensão proporcional à temperatura medida, como o sensor LM61. Com esse conhecimento, você terá não apenas a capacidade de monitorar a temperatura, mas também a oportunidade de aprender sobre a interação entre *hardware* e *software*, aprofundando suas habilidades em eletrônica e programação.

1. Crie um novo projeto usando o *Cube*, com o nome "ADC_TrigTimer", **sem inicializar os periféricos**. Ative o *Debug* e gere o código, mantendo o *clock* padrão de 64MHz.

2. Como no projeto anterior, vamos criar algumas funções, sendo uma para configurar o ADC, uma para configurar o *timer*, uma para iniciar as conversões periódicas e outra para parar as conversões. Além disso, dentro do arquivo "stm32h7xx_it.c" vamos criar uma função para indicar se há resultado novo no ADC e outra para realizar a leitura do resultado. Para estas duas funções no outro arquivo, precisamos prototipá-las no arquivo "main.c". No escopo de `/* USER CODE BEGIN PFP */`, crie os protótipos das funções:

```
void Config_Timer(void);
void Config_ADC(void);
void Start_Conv(void);
void Stop_Conv(void);

uint8_t ADC_Complete(void);
uint16_t Read_Data(void);
```

3. No escopo de `/* USER CODE BEGIN 4 */`, vamos implementar as quatro primeiras funções, começando com a de configuração de *timer*:

```
void Config_Timer(void) {
	RCC->APB1LENR |= RCC_APB1LENR_TIM6EN; // Habilita clock de TIM6
	TIM6->CR1 &= ~TIM_CR1_CEN; // Desabilita o contador
	TIM6->PSC = 63999; // Prescaler, assumindo clock de 64 MHz, timer a 1
kHz
	TIM6->ARR = 99; // Periodo do timer: 1kHz / 100 = 10Hz
	TIM6->CR2 &= ~TIM_CR2_MMS;
	TIM6->CR2 |= TIM_CR2_MMS_1; // MMS[2:0] = 010: Update Event
}
```

Nesta função estamos configurando o *timer* para realizar um *update* a cada 100ms, ou seja, 10 vezes por segundo. Não estamos ativando a interrupção de *timer*, mas estamos configurando o TIM6 para enviar um sinal de que realizou o *update* como um **evento**. Este sinal de evento é ligado a vários periféricos, podendo servir para iniciar algum processo. No nosso projeto, o *update event* do TIM6 irá disparar automaticamente o início da conversão do ADC1. Note que nesta função, a contagem não foi habilitada. Isto será feito posteriormente.

4. Na sequência, vamos implementar a função de configuração do ADC:

```
void Config_ADC(void) {
	//Configurar PC4 como analog
	RCC->AHB4ENR |= RCC_AHB4ENR_GPIOCEN;
	GPIOC->MODER |= GPIO_MODER_MODE4;
	{
	// Este trecho habilita o pino PC5 configurado para servir de GND
	// digital, evitando jumpers adicionais na conexao do sensor com AGND
	// Se usar AGND, remova este bloco de instrucoes
	GPIOC->MODER |= GPIO_MODER_MODE4;
	GPIOC->MODER &= ~GPIO_MODER_MODE5;
	GPIOC->MODER |= GPIO_MODER_MODE5_0; // PC5 em GPIO output
	GPIOC->OTYPER |= GPIO_OTYPER_OT5; // PC5 open drain
	GPIOC->BSRR = GPIO_BSRR_BR5; // Nivel low
	}
	// Habilitar o clock do ADC1
	RCC->AHB1ENR |= RCC_AHB1ENR_ADC12EN;

	// Resetar o ADC1 (garantir que o ADC esteja desabilitado antes de
configurar)
	if (ADC1->CR & ADC_CR_ADEN) {
		ADC1->CR |= ADC_CR_ADDIS;  // Desabilitar o ADC se já estiver
habilitado
		while (ADC1->CR & ADC_CR_ADEN);  // Aguardar até o ADC ser
desabilitado
	}

	ADC1->CR = 0;
	// Desabilitar o deep power down
	ADC1->CR &= ~ADC_CR_DEEPPWD;
	// Habilitar o regulador de tensão do ADC (modo intermediário)
	ADC1->CR |= ADC_CR_ADVREGEN;
	// Aguardar estabilização do regulador de tensão do ADC
	while(!(ADC1->ISR & ADC_ISR_LDORDY));

	// Definir a fonte de ADC clock: clock do sistema/2 (64MHz/2)
	// O registrador eh comum para os 2 modulos
	ADC12_COMMON->CCR &= ~(ADC_CCR_CKMODE);
	ADC12_COMMON->CCR |= ADC_CCR_CKMODE_1;

	// Calibrar o ADC1 (modo de entrada única)
	ADC1->CR &= ~ADC_CR_ADCALDIF;  // Garantir que a calibração seja no
modo single-ended
	ADC1->CR |= ADC_CR_ADCAL;      // Iniciar calibração
	while (ADC1->CR & ADC_CR_ADCAL);  // Aguardar fim da calibração
	// Após a calibração, aguardar a estabilização do ADC
	for(int i = 0; i < 10000; i++);

	// Configurar o ADC1 para conversão no canal 4
	ADC1->SQR1 = 0;
	ADC1->SQR1 &= ~ADC_SQR1_L;         // Configuração para conversão de 1
canal
	ADC1->SQR1 |= (4 << ADC_SQR1_SQ1_Pos);  // Selecionar canal 4 na
sequência regular
```

```
	// Configurar o tempo de amostragem do canal 4
	ADC1->SMPR1 &= ~ADC_SMPR1_SMP4;    // Limpar configurações anteriores
	ADC1->SMPR1 |= (4 << ADC_SMPR1_SMP4_Pos);  // Amostragem de 32.5 ciclos
de ADC
	ADC1->PCSEL |= 1UL << 4; // Pre-seleciona canal 4

	// Configurar a resolucao (16 bits)
	ADC1->CFGR &= ~ADC_CFGR_RES;

	// Configurar o ADC para disparo externo pelo TIM6 Update Event
	// Reference Manual, Tabelas 194 e  196.
	ADC1->CFGR &= ~(ADC_CFGR_EXTSEL | ADC_CFGR_EXTEN); // Limpar
configuração anterior de trigger
	ADC1->CFGR |= (13 << ADC_CFGR_EXTSEL_Pos); // EXTSEL = 01101 para TIM6
Update Event
	ADC1->CFGR |= ADC_CFGR_EXTEN_0; // Habilitar trigger em borda de subida
(EXTEN = 01)

	//Configurar interrupcao de EOC prioridade 1
	ADC1->IER |= ADC_IER_EOCIE; // Habilita interrupção EOC
	NVIC_SetPriority(ADC_IRQn, 1); // Configura NVIC para interrupcoes do
ADC
	NVIC_EnableIRQ(ADC_IRQn);

	// Habilitar o ADC1
	ADC1->ISR |= ADC_ISR_ADRDY;        // Limpar flag de prontidão
	ADC1->CR |= ADC_CR_ADEN;           // Habilitar ADC1
	while (!(ADC1->ISR & ADC_ISR_ADRDY));  // Aguardar até o ADC estar
pronto
	//Iniciar conversoes
	ADC1->CR |= ADC_CR_ADSTART;
}
```

A configuração do ADC é similar a do projeto anterior. Usamos o mesmo canal com as mesmas configurações. Porém, há uma adição de código após a configuração de conversão no canal 4. Na Tabela 194 do Manual de Referência, pode-se ver que os *bits* EXTEN do registrador ADC_CFGR precisam ter os valores “01” para que o ADC seja disparado por um sinal de *trigger* em borda de subida (ver Figura 491 do Manual, sinal UEV, a borda de subida ocorre exatamente no *update*). A Tabela 196 do Manual mostra que a fonte de *trigger* externo denominada “adc_ext_trg13” está relacionada ao “tim6_trgo” (*Trigger Out* de TIM6, ou seja, seu *update*). Assim, os *bits* EXTSEL do registrador ADC_CFGR devem conter o valor “01101”, ou seja, 13.

Além da configuração do *trigger*, a interrupção de *End of Conversion* do ADC foi habilitada, e programada no NVIC com prioridade 1. Por fim, o *bit* ADSTART do registrador CR deve ser setado. Com o *trigger* por *hardware* configurado, este *bit* agora não inicia a conversão, mas habilita o ADC a responder ao *trigger*.

Por fim, note que PC5 foi configurado como saída GPIO tipo “open drain” em nível baixo. Isto foi feito para que este pino, ao lado de PC4 no conector “ANALOG” da placa auxiliar,

possa ser usado como GND para o sensor de temperatura. Assim, podemos ligar o conector do sensor diretamente no conector da placa. Como a corrente máxima do sensor é de 125 μA, o pino será capaz de funcionar como um "terra" para o sensor. Em algumas situações, isto não é o ideal, pois estamos usando o terra digital no sensor analógico. Porém, para aplicações onde uma alta exatidão não é exigida e as conexões entre sensor e entrada A/D não são longas, pode ser uma boa opção. Nos nossos testes, o ruído máximo estimado no canal foi de 1.5mV. Opcionalmente, podemos usar o pino 2 do conector CN10 (AGND) da placa NUCLEO.

5. Vamos agora implementar as funções para iniciar e parar a conversão periódica. Logo abaixo da função de configuração de ADC, implemente as duas funções:

```
void Start_Conv(void) {
	TIM6->CR1 |= TIM_CR1_CEN; // Habilita o contador
}
void Stop_Conv(void) {
	TIM6->CR1 &= ~TIM_CR1_CEN; // Desabilita o contador
}
```

Estas funções controlam a conversão periódica habilitando e desabilitando a contagem do *timer*. Se o *timer* não conta, não ocorre evento de *update* e o ADC não é disparado.

6. Vamos agora implementar as funções do arquivo "stm32h7xx_it.c". Inicialmente, vamos definir variáveis no escopo do arquivo para auxiliar nas funções. No escopo de `/* USER CODE BEGIN PV */`, declare as variáveis:

```
uint16_t data;
uint8_t complete = 0;
```

7. Agora vamos implementar a ISR de fim de conversão e as funções para a interação com o código do arquivo "main.c". No escopo de `/* USER CODE BEGIN 1 */`, implemente as funções:

```
// ISR do ADC
void ADC_IRQHandler(void) {
	if(ADC1->ISR & ADC_ISR_EOC) { // Flag de End Of Conversion
		ADC1->ISR |= ADC_ISR_EOC; // Limpa flag
		data = ADC1->DR; // Le o dado convertido
		complete = 1; // Flag interno de dado disponivel
	}
}
uint16_t Read_Data(void) {
	complete = 0; // Apaga o flag interno na leitura do dado
	return data;
}
uint8_t ADC_Complete(void) {
	return complete;
}
```

As três funções são bastante simples, dispensando maiores comentários. Vale apenas destacar que, na ISR `ADC_IRQHandler`, **o conteúdo do registrador `ADC1->DR` deve ser armazenado em uma variável. Após cada conversão, o resultado é gravado nesse registrador, sobrescrevendo o valor da amostra anterior.** Portanto, cabe ao programador garantir que, **em cada conversão**, o conteúdo de `ADC1->DR` seja salvo para processamento posterior. Ademais, ao realizar um acesso de leitura deste registrador, o *bit* de estado de `ADC_ISR_EOC` é resetado automaticamente pelo *hardware*, dispensando a necessidade de uma limpeza manual por *software*.

8. Por fim, vamos implementar o código dentro da função "main()". Voltando ao arquivo "main.c", no escopo de `/* USER CODE BEGIN 1 */`, declare um vetor para armazenar múltiplas amostras, uma variável auxiliar para definir o número da amostra, uma variável para acumular as amostras medidas (para cálculo da média de valores), uma para calcular a tensão correspondente à média, e uma para calcular a temperatura medida.

```
uint16_t adc[50];
uint8_t i;
uint32_t media;
float tensao, temp;
```

No escopo de `/* USER CODE BEGIN 2 */`, chame as funções de configuração do ADC e do *timer*:

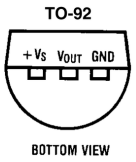
```
Config_ADC();
Config_Timer();
```

9. Agora implemente o código que vai, a cada ciclo do *loop* principal, adquirir 50 amostras pelo ADC e guardá-las no vetor, calcular o valor médio do vetor, a tensão correspondente e a temperatura equivalente, e depois aguardar 1 segundo antes de reiniciar. Abaixo da linha `/* USER CODE BEGIN 3 */`, escreva o código:

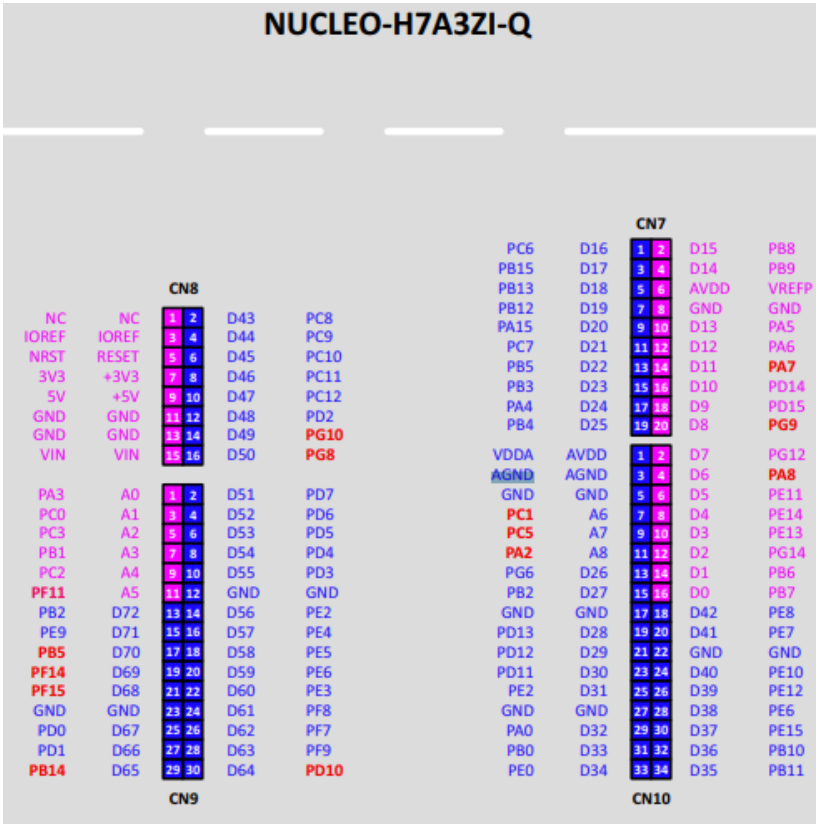
```
media = 0;
Start_Conv();
for(i = 0; i < 50; i++) {
      while(!ADC_Complete()) {}
      adc[i] = Read_Data();
      media += adc[i];
}
Stop_Conv();
media /= 50; // Faz a media das 50 amostras
tensao = (media * 3.3) / 65535.0;
// tensao = (10mV/C x temp) + 600mv
// invertendo: temp = (tensao - 600mv) / (10mV/C)
temp = (tensao - 0.6) / 0.01;
HAL_Delay(1000);
```

e coloque um *breakpoint* na linha do HAL_Delay. Faça o *Build*.

10. Conecte o sensor ao conector H9, utilizando os pinos que foram referenciados com as mesmas denominações do projeto anterior. Ligue o terminal marcado com "+" (+Vs) no pino de 3.3V, o pino central (tensão de saída, Vout) no canal 4 e o último pino (GND) no canal 8 (configurado como um "terra" neste projeto). Opcionalmente, pode-se usar o pino 3 do CN10 (AGND) no lugar do canal 8.



Pinagem do sensor LM61



Execute um *Debug*, iniciando a execução do código. Quando o programa parar no *breakpoint*, veja o conteúdo do vetor "adc", o valor médio, e os valores de tensão média e de temperatura média. Aqueça o sensor na mão e reinicie a execução. Veja novamente o conteúdo das variáveis.

Note aqui que foi usada a fórmula padrão de conversão do valor "bruto" do ADC para tensão de entrada, e depois a fórmula derivada a partir do *datasheet* do sensor de temperatura LM61. Assim, associamos os valores lidos no ADC com a grandeza física medida.

Projeto de conversor A/D disparado por *timer*, com DMA

Imagine um projeto em que você possa captar movimentos em tempo real usando um *joystick*, coletando informações sobre deslocamentos nos eixos X e Y. No projeto anterior, a CPU enfrentou um "gargalo" ao interromper seu fluxo de execução para processar os dados convertidos pelo ADC. Essa abordagem limita a velocidade de conversão e pode prejudicar a responsividade do sistema, especialmente quando lidamos com taxas de amostragem elevadas. Agora, vamos elevar nosso projeto a um novo patamar! Utilizando o recurso de DMA (*Direct Memory Access*), poderemos automatizar a transferência dos dados convertidos para um *buffer*, sem a necessidade de intervenção da CPU. Isso significa que a conversão A/D será feita de maneira mais eficiente, permitindo que você se concentre em implementar outras funcionalidades enquanto os dados são coletados em segundo plano. Além disso, vamos configurar o ADC para converter múltiplos canais, sequenciando automaticamente as leituras. Essa abordagem não apenas aprimorará a eficiência, mas também abrirá um leque de possibilidades para explorar diferentes sensores ou fontes de dados.

1. Crie um novo projeto usando o *Cube*, com o nome “ADC_Joystick”, **sem inicializar os periféricos**. Ative o *Debug* e gere o código, mantendo o *clock* padrão de 64MHz.

2. Como nos projetos anteriores, vamos criar algumas funções, sendo uma para configurar o ADC, uma para configurar o *timer*, uma para configurar o DMA uma para iniciar as conversões periódicas e outra para parar as conversões. Além disso, dentro do arquivo “stm32h7xx_it.c” vamos criar apenas uma função para indicar se há resultado novo no ADC, já que os valores obtidos serão transferidos ao *buffer* diretamente pelo DMA. Para a função no outro arquivo, precisamos prototipá-la no arquivo “main.c”. No escopo de `/* USER CODE BEGIN PFP */`, crie os protótipos das funções:

```
void Config_Timer(void);
void Config_ADC(void);
void Config_DMA(void);
void Start_Conv(void);
void Stop_Conv(void);
uint8_t Frame_Complete(void);
```

3. No escopo de `/* USER CODE BEGIN 4 */`, vamos implementar as cinco primeiras funções, começando com a de configuração de *timer*:

```
void Config_Timer(void) {
	RCC->APB1LENR |= RCC_APB1LENR_TIM6EN; // Habilita clock de TIM6
	TIM6->CR1 &= ~TIM_CR1_CEN; // Desabilita o contador
	TIM6->PSC = 63999; // Prescaler, assumindo clock de 64 MHz, timer a 1
kHz
	TIM6->ARR = 49; // Periodo do timer: 1kHz / 50 = 20Hz
	TIM6->CR2 &= ~TIM_CR2_MMS;
	TIM6->CR2 |= TIM_CR2_MMS_1; // MMS[2:0] = 010: Update Event
}
```

Esta função é idêntica à do projeto anterior, exceto pelo período de *trigger*, que agora é de 20Hz, pois iremos ler dois canais e cada *trigger* inicia a conversão de um deles.

4. Na sequência, vamos implementar a função de configuração do ADC:

```
void Config_ADC(void) {
	//Configurar PC4 e PC5 como analog
	RCC->AHB4ENR |= RCC_AHB4ENR_GPIOCEN;
	GPIOC->MODER |= GPIO_MODER_MODE4 | GPIO_MODER_MODE5;
	// Habilitar o clock do ADC1
	RCC->AHB1ENR |= RCC_AHB1ENR_ADC12EN;
	// Resetar o ADC1 (garantir que o ADC esteja desabilitado antes de
configurar)
	if (ADC1->CR & ADC_CR_ADEN) {
		ADC1->CR |= ADC_CR_ADDIS;  // Desabilitar o ADC se já estiver
habilitado
		while (ADC1->CR & ADC_CR_ADEN);  // Aguardar até o ADC ser
desabilitado
	}
	ADC1->CR = 0;
	// Desabilitar o deep power down
	ADC1->CR &= ~ADC_CR_DEEPPWD;
	// Habilitar o regulador de tensão do ADC (modo intermediário)
	ADC1->CR |= ADC_CR_ADVREGEN;
	// Aguardar estabilização do regulador de tensão do ADC
	while(!(ADC1->ISR & ADC_ISR_LDORDY));
	// Definir a fonte de ADC clock: clock do sistema/2 (64MHz/2)
	// O registrador eh comum para os 2 modulos
	ADC12_COMMON->CCR &= ~(ADC_CCR_CKMODE);
	ADC12_COMMON->CCR |= ADC_CCR_CKMODE_1;
	// Calibrar o ADC1 (modo de entrada única)
	ADC1->CR &= ~ADC_CR_ADCALDIF;  // Garantir que a calibração seja no
modo single-ended
	ADC1->CR |= ADC_CR_ADCAL;      // Iniciar calibração
	while (ADC1->CR & ADC_CR_ADCAL);  // Aguardar fim da calibração
	// Após a calibração, aguardar a estabilização do ADC
	for(int i = 0; i < 10000; i++);
	// Configurar o ADC1 para conversão nos canais 4 e 8
	ADC1->SQR1 = 0;
	//ADC1->SQR1 &= ~ADC_SQR1_L;         // Configuração para conversão de
2 canais
	ADC1->SQR1 |= ADC_SQR1_L_0;
	ADC1->SQR1 |= (4 << ADC_SQR1_SQ1_Pos);  // Selecionar canal 4 na
sequência regular
	ADC1->SQR1 |= (8 << ADC_SQR1_SQ2_Pos);  // Selecionar canal 8 na
sequencia
	// Configurar o tempo de amostragem dos canais
	ADC1->SMPR1 &= ~(ADC_SMPR1_SMP4 | ADC_SMPR1_SMP8);    // Limpar
configurações anteriores
	ADC1->SMPR1 |= (4 << ADC_SMPR1_SMP4_Pos);  // Amostragem de 32.5 ciclos
de ADC
	ADC1->SMPR1 |= (4 << ADC_SMPR1_SMP8_Pos);
	ADC1->PCSEL |= (1UL << 4) | (1UL << 8); // Pre-seleciona canais 4 e 8
```

```
      // Configurar a resolucao (16 bits)
      ADC1->CFGR &= ~ADC_CFGR_RES;
      // Configurar o ADC para disparo externo pelo TIM6 Update Event
      // Reference Manual, Tabelas 194 e  196.
   ADC1->CFGR &= ~(ADC_CFGR_EXTSEL | ADC_CFGR_EXTEN); // Limpar configuração
anterior de trigger
   ADC1->CFGR |= (13 << ADC_CFGR_EXTSEL_Pos); // EXTSEL = 01101 para TIM6
Update Event
   ADC1->CFGR |= ADC_CFGR_EXTEN_0; // Habilitar trigger em borda de subida
(EXTEN = 01)
   //Habilitar DMA
   ADC1->CFGR |= ADC_CFGR_DMNGT; //DMA Circular
      // Habilitar o ADC1
      ADC1->ISR |= ADC_ISR_ADRDY;        // Limpar flag de prontidão
      ADC1->CR |= ADC_CR_ADEN;           // Habilitar ADC1
      while (!(ADC1->ISR & ADC_ISR_ADRDY));  // Aguardar até o ADC estar
pronto
      //Iniciar conversoes
      ADC1->CR |= ADC_CR_ADSTART;
}
```

Esta também é idêntica à do projeto anterior, exceto que agora vamos converter 2 canais ao invés de um. O pino PC5 também é configurado como analógico, correspondendo ao canal 8 do ADC1. Veja a última coluna da tabela a seguir.

Pin/ball name[1] [2]															Pin name (function after reset)	Pin type	I/O structure	Alternate functions	Additional functions
LQFP100 with SMPS	TFBGA100 with SMPS	LQFP144 with SMPS	WLCSP132 with SMPS	UFBGA169 with SMPS	UFBGA176+25 with SMPS	LQFP176 with SMPS	TFBGA225 with SMPS	LQFP64	TFBGA100	LQFP100	LQFP144	UFBGA176+25	LQFP176	TFBGA216					
34	K3	46	K9	J6	N6	52	R5	23	K3	31	43	R3	53	R3	PA7	I/O	FT_ah1	TIM1_CH1N, TIM3_CH2, TIM8_CH1N, DFSDM2_DATIN1, SPI1_MOSI/I2S1_SDO, SPI6_MOSI/I2S6_SDO, TIM14_CH1, OCTOSPIM_P1_IO2, FMC_SDNWE, LCD_VSYNC, EVENTOUT	ADC12_INP7, ADC12_INN3, OPAMP1_VINM
35	H4	47	H7	K6	R6	53	M6	24	G4	32	44	N5	54	N5	PC4	I/O	FT_a	DFSDM1_CKIN2, I2S1_MCK, SPDIFRX1_IN2, FMC_SDNE0, LCD_R7, EVENTOUT	ADC12_INP4, OPAMP1_VOUT, COMP1_INM
36	J4	48	J8	N5	M7	54	N6	25	H4	33	45	P5	55	P5	PC5	I/O	FT_ah1	SAI1_D3, DFSDM1_DATIN2, PSSI_D15, SPDIFRX1_IN3, OCTOSPIM_P1_DQS, FMC_SDCKE0, COMP1_OUT, LCD_DE, EVENTOUT	ADC12_INP8, ADC12_INN4, OPAMP1_VINM

Os *bits* ADC_SQR1_L do registrador ADC_SQR1 correspondem ao número de canais sequenciais usados menos 1. Nos projetos anteriores estes *bits* eram 0000, correspondendo a 1 canal. Aqui eles são ajustados em 0001, correspondendo a 2 canais. Os *bits* `ADC_SQR1_SQx` dos registradores SQRy (y = 1.. 4) correspondem ao canal selecionado como o x-ésimo na sequência de conversões. Assim, pode-se definir cada canal na sequência carregando seu

número nos *bits* correspondentes à posição da sequência. Aqui vamos usar os *bits* `ADC_SQR1_SQ1` e `ADC_SQR1_SQ2` para os canais 4 e 8, respectivamente. Deve-se pré-selecionar todos os canais utilizados no registrador `ADC1_PCSEL`. Finalmente, o *bit* `ADC_CFGR_DMNGT` do registrador `ADC_CFGR` deve ser setado para habilitar o ADC a requisitar o DMA quando uma conversão é concluída.

Além disso, **a interrupção para o evento EOC (do inglês *End Of Conversion*) não foi habilitada para o armazenamento dos valores do registrador `ADC1->DR`**, pois o controlador DMA assumirá este gerenciamento. Veremos mais adiante que **o DMA é configurado para transferir automaticamente o conteúdo do registrador `ADC1->DR` para o vetor `adc` assim que um novo valor é registrado, sem a necessidade de intervenção da CPU**. Essa abordagem simplifica a programação do controle de fluxo de dados.

5. Precisamos ainda configurar o DMA. Logo após as funções anteriores, implemente a função correspondente:

```
void Config_DMA(void) {
	uint32_t endreg, endvetor;
	endreg = (uint32_t)&(ADC1->DR); // Endereco do registrador de dados
ADC1
	endvetor = (uint32_t)adc; // Endereco inicial do vetor de dados

	// ADC1 está na entrada de requisição 9 de DMAMUX1 (Tabela 101 do RM)
	// Saida DMAMUX1 canal 1 ligada a request de DMA1 canal 1 (secao 17.3.2
do RM)
	RCC->AHB1ENR |= RCC_AHB1ENR_DMA1EN; // Habilitar clock do DMA1
	// RM secao 17.4.3 mostra a sequencia para se congigurar o DMA e DMAMUX
	DMA1_Stream1->CR = 0; // Inicia CR com bits zerados
	DMA1_Stream1->M0AR = endvetor; // Endereco da destino
	DMA1_Stream1->PAR = endreg; // Endereco de origem
	DMA1_Stream1->NDTR = SAMPLES; // Numero de transferencias
	DMA1_Stream1->CR |= DMA_SxCR_PL | // Prioridade maxima
			DMA_SxCR_MSIZE_1 | // Elemento da memoria de 32 bits
			DMA_SxCR_PSIZE_1 | // Registrador de periferico de 32 bits
			DMA_SxCR_MINC | // Incrementa memoria
			DMA_SxCR_CIRC | // Buffer circular
			DMA_SxCR_TCIE; // Interrupcao de transfer complete
			// Do periferico para a memoria (DIR = 00)

	//Configurar interrupcao de TC prioridade 1
	NVIC_SetPriority(DMA1_Stream1_IRQn, 1); // Configura NVIC para
interrupcoes do DMA
	NVIC_EnableIRQ(DMA1_Stream1_IRQn);

	DMAMUX1_Channel1->CCR = 9; // Associa a request 9 ao canal 1 do DMAMUX1
	DMA1_Stream1->CR |= DMA_SxCR_EN; // Habilita Canal 1 do DMA1
}
```

Esta função se assemelha à usada no primeiro projeto para configurar o DMA, com algumas diferenças. A primeira é que os endereços do vetor na memória e do registrador de dados do periférico são outros (vetor "adc" e registrador DR do ADC1), assim como o número de amostras a serem transferidas (aqui coincidentemente o valor é o mesmo). Outra diferença está na configuração do registrador CR: os *bits* que definem a direção da transferência agora determinam uma transferência do periférico para a memória (DIR = 00). Além disso, a interrupção de transferência de DMA completa é habilitada, e configurada com prioridade 1 no NVIC. Finalmente, o MUX do DMA é ajustado para associar a *stream* 1 à fonte de requisição número 9, que corresponde ao ADC1 (Tabela 101 do Manual de Referência).

6. As funções "Start_Conv" e "Stop_Conv" são idênticas às do projeto anterior, podendo ser copiadas.

7. Vamos agora implementar a função do arquivo "stm32h7xx_it.c". Inicialmente, vamos definir uma variável no escopo do arquivo para auxiliar na função. No escopo de `/* USER CODE BEGIN PV */`, declare a variável:

```
uint8_t complete = 0;
```

8. Agora vamos implementar a ISR de fim de transferência DMA e a função para a interação com o código do arquivo "main.c". No escopo de `/* USER CODE BEGIN 1 */`, implemente as funções:

```
void DMA1_Stream1_IRQHandler(void) {
	if(DMA1->LISR & DMA_LISR_TCIF1) { // Flag de transfer complete
		DMA1->LIFCR |= DMA_LIFCR_CTCIF1; // Limpa flag
		complete = 1;
	}
}
uint8_t Frame_Complete(void) {
	uint8_t c;
	c = complete;
	complete = 0;
	return c;
}
```

A ISR se comporta de forma semelhante à do projeto anterior. A diferença aqui é que a variável "complete" só assume o valor 1 quando **todas** as amostras forem transferidas (no caso 100, sendo 50 para cada canal). Já a função que passa a informação de transferência concluída para o código em "main.c" automaticamente apaga a variável "complete", pois não existe mais a função para transferência do valor convertido, que realizava esta ação.

9. Voltando ao arquivo "main.c", vamos implementar o código principal. Inicialmente, precisamos de um vetor para armazenar os dados convertidos. Este vetor deve ser global, pois precisa ser visível na função de configuração do DMA. No escopo de `/* USER CODE BEGIN PV */`, declare a variável:

```
uint32_t adc[SAMPLES];
```

e lembre-se de definir o número de amostras no escopo de `/* USER CODE BEGIN PD */`:

```
#define SAMPLES 100
```

10. Dentro da função “main()”, no escopo de `/* USER CODE BEGIN 2 */`, chame as funções de configuração:

```
Config_DMA();
Config_ADC();
Config_Timer();
```

11. Finalmente, vamos definir o código do *loop* principal. Abaixo da linha `/* USER CODE BEGIN 3 */`, escreva o código:

```
Start_Conv();
while(!Frame_Complete()) {}
Stop_Conv();
HAL_Delay(1000);
```

e coloque um *breakpoint* na linha do HAL_Delay. Faça um *Build* no programa.

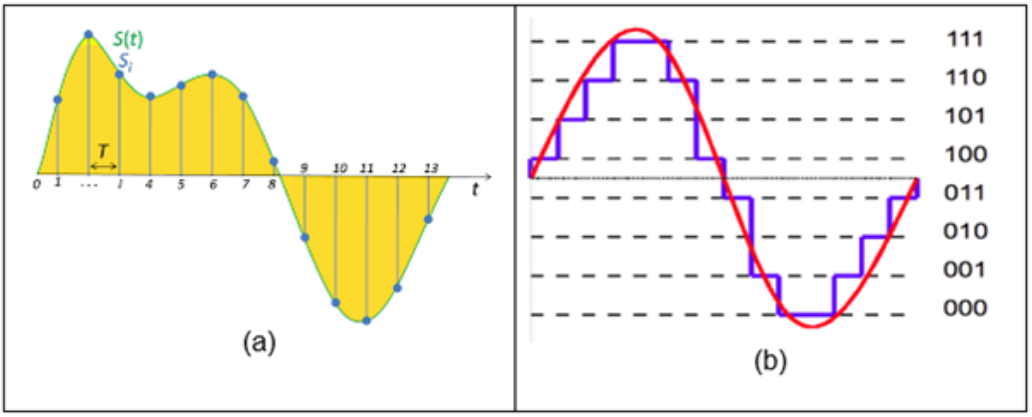
12. Adicione a conexão do outro eixo do *joystick* ao canal 8 do ADC, de acordo com a figura já apresentada no segundo projeto. Observe que o pino +5V do *joystick* deve ser conectado em 3.3V. Transfira o código executável para o microcontrolador no modo *Debug*. Abra a aba “Expressions” e insira a variável “adc”. Continue (“Resume”) a execução do programa. A cada parada no *breakpoint*, examine o conteúdo do vetor “adc”. Experimente mover apenas um eixo de cada vez enquanto executa a conversão A/D e transferência; veja como as conversões dos diferentes canais são organizadas no vetor.

13. Agora que vocês entenderam como captar os deslocamentos de um *joystick* para controlar um objeto, como vocês imaginam que essa tecnologia pode ser aplicada em diferentes contextos do dia a dia? Pensem em jogos, robótica, simulações ou até mesmo em assistências para pessoas com mobilidade reduzida. Quais ideias inovadoras vocês conseguiriam desenvolver utilizando essa abordagem?

QUANTIDADE ANALÓGICA E QUANTIDADE DIGITAL

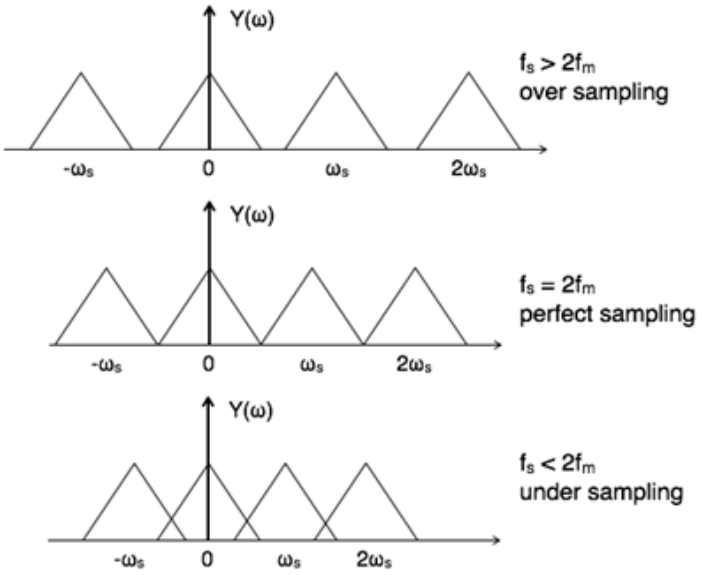
Um sinal analógico é caracterizado por uma variação contínua ao longo do tempo, abrangendo uma infinidade de valores. Para viabilizar o processamento desses valores, deve-se realizar **amostragem** do sinal (figura a), selecionando uma sequência finita de valores que representem de forma significativa o sinal original. Os valores que um sinal analógico pode assumir são contínuos, ou seja, pertencem a qualquer número da reta real. Ao representar esses sinais em sistemas computacionais, é necessário ainda realizar a

quantização (figura b), convertendo os valores contínuos em valores discretos digitalmente representáveis.



Desse modo, um **sinal digital** é um sinal analógico que foi amostrado (discretizado) e quantizado (representado por um valor discreto). Importante observar que tanto no processo de amostragem, que envolve o descarte de amostras, quanto na quantização, que implica uma aproximação de valores, pode haver perda de informações. Assim, desenvolver uma estratégia eficaz de amostragem e quantização torna-se um elemento crucial no projeto de sistemas embarcados, visando preservar a fidelidade das informações durante a conversão analógico-digital. Entre as técnicas que subsidiam o projeto de esquemas de amostragem e quantização na área de Processamento de Sinais, a mais conhecida é o **teorema de amostragem de Nyquist-Shannon**. O teorema assegura que é possível **reconstruir** integralmente um sinal analógico, limitado em banda, a partir de uma sequência de amostras com valores originais, desde que essas amostras sejam obtidas em intervalos menores que $1/(2f_m)$, onde f_m representa a maior frequência, em Hertz (Hz), da banda do sinal original.

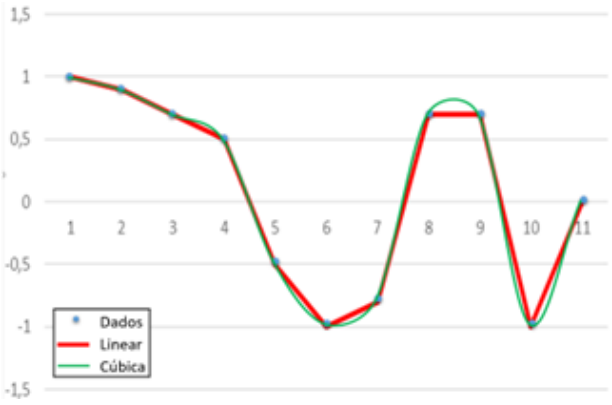
Na figura a seguir, é apresentado o modelo de amostragem no domínio de frequência. No gráfico superior, observa-se uma **superamostragem** (*oversampling*), indicando uma frequência superior a $2f_m$. No gráfico do meio, encontra-se uma **amostragem ideal** com frequência exatamente igual a $2f_m$. Já no gráfico da última linha, é ilustrada uma **subamostragem** (*downsampling*) com uma frequência inferior a $2f_m$. Nesse último caso, ocorre sobreposição de sinais de alta frequência com sinais de baixa frequência, resultando em distorção. Esse fenômeno é conhecido como **falseamento** (*aliasing*). Quando a frequência de amostragem é relativamente baixa, é prática comum aplicar uma filtragem de suavização no sinal original antes de amostrá-lo, removendo assim as componentes de altas frequências.



A fim de aprimorar a resolução dos sinais quantizados, isto é, melhorar a capacidade de distinguir a menor variação Δ, a abordagem consiste em restringir a faixa codificável entre o valor mínimo *m* e o valor máximo *M* (**fundo de escala**). Além disso, é determinado um número *n* de *bits* para representar os códigos binários associados aos valores dentro dessa faixa, pois a relação entre a menor variação diferenciável e essas grandezas pode ser expressa pela seguinte equação:

$$\Delta = \frac{(M-m)}{(2^n-1)}$$

Dado que o sinal digital é uma representação discreta, os resultados da conversão não refletem uma faixa contínua de valores. Para atingir um sinal verdadeiramente contínuo, torna-se essencial realizar uma **interpolação** entre as amostras. A figura destaca duas abordagens de interpolação: linear e cúbica. A interpolação linear, por ser uma técnica mais simples e amplamente adotada, é frequentemente utilizada. No entanto, é necessário permanecer atento ao problema de **falseamento** do sinal mencionado anteriormente.



ATERRAMENTO ANALÓGICO E ATERRAMENTO DIGITAL

No contexto do projeto de sistemas embarcados, a coexistência de sinais analógicos e digitais demanda uma abordagem especial para o aterramento, que geralmente serve como base ou referência comum (potencial elétrico em 0V) para todas as medições de tensão no circuito. Essa necessidade decorre da considerável disparidade nos componentes harmônicos dos sinais analógicos e digitais quando analisados por um analisador de espectro. Os sinais digitais, comumente encontrados em sistemas embarcados, podem abranger frequências extremamente altas, atingindo valores da ordem de megahertz (MHz) a gigahertz (GHz). Durante as transições entre os estados binários “0” e “1”, os sinais digitais experimentam picos de corrente significativos, muitas vezes na faixa de 10 mA a 100 mA. Essa dinâmica de alta frequência e os picos de corrente durante o chaveamento são características intrínsecas aos processos digitais e podem induzir ruídos nos sinais analógicos próximos, comprometendo sua qualidade.

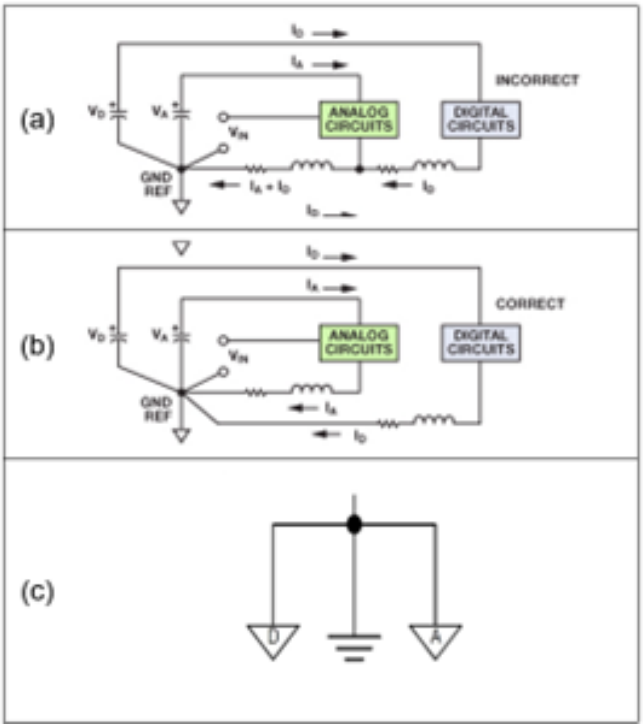
Para mitigar interferências dos sinais digitais nos sinais analógicos, uma abordagem é dividir o sistema misto em dois planos de aterramento: **aterramento analógico** (AGND) e **aterramento digital** (DGND, ou simplesmente GND). Utilizar integralmente uma das camadas de cobre da placa de circuito impresso para o aterramento pode reduzir significativamente a resistência e a indutância no caminho de retorno dos componentes de altíssima frequência dos sinais digitais, aprimorando o desacoplamento.

O desacoplamento eficaz contribui para melhorar a compatibilidade eletromagnética (do inglês, *Eletromagnetic Compatibility* -- EMC) do sistema, garantindo que ele atenda às normas e regulamentações EMC aplicáveis. A adequada gestão do aterramento é crucial para garantir que o sistema atenda aos requisitos das regulamentações e normas EMC aplicáveis. Pithadia e More discutem os papéis desses planos de aterramento em dispositivos de sinais mistos.

O ***crosstalk***, uma interferência entre os componentes digitais e analógicos, pode se manifestar quando há uma proximidade física inadequada entre os planos analógico e digital. Essa interferência indesejada pode também resultar em contaminação dos sinais analógicos por ruídos digitais ou vice-versa, comprometendo a qualidade e a integridade do sinal. Aterramentos separados podem ainda introduzir **laços de terra** indesejados, criando caminhos de corrente indesejados entre dois planos e introduzindo ruídos, afetando a precisão dos sinais analógicos. **Ao conectar os terras analógico e digital através de um filtro, cria-se referências de terra compartilhadas, o que facilita o projeto e evita problemas associados a potenciais diferentes de terra.** Embora a conexão física possa ser benéfica em muitos casos, ela deve ser cuidadosamente planejada e projetada para atender às necessidades específicas do sistema.

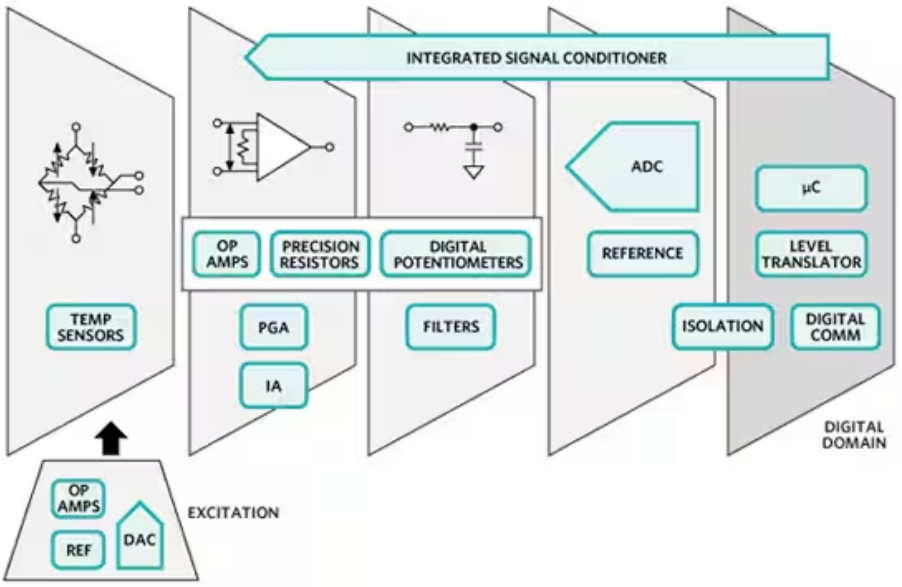
A figura ilustra a questão do aterramento em um circuito que combina sinais analógicos e digitais. Na figura (a), tentou-se isolar os terras analógico e digital por meio de um filtro. No entanto, o terra digital, com todo o seu ruído, está conectado ao caminho de terra analógico, permitindo que a corrente digital flua pelo caminho de retorno do circuito analógico,

resultando em erros na tensão desse circuito. Uma alternativa adequada para ambos os terras seria, como mostrado na figura (b), conectar os dois aterramentos por meio de filtros em um único ponto de referência GND. Assim, todos os sinais e circuitos analógicos utilizariam uma mesma referência de terra, denominada AGND (do inglês, *analog grounding*), enquanto os sinais e circuitos digitais se refeririam a uma referência DGND (do inglês, *digital grounding*). Nos esquemáticos, a existência dessas duas referências deve ser explicitada usando símbolos distintos para representar cada terra, como sugere a figura (c).



ARQUITETURA DE INTERFACE ANALÓGICA EM MICROCONTROLADORES

A arquitetura geral da interface analógica em microcontroladores compreende diversos componentes, cada um desempenhando um papel fundamental no processamento de sinais analógicos. Os principais componentes incluem amplificadores operacionais, filtros, condicionadores de sinal, e os conversores propriamente ditos.

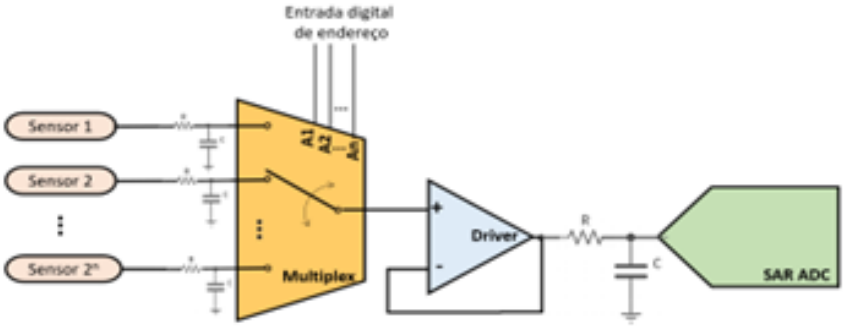


Os **amplificadores operacionais** (OP-AMP) são blocos essenciais na interface analógica, sendo usados para amplificar o sinal analógico de entrada. Sua função principal é aumentar a amplitude do sinal enquanto mantêm características como ganho, largura de banda e resposta em frequência. Eles podem inverter ou não inverter a polaridade.

Os **filtros** são utilizados para modificar ou selecionar determinadas frequências de um sinal analógico. Filtros passa-baixa, passa-alta e passa-banda são comuns na interface analógica para atender a requisitos específicos de filtragem. Por exemplo, um filtro passa-baixa pode ser empregado para atenuar componentes de alta frequência indesejados, enquanto um filtro passa-alta pode eliminar componentes de baixa frequência.

Os **condicionadores de sinal** ajustam e otimizam características específicas do sinal analógico. Isso pode incluir compensação de *offset*, nivelamento de ganho, eliminação de ruídos e calibração. Esses dispositivos garantem que o sinal analógico esteja em conformidade com os requisitos de precisão e qualidade necessários para o processamento digital subsequente.

Os **multiplexadores** (MUX) e **demultiplexadores** (DEMUX) são frequentemente empregados para direcionar seletivamente sinais para diferentes partes do circuito em aplicações com múltiplos canais de entrada/saída. Isso é útil em sistemas com sensores ou fontes analógicas diversas. A figura a seguir ilustra várias fontes de sinais analógicos para uma interface analógica. Nesta figura um multiplexador analógico coloca em sua saída o sinal analógico da sua entrada selecionada pelo código binário colocado em sua entrada digital de endereço.



PARÂMETROS DE CONVERSORES

No âmbito do projeto de sistemas digitais, uma compreensão aprofundada da implementação de um circuito DAC ou ADC não se mostra essencial, dada a vasta gama de circuitos integrados prontamente disponíveis, apresentando diversos encapsulamentos para uma integração facilitada em projetos. O foco do engenheiro projetista recai, em vez disso, na familiaridade com os parâmetros fornecidos pelos fabricantes, uma vez que estes desempenham um papel crucial na seleção criteriosa do componente mais adequado para uma aplicação específica. Essa relevância é acentuada pela estreita relação entre os transdutores e os conversores A/D e D/A, sublinhando a importância de uma escolha cuidadosa para assegurar a eficácia do sistema. Os parâmetros comumente encontrados nas folhas de dados dos fabricantes incluem:

Resolução: corresponde ao tamanho de degrau em tensão de um incremento binário no *bit* menos significativo, (LSB, do inglês *Least Significant Bit*). Tipicamente, a resolução percentual é especificada em termos de quantidade n de *bits* utilizada para representar um valor digital e o fundo de escala FS.

$$resolução\ percentual = \frac{\frac{FS}{2^n - 1}}{FS} = \frac{1}{2^n - 1}$$

Precisão: diz respeito ao grau de variação de resultados de uma medição. Entre os fabricantes de DACs, há duas formas mais usuais para especificar a precisão: erro de fundo de escala e erro de linearidade. O **erro de fundo de escala** é o desvio máximo da saída do DAC do valor esperado, expresso em percentagem do fundo de escala. A razão entre o erro do fundo de escala $\mathcal{E}_{fs}$ e o fundo de escala A_{fs} é também conhecida por **faixa dinâmica**, usualmente expressa em decibéis. Observe que a faixa de valores que um conversor AD consegue resolver é também conhecida como faixa dinâmica.

$$faixa\ dinâmcica = -20 \log \frac{\varepsilon_{fs}}{A_{fs}} dB.$$

O **erro de linearidade** é o desvio máximo no tamanho do degrau em relação tamanho esperado, também expresso em percentagem do fundo de escala. É importante observar que a resolução e a precisão (faixa dinâmica) devem ser compatíveis para evitar desperdício de recursos.

Erro de *offset* e erro de ganho: referem-se, respectivamente, ao valor de saída do conversor quando todos os *bits* de entrada estão em "0" e à diferença entre a entrada real e a entrada esperada necessária para que a saída alcance o fundo de escala. O erro de *offset*, se não corrigido, é adicionado à saída do DAC em todas as condições de entrada, resultando em uma leitura constante e incorreta. Para mitigar esses problemas, muitos DACs oferecem ajustes de *offset* e ganho. O ajuste de *offset* garante que a saída seja nula quando todos os *bits* de entrada estão em "0", enquanto o ajuste de ganho assegura que a saída atinja seu valor máximo quando a entrada atinge a condição esperada. Essa calibragem é essencial para garantir que os erros de *offset* e ganho permaneçam dentro de limites aceitáveis ao longo do tempo.

Monotonicidade: a saída aumenta conforme o número binário na entrada é incrementado.

Tempo de estabilização: corresponde ao tempo necessário para a saída do DAC estabilizar dentro do meio degrau ($\mp 1/2$) do tamanho de degrau do seu valor de fundo de escala, após uma alteração no número binário na entrada.

Tempo de conversão: é o intervalo de tempo entre o início e o fim de uma conversão. Este tempo varia muito entre os conversores. É tipicamente fornecido pelos fabricantes.

Relação sinal-ruído (SNR, do inglês Signal-to-Noise Ratio): é razão entre a raiz do valor quadrático médio (RMS, do inglês *Root Mean Square*) da amplitude do sinal físico real e a raiz do valor quadrático médio da soma de todos os componentes espectrais, exceto as 6 primeiras harmônicas e a componente DC. É um parâmetro comumente associado a conversores analógico-digitais (ADC) e digital-analógico (DACs), indicando a qualidade da conversão. Um SNR alto indica que o sinal desejado é significativamente mais forte do que o ruído presente, o que resulta em uma conversão mais precisa e confiável. Um SNR baixo pode levar a erros na leitura e na interpretação do sinal. Em aplicações de áudio, por exemplo, um DAC com alta SNR permitirá que mais detalhes sutis da música sejam percebidos. Em sistemas de medição, um ADC com SNR elevado assegura que as medições sejam precisas e confiáveis. O SNR é frequentemente expresso em decibéis (dB), permitindo uma comparação mais fácil entre diferentes sistemas. Um SNR de 20 dB, por exemplo, indica que o sinal é 10 vezes mais forte que o ruído.

Consumo de Energia: É um parâmetro importante, especialmente em sistemas embarcados, onde a eficiência energética é fundamental. Um baixo consumo de energia pode prolongar a vida útil da bateria e reduzir os custos operacionais. Muitos conversores AD e DA oferecem modos de operação de baixo consumo que podem ser ativados em condições específicas, como quando não estão em uso, contribuindo para a eficiência energética geral do sistema.

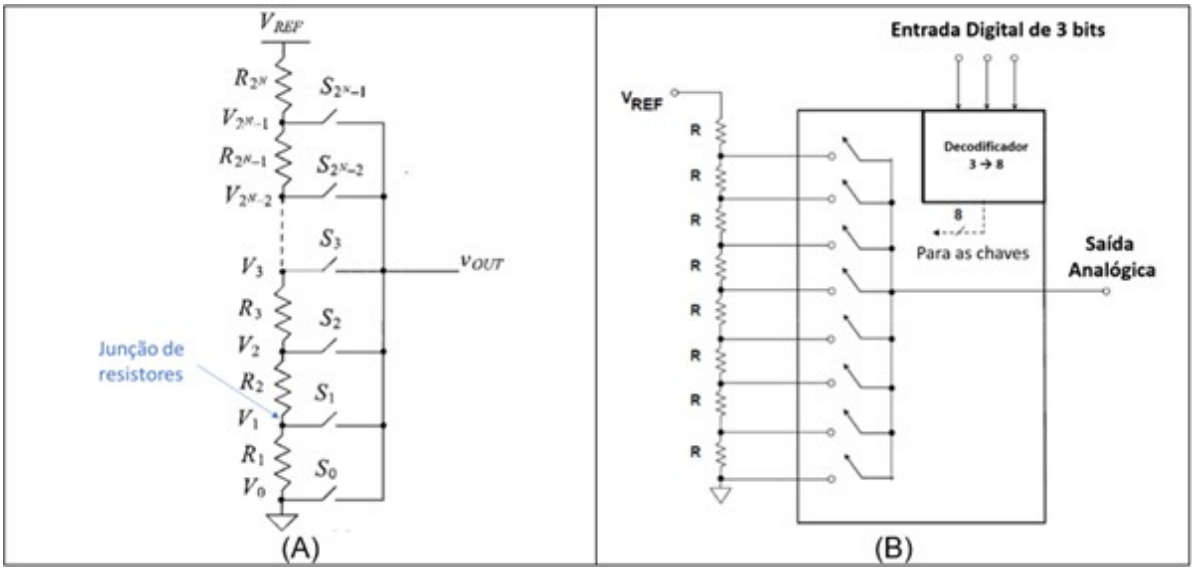
CONVERSORES DAC

Um **conversor digital-analógico** (DAC, do inglês *Digital-to-Analog Converter*) é um circuito projetado para transformar um sinal digital, comumente representado por códigos binários, em um sinal analógico, geralmente na forma de corrente ou tensão. Analogamente, pode-se conceber um conversor D/A como um potenciômetro controlado em passos discretos, cuja saída corresponde a uma fração da saída de fundo de escala ($A_{f\,s}$), determinada pela fonte de alimentação.

A implementação de um DAC pode ser feita com diversas tecnologias, cada uma com características únicas para atender a requisitos específicos. Uma abordagem comum é a utilização de DACs de comutação resistiva, baseados em resistores ponderados, nos quais a corrente ou tensão de saída é determinada por uma rede de resistores ponderados de maneira proporcional ao valor binário de entrada. Uma variante dessa topologia são os DACs com cadeia de resistores. Outra tecnologia amplamente empregada é a arquitetura R-2R, que utiliza uma rede de resistores em configuração escalonada, simplificando a implementação e proporcionando maior precisão em alguns casos. Além disso, DACs baseados em técnicas de comutação de carga, envolvendo o chaveamento de correntes de carga em uma série de capacitores, oferecem eficiência maior em termos de velocidade de conversão, sendo ideais para aplicações de alta frequência. Os mais encontrados no mercado são os de arquitetura R-2R, como o DAC0808 e o AD7524.

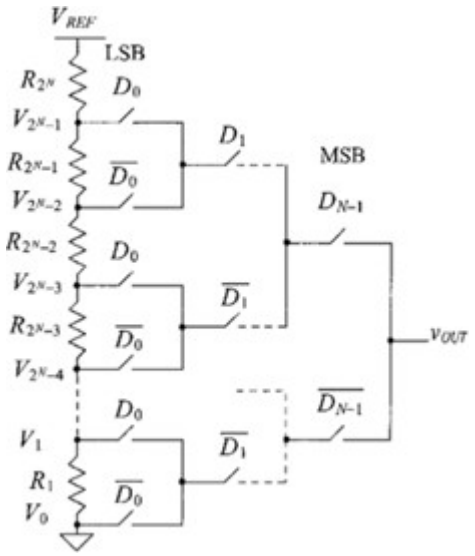
Resistor String DAC

A arquitetura mais elementar de conversão Digital-Analógica é ilustrada na figura (A) a seguir e é denominada Resistor *String* DAC, ou simplesmente *String* DAC. Essa configuração consiste em uma cadeia de 2^N resistores (R_j) de igual valor e 2^N chaves idênticas (S_i), as quais conectam as diversas junções de resistores à saída do circuito (V_{OUT}). Em qualquer instante, apenas uma chave está ativa, enquanto as demais permanecem inativas. A ativação de uma chave pode depender da decodificação do valor binário que será convertido em sinal analógico, como mostra a figura (B). Assim, a saída analógica é obtida pela divisão de tensão na junção associada à chave ativa.

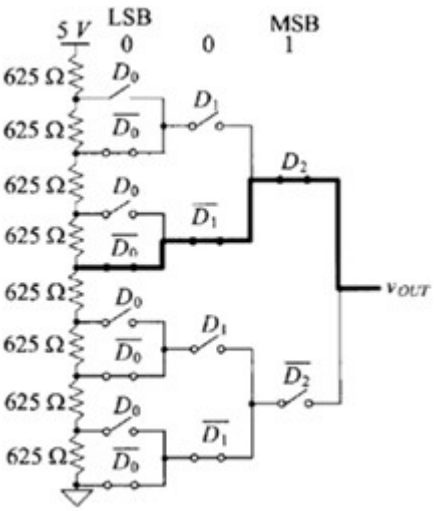


–

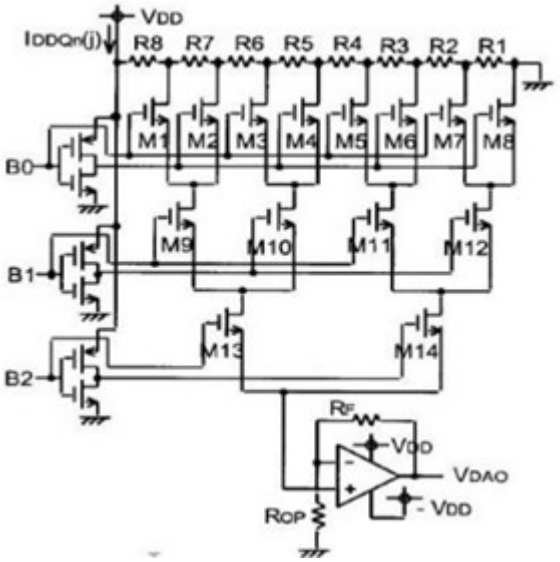
Uma grande vantagem da arquitetura de resistores em cadeia é a garantia de monotonicidade em sua saída. Contudo, um desafio associado a conversores com essa construção é a conexão permanente de 2^N-1 chaves desligadas e uma chave ligada à saída. Em casos de resoluções mais elevadas, as chaves inativas introduzem uma significativa capacitância parasita no nó de saída, resultando em uma limitação na velocidade de conversão, que, portanto, deve ser reduzida. Uma alternativa mais eficaz ao *String* DAC é ilustrada na figura que se segue. Nesse caso, uma árvore binária de chaves é adotada, garantindo que a saída esteja conectada a, no máximo, N chaves ligadas e N chaves desligadas. Essa estratégia resulta em uma redução significativa da capacitância parasita em comparação com a configuração mostrada anteriormente, o que possibilita um incremento na velocidade de conversão do DAC. Na entrada dessa árvore de chaves, uma palavra binária é utilizada, onde cada *bit* exerce controle sobre as chaves de um mesmo nível da árvore. Quando um *bit* de índice i é 1, as chaves D_i são ativadas; quando esse mesmo *bit* é 0, as chaves D_i ficam desativadas. O processo de decodificação é inerente ao arranjo das chaves, simplificando a operação do DAC.



Para ilustrar o funcionamento de um *string*-DAC com chaves em árvore, considere o exemplo de um conversor de três *bits* mostrado na seguinte figura. Para um valor de entrada binária igual a 0b100, as chaves estarão ligadas como mostrado com linha grossa. Para este valor de entrada, a saída será igual a metade da tensão de referência (5V).

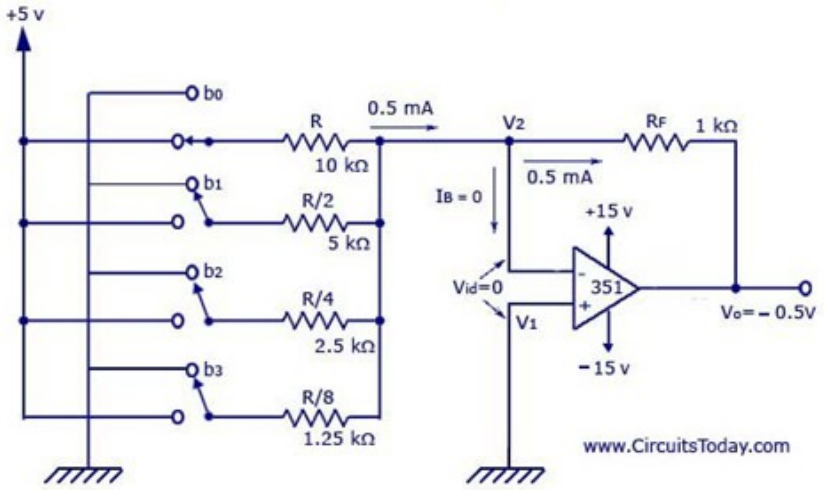


Um outro problema com o *string*-DAC é o compromisso entre a área e a dissipação de potência. Um circuito integrado com este conversor com alta resolução leva a uma grande área de pastilha dedicada ao grande número de resistores necessários. Fazendo o valor de *R* pequeno para minimizar a área necessária, a dissipação de energia se torna o problema crítico, pois a corrente sempre flui através da cadeia de resistores. Uma alternativa para enfrentar esse desafio é substituir os resistores por transistores MOS, como ilustrado na figura que se segue. Essa abordagem visa superar as limitações associadas à dissipação de potência, proporcionando uma solução mais eficiente em termos de área e consumo de energia.



Circuito DAC com resistores ponderados

Conforme a necessidade de maior precisão e resolução aumentou, a abordagem de resistores ponderados ganhou destaque. Essencialmente, o circuito consiste em gerar para cada *bit* do código binário de entrada um montante de sinal analógico e somá-lo, ponderado pela sua posição no código, através de um amplificador operacional, como mostra o esquema a seguir.



Nesta figura tem-se $A_{fs} = 7.5V$ e os resistores de diferentes resistências foram usados para ponderar a corrente de cada ramo, de maneira que a soma das correntes no nó V_2 seja

$$I_2 = b_0\left(\frac{5V}{R}\right) + b_1\left(\frac{5V}{R/2}\right) + b_2\left(\frac{5V}{R/4}\right) + b_3\left(\frac{5V}{R/8}\right) = (b_0 + b_1\times 2 + b_2\times 4 + b_3\times 8)\times\left(\frac{5V}{R}\right)$$

Definindo $K = (5V/R)$ como o **fator de proporcionalidade** do conversor D/A, onde 5V é o valor da **fonte de referência** V_{ref}, chegamos a uma expressão de proporcionalidade linear entre a entrada digital (chaves em posição conectada a 5V, "1", ou conectada ao Terra, "0") e a saída analógica (soma de correntes). Um amplificador operacional é acoplado ao ponto V_2 para amplificar a tensão de saída, operando como um ***buffer*** **de saída**. Portanto, a tensão de saída *Vo* assume uma expressão análoga:

$$V_o = -R_f I_2 = -R_f \times (b_0 + b_1\times 2 + b_2\times 4 + b_3\times 8)\times\left(\frac{V_{ref}}{R}\right) = -(b_0 + b_1\times 2 + b_2\times 4 + b_3\times 8)\times\left(\frac{V_{ref}\times R_f}{R}\right)$$

sendo neste caso o fator de proporcionalidade $K = \frac{V_{ref}\times R_f}{R}$.

Observe que a menor variação representável ΔV_o é igual ao peso do *bit* menos significativo b_0

$$\Delta V_o = b_0 \times \frac{5V\times R_f}{R} = 1\times K = K$$

Como ΔV_o corresponde à diferença de saída analógica de um degrau de incremento no sinal de entrada digital, a variação é conhecida como o **tamanho do degrau**. É também denominada a **resolução** do conversor D/A, porque, se a entrada digital for representada por n bits, então a saída de fundo de escala será quando todos os n bits estiverem em 1.

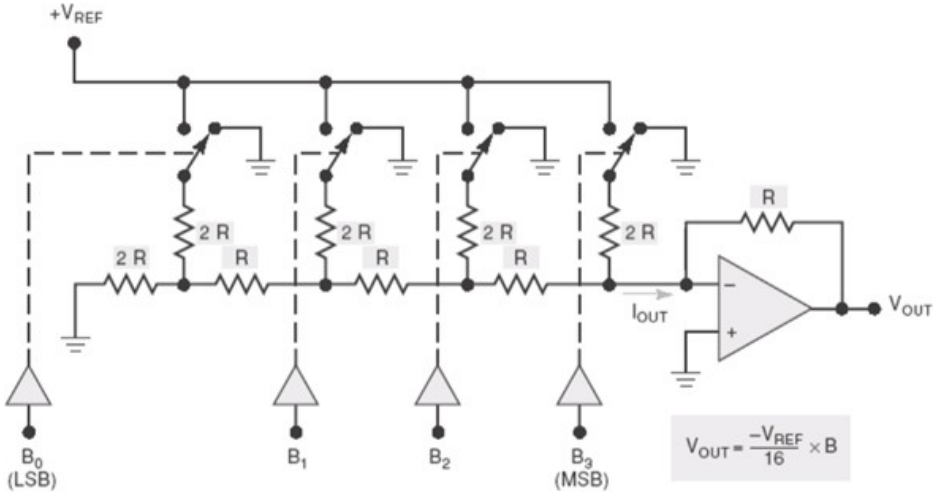
$$A_{fs} = K(2^n - 1) \rightarrow K = \frac{A_{fs}}{(2^n - 1)}$$

A precisão do circuito DAC com resistores ponderados depende da precisão dos resistores R, do resistor de realimentação R_f e da fonte de referência.

O principal problema do esquema paralelo usando resistores como pesos de ponderação é o aumento da faixa de variação das resistências quando se deseja construir um conversor de alta resolução (com *n* muito grande). Por exemplo, para *n*=12, a razão entre a maior resistência e a menor resistência pode ser 2^{11}, sendo difícil implementar tantos resistores de valores distintos mantendo a precisão dos mesmos. Embora essa arquitetura não seja comumente encontrada em aplicações comerciais devido a esses desafios práticos, o entendimento dos princípios subjacentes é fundamental para o desenvolvimento de conhecimentos em princípio de conversão D/A usando circuitos eletrônicos.

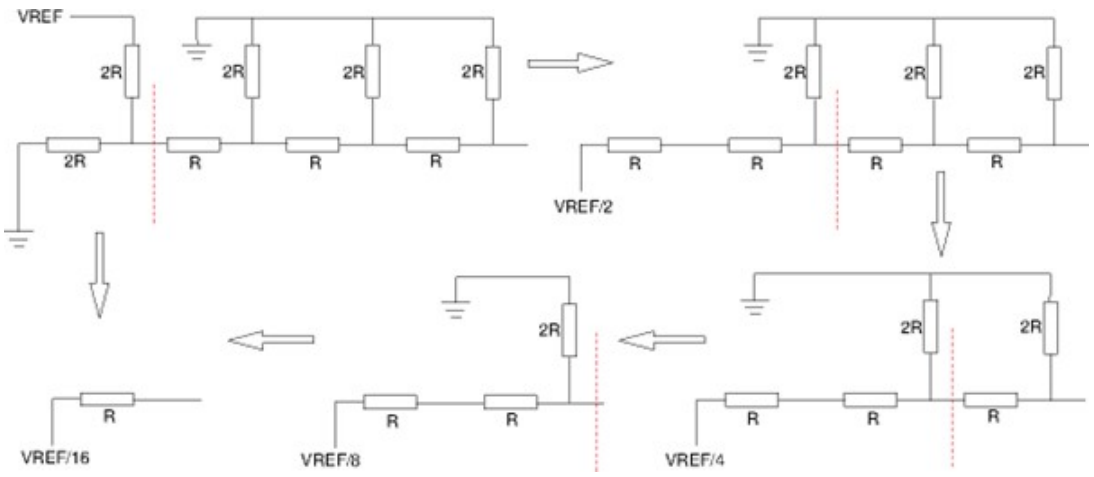
Circuito DAC com rede resistiva R/2R

Um circuito alternativo ao circuito DAC com resistores ponderados é a **rede R/2R** que envolve somente dois valores de resistências R e 2R, como mostra a figura que se segue.

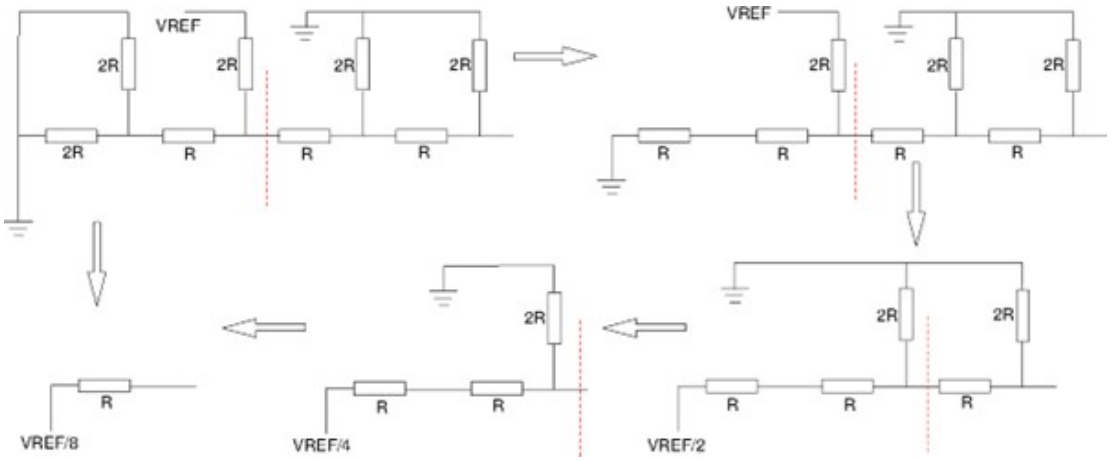


Para analisar a rede, utilizaremos o **Teorema de Sobreposição** e **circuitos equivalentes de Thévenin** para a fonte *VREF* em cada ramo de *bit* (em 1) vistos da entrada negativa do amplificador operacional.

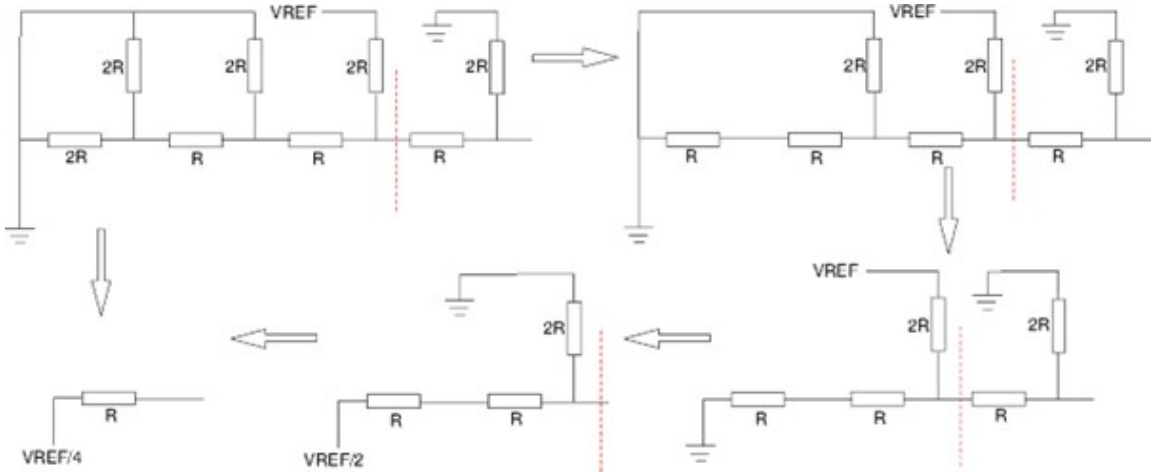
Para o *bit* b_0=1, o circuito equivalente de Thévenin é uma tensão *VREF*/16 com um resistor série equivalente *R*, conforme mostra o processo de redução esquematizado.



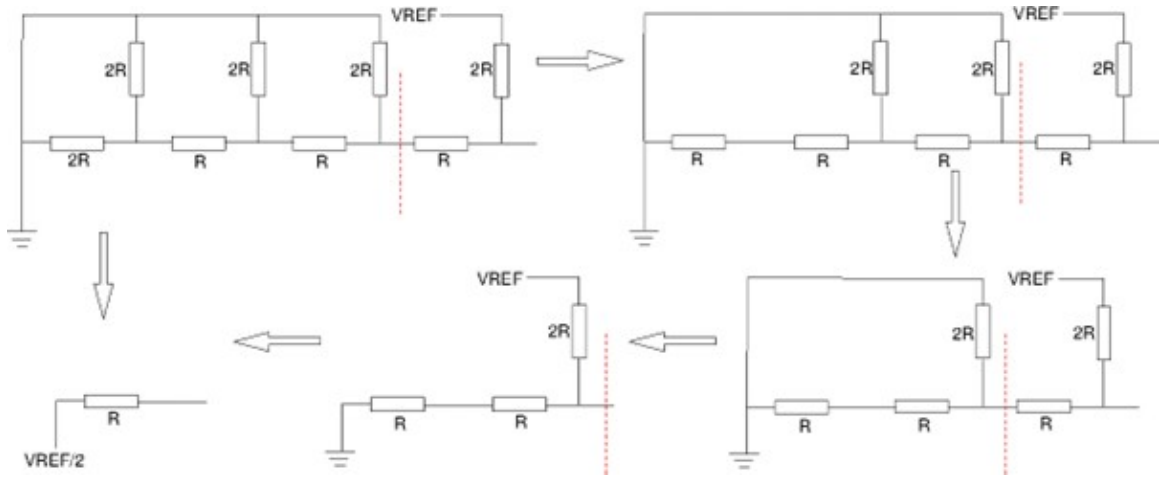
Para o *bit* b_1=1, o circuito equivalente de Thévenin é uma tensão *VREF*/8 com um resistor série equivalente *R*, conforme mostra o processo de redução a seguir.



Para o *bit* b_2=1, o circuito equivalente de Thévenin é uma tensão *VREF*/4 com um resistor série equivalente *R*, conforme mostra o seguinte processo de redução.



Finalmente, para o *bit* b_3=1, o circuito equivalente de Thévenin é *VREF*/2 com um resistor série equivalente R, conforme ilustra o seguinte processo de redução.



Pelo Teorema de Sobreposição, a tensão na entrada negativa do amplificador operacional é a soma das tensões de cada ramo:

$$V = b_0\left(\frac{VREF}{16}\right) + b_1\left(\frac{VREF}{8}\right) + b_2\left(\frac{VREF}{4}\right) + b_3\left(\frac{VREF}{2}\right) = \left(b_0 + 2b_1 + 4b_2 + 8b_3\right) \times \left(\frac{VREF}{16}\right)$$

Sendo R a resistência equivalente, a corrente I_{OUT} é:

$$I_{OUT} = \frac{V}{R} = \frac{\left(b_0 + 2b_1 + 4b_2 + 8b_3\right) \times \left(\frac{VREF}{16}\right)}{R},$$

pois a entrada negativa do amplificador operacional está virtualmente no potencial TERRA. A realimentação negativa do amplificador operacional força uma corrente igual a I_{OUT} pelo resistor R de realimentação, de forma que a tensão V_{OUT} seja

$$V_{OUT} = -\left(b_0 + 2b_1 + 4b_2 + 8b_3\right) \times \left(\frac{VREF}{16}\right) = B \times \left(-\frac{VREF}{16}\right),$$

onde B é a entrada digital.

Um exemplo de circuito integrado de conversão digital-analógico (DAC) comercial, que utiliza a arquitetura de rede resistiva R/2R, é o DAC7571 da *Texas Instruments* (*Texas Instruments*, 2014).

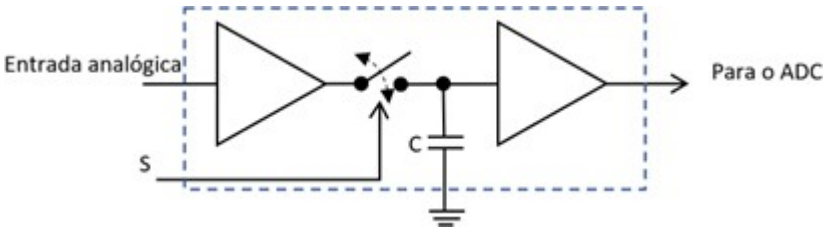
CONVERSORES ADC

Um **conversor analógico-digital** (em inglês *Analog-to- Digital Converter* – ADC) é um circuito projetado para transformar um sinal analógico, geralmente expresso como corrente ou tensão, em um sinal digital representado por códigos binários. Esse processo implica as etapas de amostragem, capturando instantâneos do sinal em intervalos de tempo definidos, e quantização, convertendo esses instantâneos em valores digitais discretos. Todas as conversões são iniciadas por disparos (em inglês, *triggers*), que podem ser por *software* or por *hardware*. Embora esses conversores estejam disponíveis comercialmente, compreender as técnicas

empregadas em sua implementação é fundamental para uma compreensão aprofundada dos parâmetros envolvidos na especificação do fabricante. Isso facilita a seleção apropriada do ADC para uma aplicação específica. Antes de explorar as variadas tecnologias utilizadas na implementação de um conversor analógico-digital, é pertinente apresentar um módulo auxiliar crucial para esses conversores, o **módulo de amostragem e retenção** (S/H).

Módulo de amostragem-e-retenção (*sample-and-hold*)

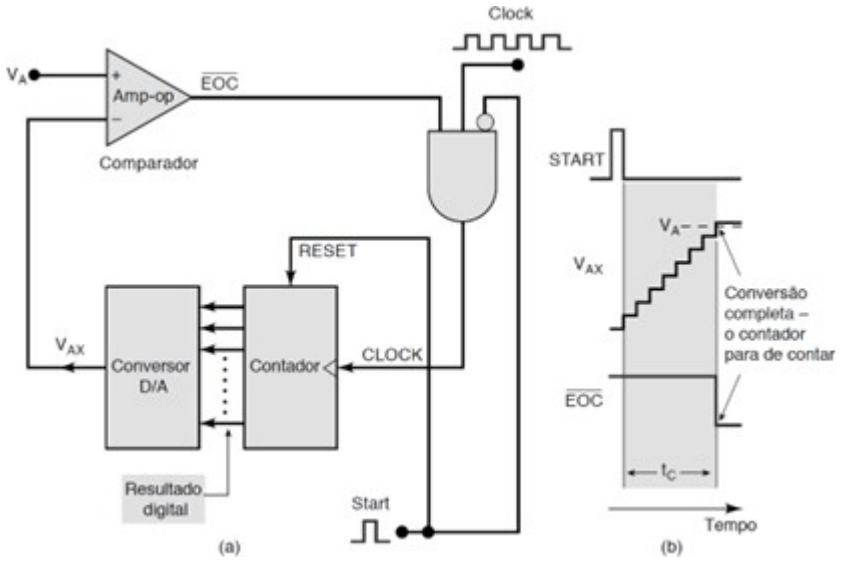
O uso de um **circuito de amostragem-e-retenção** (S/H, do inglês *sample-and-hold*) na entrada do sinal analógico antes do conversor é uma prática estabelecida. Esse módulo desempenha a função de "congelar" o sinal analógico por um intervalo de tempo enquanto a conversão está em andamento. A figura que se segue facilita a compreensão do funcionamento desse circuito.



O componente central é o capacitor (C), responsável pela retenção do sinal. O sinal de controle (S) regula a abertura e fechamento de uma chave. Quando a chave está fechada, o sinal da entrada analógica é amostrado e armazenado no capacitor durante o ciclo de amostragem. Posteriormente, quando a chave é aberta, a tensão armazenada no capacitor fica disponível para o ADC durante o ciclo de retenção. Circuitos integrados especializados desse tipo, que incorporam a função de amostragem/retenção (S/H), estão disponíveis, e um exemplo é a pastilha AD585 da *Analog Devices*.

ADC de Rampa Digital

A versão mais simples de ADC é o ADC de rampa digital, onde o valor de um contador é incrementado binariamente, conforme ilustra a seguinte figura. A saída do contador, V_{AX} , é convertida em um sinal analógico por meio de um DAC, e esse sinal é comparado com o sinal de entrada analógico, V_A. O sinal de saída do comparador, $\overline{EOC}$ (do inglês *End-Of-Conversion*), serve como entrada para uma porta lógica AND, que, por sua vez, realimenta um novo pulso para o contador enquanto $V_{AX} < V_A$. E o processo se repete iterativamente até que $\overline{EOC}$ se torne igual a 0.



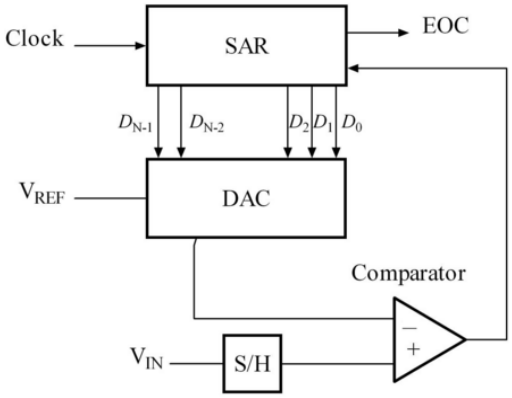
Essa arquitetura de rampa digital é especialmente valiosa do ponto de vista educacional, sendo frequentemente utilizada para fins didáticos. Ela oferece uma compreensão clara dos princípios fundamentais da conversão analógico-digital por meio de circuitos eletrônicos. No entanto, uma das principais desvantagens do circuito conversor ADC de rampa digital é o tempo médio de conversão, que dobra a cada *bit* adicional na saída digital, ou seja, com o aumento da resolução do conversor. O tempo de conversão se torna ainda maior ao se atingir o último degrau, próximo ao fundo de escala. Para um conversor de *n bits*, o tempo de conversão pode ser expresso como:

$$t_c(max) = (2^n - 1) \times T_{relógio} \ ,$$

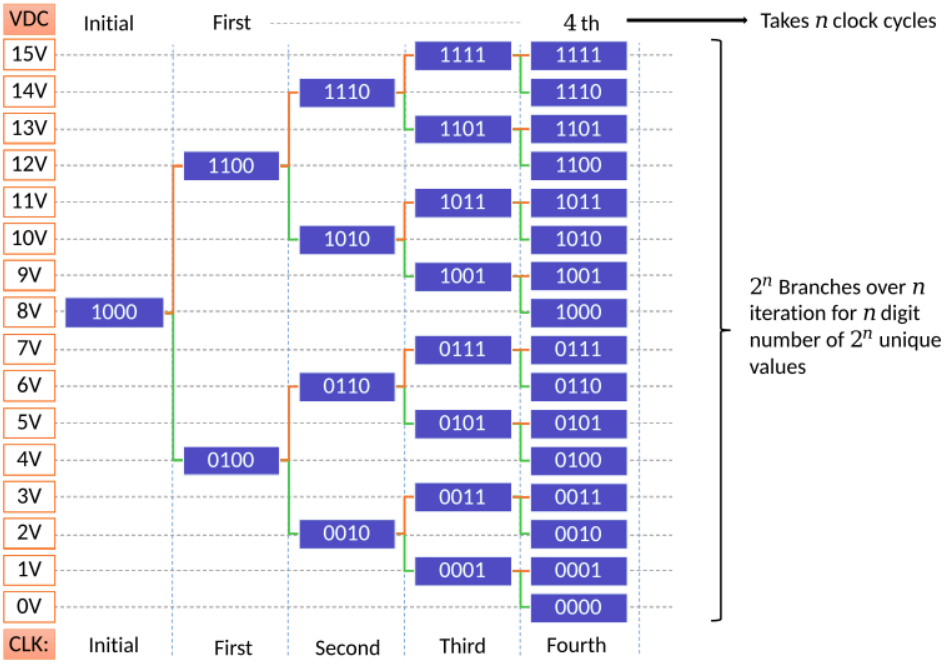
onde $T_{relógio}$ e n denotam, respectivamente, o período do relógio e a quantidade de *bits* do contador. Essa característica torna essa arquitetura uma escolha menos comum em aplicações comerciais. Além disso, o tempo de conversão varia bastante com o valor da tensão de entrada. Geralmente, o tempo médio de conversão $t_c(med)$ é especificado, sendo a metade do tempo máximo de conversão.

ADC com Registrador de Aproximações Sucessivas

A arquitetura de **conversor de aproximações sucessivas** (SAR) é uma das mais amplamente utilizadas em conversores analógico-digitais (ADC), oferecendo um tempo de conversão fixo, independentemente do valor analógico de entrada. Ao contrário do ADC de rampa digital, que utiliza um contador, o conversor SAR emprega um registrador de aproximação sucessiva (SAR) para determinar o valor digital correspondente ao sinal analógico. A figura a seguir ilustra os componentes básicos de um conversor SAR.



O funcionamento do conversor de aproximações sucessivas (SAR) é fundamentado em um processo binário iterativo de ajuste *bit* a *bit*, que começa pelo *bit* mais significativo (MSB) e avança até o *bit* menos significativo (LSB). Essa abordagem permite uma busca eficiente pelo valor digital correspondente ao sinal analógico de entrada. Inicialmente, todos os *bits* do registrador SAR são definidos como "0". O processo começa alterando o MSB para "1". Essa mudança reduz pela metade o intervalo de possíveis valores, pois um "1" no MSB representa uma faixa de valores superior à metade do total. O sistema, então, compara o sinal gerado pelo DAC (Conversor Digital-Analógico) com a tensão de entrada V_{IN}. Se a saída do DAC for maior que V_{IN}, isso indica que o valor digital atual é muito alto e o valor digital correspondente deve estar contido na metade com "0" no *bit* em análise. Nesse caso, o *bit* correspondente (o MSB) é redefinido para "0", e o sistema prossegue para o próximo *bit*, que também é ajustado. Se, por outro lado, a saída do DAC for menor que V_{IN}, o *bit* permanece como "1". Esse processo se repete para cada *bit* subsequente, refinando continuamente a aproximação até que todos os *bits* tenham sido avaliados e o sinal EOC seja ativado.



Exemplos de circuitos integrados que implementam a arquitetura SAR incluem o ADC0804 e o ADS8866, ambos da Texas Instruments.

Outros tipos de ADC

Existem outras técnicas para implementar um ADC, como por exemplo o sigma-delta, porém são mais sofisticadas e não serão abordadas aqui. O ADC do microcontrolador usado no curso é o de aproximações sucessivas, o mais utilizado em microcontroladores.

Recursos avançados em ADC

Neste roteiro foram abordados os recursos básicos do ADC. Alguns fabricantes implementam ADCs com recursos mais avançados, adequados em determinadas situações. Segue um resumo dos principais recursos avançados encontrados em ADCs:

- **Entrada diferencial:** Neste modo, dois pinos de entrada são usados para medir a diferença de tensão entre eles. Em vez de medir a tensão absoluta de um único canal em relação ao GND (como no modo de entrada simples), o ADC mede a diferença de potencial entre dois canais de entrada, geralmente chamados de **positivo** e **negativo**. O modo diferencial é mais resistente a ruídos comuns, pois ele cancela os sinais de interferência que afetam igualmente ambos os pinos.
- **Canais "injetados":** São usados para realizar conversões com prioridade mais alta, "injetadas" entre as conversões regulares. Essas conversões podem ser acionadas de forma **imediata** por interrupções externas, o que significa que podem ocorrer de maneira **assíncrona** em relação às conversões regulares. São úteis em situações em que é necessário obter uma leitura rápida e pontual, sem esperar pela conclusão da sequência de canais regulares. Além disso, cada canal injetado possui seu próprio registrador de dados, o que facilita a manipulação dos resultados.
- ***Oversampling*:** é uma técnica que consiste em realizar múltiplas amostragens do mesmo canal a cada disparo de conversão, e, em seguida, processar essas amostras para melhorar a resolução ou reduzir o ruído do sinal digitalizado. Normalmente, cada resultado de conversão é somado cumulativamente no registrador de dados, e ao final o valor da soma é dividido pelo número de amostras, realizando-se uma média da amostra. Esse processo aumenta a relação sinal-ruído (SNR) e permite que o sistema melhore a resolução efetiva, mesmo que o ADC tenha uma resolução nominal limitada. Normalmente o número de amostras por disparo é uma potência de dois, pois isso permite um processo de divisão simples ao final, usando-se deslocamento à direita dos *bits*.
- ***Watchdog* analógico:** é uma funcionalidade que monitora continuamente o valor de conversão de uma entrada analógica, e gera uma interrupção quando o valor lido sai de um intervalo de valores pré-definidos. O usuário escreve em registradores específicos os limites superior e inferior para o valor da tensão convertida. Durante as conversões, o *watchdog* compara os resultados do ADC com esses limites. Se a tensão lida estiver fora do intervalo especificado (acima do limite superior ou abaixo do limite inferior), o *watchdog* gera uma interrupção.

CONVERSÃO DOS RESULTADOS DO ADC EM UNIDADES FÍSICAS

É importante ressaltar que o resultado de uma conversão A/D, acessível pelo registrador ADCx_DR ou guardado na memória pelo DMA, 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] é diretamente proporcional ao código binário entre 0 e 2^N - 1. Dessa forma, o valor da amostra de tensão, ***Tensão_amostrada***, pode ser obtido a partir do código binário armazenado em ADCx_DR por uma regra de três simples usando operações em ponto flutuante:

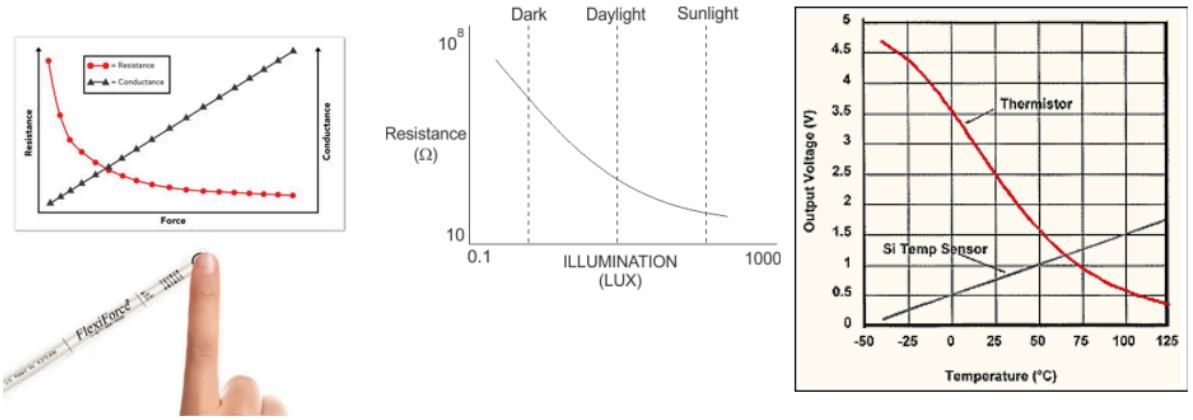
(Tensão amostrada-VREFL) → [ADCx_DR]

(VREFH – VREFL) → 2^N - 1

$$Tensão\ Amostrada - VREFL = \frac{ADCx\_DR \times (VREFH - VREFL)}{2^N - 1}$$

$$Tensão\ Amostrada = \frac{ADCx\_DR \times (VREFH - VREFL)}{2^N - 1} + VREFL \quad (1)$$

Se for necessário obter os valores originais das grandezas físicas dos sinais amostrados, é essencial realizar um pós-processamento das amostras de tensão recuperadas, *Tensão_amostrada*, convertendo-as para os valores nas grandezas físicas originais. Isso geralmente requer consulta aos datasheets dos fabricantes dos sensores, como se pode ver na figura abaixo (da direita para a esquerda, resistência e condutância x força em um FSR, resistência x luminosidade em um sensor de luminosidade, e tensão de saída x temperatura em um termistor e em um sensor integrado).



Joystick Analógico

O *joystick* analógico é um componente que atua como um controle bidimensional (X e Y) e inclui um botão integrado. Amplamente utilizado em projetos de controle de movimento,

como robótica e interfaces de jogos, ele é composto por dois potenciômetros, um para cada eixo (X e Y), além de um botão pressionável, permitindo tanto o controle direcional quanto a detecção de cliques. Cada eixo é controlado por um potenciômetro, e ao inclinar o *joystick* em uma direção, a resistência dos potenciômetros varia, resultando na geração de um sinal de tensão analógica. Esse sinal é então lido por canais de ADC, indicando a posição do *joystick*. Quando o *joystick* está na posição central, ambos os eixos normalmente fornecem um valor próximo da metade da tensão máxima, correspondente à posição de descanso. Ao mover o *joystick* para um dos extremos, o valor de um dos eixos se aproxima de zero ou do valor máximo do ADC. Além dos eixos analógicos, o *joystick* possui um botão que é acionado ao pressionar o bastão para baixo. Esse botão funciona como um interruptor, gerando um sinal digital (0 ou 1) que pode ser lido por uma entrada digital.

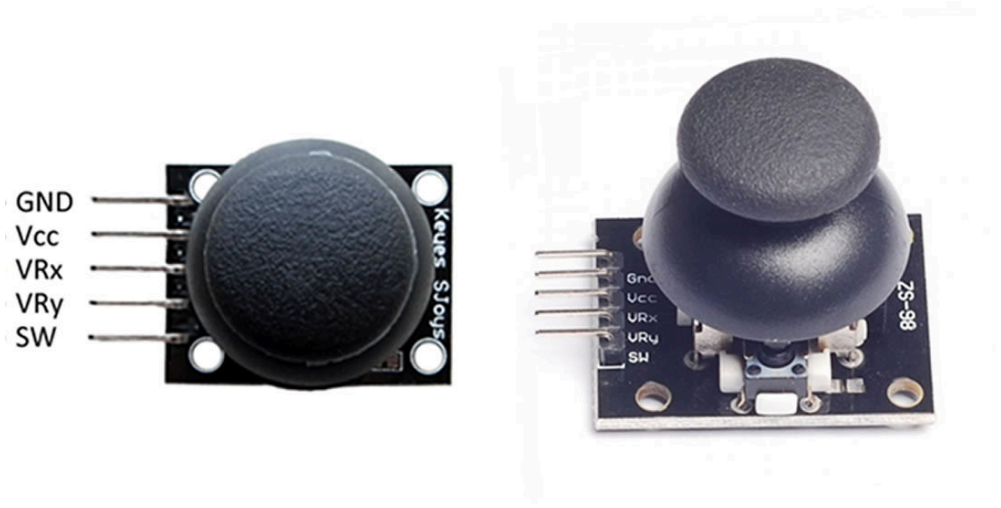
O módulo de *joystick* analógico normalmente tem cinco pinos principais:

- VCC: Alimentação. Muitas vezes marcado como "+5V", mas pode ser qualquer valor de tensão, definindo o valor máximo de tensão. No projeto deste roteiro, usamos 3.3V, pois este é o valor máximo aceito pelo ADC.
- GND: Terra
- VRx: Saída analógica do eixo X
- VRy: Saída analógica do eixo Y
- SW: Saída digital do botão

Os pinos VRx e VRy são conectados aos pinos de ADC do microcomtrolador para que ele possa ler os valores correspondentes à posição do *joystick*. O pino SW é conectado a uma entrada digital para ler o estado do botão.

Note que não é necessário converter o valor obtido no ADC em um valor de tensão. No *joystick*, o que importa é o valor relativo do eixo em relação ao valor máximo. Assim, podemos usar os valores "brutos" do `ADCx_DR`, comparando-os com o valor máximo para termos uma estimativa dos deslocamentos relativos.

$$deslocamento\,relativo = \frac{(ADCx\_DR - 0)}{(2^N - 1) - 0}$$



Potenciômetro

O potenciômetro é um componente eletrônico que pode ser usado como divisor de tensão, permitindo a variação da tensão de saída conforme o ajuste do usuário. Ele é constituído por uma resistência fixa e um contato deslizante, conhecido como cursor, que se move ao longo dessa resistência.

O potenciômetro possui três terminais. Dois deles estão conectados às extremidades da resistência fixa e o terceiro ao cursor. Quando aplicamos uma tensão fixa entre as extremidades da resistência (os dois terminais fixos), a tensão entre um dos terminais e o cursor varia conforme o movimento deste último. Isso ocorre porque o cursor divide a resistência total em duas partes, cada uma com um valor proporcional à distância entre o cursor e os extremos. Assim, o terminal central (cursor) fornece uma tensão variável em relação a uma das extremidades (geralmente ligada ao "terra" do sistema), que pode ser ajustada movendo o cursor. Quanto mais próximo o cursor estiver de um extremo, menor será a resistência nesse lado e, portanto, maior será a tensão nesse ponto.

Se amostrarmos as tensões nos terminais a ou b de um potenciômetro com o terminal c aterrado, utilizando um módulo ADC, podemos obter essas tensões a partir dos resultados binários armazenados no registrador ADCx_DR, conforme indicado pela Equação (1).

Fonte: [Squids]

Sensor de Temperatura Integrado LM61

Segundo o que o *datasheet* do componente diz, traduzido para o português (grifo dos autores do roteiro):

"O dispositivo LM61 é um circuito integrado sensor de temperatura de precisão, que pode medir uma faixa de temperatura de –30°C a 100°C operando com uma fonte de alimentação

simples de 2,7 V. A tensão de saída do LM61 é linearmente proporcional à temperatura (10 mV/°C) e possui um *offset* DC de 600 mV. Esse *offset* permite a leitura de temperaturas negativas sem a necessidade de uma fonte de alimentação negativa. A tensão de saída nominal do LM61 varia de 300 mV a 1600 mV para uma faixa de temperatura de –30°C a 100°C. O LM61 é calibrado para fornecer precisões de ±2°C à temperatura ambiente e ±3°C em toda a faixa de temperatura de –25°C a 85°C. A saída linear do LM61, o deslocamento de 600 mV e a calibração de fábrica simplificam o circuito externo necessário em um ambiente de fonte única, onde é necessário ler temperaturas negativas. Como a corrente de repouso é inferior a 125 µA, o aquecimento automático é limitado a um valor muito baixo de 0,2°C em ar parado. A capacidade de desligamento do LM61 é intrínseca, pois **seu consumo de energia inerentemente baixo permite que ele seja alimentado diretamente pela saída de muitos circuitos lógicos**.

O LM61 tem a aparência de um transistor de baixa potência com 3 terminais apenas: Alimentação positiva, GND e sinal de saída. Ele foi projetado para permitir a medição de temperaturas em graus Celsius com ADCs de tensão exclusivamente positiva em uma faixa significativa, incluindo valores negativos. O uso de um *offset* permite que temperaturas negativas possam ser representadas por tensões positivas. A zero graus, a tensão de saída é de 600mV, sendo que para cada grau a mais a tensão aumenta em 10mV, e para cada grau a menos a tensão diminui pelo mesmo fator. Ele pode ser alimentado por tensões que vão de 2.7V até 10V, sendo o *offset* e o fator de 10mV por grau independentes desta tensão. Seu consumo é de apenas 125 μA. Sua faixa de tensão de saída é de 300 a 1600mV, sendo adequada para a grande maioria dos ADCs de microcontroladores. Sua impedância de saída é de cerca de 800 Ω, o que permite um tempo de *sample* bastante curto.”

O fabricante especifica a relação entre a temperatura medida e a tensão gerada por meio da função de transferência:

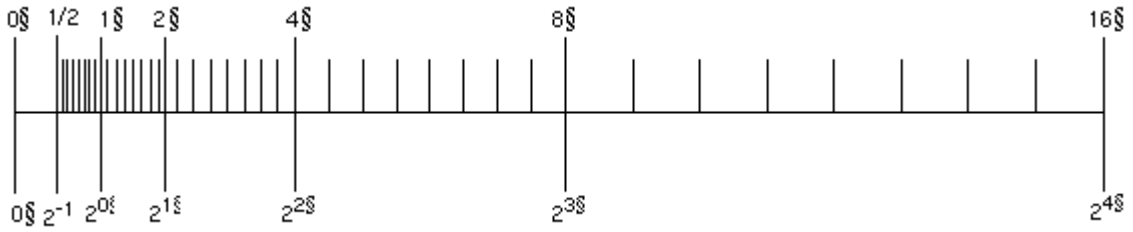
$$V_O = 10\ mV/°C \times T°C + 600\ mV$$

Essa relação permite estimar a temperatura com base no valor da tensão amostrada pelo ADC, que é obtido a partir do resultado da conversão no registrador ADCx_DR, conforme descrito na Equação (1).

EXIBIÇÃO DE NÚMEROS EM PONTO FLUTANTE EM *DISPLAYS*

Muitas medidas físicas são valores reais e representadas como números em ponto flutuante em sistemas digitais. As representações em ponto flutuante, seguindo o padrão IEEE754, são as mais difundidas para representar as aproximações dos números reais. Essas representações permitem descrever, com uma acurácia maior, a parte fracionária dos valores de diferentes ordens de grandeza usando uma quantidade fixa de *bytes* (4 para precisão simples e 8 para precisão dupla). Ao invés de espaçamentos equidistantes dos valores do tipo inteiro, o padrão IEEE754 da representação de pontos flutuantes espaça a parte inteira dos valores do tipo float de forma não uniforme ao longo da reta real, de forma que quanto menores são os valores

menor é o espaçamento entre eles. Porém, a quantidade de pontos entre dois valores subsequentes, representando a parte fracionária entre eles, é a mesma. Assim, a resolução da parte fracionária varia conforme o valor da parte inteira. Quanto menor o valor da parte inteira, maior é a resolução da parte fracionária.



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

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

Entretanto, muitos *displays*, como terminais seriais e telas LCD, processam apenas vetores de caracteres (*strings*), o que exige que os valores gerados pelos sistemas digitais sejam convertidos para a representação ASCII adequada antes de serem enviados a esses dispositivos de saída. Essa conversão é tipicamente realizada por uma função chamada ftoa.

Como o número de casas decimais pode ser indefinido, é comum utilizar truncamento ou arredondamento para padronizar a exibição em um número fixo de casas decimais. O **truncamento** é o processo de remover as casas decimais indesejadas sem alterar o valor restante. Por exemplo, se truncar o número 3,456 para duas casas decimais, o resultado será 3,45. A parte decimal extra é simplesmente descartada. O arredondamento, por outro lado, é o processo de ajustar o valor para a casa decimal mais próxima. Por exemplo, se arredondar o número 3,456 para duas casas decimais, o resultado será 3,46, pois a última casa decimal (6) indica que a parte anterior deve ser aumentada.

Um algoritmo amplamente utilizado para essa conversão envolve os seguintes passos: primeiro, é necessário tratar o sinal do número. Em seguida, o número `n` é dividido em sua parte inteira (`ipart`) e parte fracionária (`fpart = n - ipart`). A parte fracionára, `fpart`, é convertida para um tipo de dado inteiro, denominado `ifpart`. Ambas as partes são, então, transformadas em *strings* de caracteres utilizando um algoritmo específico para conversão de

inteiros para sua representação ASCII, comumente conhecido como itoa. Por fim, as duas *strings* são concatenadas com um separador, como um ponto (".") ou uma vírgula (",").

ACESSO DIRETO À MEMÓRIA

O conceito DMA (do inglês, *Direct Memory Access*) foi rudimentarmente implementado, pela primeira vez, em 1958 num computador baseado em válvulas de vácuo IBM 709. Somente na década de 1970, foram lançados controladores DMA como o entendemos hoje. Nesta década foi lançado um dos primeiros controladores DMA, o Intel 8237. Os computadores PCs IBM, lançados em 1981, usaram uma arquitetura baseada no processador de 16 *bits* Intel 8088 e um controlador DMA Intel 8237A, que é uma versão aprimorada do 8237, projetada para melhor desempenho e maior flexibilidade em comparação com seu antecessor. Essa combinação de processador e controlador DMA permitiu que os PCs IBM realizassem eficientemente transferências diretas de dados para a memória, contribuindo para o seu sucesso e ajudando a estabelecer os padrões que moldaram a computação pessoal moderna. A evolução da eletrônica digital e das interfaces de barramento, como PCI e PCIe, impulsionou o DMA a novos patamares, tornando-o uma parte essencial de sistemas modernos. Hoje, o DMA é uma parte crítica de praticamente todos os sistemas computacionais, desde PCs até dispositivos móveis e sistemas embarcados.

O DMA é um recurso que desempenha um papel crucial na otimização de transferências de dados. Enquanto a CPU controla transferências de dados entre periféricos e memória principal, o DMA oferece uma abordagem mais eficiente e autônoma, permitindo que dispositivos periféricos acessem diretamente a memória. Isso reduz a carga da CPU e aumenta a velocidade das transferências. Existem essencialmente duas maneiras de implementar transferências diretas de dados entre periféricos e a memória principal, sem a intervenção do processador.

A primeira, conhecida como **sistema de acesso direto à memória de primeira parte** (em inglês, *first-party DMA*), baseia-se na arbitragem do uso de um recurso compartilhado, o barramento, entre a memória principal e os periféricos que podem se comunicar diretamente. Nesse caso, o periférico que ganha o direito de usar o barramento passa a ser o mestre do barramento e assume integralmente o controle dele. A segunda abordagem, considerada a técnica padrão de acesso direto à memória, é conhecida como **acesso direto à memória de terceira parte** (em inglês, *third-party DMA*). Essa técnica envolve o uso de um **Controlador para Acesso Direto à Memória** (em inglês, *Direct Memory Access Controller* – DMAC). O controlador DMA é um circuito dedicado projetado para gerenciar transferências diretas entre periféricos e a memória principal. Ele assume a função de mestre no lugar do processador, colocando no barramento de endereços as posições de memória que se deseja acessar e gerando sinais de controle de acesso.

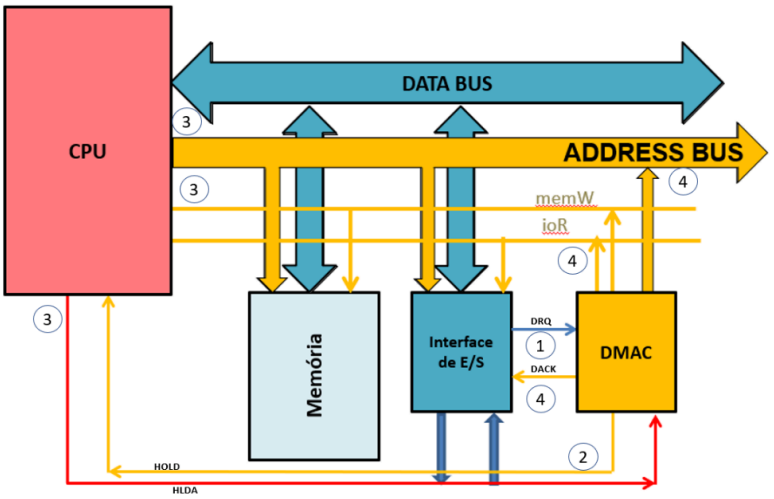
Nos microcontroladores modernos, a abordagem de DMA de terceira parte é a mais prevalente. Isso se deve à sua capacidade de aumentar a eficiência do sistema, permitindo que a CPU se concentre em outras tarefas enquanto o controlador DMA realiza as transferências de dados em segundo plano. Essa configuração é especialmente vantajosa em aplicações que

exigem processamento em tempo real, como controle de dispositivos, aquisição de dados e comunicação em redes, onde o desempenho e a rapidez são fundamentais.

Controlador de Acesso Direto à Memória

Quando um periférico deseja realizar uma transferência direta de dados para ou da memória principal, ele negocia o acesso ao barramento com o DMAC. O DMAC atua como um gerenciador de tráfego, arbitra o acesso ao barramento e controla as transferências de dados entre periféricos e memória, sem a necessidade de intervenção direta da CPU. Isso permite que a CPU continue suas operações enquanto o DMAC cuida das transferências de dados, melhorando a eficiência do sistema.

Essa negociação pode ser entendida com o auxílio da figura, em que se distinguem 4 etapas básicas de acesso aos barramentos para transferências diretas sem intervenção da CPU.



1. Solicitação de DMA pelo periférico, enviando o sinal DRQ ao DMAC.
2. Solicitação de liberação dos barramentos da CPU (HOLD) pelo DMAC.
3. Assim que concluir a execução da instrução, a CPU reconhece a solicitação de HOLD e libera os seus barramentos de dados, de endereços e de controle, colocando seus acessos em alta impedância; além disso o processador informa ao DMAC que reconheceu a solicitação e liberou os barramentos para o DMAC com o **sinal de reconhecimento de HOLD** (do inglês, *Hold Acknowledgement* – HLDA).
4. O DMAC informa ao periférico que a sua solicitação de DMA foi reconhecida pelo sinal de **reconhecimento (da solicitação) de DMA** (do inglês, *DMA Acknowlegment* – DACK). Este sinal é usado para colocar no barramento de dados do sistema os dados a serem transferidos. Além disso, o DMAC gera o endereço da posição de memória a ser acessada e os sinais de controle necessários para a transferência com a memória, como memW (memória escrita) ou memR (memória leitura). Ele também emite os sinais de controle para a transferência com a interface, como ioR (entrada) ou ioW (saída).

O período de tempo durante o qual o controlador DMA executa uma única transferência de dados entre um dispositivo periférico e a memória principal do sistema é conhecido por **ciclo de transferência de um controlador DMA**. Esse ciclo é uma unidade fundamental de operação do DMA.

Os modos de operação de um DMA são projetados para atender a diferentes cenários de uso. A seleção do modo apropriado depende das necessidades específicas do sistema e das características dos dispositivos periféricos envolvidos. Essa flexibilidade ajuda a otimizar o desempenho do sistema e a minimizar o envolvimento da CPU em tarefas de transferência de dados.

Modos de Operação do DMA

Os modos de operação de um DMA se referem às diferentes configurações que definem como o controlador DMA gerencia as transferências de dados entre a memória e os periféricos, ou entre diferentes áreas de memória. Cada modo de operação tem características específicas que se adequam a diferentes necessidades de transferência. Aqui estão alguns dos modos de operação mais populares:

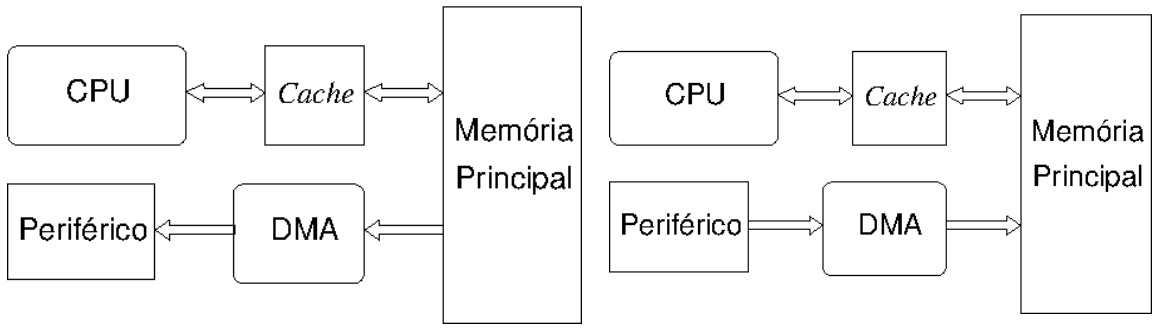
Transferência por Rajada (em inglês, *Burst Transfer Mode*): Esse modo permite a execução de transferências de uma série contínua de blocos de dados sem a necessidade de ativar a transferência para cada bloco separadamente. Isso é especialmente eficiente para operações de leitura ou escrita contínuas, pois reduz a sobrecarga de controle e minimiza o envolvimento da CPU. No entanto, durante a execução de transferências em modo de rajada, o controlador DMA assume o controle do barramento e bloqueia as operações da CPU temporariamente. Isso ocorre porque enquanto o controlador DMA está realizando uma rajada de transferências, ele mantém controle contínuo sobre o barramento, para garantir a eficiência das operações. Como resultado, outras operações da CPU, como transferências entre periféricos de entrada e saída e o processador, podem ser temporariamente interrompidas.

Roubo de Ciclo (em inglês, *Cycle Stealing Mode*): Nesse modo, a CPU é bloqueada quando o DMAC assume o controle do barramento, mas ao contrário do modo de rajada, o controlador DMA “rouba” ciclos de barramento da CPU quando precisa realizar uma transferência de dados. Durante esses ciclos, a CPU é interrompida brevemente para permitir que o DMA acesse a memória. Essa abordagem é benéfica em sistemas onde a CPU precisa estar disponível para interações frequentes com periféricos ou outras operações críticas.

Transferência Intercalada (em inglês, *Interleaving Transfer Mode*) ou **Transferência Transparente** (em inglês, *Transparent Transfer Mode*): Nesse modo, o controlador DMA realiza transferências de dados em paralelo com a operação da CPU, mas com um padrão de *interleaving* (ou intercalamento). Isso significa que as transferências DMA são feitas de forma a intercalar os dados, permitindo que a CPU e o DMA operem em um padrão mais regular e alternado. A principal vantagem desse modo é que a CPU nunca pára de executar seus programas e a transferência DMA não consome tempo da CPU, enquanto a desvantagem é que o *hardware* precisa determinar quando a CPU não está utilizando os barramentos do sistema, o que pode ser complexo. Esse modo leva mais tempo para transferir um bloco de dados, mas é também o modo mais eficiente em termos de desempenho geral do sistema.

Coerência de *Cache*

É importante notar que o DMAC permite que dispositivos periféricos acessem diretamente a memória, o que realmente alivia a carga da CPU e aumenta a velocidade das transferências. No entanto, essa operação pode criar situações de incoerência de *cache*. Quando o DMAC realiza transferências de dados diretamente entre um periférico e a memória principal, essas operações podem não ser refletidas imediatamente na memória *cache* utilizada pela CPU. Essas incoerências podem ocorrer tanto em operações de leitura, onde a CPU pode ler um valor obsoleto da *cache*, quanto em operações de escrita, onde dados escritos diretamente na memória pelo DMAC não são refletidos na *cache* da CPU.



Leitura Escrita

Quando ocorre uma leitura DMA, os dados são lidos diretamente da memória principal, mas a *cache* pode conter informações atualizadas, pois a CPU pode ter atualizado a *cache* sem refletir essas atualizações imediatamente na memória principal. Portanto, é necessário realizar uma limpeza de *cache* (do inglês, *cache flush*) antes de uma operação DMA de leitura para garantir que a memória principal contenha a versão mais recente dos dados. A limpeza de *cache* envolve a escrita dos dados modificados (ou seja, dados sujos) de volta na memória principal. Essa é uma medida importante para garantir que os dados lidos sejam sempre os mais recentes.

Quando ocorre uma escrita DMA, os dados são escritos diretamente na memória principal, sem atualizar a *cache*. Para garantir a coerência dos dados, qualquer dado na *cache* que corresponda à área de memória escrita pela operação DMA deve ser marcada como inválida (do inglês, *cache invalidation*). Dessa forma, a próxima vez que a CPU tentar acessar esses dados na *cache*, ela será forçada a buscar os dados mais recentes da memória principal.

ESTRUTURA DE DADOS: *BUFFER* CIRCULAR

Para harmonizar os diferentes passos de processamento de dois módulos, sem comprometer a capacidade do processador, é comum utilizar nos projetos de sistemas embarcados os ***buffers circulares*** para armazenar os dados para que a comunicação ocorra de forma fluida e responsiva.

Um *buffer* circular é, de fato, uma variante da estrutura de dados **fila** (FIFO) com as pontas conectadas (o elemento seguinte ao último é o primeiro da fila). O diferencial dessa estrutura em relação às clássicas filas está na forma como é feita a reorganização dos dados em cada remoção e na forma como o espaço de memória é ocupado. Na fila, todos os elementos devem ser deslocados de uma casa quando se remove o primeiro elemento. Porém, no *buffer* circular, os seus elementos não são deslocados quando se retira um elemento da fila. A manipulação dos dados é feita por dois ponteiros, denominados *head* (cabeça da fila) e *tail* (final da fila). Quando se adiciona um elemento, o *head* é incrementado ciclicamente, e quando se retira um elemento do final da fila, o *tail* é incrementado ciclicamente. Na Figura 4, sob a perspectiva de uma fila clássica, se removermos o elemento '0' da fila, os elementos '1' a '14' serão deslocados de uma casa para esquerda e *pointer* tem o seu endereço deslocado de uma casa. Já no *buffer* circular, a remoção do elemento '0' faz com que somente *tail* tenha o endereço deslocado de uma casa. O conteúdo da memória não é alterado. Como os ponteiros são incrementados ciclicamente no *buffer* circular, os endereços de memória acessados estão sempre dentro do espaço pré-reservado, enquanto na fila, o *pointer* pode vazar para fora do espaço previamente reservado.

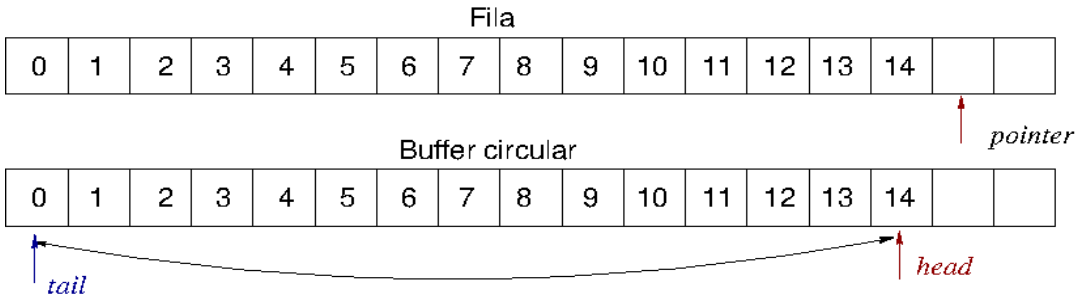


Figura 4: Fila e *buffer* circular.

Em C, é comum definir um novo tipo de dado struct, como `BufferCirc_type`, para agrupar uma fila cíclica de `dados` e os seus ponteiros `tail` e `head` em uma única variável.

```
typedef struct BufferCirc_tag
{
	char dados[MAX];		// buffer de dados com um tamanho de MAX elementos
	unsigned int tamanho;	// quantidade total de elementos
	unsigned int leitura;	// indice de leitura (tail)
	unsigned int escrita;	// indice de escrita (head)
} BufferCirc_type;
```

STM32H7A3

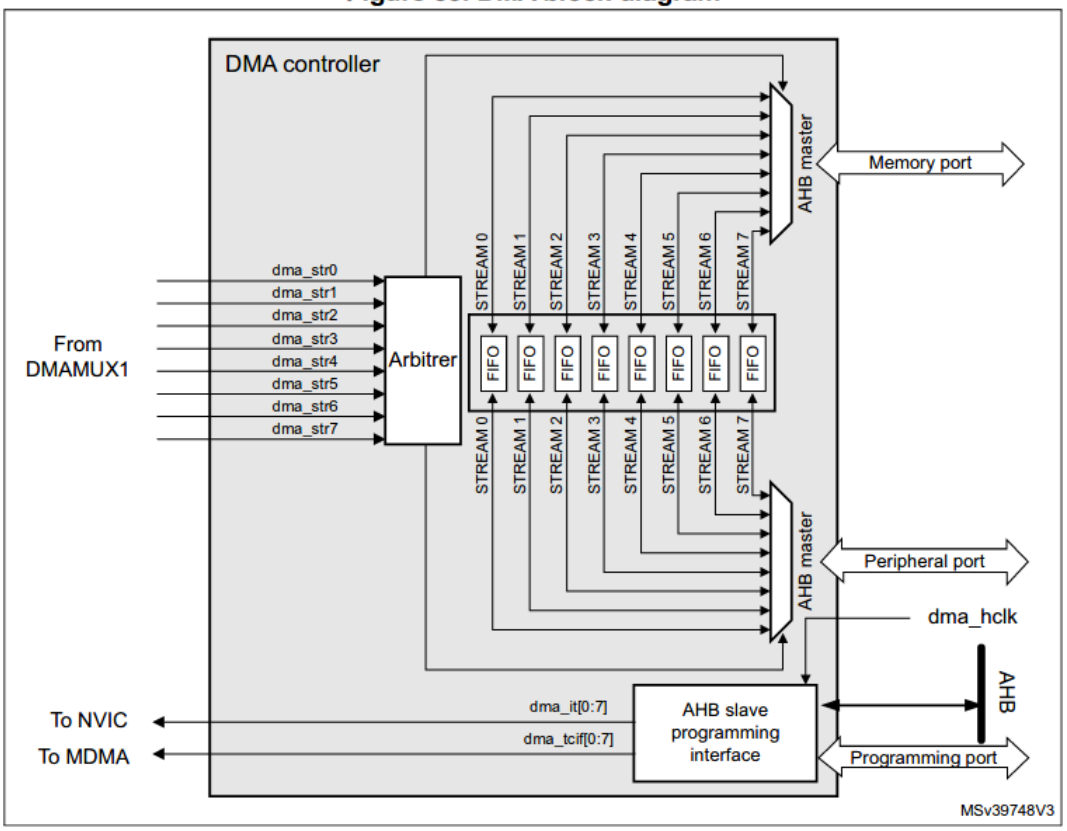
Nesta seção, exploraremos em detalhes as funcionalidades oferecidas pelos módulos DMA, DAC e ADC integrados no microcontrolador STM32H7A3.

Módulo DMA

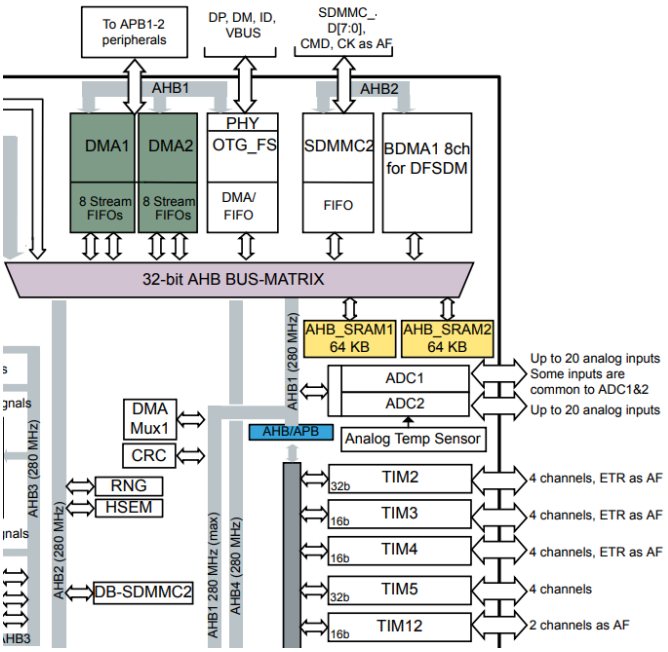
O microcontrolador STM32H7A3 possui um sistema de acesso direto à memória (em inglês, *Direct Memory Acces* – DMA) que facilita a transferência rápida de dados entre periféricos e

memória, ou mesmo entre diferentes áreas da memória, sem intervenção direta da CPU. Esse recurso aumenta a eficiência do sistema, liberando a CPU para realizar outras operações. O STM32H7A3 integra três variantes de controladores DMA: DMA, BDMA (do inglês *Basic Direct Memory Access*) e MDMA (do inglês *Master Direct Memory Access*). Sendo o DMA a implementação padrão, focaremos neste roteiro a descrição desta variante, cujo diagrama de blocos é ilustrado a seguir.

Figure 83. DMA block diagram



O **controlador DMA** é composto por dois controladores DMA (DMA1 e DMA2), que estão conectados no barramento AHB1. Projetados para seguir a abordagem de *clock gating*, esses controladores devem ser ativados individualmente por meio dos *bits* `RCC_AHB1ENR_DMA2EN` e `RCC_AHB1ENR_DMA1EN` antes da configuração e inicialização. Cada controlador oferece 8 canais lógicos `dma_strx`, que possibilitam a transferência de dados entre dispositivos periféricos e a memória principal sem a intervenção da CPU. Isso resulta em um total de 16 canais, denominados *streams*, para gerenciar as solicitações de acesso à memória de diversos periféricos.



Para otimizar o acesso ao barramento mestre, o DMA conta com duas portas mestres AHB, AHB *Memory Port* e AHB *Peripheral Port*, permitindo transferências simultâneas entre:

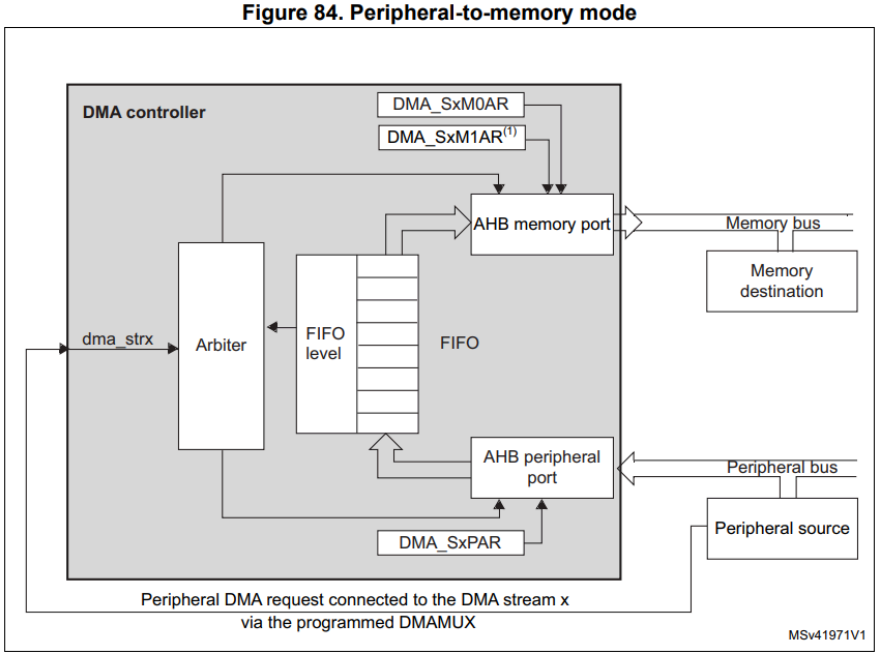
- **Periférico para Memória:** O DMA lê dados do periférico e os grava na memória.
- **Memória para Periférico:** O DMA lê dados da memória e os grava no periférico.
- **Memória para Memória:** O DMA copia dados de uma área da memória para outra.

A direção de transferência é configurada pelos *bits* `DMA_SxCR_DIR`. Além disso, cada *stream* é equipado com um *buffer* FIFO (do inglês *First-In*, *First-Out*), que suporta quatro palavras (32 *bits*), e pode operar em dois modos: FIFO, com níveis de limiar configuráveis, ou em modo direto, onde cada solicitação DMA inicia uma transferência imediatamente. No modo FIFO, distingue-se ainda o modo circular e o modo não-circular (fila).

Antes de iniciar transferências via DMA, é necessário configurar o canal x de transferência via o registrador `DMA_SxCR`. Este registrador configura o modo de operação do *stream*, a direção da transferência, o tamanho da transferência, o endereçamento, o modo circular ou de fila, as interrupções e a habilitação do *stream*. Muitos dos seus *bits* de configuração são protegidos e só permitem acessos de escrita quando o *bit* `DMA_SxCR_EN` é resetado em “0”.

A quantidade de dados a ser transferida (de 1 até 65535) é programável e está relacionada à largura da fonte do periférico que solicita a transferência DMA conectada à porta AHB do periférico. O registrador `DMA_SxNDTR` que contém a quantidade de itens de dados a serem transferidos é decrementado após cada transação. Além de especificar o número de itens de dados a serem transferidos, é necessário configurar o endereço do periférico e os endereços de memória envolvidos na transferência nos registradores, `DMA_SxPAR`, `DMA_SxM0AR` e `DMA_SxM1AR`, como ilustra o seguinte fluxo de dados no modo de transferência de um periférico para memória.

Figure 84. Peripheral-to-memory mode

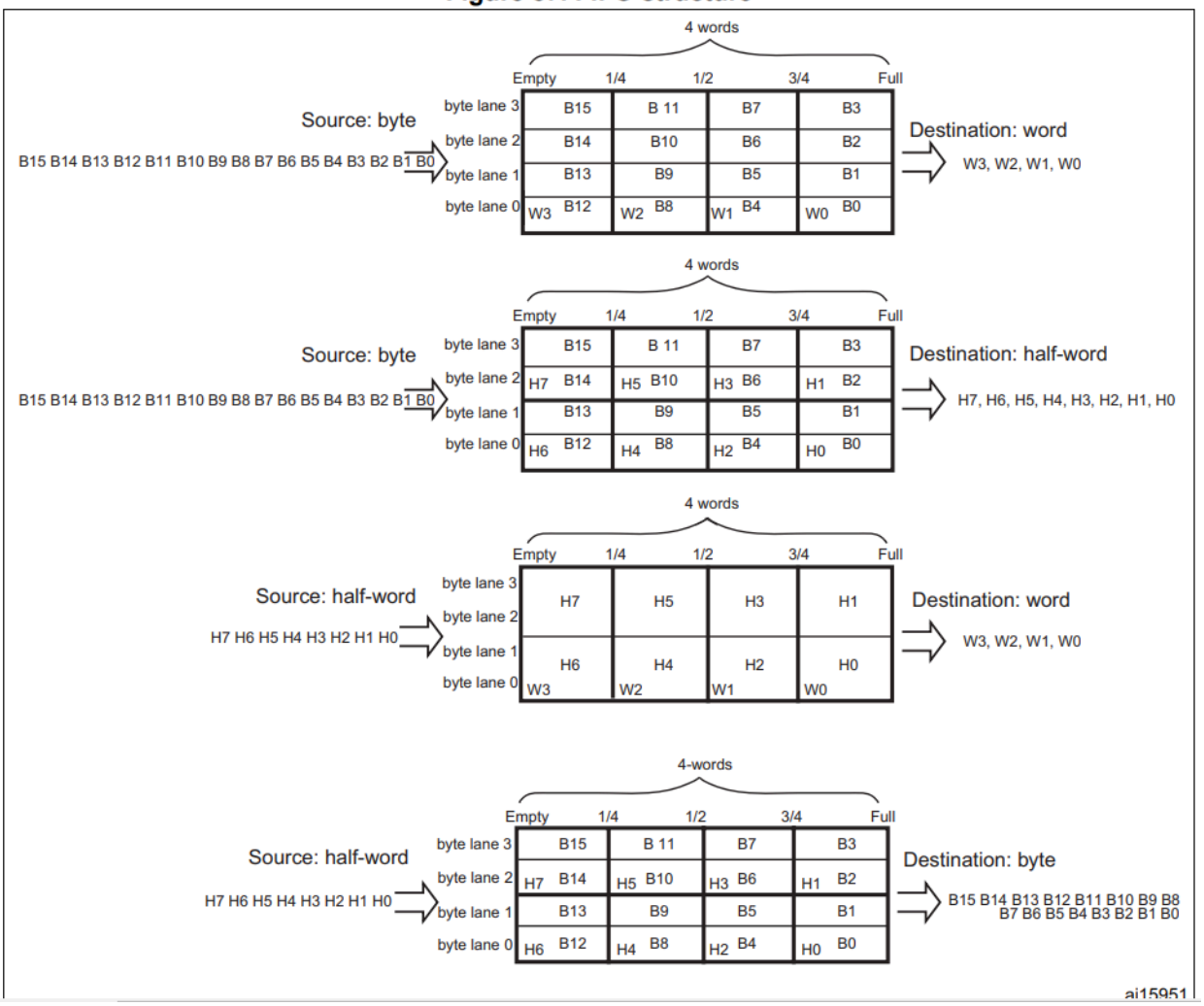


1. For double-buffer mode.

Quando há mais de um *stream* habilitado, um arbitador gerencia os pedidos de *stream* DMA com base em sua prioridade para cada uma das duas portas mestres AHB e inicia as sequências de acesso a periféricos/memória. As prioridades são gerenciadas em duas etapas:

- ***Software***: A prioridade de cada *stream* pode ser configurada no registrador `DMA_SxCR_PL` em um dos quatro níveis: Muito alta prioridade (0b11), Alta prioridade (0b10), Prioridade média (0b01) e Baixa prioridade (0b00).
- ***Hardware***: Se dois pedidos tiverem o mesmo nível de prioridade de *software*, o *stream* com o número mais baixo tem prioridade sobre o *stream* com o número mais alto. Por exemplo, o stream 2 tem prioridade sobre o stream 4.

Figure 87. FIFO structure



O FIFO é utilizado para armazenar temporariamente os dados que vêm da fonte antes de transmiti-los para o destino. O nível limiar de cada *stream* pode ser configurado por *software* entre 1/4, 1/2, 3/4 ou cheio. Para habilitar o uso do nível de limiar do FIFO, o modo direto deve ser desativado configurando o *bit* `DMA_SxFCR_DMDIS` em "1". A estrutura do FIFO varia conforme as larguras de dados da fonte e do destino, seguindo o princípio de "primeiro que entra, primeiro que sai" para o armazenamento e a recuperação dos *bytes* no *buffer* conforme ilustrado na figura 87 do Manual de Referência.

Os módulos DMA suportam uma variedade de modos de transferência, cujas configurações são realizadas por meio de diversos registradores específicos do DMA. Um dos modos é a **transferência por rajada** (em inglês, *Burst Transfer Mode*), que permite a transferência de um bloco contíguo de dados em uma única requisição, otimizando a transferência de grandes volumes de dados. Esse modo é ativado configurando um tamanho de *burst* maior que 1 nos *bits* `DMA_SxCR_MBURST` (*Memory Burst Transfer Configuration*) ou `DMA_SxCR_PBURST` (*Peripheral Burst Transfer Configuration*).

Outro modo é o **roubo de ciclo** (em inglês, *Cycle Stealing Mode*), no qual o DMA transfere um único dado por vez e libera o barramento imediatamente após a transferência, permitindo

que a CPU continue suas operações com mínima interrupção. Para ativá-lo, o tamanho de *burst* deve ser configurado como 1 (transferência única) nos *bits* DMA_SxCR_MBURST ou DMA_SxCR_PBURST.

O modo de **transferência intercalada/transparente** (em inglês, *Interleaving/Transparent Transfer Mode*) opera transferindo dados apenas quando a CPU não está utilizando o barramento, minimizando o impacto no desempenho. Este modo não possui um *bit* de configuração específico, sendo implementado através da lógica de arbitragem interna do DMA, que monitora o uso do barramento pela CPU.

Por fim, o modo de ***buffer* duplo** (em inglês, *Double-Buffer Mode*) utiliza dois *buffers* de memória, permitindo que a CPU preencha um *buffer* enquanto o DMA transfere dados do outro, garantindo um fluxo contínuo de dados. Esse modo é habilitado configurando o *bit* DMA_SxCR_DBM (*Double-Buffer Mode*) no registrador como 1.

Os módulos DMA utilizam interrupções para sinalizar eventos relacionados às transferências de dados, como conclusão de transferência, erros ou a necessidade de intervenção do software. A tabela abaixo descreve os *bits* de habilitação (DMA_SxCR), estado (DMA_LISR e DMA_HISR) e limpeza (DMA_LIFCR e DMA_HIFCR) associados às interrupções que podem ser geradas por cada canal do DMA:

Interrupção	Descrição	Habilitação	Estado	Limpeza
TC	Transferência Completa: Setada quando uma transferência é concluída.	TCIE	TCIF	CTCIF
HT	Meia Transferência: Setada quando metade da transferência é concluída.	HTIE	HTIF	CHTIF
TE	Erro de Transferência: Setado quando ocorre um erro durante a transferência.	TEIE	TEIF	CTEIF
FE	Erro de FIFO: Setado para erros relacionados ao FIFO do canal.	FEIE	FEIF	CFEIF
DME	Erro de Modo Direto: Setado em caso de erro no modo de acesso direto.	DMEIE	DMEIF	CDMEIF

É necessário que as linhas de requisição de interrupção correspondentes sejam habilitadas e devidamente configuradas no controlador de interrupções NVIC.

dma1_it0	18	11	DMA1_STR0	DMA1 Stream0 global interrupt	0x0000 006C

Signal	Priority	NVIC position	Acronym	Description	Address offset
dma1_it1	19	12	DMA1_STR1	DMA1 Stream1 global interrupt	0x0000 0070
dma1_it2	20	13	DMA1_STR2	DMA1 Stream2 global interrupt	0x0000 0074
dma1_it3	21	14	DMA1_STR3	DMA1 Stream3 global interrupt	0x0000 0078
dma1_it4	22	15	DMA1_STR4	DMA1 Stream4 global interrupt	0x0000 007C
dma1_it5	23	16	DMA1_STR5	DMA1 Stream5 global interrupt	0x0000 0080
dma1_it6	24	17	DMA1_STR6	DMA1 Stream6 global interrupt	0x0000 0084
dma1_it7	54	47	DMA1_STR7	DMA1 Stream7 global interrupt	0x0000 00FC

Além dos controladores padrão DMA1 e DMA2, o STM32H7A3 possui duas variantes adicionais de controlador DMA: o MDMA (do inglês *Master Direct Memory Access*) e o BDMA (do inglê *Basic Direct Memory Access*). O MDMA é projetado para trabalhar em conjunto com os controladores DMA padrão, oferecendo transferências de dados ainda mais rápidas. Ele atua como um mestre AXI/AHB, controlando a matriz de barramento AXI/AHB para iniciar transações e gerenciar acessos à memória de maneira eficiente. Por outro lado, o BDMA é uma versão mais simples do DMA, voltada para transferências de dados básicas. Ele é capaz de realizar transferências entre periféricos e memória, entre diferentes áreas de memória e também entre periféricos. O BDMA é implementado em duas instâncias, BDMA1 e BDMA2, cada uma com 8 canais configuráveis de forma independente, permitindo o acesso a diferentes áreas de memória e periféricos.

Uma vez configurado um módulo DMA, o periférico envia um sinal de solicitação de DMA (em inglês DMA *request signal*) quando precisa realizar uma transferência de dados via DMA. Essa solicitação permanece pendente até que seja atendida pelo controlador de DMA, que, ao receber a solicitação, gera um sinal de reconhecimento de DMA. Assim que o controlador confirma a solicitação, o sinal de solicitação do periférico é desativado. Um conjunto de sinais de controle seria necessário para gerenciar o protocolo de solicitação e reconhecimento de DMA, conhecido como **linha de solicitação de DMA** (em inglês, *DMA request line*).

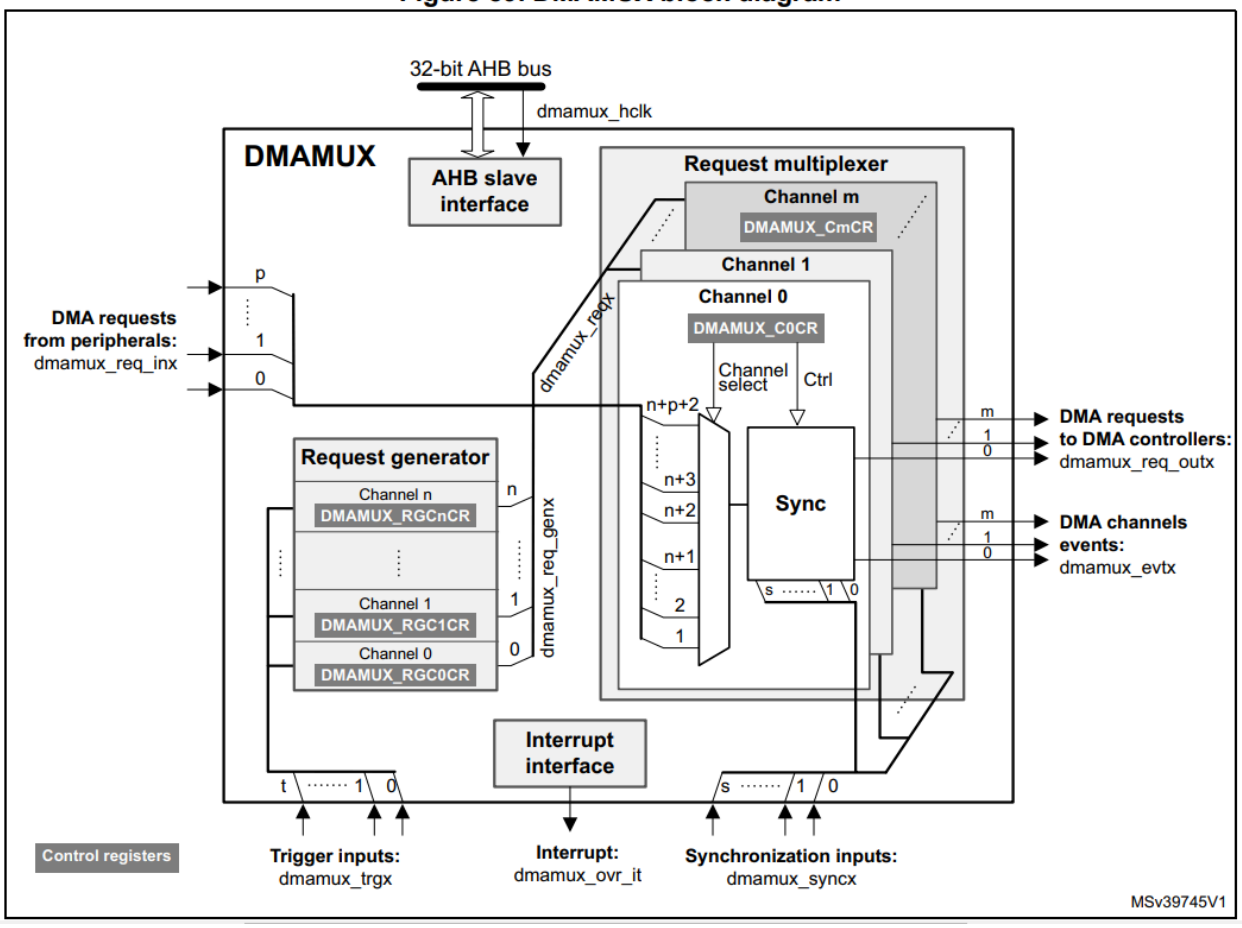
Para otimizar a comunicação entre os periféricos e o controlador de DMA, utiliza-se um multiplexador de solicitações de DMA, conhecido como DMAMUX (do inglês DMA *Request*

Line Multiplexer). O STM32H7A3 integra duas instâncias do DMAMUX, DMAMUX1 e DMAMUX2, que são responsáveis por rotear as solicitações DMA (`dmamux_reqx`) para os *streams* de DMA (`dmamux_req_outx`), conforme detalhado no Manual de Referência:

- DMAMUX1: Conectado aos controladores DMA1 e DMA2.
- DMAMUX2: Conectado ao controlador BDMA.

Cada componente DMAMUXn possui múltiplos canais, permitindo que cada canal selecione uma linha de solicitação de DMA específica através dos *bits* `DMAMUXn_CxCR_DMAREQ_ID[6:0]` (`dmamux_req_inx`), onde x é o número de canal.

Figure 89. DMAMUX block diagram



O Manual de Referência lista as 127 fontes de linhas de solicitação de periféricos. A seleção de uma dessas fontes pode ser feita de maneira incondicional ou em sincronização com eventos externos.

Table 101. DMAMUX1: assignment of multiplexer inputs to resources (continued)

DMA request MUX input	Resource	DMA request MUX input	Resource	DMA request MUX input	Resource
37	SPI1_RX	79	UART7_RX	121	Reserved
38	SPI1_TX	80	UART7_TX	122	Reserved
39	SPI2_RX	81	UART8_RX	123	Reserved
40	SPI2_TX	82	UART8_TX	124	Reserved
41	USART1_RX	83	SPI4_RX	125	Reserved
42	USART1_TX	84	SPI4_TX	127	Reserved

Table 101. DMAMUX1: assignment of multiplexer inputs to resources

DMA request MUX input	Resource	DMA request MUX input	Resource	DMA request MUX input	Resource
1	dmamux1_req_gen0	43	USART2_RX	85	SPI5_RX
2	dmamux1_req_gen1	44	USART2_TX	86	SPI5_TX
3	dmamux1_req_gen2	45	USART3_RX	87	SAI1_A
4	dmamux1_req_gen3	46	USART3_TX	88	SAI1_B
5	dmamux1_req_gen4	47	TIM8_CH1	89	SAI2_A
6	dmamux1_req_gen5	48	TIM8_CH2	90	SAI2_B
7	dmamux1_req_gen6	49	TIM8_CH3	91	SWPMI_RX
8	dmamux1_req_gen7	50	TIM8_CH4	92	SWPMI_TX
9	ADC1	51	TIM8_UP	93	SPDIFRX_DT
10	ADC2	52	TIM8_TRIG	94	SPDIFRX_CS
11	TIM1_CH1	53	TIM8_COM	95	Reserved
12	TIM1_CH2	54	Reserved	96	Reserved
13	TIM1_CH3	55	TIM5_CH1	97	Reserved
14	TIM1_CH4	56	TIM5_CH2	98	Reserved
15	TIM1_UP	57	TIM5_CH3	99	Reserved
16	TIM1_TRIG	58	TIM5_CH4	100	Reserved
17	TIM1_COM	59	TIM5_UP	101	DFSDM1_dma0
18	TIM2_CH1	60	TIM5_TRIG	102	DFSDM1_dma1
19	TIM2_CH2	61	SPI3_RX	103	DFSDM1_dma2
20	TIM2_CH3	62	SPI3_TX	104	DFSDM1_dma3
21	TIM2_CH4	63	UART4_RX	105	TIM15_CH1
22	TIM2_UP	64	UART4_TX	106	TIM15_UP
23	TIM3_CH1	65	UART5_RX	107	TIM15_TRIG
24	TIM3_CH2	66	UART5_TX	108	TIM15_COM
25	TIM3_CH3	67	DAC1_out1	109	TIM16_CH1
26	TIM3_CH4	68	DAC1_out2	110	TIM16_UP
27	TIM3_UP	69	TIM6_UP	111	TIM17_CH1
28	TIM3_TRIG	70	TIM7_UP	112	TIM17_UP
29	TIM4_CH1	71	USART6_RX	113	Reserved
30	TIM4_CH2	72	USART6_TX	114	Reserved
31	TIM4_CH3	73	I2C3_RX	115	Reserved
32	TIM4_UP	74	I2C3_TX	116	UART9_RX
33	I2C1_RX	75	DCMI_PSSI	117	UART9_TX
34	I2C1_TX	76	CRYP_IN	118	USART10_RX
35	I2C2_RX	77	CRYP_OUT	119	USART10_TX
36	I2C2_TX	78	HASH_IN	120	Reserved

Quando se deseja iniciar transferências de DMA com base em eventos programáveis recebidos por meio de sinais de disparo (`dmamux_trgx`), em vez de depender exclusivamente das solicitações dos periféricos, o DMAMUXn pode gerar solicitações de DMA (`dmamux_req_genx`) em múltiplos canais. Cada canal pode ser configurado para

responder a um evento de disparo específico, definido pelos *bits* DMAMUX_RGxCR_SIG_ID[2:0]. O excerto do Manual de Referência resume os sinais que podem ser utilizados como disparos para o DMAMUX1, juntamente com seus respectivos SIG_ID.

Table 102. DMAMUX1: assignment of trigger inputs to resources

Trigger input	Resource	Trigger input	Resource
0	DMAMUX1_evt0	4	lptim2_out
1	DMAMUX1_evt1	5	lptim3_out
2	DMAMUX1_evt2	6	extit0
3	lptim1_out	7	TIM12_TRGO

Um evento de disparo pode ser configurado como uma borda de subida, uma borda de descida ou ambas, por meio dos *bits* DMAMUXn_RGxCR_GPOL[1:0]. Para que o canal do gerador responda a esses eventos de disparo e comece a gerar solicitações de DMA, é necessário ativar o *bit* DMAMUXn_RGxCR_GE. O número total de solicitações de DMA geradas após um evento de disparo é determinado por (GNBREQ + 1), onde GNBREQ é o valor configurado nos *bits* DMAMUXn_RGxCR_GNBREQ[4:0]. O valor 1 é adicionado porque o contador começa em GNBREQ e é decrementado até chegar a zero. Quando o contador atinge zero (*underrun*), o canal do gerador pára de gerar solicitações de DMA. O contador é então recarregado automaticamente com o valor programado nos *bits* DMAMUXn_RGxCR_GNBREQ[4:0], aguardando o próximo evento de disparo.

O módulo Sync no DMAMUX e suas entradas de sincronização (em inglês, *synchronization inputs*), dmamux_syncx, permitem que as transferências de dados sejam iniciadas não apenas por solicitações de periféricos ou por gerações de solicitações pelo DMAMUXn, mas também por eventos sincronizados. Cada canal do DMAMUXn pode ser individualmente sincronizado ativando o *bit* DMAMUX_CxCR_SE. O DMAMUX possui múltiplas entradas, que são conectadas em paralelo a todos os canais do multiplexador de solicitação. Essas entradas de sincronização podem ser configuradas através dos *bits* DMAMUX_CxCR_SYNC_ID[3:0] para receber sinais de eventos de fontes internas ou externas ao microcontrolador. A tabela a seguir mostra as fontes de sincronização e respectivos SYNC_ID.

Table 103. DMAMUX1: assignment of synchronization inputs to resources

Sync. input	Resource	Sync. input	Resource
0	DMAMUX1_evt0	4	lptim2_out
1	DMAMUX1_evt1	5	lptim3_out
2	DMAMUX1_evt2	6	extit0
3	lptim1_out	7	TIM12_TRGO

Quando um canal está configurado no modo de sincronização, a linha de solicitação de DMA selecionada é transmitida para a saída do canal do multiplexador apenas após a detecção de uma borda configurável (subida ou descida) no sinal de sincronização escolhido. O tipo de borda é determinado pelos *bits* DMAMUXn_CxCR_SPOL[1:0]. Ao detectar o evento de

sincronização e, se houver uma solicitação de DMA pendente, o DMAMUXn inicia a transferência. O contador de solicitações é decrementado a cada solicitação atendida pelo controlador de DMA. Quando o contador atinge zero (*underrun*), o canal DMAMUXn pára de gerar solicitações de DMA, e a linha de solicitação de DMA é desconectada da saída do canal. O contador é automaticamente recarregado com o valor programado nos *bits* `DMAMUX_CxCR_NBREQ[4:0]` na ocorrência do próximo evento de sincronização.

O sinal `dmamux_req_outx` representa a saída de solicitação de DMA do canal x do DMAMUX. Ele é conectado à entrada de solicitação de um canal específico do controlador DMA. Quando um periférico ou um evento interno precisa de uma transferência DMA, ele envia uma solicitação para o DMAMUX. O DMAMUX, por sua vez, multiplexa essa solicitação, selecionando o canal DMA apropriado e ativando o sinal `dmamux_req_outx` correspondente.

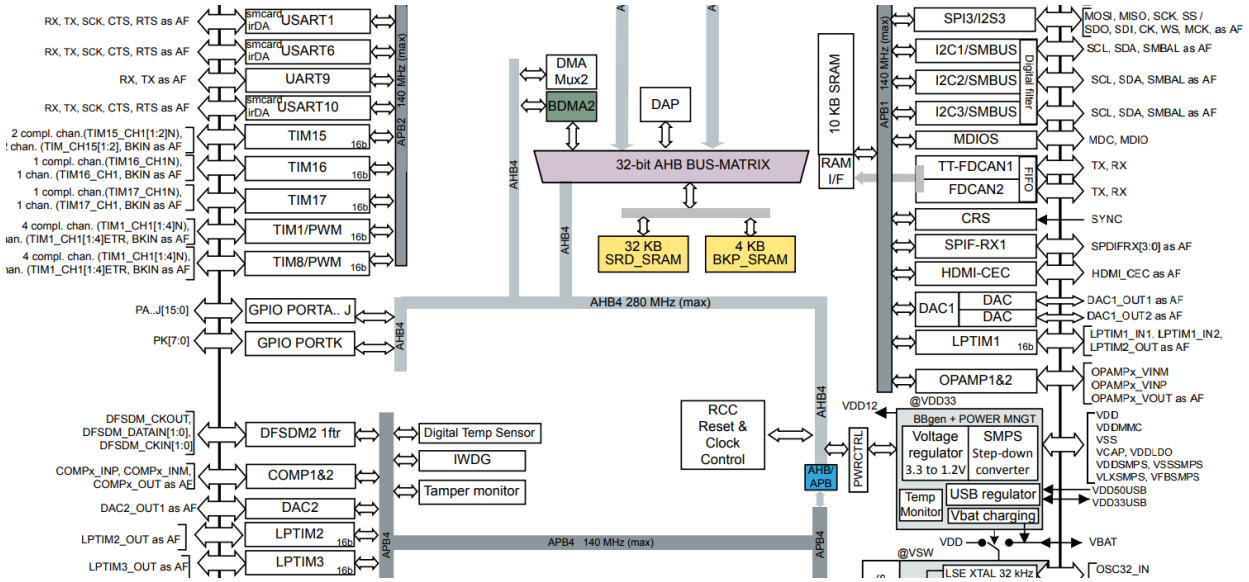
O sinal `dmamux_evtx` indica um evento gerado pelo canal x do DMAMUX. Esses eventos podem ser usados para sincronizar ou disparar ações em outros componentes do sistema. Um evento é, por exemplo, gerado quando o contador de solicitações DMA do canal, configurado pelos *bits* `DMAMUX_CxCR_NBREQ[4:0]`, atinge zero (*underrun*) após uma série de solicitações DMA serem atendidas pelo controlador. Além disso, alguns desses eventos, relacionados à configuração e controle das requisições de DMA, podem ser utilizados para interromper o processador, desde que os *bits* `DMAMUX_CxCR_SOIE` e/ou `DMAMUX_RGxCR_OIE` estejam configurados como "1" e que as linhas de requisição de interrupção do NVIC correspondentes estejam habilitadas.

Signal	Priority	NVIC position	Acronym	Description	Address offset
dmamux1_ovr_it	109	102	DMAMUX1_OV	DMAMUX1 overrun interrupt	0x0000 01D8
dmamux2_ovr_it	135	128	DMAMUX2_OVR	DMAMUX2 overrun interrupt	0x0000 0240

As interrupções permitem que o sistema responda a situações como a conclusão de uma sequência de transferências ou a detecção de condições de erro, sem a necessidade de *polling* constante dos *bits* de estado `DMAMUX_CSR_SOFx` e `DMAMUX_RGSR_OFx`. Após o tratamento da interrupção, é essencial limpar as *flags* correspondentes para que novas interrupções do mesmo tipo possam ser detectadas. Para limpar a *flag* `DMAMUX_CSR_SOFx`, o *software* deve configurar o *bit* correspondente `DMAMUX_CFR_CSOFx`. De forma similar, para limpar a *flag* `DMAMUX_RGSR_OFx`, deve-se escrever "1" no *bit* `DMAMUX_RGCFR_COFx` pelo *software*.

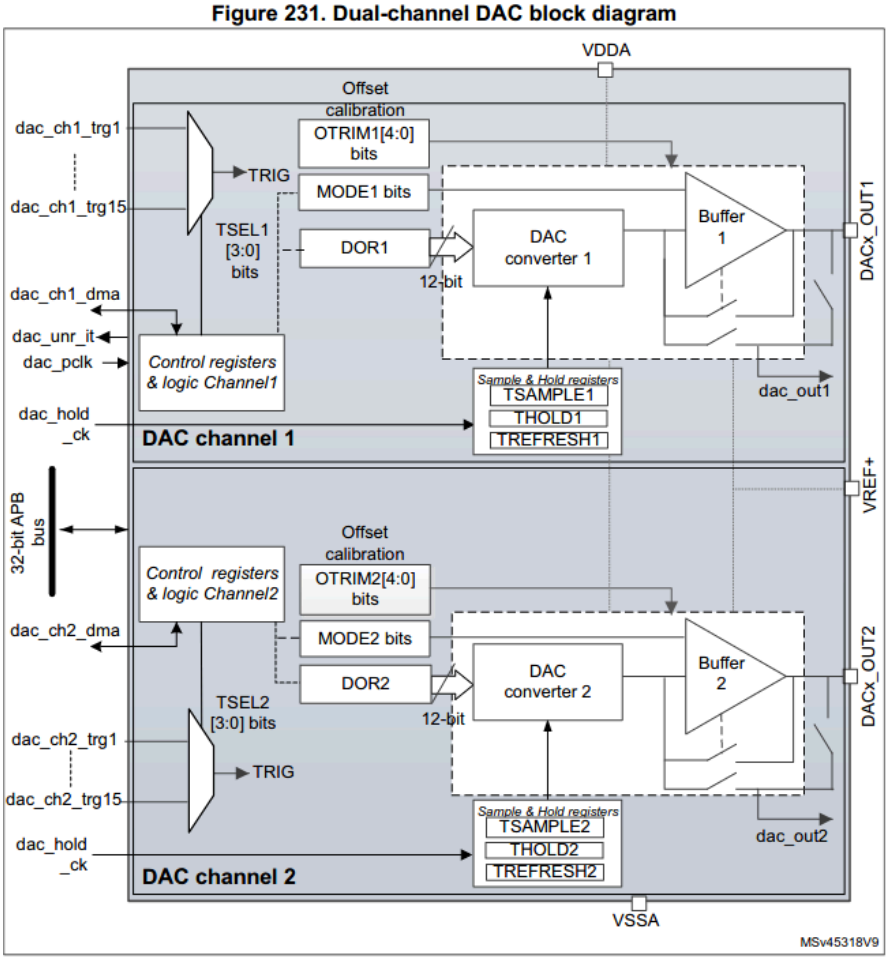
Módulo DAC

O STM32H77A3 possui duas unidades conversoras digital-analógico (DAC), denominadas DAC1 e DAC2. O conversor DAC1 conta com um canal de saída, enquanto o DAC2 oferece dois canais de saída independentes, permitindo a conversão simultânea de dois valores digitais em suas respectivas saídas analógicas. Os DACs estão conectados ao sistema por meio dos barramentos APB1 e APB4, respectivamente. Na abordagem de *clock gating*, é fundamental ativar explicitamente os sinais de relógio para esses módulos. Isso é feito pelos *bits* RCC_APB1LENR_DAC1EN e RCC_APB4ENR_DAC2EN. Essa ativação deve ocorrer antes da configuração, garantindo que o *clock* esteja habilitado. Dessa forma, asseguramos o funcionamento correto dos DACs e evitamos problemas operacionais relacionados à falta de *clock*.



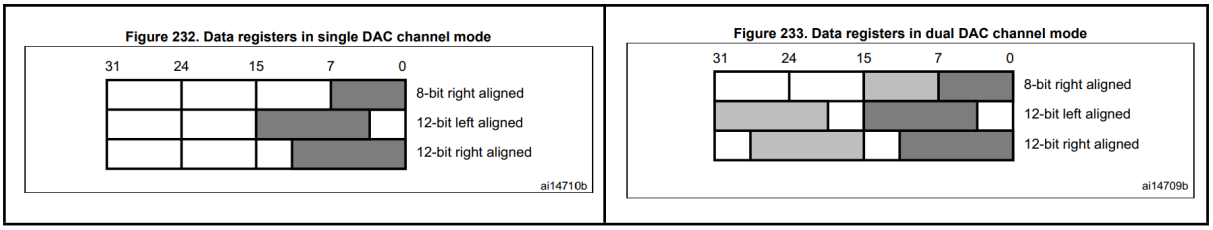
O diagrama de blocos, retirado do Manual de Referência, ilustra claramente a funcionalidade do DAC2 com os blocos distintos “DAC channel 1” e “DAC channel 2”. Cada canal possui seu próprio caminho de entrada de dados e registradores de controle para os pinos de saída, `DACn_OUT1` e `DACn_OUT2`. Essa configuração expande significativamente a capacidade do módulo DAC2 de lidar com dois sinais separados, eliminando a necessidade de um segundo DAC e, assim, economizando espaço na placa e reduzindo a complexidade do circuito.

Figure 231. Dual-channel DAC block diagram



Antes de iniciar a conversão em um canal do DAC, é necessário designar um pino de saída, que deve ser ativado com o registrador RCC_AHB4ENR e configurado como analógico, para esse canal. Além disso, deve-se definir o formato de dados a ser utilizado na conversão por meio dos registradores do bloco "Control registers & logic Channel m". Somente após concluir a configuração e a inicialização do canal é que se deve habilitá-lo individualmente, utilizando o *bit* DAC_CR_ENm. O formato dos dados a serem carregados nos registradores de retenção (`DAC_DHRxR1` para o canal 1 e `DAC_DHRxR2` para o canal 2, onde x representa a resolução desejada (8 *bits* ou 12 *bits*)), varia de acordo com o modo de configuração do DAC, como mostram as figuras extraídas do Manual de Referência:

1. **Canal DAC Único**: Neste modo, os dados podem ter 8 ou 12 *bits*, com alinhamento à esquerda ou à direita. Para alinhamento à direita de 8 *bits*, o *software* deve carregar os dados nos *bits* `DAC_DHR8Rx[7:0]` (armazenados nos *bits* `DAC_DHRx[11:4]`). Para alinhamento à esquerda de 12 *bits*, o *software* deve carregar os dados nos *bits* `DAC_DHR12Lx[15:4]` (armazenados nos *bits* `DAC_DHRx[11:0]`). E para alinhamento à direita de 12 *bits*: o *software* deve carregar os dados nos *bits* `DAC_DHR12Rx[11:0]` (armazenados nos *bits* `DAC_DHRx[11:0]`)



2. **Canais DAC Duplos**: Nesse modo, os dados devem ser de 8 ou 12 *bits*, sempre alinhados à direita. Para alinhamento à direita de 8 *bits*, os dados para o canal 1 do DAC devem ser carregados nos *bits* DAC_DHR8RD[7:0] (armazenados nos *bits* DAC_DHR1[11:4]) e os dados para o canal 2 do DAC devem ser carregados nos *bits* DAC_DHR8RD[15:8] (armazenados nos *bits* DAC_DHR2[11:4]). Para alinhamento à esquerda de 12 *bits*, os dados para o canal 1 do DAC devem ser carregados nos *bits* DAC_DHR12LD[15:4] (armazenados nos *bits* DAC_DHR1[11:0]) e os dados para o canal 2 do DAC devem ser carregados nos *bits* DAC_DHR12LD[31:20] (armazenados nos *bits* DAC_DHR2[11:0]). E para alinhamento à direita de 12 *bits*: os dados para o canal 1 do DAC devem ser carregados nos *bits* DAC_DHR12RD[11:0] (armazenados nos *bits* DAC_DHR1[11:0]) e os dados para o canal 2 do DAC devem ser carregados nos *bits* DAC_DHR12RD[27:16] (armazenados nos *bits* DAC_DHR2[11:0]).

Note que, dependendo do registrador DAC_DHRyyyx carregado, os dados escritos pelo usuário são deslocados e armazenados no correspondente DAC_DHRx, que são registradores de retenção de dados. Estes registradores internos não estão mapeados na memória. Os dados armazenados no registrador DAC_DHRx são transferidos automaticamente para o registrador DAC_DORm após um ciclo de relógio dac_pclk, caso nenhum disparo de *hardware* esteja selecionado (com o *bit* DAC_CR_TENm em "0"). A conversão inicia assim que o *bit* DAC_SWTRGR_SWTRIGm é ativado. Este *bit* é resetado por *hardware* assim que o registrador DAC_DORm for carregado com o conteúdo do registrador DAC_DHRx.

Entretanto, se um disparo de *hardware* for ativado (com o *bit* DAC_CR_TENm em "1"") e ocorrer um sinal de disparo, a transferência será realizada três ciclos de relógio dac_pclk após o evento. Os *bits* de controle DAC_CR_TSELm[3:0] determinam qual dos 16 eventos possíveis, mostrados no Manual de Referência, irá disparar a conversão. Cada vez que a interface do DAC detecta uma borda de subida na fonte de disparo selecionada, os últimos dados armazenados no registrador DAC_DHRx são transferidos para o registrador DAC_DORm. O registrador DAC_DORm é atualizado três ciclos de clock dac_pclk após a ocorrência do disparo. Os *bits* DAC_CR_TSELm[3:0] não podem ser alterados enquanto o *bit* DAC_CR_ENx estiver ativado.

Table 219. DAC1 interconnection

Signal name	Source	Source type
dac_hold_ck	lsi_ck (selected in the RCC)	LSI clock selected in the RCC
dac_chx_trg1 (x = 1, 2)	tim1_trgo	Internal signal from on-chip timers
dac_chx_trg2 (x = 1, 2)	tim2_trgo	Internal signal from on-chip timers
dac_chx_trg3 (x = 1, 2)	tim4_trgo	Internal signal from on-chip timers
dac_chx_trg4 (x = 1, 2)	tim5_trgo	Internal signal from on-chip timers
dac_chx_trg5 (x = 1, 2)	tim6_trgo	Internal signal from on-chip timers
dac_chx_trg6 (x = 1, 2)	tim7_trgo	Internal signal from on-chip timers
dac_chx_trg7 (x = 1, 2)	tim8_trgo	Internal signal from on-chip timers
dac_chx_trg8 (x = 1, 2)	tim15_trgo	Internal signal from on-chip timers
dac_chx_trg11 (x = 1, 2)	lptim1_out	Internal signal from on-chip timers
dac_chx_trg12 (x = 1, 2)	lptim2_out	Internal signal from on-chip timers
dac_chx_trg13 (x = 1, 2)	exti9	External pin
dac_chx_trg14 (x = 1, 2)	lptim2_out	Internal signal from on-chip timers

Quando o registrador `DAC_DORm` é carregado com o conteúdo do `DAC_DHRx`, a tensão de saída analógica se torna disponível após um tempo de estabilização $t_{SETTLING}$, que varia conforme a tensão de alimentação e a carga conectada à saída analógica. As entradas digitais são convertidas em tensões de saída por meio de uma conversão linear entre 0 e a tensão de referência positiva VREF+ pela seguinte equação:

$$DAC_{OUTPUT} = V_{REF} \times \frac{DAC\_DORx}{4096}$$

A saída de cada canal do DAC pode ser configurado em dois modos. Além disso, há um *buffer* de saída que pode ser ativado para melhorar o fluxo de saída. No entanto, antes de habilitar o *buffer* de saída, é necessário calibrar o conversor. Essa calibração é realizada na fábrica (carregada após o *reset*) e pode ser ajustada por *software* durante a operação da aplicação. Assim, existem quatro combinações possíveis, que variam conforme o estado do *buffer* e as interconexões do pino DACx_OUTm para cada modo de saída:

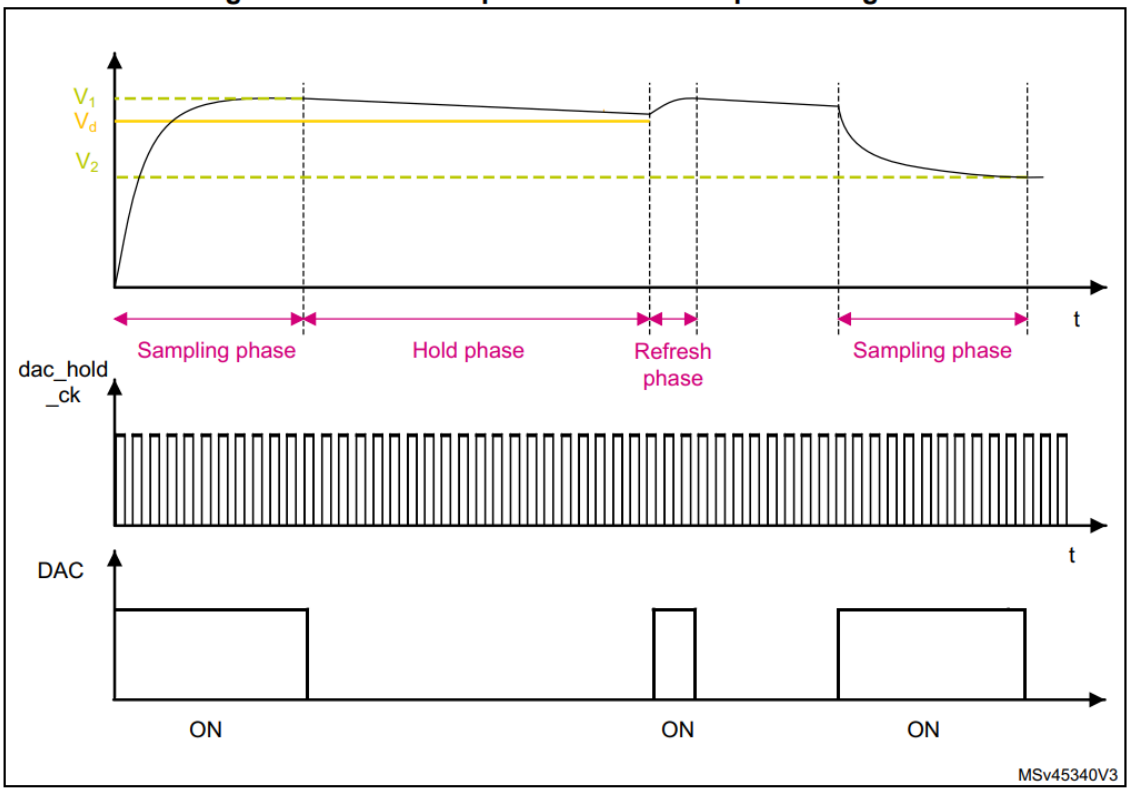
- **Modo Normal:** Para habilitar o *buffer* de saída, os *bits* `DAC_MCR_MODEm[2:0]` devem ser configurados em 0b000, se o DAC está conectado ao pino externo, e em 0b001, se o DAC está conectado ao pino externo e a periféricos internos. Para desabilitar o *buffer* de saída, os *bits* `DAC_MCR_MODEm[2:0]` devem ser configurados em 0b010, se o DAC está conectado ao pino externo, e em 0b011, se o DAC está conectado apenas a periféricos internos.

Table 222. Channel output modes summary

MODEx[2:0]			Mode	Buffer	Output connections
0	0	0	Normal mode	Enabled	Connected to external pin
0	0	1			Connected to external pin and to on chip-peripherals (such as comparators)
0	1	0		Disabled	Connected to external pin
0	1	1			Connected to on chip peripherals (such as comparators)

- **Modo Amostragem-e-Retenção** (modo *sample and hold*): o DAC converte os dados em um valor analógico e, em seguida, mantém a tensão convertida em um capacitor. Quando não está realizando conversões, o DAC e o *buffer* são completamente desligados entre as amostras, e a saída do DAC fica em estado *tri-state*, reduzindo assim o consumo de energia. O tempo de amostragem (em inglês, *sampling phase*) é configurado com os *bits* `DAC_SHSRm_TSAMPLEm[9:0]`. Durante a escrita destes *bits*, o *bit* `DAC_SR_BWSTm` no registrador é setado em "1". Na fase de retenção (em inglês, *hold phase*), o canal de saída do DAC fica em estado *tri-state*, e o DAC e o *buffer* são desligados para reduzir o consumo de potência. O tempo de retenção é configurado com os *bits* `DAC_SHHR_THOLDm[9:0]`, enquanto o tempo de recarga (em inglês, *refresh phase*) é configurado com os *bits* `DAC_SHRR_TREFRESHm[7:0]` no registrador.

Figure 239. DAC Sample and hold mode phase diagram



De forma análoga ao modo Normal, distinguem-se quatro combinações possíveis no modo *sample and hold*.

Table 222. Channel output modes summary (continued)

MODEx[2:0]			Mode	Buffer	Output connections
1	0	0	Sample and hold mode	Enabled	Connected to external pin
1	0	1			Connected to external pin and to on chip peripherals (such as comparators)
1	1	0		Disabled	Connected to external pin and to on chip peripherals (such as comparators)
1	1	1			Connected to on chip peripherals (such as comparators)

A **cooperação entre o DMA, o DAC e o DMAMUX** é fundamental para otimizar a transferência de dados em sistemas microcontrolados. Cada canal do DAC possui capacidade de DMA, e são utilizados dois canais de DMA para atender às requisições de DMA dos canais do DAC. Quando um disparo externo ocorre, e o *bit* DAC_CR_DMAENm está ativado, o valor do registrador DAC_DHRx é transferido para o registrador DAC_DORm. Essa transferência é concluída e um pedido de DMA é gerado. No modo dual, ambos os canais podem gerar pedidos simultaneamente, ou um único pedido pode ser emitido para gerenciar as operações de ambos os canais.

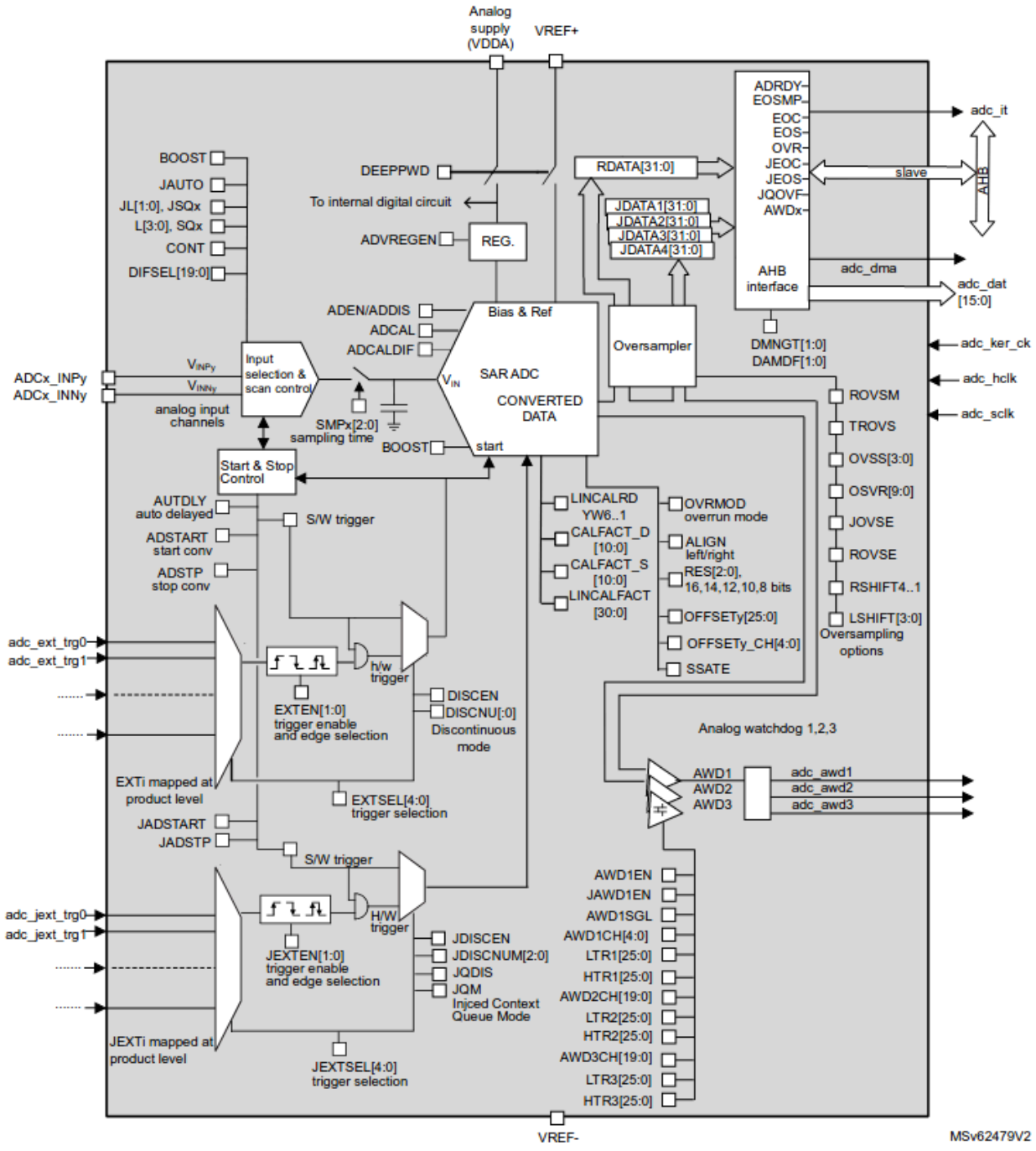
Entretanto, os dados devem ser escritos no DAC_DHRx antes do primeiro evento de disparo, pois a transferência ocorre antes do pedido de DMA. Um aspecto importante a considerar é a subcarga de DMA: como os pedidos de DMA do DAC não são enfileirados, um segundo disparo externo que chegue antes do reconhecimento do primeiro não gerará um novo pedido, ativando a *flag* de subcarga de DMA (DAC_SR_DMAUDRm) e reportando um erro. Isso significa que o canal do DAC continuará a converter dados antigos até que a situação seja resolvida.

Para corrigir uma subcarga de DMA, o *software* deve limpar a *flag* DAC_SR_DMAUDRm, desativar o *bit* DAC_CR_DMAENm do fluxo de DMA utilizado e reinicializar tanto o DMA quanto o canal do DAC. Além disso, ajustes na frequência de conversão do disparo do DAC ou a redução da carga de trabalho do DMA podem ajudar a evitar futuras sobrecargas. Quando tudo estiver configurado corretamente, a conversão do DAC pode ser retomada habilitando as transferências de DMA e os disparos de conversão. Se o *bit* DAC_CR_DMAUDRIEm estiver habilitado, um evento de interrupção será gerado para cada canal do DAC. Além disso, caso a linha de requisição de interrupção IRQ127 esteja ativada, o evento será processado pelo controlador NVIC,

dac2_unr_it	134	127	DAC2	DAC2 underrun interrupt	0x0000 023C

Módulo ADC

O módulo ADC é um periférico integrado no STM32H7A3, responsável por converter sinais analógicos de sensores ou outras fontes em dados digitais, que podem ser processados pelo núcleo do processador. O diagrama de blocos apresentado no Manual de Referência oferece uma visão geral detalhada da arquitetura do módulo ADC, mostrando os vários blocos funcionais e como eles interagem entre si.



Antes de utilizar o ADC, é necessário habilitar o sinal de relógio para o módulo e para o GPIO correspondente aos pinos analógicos. Isto é feito configurando os *bits* apropriados nos registradores RCC_AHB1ENR (para o *clock* do ADC) e RCC_AHB4ENR (para o *clock* do GPIO). Por exemplo, para habilitar o ADC1 e o GPIOC, seria necessário configurar os *bits* `RCC_AHB1ENR_ADC12EN` e `RCC_AHB4ENR_GPIOCEN`, respectivamente.

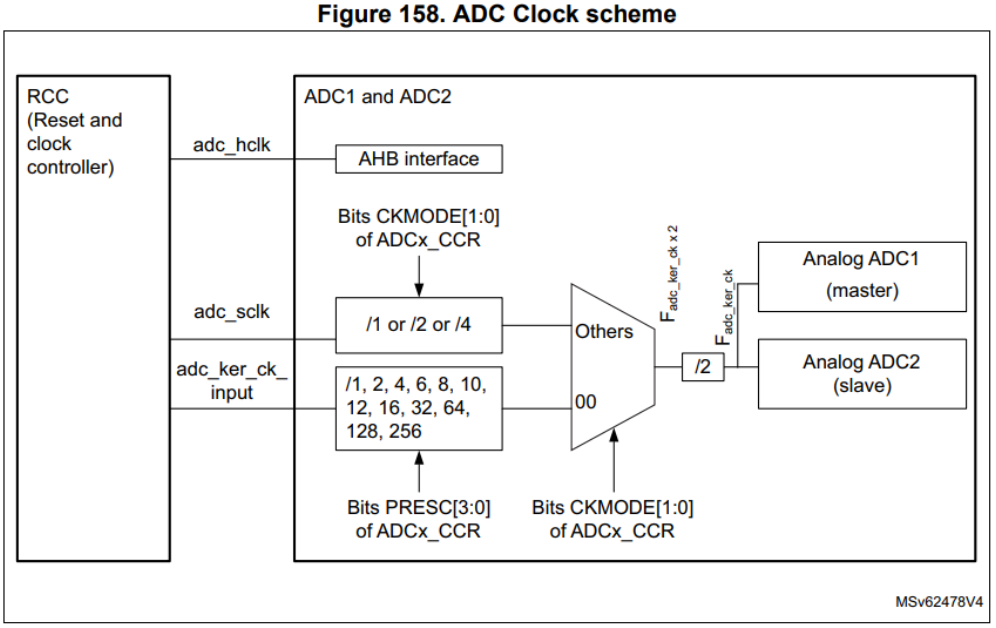
A configuração do ADC envolve diversos parâmetros, que podem ser ajustados através dos registradores do ADC.

Seleção da Fonte de Clock: Os conversores ADC usam um sinal de relógio dedicado, denominado *clock* do ADC, distinto do *clock* do barramento AHB (`adc_hclk`) usado para

acessar os registradores do ADC. Essa abordagem visa minimizar o ruído e o *jitter* do clock, garantindo medições mais precisas e confiáveis. Através dos *bits* `ADC12_CCR_CKMODE[1:0]` do registrador `ADC12_CCR`, que é **comum a dois conversores ADC1 e ADC2 integrados no STM32H7A3**, pode-se definir como o *clock* do ADC:

- uma fonte de *clock* específica, chamada de `adc_ker_ck_input`, que é independente e assíncrona em relação ao *clock* do AHB. Nesta opção, o *clock* do ADC pode ser dividido pelos fatores: 1, 2, 4, 6, 8, 10, 12, 16, 32, 64, 128, 256, configurado os *bits* `ADC12_CCR_PRESC[3:0]`.
- o *clock* do sistema (`RCC_CDCFGR1_HPRE[3:0]` = 0b0xxx) ou o *clock* do sistema dividido por 2 (`RCC_CDCFGR1_HPRE[3:0]` != 0b0xxx), denotado por `adc_sclk`. Nesta opção, um fator de divisor programável pode ser selecionado (/1, 2 ou 4, de acordo com os *bits* `ADC12_CCR_CKMODE[1:0]`, se o *bit* mais significativo de `HPRE` está setado em 0.

Figure 158. ADC Clock scheme



1. Refer to the RCC section to see how adc_hclk and adc_ker_ck_input can be generated.

Todas as interfaces ADC devem usar a mesma fonte de clock se elas não estiverem usando um *prescaler*. Isso sugere que, embora várias interfaces ADC possam compartilhar a mesma fonte de *clock*, elas ainda podem ter um *clock* dedicado separado de outros periféricos.

A descrição funcional do módulo, etapas de configuração e os registradores relevantes são descritas a seguir.

Modos de Operação: O ADC pode operar em três modos distintos: modo **único**, realizando uma conversão a cada disparo; modo **contínuo**, executando conversões de forma ininterrupta; e modo **descontínuo**, realizando conversões em grupos com pausas. Essas conversões regulares seguem uma sequência predefinida nos registradores `ADCn_SQRm`. O modo de operação pode ser configurado por meio dos *bits* `ADC_CFGR_CONT` e `ADC_CFGR_DISCEN`.

Modos de Entrada: O ADC pode ser configurado, através do registrador ADCn_DIFSEL, para operar em modo ***single-ended*** (unipolar), onde a tensão de entrada é medida em relação a uma tensão de referência fixa, geralmente VREF+. Nesse modo, o pino de entrada positivo (ADCx_INPx) é utilizado para medir a tensão de entrada, enquanto o pino negativo (ADCx_INNx) é desconsiderado. Alternativamente, o ADC pode operar em modo diferencial, no qual a diferença de tensão entre dois pinos de entrada é medida, proporcionando uma leitura mais precisa em ambientes com ruído elétrico. O pino ADCx_INPx atua como entrada positiva, e o pino ADCx_INNx atua como entrada negativa. Os valores de saída para um canal m, configurado no modo diferencial, são representados por um tipo de dado sem sinal. Quando VINP[i] é igual a VREF- e VINN[i] é igual a VREF+, a saída é 0x0000 (modo de resolução de 16 *bits*). Por outro lado, quando VINP[i] é igual a VREF+ e VINN[i] é igual a VREF-, a saída é 0xFFFF.

Seleção de Canal: Cada ADC possui até 20 canais multiplexados, que podem ser selecionados através do registrador ADCn_PCSEL. Os pinos que podem ser multiplexados a um canal específico já são definidos pelo *hardware*. Ao selecionar um canal, é importante considerar o pino configurado como entrada analógica no registrador GPIOx_MODER. Com a identificação do pino, podemos consultar a última coluna da Tabela 7 no *Datasheet* para determinar o canal correspondente a ele e, em seguida, habilitar o canal em ADCn_PCSEL. O pino PC4, por exemplo, está mapeado como a entrada positiva (INP) do canal 4 nos módulos ADC1 e ADC2 quando configurado em modo diferencial. Portanto, além de configurar o pino PC4 como analógico, é necessário definir o *bit* 4 do registrador ADCn_PCSEL como "1" para que PC4 funcione como entrada analógica do canal 4 do módulo ADC1.

Pin/ball name[1] [2]															Pin name (function after reset)	Pin type	I/O structure	Alternate functions	Additional functions
LQFP100 with SMPS	TFBGA100 with SMPS	LQFP144 with SMPS	WLCSP132 with SMPS	UFBGA169 with SMPS	UFBGA176+25 with SMPS	LQFP176 with SMPS	TFBGA225 with SMPS	LQFP64	TFBGA100	LQFP100	LQFP144	UFBGA176+25	LQFP176	TFBGA216					
34	K3	46	K9	J6	N6	52	R5	23	K3	31	43	R3	53	R3	PA7	I/O	FT_ah1	TIM1_CH1N, TIM3_CH2, TIM8_CH1N, DFSDM2_DATIN1, SPI1_MOSI/I2S1_SDO, SPI6_MOSI/I2S6_SDO, TIM14_CH1, OCTOSPIM_P1_IO2, FMC_SDNWE, LCD_VSYNC, EVENTOUT	ADC12_INP7, ADC12_INN3, OPAMP1_VINM
35	H4	47	H7	K6	R6	53	M6	24	G4	32	44	N5	54	N5	PC4	I/O	FT_a	DFSDM1_CKIN2, I2S1_MCK, SPDIFRX1_IN2, FMC_SDNE0, LCD_R7, EVENTOUT	ADC12_INP4, OPAMP1_VOUT, COMP1_INM

Calibração: Cada ADC possui uma calibração de fábrica para garantir precisão em condições normais. No entanto, variações de tensão, temperatura e processo de fabricação podem afetar a precisão do ADC. A calibração compensa esses fatores. Antes de realizar as conversões, é recomendado calibrar o ADC. O procedimento de calibração envolve os seguintes passos:

1. **Configuração:**
    a. **Habilitar o Regulador de Tensão:** Assegurar que o regulador de tensão interno do ADC esteja ativo configurando o *bit* ADC_CR_ADVREGEN. em "1". Aguardar a estabilização do regulador verificando a *flag* ADC_ISR_LDORDY no registrador.
    b. Selecionar o Tipo de Calibração: Escolher entre calibração *single-ended* (ADC_CR_ADCALDIF=0) ou diferencial (ADC_CR_ADCALDIF=1). Para STM32H7A3, a calibração deve ser realizada separadamente para cada modo de entrada: *single-ended* e diferencial.
    c. **Ativar a Calibração de Linearidade (opcional):** Se necessário, habilitar a correção de linearidade sentando em "1" o *bit* ADC_CR_ADCALLIN.
2. **Iniciar a Calibração:** Iniciar o processo de calibração escrevendo "1" no *bit* ADC_CR_ADCAL. O *hardware* realiza a calibração automaticamente.
3. **Aguardar a Conclusão:** Monitorar o *bit* ADC_CR_ADCAL. Ele é resetado automaticamente em "0" pelo *hardware* quando a calibração é concluída.
4. **Leitura dos Fatores de Calibração (opcional):** Os fatores de calibração, calculados pelo *hardware*, podem ser lidos nos registradores ADCn_CALFACT1 e ADCn_CALFACT2, se necessário.

Tempo de Amostragem: O tempo de amostragem determina quanto tempo o ADC leva para amostrar o sinal analógico. Antes de iniciar uma conversão, o ADC deve relacionar a fonte de tensão que está sendo medida e o capacitor de amostragem embutido no ADC. Esse tempo de amostragem deve ser suficiente para que a fonte de tensão carregue o capacitor embutido até o nível de tensão de entrada. Ou seja, se a fonte de tensão apresentar uma impedância de saída elevada, o tempo de amostragem deverá ser maior (constante RC). Cada canal pode ser amostrado com um tempo de amostragem diferente, que é programável utilizando os *bits* ADC_SMPR1_SMP[2:0] (canais 0 até 9) ou ADC_SMPR2_SMP[2:0] (canais 10 até 19). Para as conversões regulares, o ADC notifica o término da fase de amostragem ao ativar o *bit* de estado ADC_ISR_EOSMP.

Resolução: O ADC oferece resoluções programáveis de 16, 14, 12, 10 e 8 *bits*, definidas pelos *bits* ADC_CFGR_RES[1:0]. Resoluções mais baixas resultam em tempos de conversão mais rápidos, o que é essencial em aplicações sensíveis ao tempo, além de consumir menos energia, beneficiando dispositivos alimentados por bateria. Por outro lado, resoluções mais altas proporcionam maior precisão nas medições. A tabela extraída do Manual de Referência ilustra o efeito de resoluções sobre os tempos de conversão

Table 198. T_{SAR} timings depending on resolution

RES [2:0]	T_{SAR} (ADC clock cycles)	T_{SAR} (ns) at $F_{adc\_ker\_ck}$=24 MHz	$T_{adc\_ker\_ck}$ (ADC clock cycles) (with Sampling Time= 1.5 ADC clock cycles)	$T_{adc\_ker\_ck}$ (ns) at $F_{adc\_ker\_ck}$=24 MHz
16 bits	8.5 ADC clock cycles	354.2	10 ADC clock cycles	416.7
14 bits	7.5 ADC clock cycles	312.5	9 ADC clock cycles	375
12 bits	6.5 ADC clock cycles	270.8	8 ADC clock cycles	333.3
10 bits	5.5 ADC clock cycles	229.2	7 ADC clock cycles	291.7
8 bits	4.5 ADC clock cycles	187.5	6 ADC clock cycles	250.0

Inicialização do ADC: Após a configuração dos parâmetros desejados, o ADC precisa ser habilitado e inicializado para iniciar as conversões. Isso é realizado configurando os *bits* ADC_CR_ADEN, que ativam o módulo ADC, e verificando o *bit* ADC_ISR_ADRDY, que indica a prontidão do ADC para iniciar as conversões.

Disparo de Conversão: O disparo de conversão se refere ao evento que inicia o processo de transformação de um sinal analógico em um valor digital. Essa conversão pode ser acionada por *software*, *hardware* (como um *timer* ou um sinal externo) ou por uma combinação de ambos. No caso do disparo por *software*, a conversão é iniciada ao se escrever "1" no *bit* ADC_CR_ADSTART. Já o disparo por *hardware* ocorre quando um evento externo, gerado por outro periférico, como um *timer*, inicia a conversão. A seleção da fonte de disparo externo é feita por meio dos *bits* ADC_CFGR_EXTSEL[4:0], que permitem escolher a partir de uma lista de eventos disponíveis. Além disso, a polaridade do sinal de disparo externo (borda de subida, borda de descida ou ambas) pode ser configurada utilizando os *bits* ADC_CFGR_EXTEN[1:0]. Essa flexibilidade possibilita a sincronização das conversões do ADC com outros eventos do sistema, garantindo uma operação mais integrada e eficiente.

Leituras do ADC e Resultados: O resultado da conversão, com até 16 *bits* de precisão, é armazenado em um registrador de dados ADCn_DR de 32 *bits,* que pode ser alinhado à esquerda ou à direita. O alinhamento dos dados armazenados após a conversão é selecionado pelos *bits* ADC_CFGR2_OVSS[3:0] e ADC_CFGR2_LSHIFT[3:0]. O valor convertido é um número binário que representa a amplitude do sinal analógico amostrado. Ele reflete a relação entre as tensões amostradas, V_{INP} e V_{INN}, a tensão de referência V_{REF+} e a escala cheia, conforme descrito a seguir:

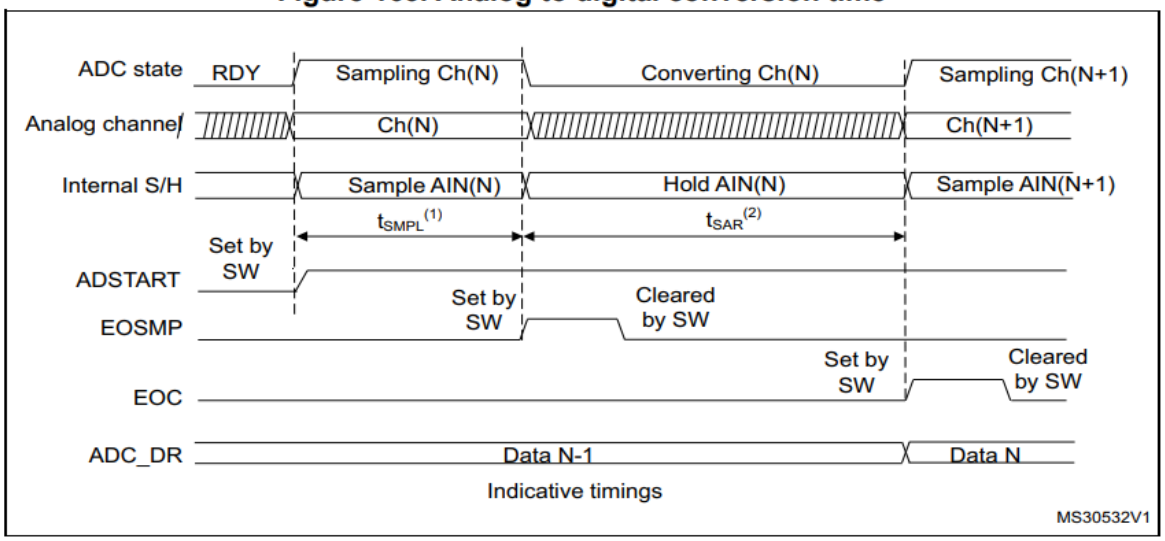
$$\text{Converted value} = \frac{\text{ADC\_Full\_Scale}}{2} \times \left[1 + \frac{V_{INP} - V_{INN}}{V_{REF+}}\right]$$

O diagrama de tempo apresentado no Manual de Referência sintetiza os principais tempos envolvidos numa conversão completa de uma amostra analógica.

$$T_{CONV} = T_{SMPL} + T_{SAR} = [1.5\ |_{min} + 7.5\ |_{14bit}] \times T_{adc\_ker\_ck}$$

$$T_{CONV} = T_{SMPL} + T_{SAR} = 62.5\ ns\ |_{min} + 312.5\ ns\ |_{14bit} = 375.0\ ns\ (for\ F_{adc\_ker\_ck} = 24\ MHz)$$

Figure 165. Analog to digital conversion time



1. T_{SMPL} depends on SMP[2:0]
2. T_{SAR} depends on RES[2:0]

O ADC do STM32H7A3 suporta DMA para facilitar a transferência de dados amostrados diretamente para a memória, sem a intervenção da CPU. Para habilitar esse acesso, é necessário seguir algumas etapas de configuração:

1. **Habilitar o DMA no ADC**: Deve-se setar em "1" o *bit* ADC_CFGR_DMNGT[1:0] para habilitar as solicitações de DMA após a conclusão de uma conversão. Isso permite que o ADC gere um sinal de solicitação ao DMA.

DMA request MUX input	Resource
1	dmamux1_req_gen0
2	dmamux1_req_gen1
3	dmamux1_req_gen2
4	dmamux1_req_gen3
5	dmamux1_req_gen4
6	dmamux1_req_gen5
7	dmamux1_req_gen6
8	dmamux1_req_gen7
9	ADC1
10	ADC2
11	TIM1_CH1
12	TIM1_CH2

2. **Configuração do DMAMUX**: O DMAMUX deve ser configurado para direcionar a solicitação de *trigger* do ADC para o canal DMA correspondente. Isso é realizado configurando os *bits* DMAMUX_CxCR_DMAREQ_ID[6:0] com a identificação do canal associado ao módulo ADC. Para o ADC1, essa identificação é 9, enquanto para o ADC2, é 10. Essa configuração assegura que o DMAMUX reconheça corretamente qual canal deve receber as solicitações de DMA provenientes do ADC.
3. **Configuração do canal DMA**: É necessário configurar o canal DMA, definindo os endereços de origem e destino nos registradores DMA_SxPAR (endereço periférico, que aponta para o registrador de dados do ADC) e DMA_SxM0AR (endereço de memória, que aponta para o *buffer* onde os dados serão armazenados).
4. **Direção e modo da transferência**: Configurar o registrador DMA_SxCR para especificar a direção da transferência (normalmente de periférico para memória), o tamanho dos dados, o modo de operação (por exemplo, circular para transferências contínuas) e a prioridade da solicitação.
5. **Habilitar o canal DMA**: Finalmente, o canal DMA deve ser habilitado definindo o *bit* DMA_SxCR_EN.

Adicionalmente, o ADC pode gerar interrupções para sinalizar eventos importantes, como a conclusão de uma conversão ou a ocorrência de um erro. Essas interrupções podem ser habilitadas e configuradas através dos bits do registrador ADCn_IER, além dos *bits* correspondentes à IRQ18 nos registradores do controlador NVIC. As *flags* no registrador de estado ADCn_ISR podem ser resetadas escrevendo "1" nos *bits* correspondentes, permitindo a correta monitorização dos eventos do ADC.

adc1_it adc2_it	25	18	ADC1_2	ADC1 and ADC2 global interrupt	0x0000 0088