DISCIPLINA EA701
Introdução aos Sistemas Embarcados

ROTEIRO 9: Entradas/Saídas Analógicas
Domínios Analógicos e Digitais, Conversores DAC e ADC, Interface Analógica em Microcontroladores, DMA (DMAC, DMAMUX e *Buffer* Circular), Sensores Analógicos (Sensor de Temperatura LM61, Potenciômetro, *Joystick*), Atuador Analógico (Disco Piezoelétrico)

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

FEEC / UNICAMP

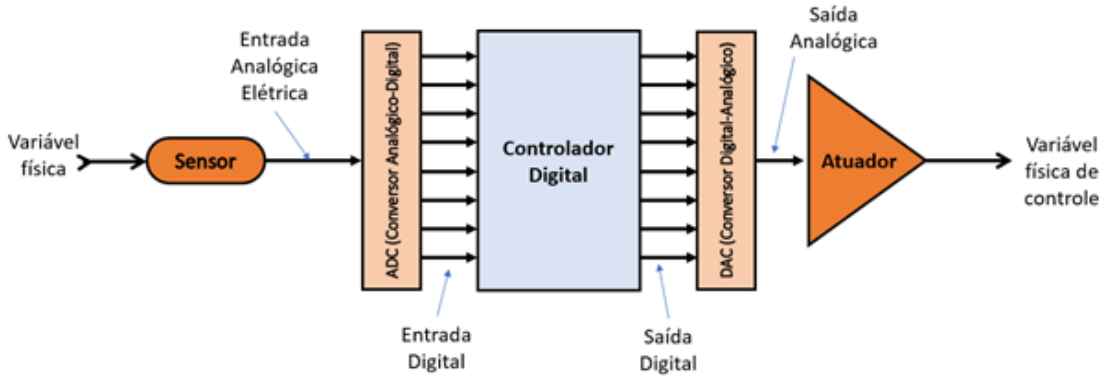
Revisado e modificado em abril de 2025 por Ting com auxílio do Chatgpt

Revisado em outubro de 2024

INTRODUÇÃO

O mundo elétrico pode ser entendido de duas formas principais: analógica e digital. O mundo digital representa grandezas físicas com valores discretos (códigos binários), interpretados por computadores. Já o mundo analógico lida com grandezas contínuas (tensão, corrente, temperatura, som), que variam suavemente. A manipulação de sinais é mais eficiente no digital, mas a interação de sistemas digitais com o mundo físico, predominantemente analógico, exige superar desafios de compatibilidade entre esses dois domínios.

Para aproveitar a capacidade de processamento dos sistemas digitais no controle de dispositivos com base nos sinais analógicos amostrados 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** (em inglês, *Digital to Analog Converter* – DAC). Por sua vez, a conversão de sinais analógicos para digitais é feita pelos **Conversores Analógico-Digitais** (em inglês, *Analog to Digital Converter* – ADC). 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

Desenvolveremos cinco projetos neste Roteiro. Inicialmente, no primeiro projeto, sintetizaremos formas de onda triangular e senoidal utilizando os DACs do microcontrolador. Em seguida, nos dois projetos subsequentes, concentraremos na leitura de sinais analógicos através do ADC, explorando duas abordagens distintas: conversão simples iniciada por *software* e conversão iniciada por *hardware* via *timer*. Finalmente, nos dois últimos projetos, utilizaremos os módulos DMA e DMAMUX para a transferência direta de dados entre os conversores e a memória, empregando o controlador de DMA e o roteador DMAMUX.

Projeto de conversão D/A básica

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 analógicos a partir das amostras digitais, uma das maneiras de se fazer isto é usando o Conversor Digital-Analógico (DAC). Vamos explorar neste projeto uma forma direta de transferência de amostras digitais para o DAC, com a taxa de transferência controlada por interrupção de *timer*. Utilizaremos os dois DACs do microcontrolador para gerar, de forma simultânea, duas formas de onda distintas: uma triangular e outra senoidal. As amostras inteiras, com valores entre 0 e 4096, serão calculadas e convertidas para tensões analógicas no intervalo de 0 a 3.3V a cada interrupção do núcleo.

1. Crie um novo projeto usando o *Cube*, com o nome “DAC_Basico”, **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. Ainda nesta etapa de preparação, vamos criar os protótipos das funções de configuração dos periféricos que vamos utilizar (DAC para conversão e TIM7 para intterrupções periódicas). No escopo de `/* USER CODE BEGIN PFP */`, declare os protótipos das funções de configuração desses periféricos:

```
void Config_DAC(void);
void Config_Timer(void);
```

e funções definidas no arquivo `stm32h7xx_it.c`:

```
uint8_t leFlag();
void resetFlag();
```

5. 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 as variáveis locais:

```
int8_t step = 0;         // indice de amostra da onda triangular
int8_t direction = 1;  // 1 para subir, -1 para descer
uint8_t i = 0;          // indice de amostra da onda senoidal
uint32_t dac_value, sine_wave;
```

6. Antes de configurar os periféricos, vamos chamar as funções de configuração dos periféricos.

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

7. Dentro do laço infinito, a cada interrupção do *timer*, o canal DAC1_OUT2 (PA5) e o canal DAC2_OUT1 (PA6) são carregados com novas amostras digitais da onda triangular e senoidal, respectivamente. Insira no escopo `/* USER CODE BEGIN 3 */` os cálculos das amostras digitais `dac_value` e `sine_wave` correspondentes aos índices `step` e `i` das ondas triangular e senoidal, respectivamente.

```
if (leFlag()) {
	//Reseta flag
	resetFlag();
	// 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)
	dac_value = (step * 4095) / 50;
	//Calcula o proximo valor da onda senoidal
```

```
        sine_wave = (uint32_t)(OFFSET + (OFFSET * sin(2 * PI * i /
SAMPLES)));
        if(sine_wave > DAC_MAX_VALUE) {
                sine_wave = DAC_MAX_VALUE;
        }
        i++;
        if (i == SAMPLES) i = 0;
        // Atualizar os valores do DAC1_OUT2 e DAC2_OUT1
        DAC1->DHR12R2 = dac_value;    //PA5
        DAC2->DHR12R1 = sine_wave;    //PA6
}
```

O procedimento implementa uma rampa com comportamento ascendente nos primeiros 50 passos e descendente nos 50 subsequentes, além de gerar uma senóide completa ao longo de 100 passos.

8. Agora, precisamos 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 os pinos 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*.

9. Ainda no mesmo escopo, vamos definir a função de configuração do *timer*. Vamos usar o *timer* 7 para a cadência das duas ondas. Abaixo da função anteriormente criada, escreva essa nova função:

```
void Config_Timer(void) {
	// Enable clock for TIM6 e TIM7
	RCC->APB1LENR |= RCC_APB1LENR_TIM7EN;
	// TIM7: Faz interrupcao periodica
	// Resetar os registradores de TIM7
	TIM7->EGR |= TIM_EGR_UG_Msk;
	while (TIM7->EGR & TIM_EGR_UG);
	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* 7. foi realizado o procedimento padrão para interrupção periódica, com um período de 50kHz. Como vamos gerar as ondas com 100 amostras por ciclo, teremos 500 ciclos por segundo, ou seja, uma frequência de 500Hz.

10. Por fim, é necessário definir a ISR para sinalizar as ocorrências de interrupções periódicas para atualização das amostras. Abra o arquivo “stm32h7xx_it.c”. Vamos inicialmente criar 1 variável global no escopo `/* USER CODE BEGIN PV */`

```
uint8_t flag;
```

declarar protótipos de duas funções no escopo `/* USER CODE BEGIN PFP */`

```
uint8_t leFlag();
void resetFlag();
```

e implementar essas funções no escopo `/* USER CODE BEGIN EV */`

```
uint8_t leFlag() {
	return flag;
}
void resetFlag() {
	flag = 0;
}
```

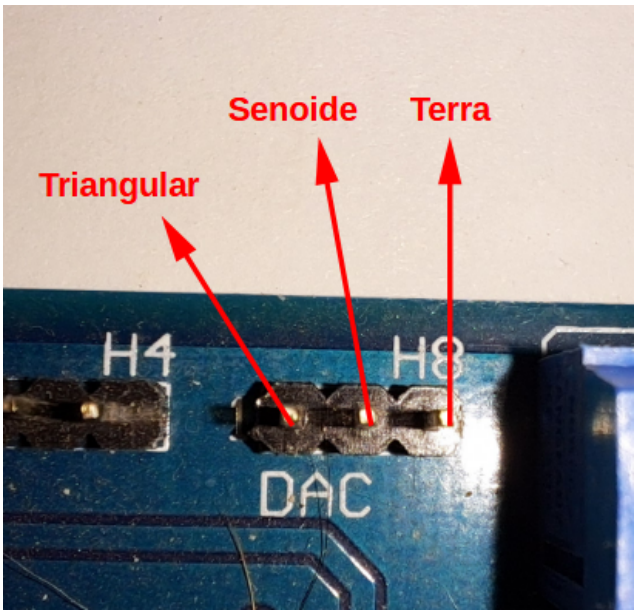
11. No escopo de `/* USER CODE BEGIN 1 */`, implemente a ISR:

```
void TIM7_IRQHandler(void) {
   // Verificar se a interrupcao e de update
   if (TIM7->SR & TIM_SR_UIF) {
       TIM7->SR &= ~TIM_SR_UIF;  // Limpar o flag de interrupcao
       flag = 1;
```

```
    }
}
```

Essencialmente, esta rotina de serviço indica uma interrupção periódica, a qual dispara o envio de um novo par de amostras para gerar as formas de onda.

12. 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 o pino 7 do conector H9 (ADC). Veja a figura a seguir.



13. Transfira o código para o microcontrolador 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 configurada para o *timer*.

14. Dê um *zoom* nos sinais para ver a discretização dos mesmo, caracterizada por pequenos "degraus". Para reduzir a percepção desses degraus, podemos simplesmente aumentar o número de amostras? No caso da senóide, a geração de amostras e o controle da transferência de dados são parametrizados pela macro `SAMPLES`. Portanto, alterar o valor de `SAMPLES` em sua definição global aplicará a modificação a todo o programa. Experimente modificar o valor de `SAMPLES` de 100 para 200, recompilar e re-executar o programa. Essa alteração afetou o intervalo entre duas amostras? E o período do sinal gerado, foi alterado? Qual o efeito visual do aumento do número de amostras na forma de onda exibida?

Considerando agora a redução do valor do registrador de auto-recarga pela metade (para 10-1), qual seria o novo intervalo entre as amostras e o período do sinal gerado?

Você consegue descrever como os sinais podem ser suavizados? Caso nenhuma solução para a suavização surja neste momento, podemos prosseguir e revisitar essa questão posteriormente.

15. Para fazer o mesmo com a onda triangular, todos os valores "50" dentro da ISR devem ser mudados para "100, mantendo o valor de auto-recarga em (10-1). Estes valores ocorrem em 3

das linhas de código da `main`, sendo de fácil localização. Certifique se mudou as 3 ocorrências do valor 50 para 100 antes de recompilar. Re-execute o programa. Veja agora os "degraus" no sinal. O que mudou? Você consegue explicar essa mudança? Já consegue dizer quais são os parâmetros que controlam a suavidade de um sinal digital? Caso não tenha uma solução ainda, prossigamos e veremos se essa ideia surge mais adiante.

16. Vamos analisar a relação entre as amostras digitais (entre 0 e 4096) e os correspondentes valores analógicos (na faixa de 0V a 3.3V). Para isso, insira um breakpoint na linha de instrução "i++;". Quando a execução do programa pausar nesse ponto, utilize um **multímetro** para medir a tensão nos pinos PA5 e PA6 para os valores da variável `step` indicados na primeira coluna da tabela a seguir. Registre tanto os valores de tensão medidos quanto os valores das amostras correspondentes.

`step`	`i`	`sine_wave`	PA6 (em Volts)	`dac_value`	PA5 (em Volts)
10					
25					
50					
75					
99					

Posteriormente, investigue a correlação entre os valores das amostras e os resultados das conversões nos canais analógicos. Essa relação é linear? Se você quiser quantificar essa relação, use uma técnica de regressão.

17. Se conectarmos um *buzzer* entre o pino 1 e o terra ou entre o pino 2 e o terra, e aproximá-lo do seu ouvido, qual será sua percepção sonora? Você consegue explicar o fenômeno observado? Nessa configuração específica, o *buzzer* funciona como um sensor ou um atuador? Trata-se de um periférico digital ou analógico? Caso esses conceitos ainda não estejam totalmente claros, não se preocupe. Eles serão detalhados em seções futuras, e então você terá as ferramentas para explicar suas observações.

18. Declaramos `dac_value` e `sine_wave` como inteiros de 32 *bits*. Precisamos de tantos *bits*? Investigue o tamanho dos dados para os quais essas variáveis irão passar para descobrir.

19. Você já se questionou sobre o momento em que a conversão de cada valor digital tem início? É possível para um desenvolvedor configurar com precisão o início de cada conversão no STM32H7? A resposta a estas questões será abordada posteriormente.

20. Qual técnica de conversão digital-analógica é utilizada nos conversores DAC do microcontrolador STM32H7A? Caso a resposta não seja imediata, não se preocupe, ela será apresentada em breve.

Projeto de conversão A/D básica

Sabe como um voltímetro 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 das amostras coletadas 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 |= ADC_SQR1_SQ1_2;  // 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 |= ADC_SMPR1_SMP4_2;  // Amostragem de 32.5 ciclos de ADC
	ADC1->PCSEL |= ADC_PCSEL_PCSEL_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 o pino 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 circuito 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 iniciada com um disparo por *software*, aguarda por *polling* a conclusão da conversão 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
	// (sobrescrita do valor anterior ainda não acessado)
	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 por *software*** (setando o *bit* ADSTART) e entra em um *loop* testando o *flag* EOC (*End of Conversion*). Por fim, limpa o *flag* de *overrun* (evento de estouro do *buffer*, quando os dados de conversão não são lidos pelo processador) 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 uma conversão (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);
```

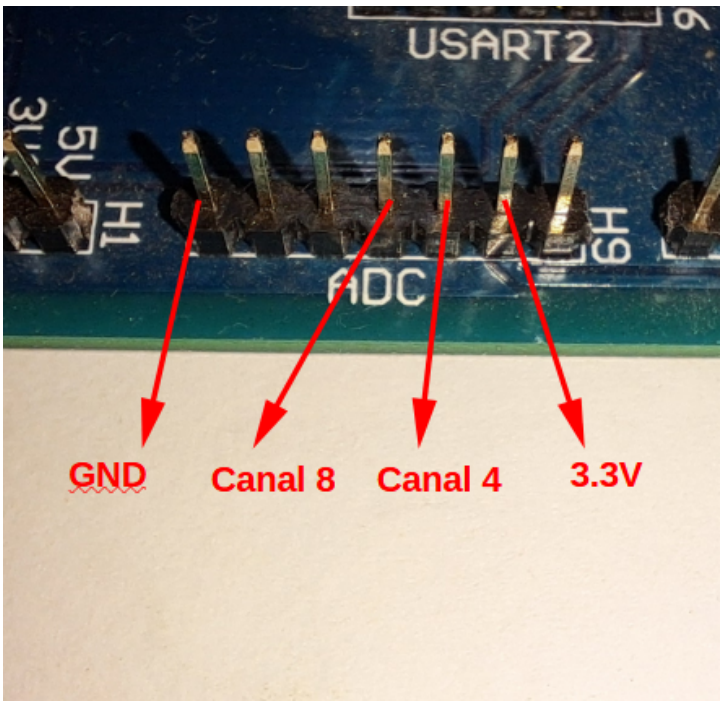
A primeira linha realiza a conversão A/D, guardando o resultado na variável "adc". A segunda linha calcula o valor de tensão correspondente ao valor lido no ADC, sendo o valor resultante expresso em Volts, em uma variável de ponto flutuante. A terceira linha realiza uma espera de 100ms antes de uma nova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

A implementação da função `HAL_Delay(X)`, que introduz um atraso de X milissegundos no fluxo de controle, é semelhante à da função `void Delay(uint32_t ms)` presente no projeto `Delay_1ms` do Roteiro 5. Por padrão, o módulo `SysTick`, configurado com um período de 1ms, é inicializado no código gerado pelo STM32CubeMX. Quando `HAL_Delay` é chamada, a interrupção do `SysTick` é habilitada, e o incremento do seu contador é monitorado dentro da rotina de serviço de interrupção `SysTick_Handler`, localizada no arquivo `stm32h7xx_it.c`.

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 *cursor* (braço móvel) do potenciômetro para outra posição e dê um *Resume*, experimentando vários valores, sempre comparando a tensão medida no potenciômetro com o valor calculado no programa. Para comparação, preencha a seguinte tabela com os valores de 5 amostras.

Amostra	“`adc`”	“`tensao`” (V)	Tensão Medida (V)
1			
2			
3			

4			
5			

Como você explica a fórmula usada para converter o valor “bruto” do ADC (“`adc`”) em um valor de tensão correspondente (“`tensao`”)? Qual é a diferença média entre os valores obtidos pelo conversor ADC e pelo multímetro?

Qual é a relação entre as amostras analógicas (Tensão Medida) e os valores digitais (`adc`) observados? Essa relação é linear? A equação obtida por meio de uma técnica de regressão se assemelha à fórmula utilizada para converter o valor `adc` em `tensao`?

9. Nesse projeto, o potenciômetro atua como um sensor ou um atuador analógico? Caso esses conceitos ainda não estejam claros, não se preocupe, eles serão detalhados posteriormente e você conseguirá responder.

10. Para iniciar uma conversão do ADC via *software*, qual registrador é acionado? Após essa ativação, o ADC efetua uma única leitura ou prossegue com conversões em sequência? Qual é o impacto da ausência dessa inicialização no processo de conversão? As respostas para essas perguntas serão exploradas no próximo projeto-exemplo.

11. Quais modificações você faria no projeto se mudarmos a resolução de conversão de 16 *bits* para 10 *bits*? Nós veremos mais adiante a resposta para esta pergunta.

12. Qual é a taxa de superamostragem configurada para o controlador ADC neste projeto? Onde está essa configuração? Se não tiver nehuma ideia, vamos descobrir onde está essa configuração juntos.

13. Considerando que o registrador `ADC1->SQR1` define a sequência de conversão e, por conseguinte, a ordem dos canais a serem convertidos, qual a justificativa para a necessidade de pré-selecionar um canal separadamente, através do registrador `ADC1->PCSEL`, se este já está explicitamente incluído na referida sequência? Parece redundante, não é? Adiante, exploraremos como essa aparente duplicação reflete as nuances das soluções de *hardware*, que podem variar significativamente entre projetos, resultando em diferentes arquiteturas de registradores de configuração.

Projeto de conversor A/D disparado por *hardware* e geração de evento de “fim 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 informações. Vamos explorar como obter amostras periódicas de um sensor de temperatura. Você poderia fazer uma conversão A/D única, controlando o momento de aquisição pelo *software* e aguardando o fim de uma conversão por *polling*, como vimos no projeto anterior. Mas e se quisermos uma

coleta contínua, com uma taxa de amostragem consistente? É possível aprimorarmos o nosso projeto com mínima sobrecarga de *software*?

Uma alternativa para eliminar o *polling* seria habilitar o evento de interrupção de "fim de conversão". Dessa forma, o fluxo de controle seria desviado por *hardware* para a rotina de serviço `ADC_IRQHandler` assim que uma conversão fosse concluída. Para implementar uma coleta periódica, podemos utilizar um temporizador configurado para gerar interrupções periódicas. Dentro da rotina de tratamento dessas interrupções, iniciaríamos uma nova conversão, garantindo a periodicidade das aquisições. Com essas interrupções periódicas e a habilitação das interrupções de "fim de conversão", apenas duas intervenções por *software* seriam necessárias: uma para iniciar a conversão por *software* e outra para ler o resultado.

E se substituirmos as conversões iniciadas por *software* por conversões disparadas por *hardware*? Imagine o sinal de fim de contagem de um período de um temporizador conectado diretamente ao pino de inicialização da conversão ADC. Nesse cenário, a conversão seria iniciada automaticamente a cada evento de interrupção periódica gerado pelo temporizador. Dessa forma, o programa se tornaria mais eficiente, dedicando-se exclusivamente ao processamento dos resultados da conversão, acessíveis na rotina de serviço `ADC_IRQHandler`.

Considere agora que esse sinal analógico provém de um sensor de temperatura, como o sensor LM61, que produz uma tensão proporcional à temperatura medida. Com essa abordagem, você não apenas monitoraria a temperatura de forma periódica, mas também teria a oportunidade de explorar 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->EGR |= TIM_EGR_UG_Msk;     // atualizacao inicial dos registradores
	while (TIM6->EGR & TIM_EGR_UG);
	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, configuramos o temporizador para gerar um evento de atualização a cada 100 ms, o que corresponde a uma frequência de 10 vezes por segundo. Embora a interrupção do temporizador não esteja sendo habilitada, o TIM6 está sendo configurado para emitir um sinal de evento ao ocorrer cada atualização através de TIM6_CR2. Esse sinal de evento pode ser roteado para diversos periféricos e, caso estes estejam configurados para tal, pode deflagrar o início de suas operações. Em nosso projeto, o evento de atualização do TIM6 será utilizado para disparar automaticamente o início da conversão do ADC1, que será configurado para essa finalidade. Observe que, nesta etapa da configuração, a contagem do temporizador ainda não foi habilitada. Essa ação será realizada 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 |= ADC_SQR1_SQ1_2;  // Selecionar canal 4 na sequência
regular
	// Configurar o tempo de amostragem do canal 4
	ADC1->SMPR1 &= ~ADC_SMPR1_SMP4;    // Limpar configurações anteriores
	ADC1->SMPR1 |= ADC_SMPR1_SMP4_2;  // Amostragem de 32.5 ciclos de ADC
	ADC1->PCSEL |= ADC_PCSEL_PCSEL_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 |= (ADC_CFGR_EXTSEL_3 |
		ADC_CFGR_EXTSEL_2 |
		ADC_CFGR_EXTSEL_0); // 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" para o ADC1 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. Com o TIM6 configurado para gerar o sinal de atualização para outros periféricos e o ADC1 habilitado para utilizar o sinal do TIM6 como gatilho para conversões por *hardware*, o circuito de disparo das conversões do ADC1 está completo. Assim, a cada evento de atualização do TIM6, uma nova conversão será iniciada no ADC1.

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 ADC1 a responder ao *trigger*.

Por fim, note que o pino PC5 foi configurado como saída GPIO tipo "*open drain*" em nível baixo. Isto foi feito para que este pino, ao lado do pino 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. Preferivelmente, devemos 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, ativando e desativando a contagem do *timer*. Se o *timer* estiver parado, o evento de atualização não ocorre e, consequentemente, o ADC configurado para iniciar as conversões por *hardware* não é disparado, mesmo estando habilitado e iniciado.

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;
```

E incluir no escopo de `/* USER CODE BEGIN PFP */` os protótipos

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

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 a função:

```
// 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
		if (!complete) {
			data = ADC1->DR; // Le o dado convertido
			complete = 1; // Flag interno de dado disponivel
		}
	}
}
```

E no escopo de `/* USER CODE BEGIN 0 */`, adicione as funções:

```
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 e controlar a sua sobrescrita. Neste projeto, implementou-se a estratégia de sobrescrever `data` somente após sua leitura; caso contrário, o valor convertido é descartado. 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();
```

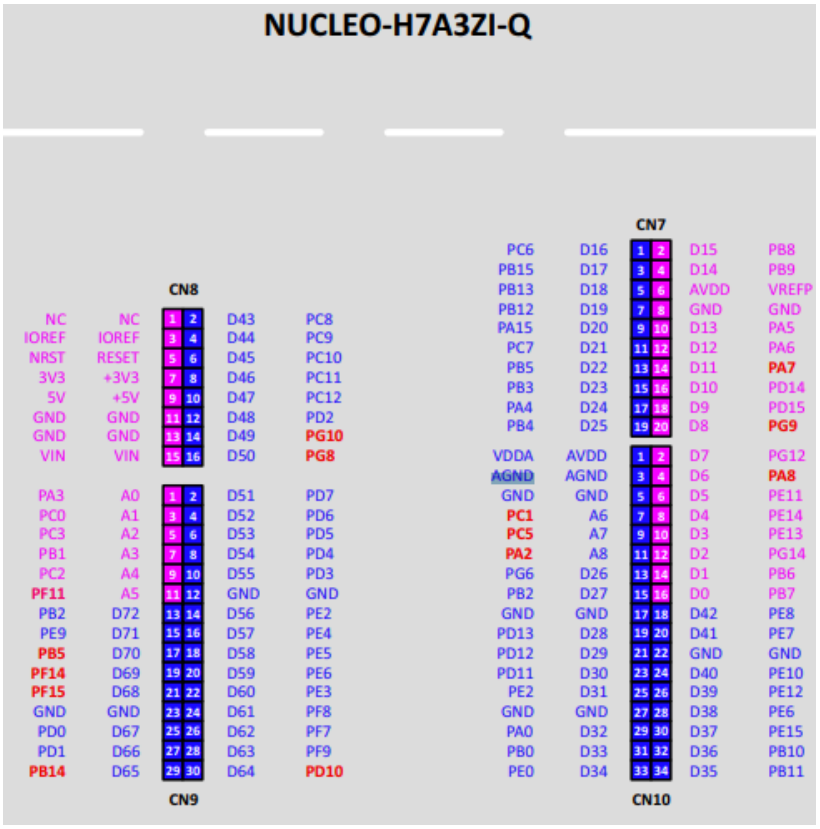
9. Agora implemente o código que vai, a cada ciclo do *loop* principal, adquirir 50 amostras pelo ADC e guardá-las no vetor, calcular o valor médio do vetor, a tensão correspondente e a temperatura equivalente, e depois aguardar 1 segundo antes de reiniciar. Abaixo da linha `/* USER CODE BEGIN 3 */`,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



Transfira o código executável para o microcontrolador e inicie 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 ("`media`") e de temperatura média ("`temp`"). Aqueça o sensor na mão e retome a execução. Veja novamente o conteúdo das variáveis.

11. Como os valores de "`adc`" e "`tensao`" foram calculados neste projeto? Compare o procedimento adotado aqui com o método aplicado no segundo projeto. Observe que a fórmula utilizada para obter "`temp`" neste projeto foi derivada do *datasheet* do sensor de temperatura LM61. Qual é a finalidade desta segunda fórmula?

12. Ao contrário do segundo projeto, as conversões neste projeto são disparadas por sinais provenientes do *timer* TIM6. Em comparação com o segundo projeto, onde as conversões são iniciadas por *software*, quais configurações adicionais são necessárias para associar o canal 4 (PC4) à entrada do ADC1, configurar o ADC1 a operar no modo de disparo por *hardware* e o TIM6 como fonte desses disparos de inicialização de conversões? Praticaremos neste Roteiro essas configurações.

13. Como realizar conversões periódicas, análogas a este projeto, em microcontroladores que possuem temporizadores periódicos e um *hardware* de conversão com suporte exclusivo para iniciação via *software* e eventos de interrupção de fim de conversão? Considerando essa limitação de *hardware*, quais seriam os desafios de implementação em *software* para garantir a periodicidade desejada? Que mecanismos e lógicas adicionais seriam necessários, e como eles impactariam a complexidade e o consumo de recursos do *software* em comparação com uma solução com disparo de conversão por *hardware*? No entanto, a vantagem de um *software* mais limpo, conciso e de fácil manutenção pode implicar uma maior complexidade na configuração de um *hardware* intrinsecamente mais potente, flexível, eficiente e preciso. Vamos explorar juntos essa complexidade e aprender domá-la?

Projeto de conversão DAC com DMA

No primeiro projeto-exemplo, demonstramos uma técnica de conversão periódica de amostras digitais em valores analógicos, utilizando as interrupções periódicas de um temporizador para controlar a taxa de transferência das amostras para o conversor. A cada interrupção, uma nova amostra digital era calculada para conversão. Agora, exploraremos uma abordagem direta para transferir amostras digitais pré-calculadas para o DAC através do controlador DMA (do inglês *Direct Memory Access*), com a taxa de transferência ainda controlada pelo temporizador, mas sem a necessidade de tratamento dos seus eventos de interrupção pelo núcleo do processador. Especificamente, armazenaremos todas as amostras da forma de onda senoidal num espaço contíguo da memória e configuraremos o *hardware* para transferi-las sequencialmente para o conversor DAC, dispensando a intervenção da CPU.

1. Crie um novo projeto do tipo “STM32 Project from an Existint STM32CubeMX Configuration File (.ioc)” usando o *Cube*, a partir do “File” <caminho>/DAC_basico.ioc com o “Project Name” “DAC_DMA”. Gere o código que terá a mesma configuração básica do primeiro projeto-exemplo.

2. Vamos definir o vetor que irá guardar as amostras da senóide como variável global no escopo de `/* USER CODE BEGIN PV */`, pois ele deve ser acessível a partir da função “main” e também pela função de configuração do DMA:

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

3. Vamos adicionar configuração do DMA no escopo de `/* USER CODE BEGIN PFP */`

```
void Config_Timer(void);
```

4. Dentro da função `main`, vamos montar o vetor de amostras da senóide no vetor sine_wave declarado dentro do escopo de `/* USER CODE BEGIN 1 */`:

```
uint8_t i;
int8_t step = 0;
int8_t direction = 1;  // 1 para subir, -1 para descer
uint32_t dac_value;
```

```
	//Inicializa o vetor de valores na memoria
	// a serem transferidos para DAC via DMA
	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;
		}
	}
```

Observe que foi removida a variável sine_wave declarada no primeiro projeto.

5. No escopo de `/* USER CODE BEGIN 2 */`, insira a chamada da função que configura DMA:

```
Config_DMA();
```

6. No laço infinito, removeremos os cálculos das amostras da senóide e suas transferências para DAC no escopo de `/* USER CODE BEGIN 3 */`. A versão final é:

```
if (leFlag()) {
	//Reseta flag
	resetFlag();
	// 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)
	dac_value = (step * 4095) / 50;
	// Atualizar o valor do DAC1_OUT2
	DAC1->DHR12R2 = dac_value;   //PA5
}
```

7. O que precisamos adicionar é a função de configuração do DMA no escopo de `/* USER CODE BEGIN 4 */`, 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
```

```
	// TIM7_UP está na entrada de requisição 70 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 conqifurar 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 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 |= (DMAMUX_CxCR_DMAREQ_ID_6 |
		DMAMUX_CxCR_DMAREQ_ID_2 |   //Associa a request 70=0b0100 0110
		DMAMUX_CxCR_DMAREQ_ID_1 );  //(TIM7 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 destino (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).

O registrador DMAMUX1_Channel1 é um registrador de configuração do multiplexador do canal 1 do controlador DMA1, usado para rotear dados entre a memória e o registrador de dados do canal 1 do DAC2. Foi configurada a fonte de evento número 70 (*update* do *timer* 7), correspondente ao evento de *update* do TIM7, como *trigger* do canal 1 do DMA1. Assim, a cada *update* do TIM7, o canal 1 do DMA1 transfere uma amostra do vetor `sine_wave`, localizado na memória, para o registrador de dados do canal 1 de DAC2. O modo circular está habilitado, de modo que ao atingir a centésima amostra, o *hardware* reinicia automaticamente o ciclo a partir da primeira amostra.

8. Agora vamos atualizar a função de configuração do *timer* TIM7 para habilitar a requisição do controlador DMA na ocorrência do evento de Update. Segue-se a nova versão com a inclusão da instrução de habilitação "`TIM7->DIER |= TIM_DIER_UDE;`":

```
void Config_Timer(void) {
	// Enable clock for TIM6 e TIM7
	RCC->APB1LENR |= RCC_APB1LENR_TIM6EN | RCC_APB1LENR_TIM7EN;
	// TIM7: Faz interrupcao periodica
	TIM7->EGR |= TIM_EGR_UG_Msk;
	while (TIM7->EGR & TIM_EGR_UG);
	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->DIER |= TIM_DIER_UDE;  // Habilitar requisição de DMA em evento
de atualização
	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);
}
```

9. Após as atualizações, faça um *Build* e transfira o código executável para o microcontrolador. Conecte as pontas de prova do **osciloscópio** nos pinos 1 e 2 do conector H8 (DAC) e ligue o terra do **osciloscópio** no pino 3 do mesmo conector, como no primeiro projeto para visualizar as duas formas de onda geradas.

10. Reavalie os testes de validação executados no primeiro projeto e compare os resultados com os obtidos nesta nova implementação. Note que a primeira implementação dependia da intervenção da CPU para transferir cada amostra individualmente, enquanto nesta, a CPU apenas inicia a contagem do temporizador TIM7, delegando a transferência cíclica das amostras ao *hardware* (controlador DMA1).

Em termos de periféricos, realize uma análise comparativa dos recursos utilizados em ambas as abordagens. No que concerne ao código, investigue as diferenças nos fluxos de controle, nas configurações dos periféricos e na proporção de responsabilidade entre *software* e *hardware* na geração das respectivas formas de onda. Caso não tenha uma resposta clara no momento, não se preocupe, retomaremos este ponto posteriormente.

11. A partir dessas comparações, você consegue discernir o papel fundamental do controlador DMA e do multiplexador DMAMUX? Ao transferir a responsabilidade da transferência de dados para esses componentes de *hardware* especializados, o benefício obtido em relação a uma implementação puramente via *software* supera a complexidade adicional na configuração desses *hardwares*? Vamos aprofundar nossa compreensão desses elementos de *hardware* para otimizar seu uso?

12. Em um sistema onde o *timer* TIM7 dispara requisições DMA a uma frequência excessivamente alta, excedendo a capacidade de processamento individual de cada requisição, o que leva a eventos de subcarga (em inglês, *underrun*), como o DMA lidará com essa situação? As requisições subsequentes serão descartadas ou existe algum mecanismo de tratamento para essa condição de alta frequência? Abordaremos essa questão adiante.

13. Em uma transferência DMA configurada em modo circular para leitura contínua de um *buffer* na memória, o DMA retorna automaticamente ao início ao alcançar o final do *buffer*. Considerando que o vetor `sine_wave` é atualizado sequencial e dinamicamente, se o processamento dos dados pela CPU for mais rápido que a taxa de leitura do DMA, o que acontecerá com os dados recém-atualizados? Eles serão enfileirados em outro *buffer* para aguardar processamento? Como podemos garantir a integridade dos dados nesse cenário? A resposta será explorada adiante.

Projeto de conversor A/D disparado por *hardware*, 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 terceiro projeto, 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. 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. Por ser uma função definida 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);
void Reset_FrameComplete (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->EGR |= TIM_EGR_UG_Msk;	// atualizacao inicial dos
registradores
	while (TIM6->EGR & TIM_EGR_UG);
	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 similar à do terceiro projeto, 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 |= (ADC_SQR1_SQ1_2 |  // Selecionar canal 4 na sequência
regular (PC4)
                   ADC_SQR1_SQ2_3);  // Selecionar canal 8 na sequencia
regular (PC5)
    // Configurar o tempo de amostragem dos canais
    ADC1->SMPR1 &= ~(ADC_SMPR1_SMP4 | ADC_SMPR1_SMP8);     // Limpar
configurações anteriores
    ADC1->SMPR1 |= (ADC_SMPR1_SMP4_0 |
                   ADC_SMPR1_SMP8_0); // Amostragem de 2.5 ciclos de ADC
    // Pre-selecionar os canais
    ADC1->PCSEL |= (ADC_PCSEL_PCSEL_4 |
                   ADC_PCSEL_PCSEL_8); // Pre-seleciona canais 4, 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 |= (ADC_CFGR_EXTSEL_3 |
                   ADC_CFGR_EXTSEL_2 |
                   ADC_CFGR_EXTSEL_0); // EXTSEL = 01101 para TIM6 Update
Event
    ADC1->CFGR |= ADC_CFGR_EXTEN_0; // Habilitar trigger em borda de
subida (EXTEN = 01)
    //Habilitar DMA modo circular
    ADC1->CFGR |= ADC_CFGR_DMNGT; //DMA (modo 0b11)
    // 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 análoga à do terceiro projeto, 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 definem o comprimento da sequência de conversão, correspondendo ao número de conversões menos 1. Em projetos anteriores, esses *bits* eram configurados como 0000, indicando uma única conversão. Neste caso, são ajustados para 0001, o que equivale a duas conversões sequenciais. A seleção dos canais que compõem a sequência é feita por meio dos *bits* `ADC_SQRx_SQy` nos registradores `ADC1_SQR1` a `ADC1_SQR4` (com x = 1 a 4), onde cada campo `SQy` define o canal a ser convertido na posição y da sequência. Neste projeto, configuramos os campos `ADC_SQR1_SQ1` e `ADC_SQR1_SQ2` para selecionar os canais 4 e 8, respectivamente. Além disso, é necessário habilitar previamente todos os canais utilizados no registrador `ADC1_PCSEL`, que controla a pré-seleção dos canais permitidos para conversão. Por fim, para otimizar a transferência de dados, configure o campo `DMNGT` no registrador `ADC1_CFGR` para habilitar a integração com o controlador DMA. Essa configuração permite que o ADC1 gere automaticamente uma requisição de DMA ao final de cada conversão ou sequência, dependendo do modo operacional. Neste cenário específico, o modo DMA com *buffer* circular está ativado.

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 controlador 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.

É importante observar a diferença entre o DMA e o NVIC. Enquanto o **DMA opera de forma totalmente independente do núcleo** durante a maior parte do processo de

transferência de dados, movendo blocos de informação entre periféricos e memória sem intervenção da CPU, o **NVIC é o responsável por gerenciar as interrupções que *são* processadas pelo núcleo**.

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_MINC | // Incrementa memoria
			DMA_SxCR_CIRC | // Buffer circular
			DMA_SxCR_TCIE; // Interrupcao de transfer complete
	// Do periferico para a memoria (DIR = 00)
	DMA1_Stream1->CR &= ~DMA_SxCR_MSIZE_Msk;
	DMA1_Stream1->CR |= DMA_SxCR_MSIZE_0; // Elemento da memoria de 16
bits
	DMA1_Stream1->CR &= ~DMA_SxCR_PSIZE_Msk;
	DMA1_Stream1->CR |= DMA_SxCR_PSIZE_0; // Elemento da periferico de 16
bits
	//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 = (DMAMUX_CxCR_DMAREQ_ID_0 |
			DMAMUX_CxCR_DMAREQ_ID_3); // Associa a request 9 ao canal
1 do DMAMUX1
	DMA1_Stream1->CR |= DMA_SxCR_EN; // Habilita Canal 1 do DMA1
}
```

A configuração do DMA nesta função apresenta semelhanças com o projeto anterior, embora com algumas distinções importantes. Primeiramente, os endereços de memória do vetor (`adc`) e do registrador de dados do periférico (`ADC1->DR`) são distintos, assim como a fonte de requisição. O número de amostras a serem transferidas coincide com o projeto anterior. Adicionalmente, a configuração do registrador `DMA1_Stream1->CR` difere: os *bits* de direção da transferência (`DIR`=0b00) agora especificam uma transferência do periférico para a

memória, ao invés de memória para periférico (DIR=0b01). Além disso, o tamanho das amostras, tanto na memória quanto no periférico, foi ajustado para 16 *bits*, substituindo a configuração anterior de 32 *bits*. Por fim, o multiplexador (MUX) do DMA é configurado para associar o canal 1 à fonte de requisição número 9, correspondente ao ADC1 (conforme a Tabela 101 do Manual de Referência), para transferir dados do ADC para a memória. Em contraste com o projeto anterior, onde o *timer* TIM7 requisitava um canal DMA para transferir dados da memória para o DAC.

Cabe ressaltar que quando um periférico, como o DMA, precisa sinalizar um evento para o núcleo (por exemplo, a conclusão de uma transferência de dados), ele gera um pedido de interrupção. Esse pedido é então gerenciado pelo NVIC, que determina a prioridade e direciona o núcleo para executar a rotina de tratamento de interrupção (ISR) correspondente. Portanto, embora o DMA realize a transferência de dados de forma autônoma, a *notificação* e o *tratamento* de eventos relacionados ao DMA (como a conclusão da transferência) envolvem a ativação do NVIC e a execução de código pelo núcleo. Neste projeto, a interrupção de transferência completa do DMA é habilitada com prioridade 1 no NVIC. Essa colaboração permite que o sistema reaja de forma eficiente aos eventos gerados pelos periféricos, mesmo com a transferência de dados ocorrendo em paralelo.

6. As funções “Start_Conv” e “Stop_Conv” são idênticas às do terceiro projeto, podendo ser copiadas. Elas são responsáveis por iniciar periodicamente uma conversão no ADC1 com os seus eventos de *Update*.

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;
```

E no escopo de `/* USER CODE BEGIN PFP */`, os protótipos das funções

```
uint8_t Frame_Complete(void);
void Reset_FrameComplete (void);
```

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;
	}
}
```

E no escopo de `/* USER CODE BEGIN 0 */`

```
uint8_t Frame_Complete(void) {
```

```
	uint8_t c;
	c = complete;
	return c;
}
void Reset_FrameComplete (void) {
	complete = 0;
}
```

A ISR opera de maneira similar ao terceiro projeto. A principal diferença reside na atualização da *flag* `complete`, que só assume o valor 1 após a transferência de todas as amostras (100 no total, sendo 50 por canal) para a área de memória `adc`. Essa área é sobrescrita em blocos nas transferências subsequentes. Ao contrário do terceiro projeto, que descartava novas conversões se `data` não tivesse sido lido, neste projeto a sobrescrita é incondicional após um atraso de aproximadamente 1000ms. Esse atraso ocorre quando o *timer* TIM6 é reativado e a *flag* `complete` é resetada para 0.

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();
Reset_FrameComplete();     //Resetar o buffer
while(!Frame_Complete()) {} //Aguardar o preenchimento
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 "`Live 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 ADC e transferência; veja como as conversões dos diferentes canais são organizadas no vetor.

13. No terceiro projeto, embora o início da conversão fosse por *hardware*, a CPU ainda participava ativamente no armazenamento das amostras dos sinais a cada amostragem. Neste projeto, a atuação da CPU se limita a habilitar o *timer* TIM6, configurado para direcionar seu evento de *Update* a outros periféricos, enquanto o controlador DMA1 e o multiplexador DMAMUX1 assumem a responsabilidade pela transferência dos dados, conforme previamente configurado. Com a delegação completa das transferências ao *hardware*, o papel do desenvolvedor se resume à configuração desses periféricos. Embora represente uma abordagem distinta daquela com a qual estamos familiarizados, o tempo dedicado ao aprendizado dessa configuração tende a ser menor do que o necessário para emular o comportamento do *hardware*. Que tal dedicarmos um tempo para aprofundar nosso conhecimento nesse novo paradigma de programar microcontroladores?

14. No modo de transferência circular DMA, os dados são gravados continuamente em um *buffer*. Ao alcançar o final do *buffer*, o DMAC reinicia a gravação a partir do início. Se a CPU processar os dados a uma velocidade inferior à da escrita do DMAC, os dados não lidos serão sobrescritos pelos novos dados, resultando em eventos de sobrecarga (em inglês, *overrun*)? Quais mecanismos podem ser implementados para garantir a integridade dos dados nesse cenário? Exploraremos essas questões em detalhes adiante.

15. Agora que vocês entenderam como captar os deslocamentos de um *joystick*, 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?

FUNDAMENTOS TEÓRICOS

Explororaremos a integração entre os **domínios analógico e digital**. Inicialmente, revisitaremos os conceitos fundamentais de **quantidades analógicas e digitais**, bem como a natureza distinta dos domínios em que operam e os desafios na sua **integração**. Em seguida, são mostradas as diversas **arquiteturas de conversores ADC e DAC**, ilustrando as diferentes técnicas empregadas para realizar essa conversão essencial.

Hoje, há muitos ADCs e DACs prontos para uso, feitos por especialistas. O projetista de sistemas embarcados não precisa criar um do zero, mas sim escolher o melhor disponível. Essa seleção exige um entendimento claro de **especificações e métricas de desempenho** dos

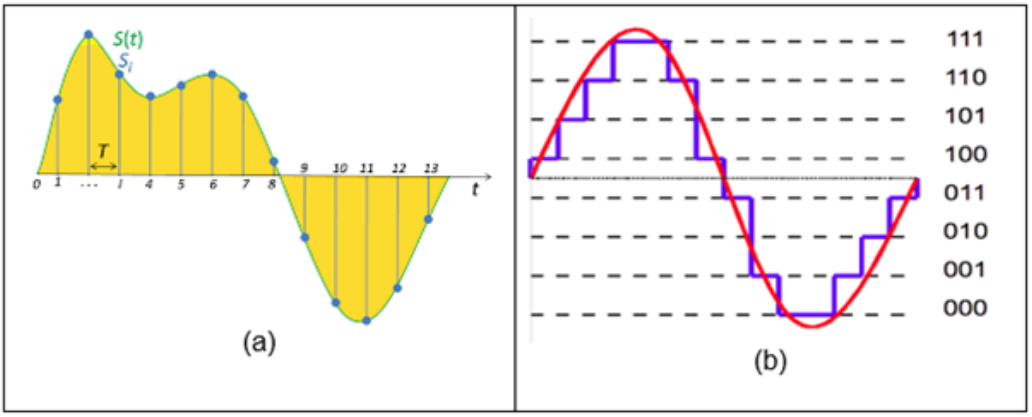
conversores. Uma seção será dedicada para detalhar os parâmetros mais relevantes presentes nas folhas de dados dos fabricantes. Para facilitar o desenvolvimento de sistemas embarcados que interagem com o mundo analógico, os microcontroladores modernos incorporam, junto aos conversores, uma sofisticada **arquitetura de interface analógica**. Veremos os componentes e as estratégias utilizadas para simplificar a conexão e o controle de **transdutors** (sensores e atuadores).

Finalmente, para lidar com a demanda por aquisição e geração de dados em tempo real de forma eficiente, discutiremos o papel do **Acesso Direto à Memória** (em inglês, *Direct Memory Access* – DMA) e dos **buffers circulares**. Mostraremos como esses mecanismos otimizam a transferência de dados entre os conversores e a memória do sistema, liberando o processador para outras tarefas críticas e garantindo um fluxo de informações contínuo e sem perdas.

QUANTIDADE ANALÓGICA E QUANTIDADE DIGITAL

No domínio digital, a manipulação de sinais e dados alcança elevada eficiência, permitindo o processamento, armazenamento e transmissão de informações sem as restrições impostas pela natureza contínua e inerentemente variável dos sinais analógicos. Contudo, o ambiente físico que nos cerca é predominantemente analógico. Para que sistemas digitais possam interagir com o mundo real, como em sistemas de áudio, vídeo ou na leitura de sensores de temperatura, a conversão entre os domínios analógico e digital torna-se imprescindível. Surge então a questão fundamental: como representar sinais analógicos através de sinais digitais?

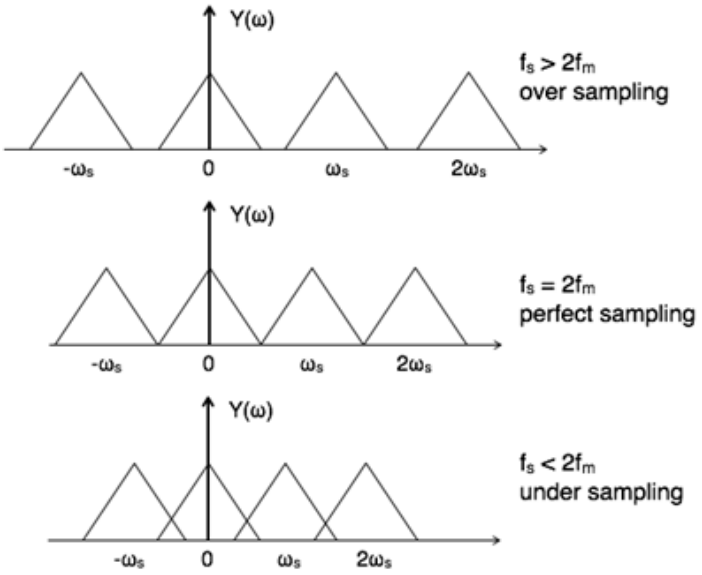
Um sinal analógico distingue-se por sua variação contínua no tempo e pela capacidade de assumir uma infinidade de valores dentro de sua faixa de amplitude, pertencendo, teoricamente, a qualquer ponto da reta real. A transição dessa infinidade de valores para uma quantidade finita, manipulável digitalmente, inicia-se com o processo de **amostragem** do sinal (Figura a). Nesta etapa, seleciona-se uma sequência discreta de valores do sinal analógico em intervalos de tempo regulares, buscando representar o sinal original de maneira significativa. No entanto, mesmo com a amostragem, os valores obtidos ainda pertencem ao domínio contínuo. Para convertê-los em códigos binários, que são intrinsecamente discretos, é necessário o processo de **quantização** (Figura b). A quantização mapeia esses valores contínuos amostrados para um conjunto finito de níveis discretos, que podem então ser codificados digitalmente. O processo de **codificação** (**digital**) atribui um código binário único a cada um desses níveis discretos. Por exemplo, se nosso quantizador definiu 8 níveis distintos, podemos representá-los utilizando códigos binários de 3 dígitos (*bits*), que nos permitem ter $2^3=8$ combinações possíveis. Assim, o nível de menor amplitude poderia ser codificado como 0b000, o próximo como 0b001, e assim sucessivamente até o nível de maior amplitude, que seria representado por 0b111. Essa representação binária é o formato fundamental para o processamento, armazenamento e transmissão do sinal em sistemas digitais.



(Fonte: (a) Wikipedia e (b) DifferenceBetween.)

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.

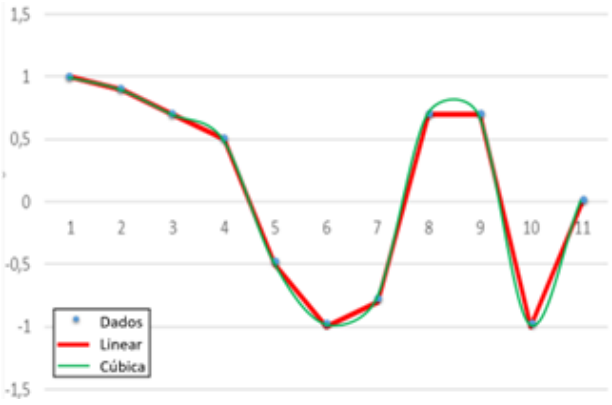
Na figura a seguir, é apresentado o modelo de amostragem no domínio de frequência. No gráfico superior, observa-se uma **superamostragem** (*oversampling*), indicando uma frequência superior a $2f_m$. No gráfico do meio, encontra-se uma **amostragem ideal** com frequência exatamente igual a $2f_m$. Já 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.

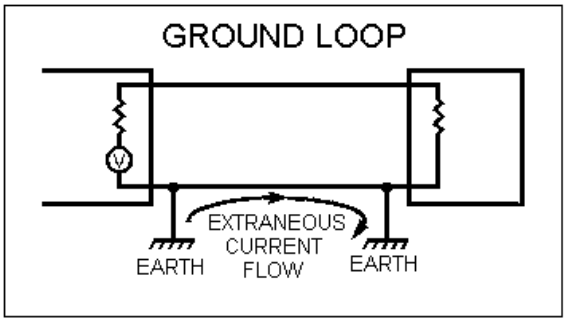


COEXISTÊNCIA DE DOMÍNIOS ANALÓGICOS E DIGITAIS

A integração dos mundos analógico e digital em um mesmo circuito impõe desafios significativos, especialmente em relação ao aterramento e à mitigação de ruídos. Em sistemas embarcados, onde é necessário combinar sinais de ambos os domínios, o aterramento desempenha um papel crucial, servindo como referência comum (potencial elétrico de 0V) para todas as medições de tensão no circuito.

Os sinais analógicos e digitais apresentam características distintas, especialmente quando analisados por um analisador de espectro. Os sinais digitais, típicos em sistemas embarcados, operam em frequências extremamente altas, variando da ordem de megahercios (MHz) a gigahercios (GHz). Durante as transições entre os estados binários (0 e 1), os sinais digitais geram picos de corrente significativos, na faixa de 10 mA a 100 mA, o que pode induzir ruídos nos sinais analógicos próximos, afetando sua qualidade. Por outro lado, os sinais analógicos são mais sensíveis a variações de tensão e interferências, o que torna ainda mais crítica a separação e o controle adequado de seus **aterramentos**.

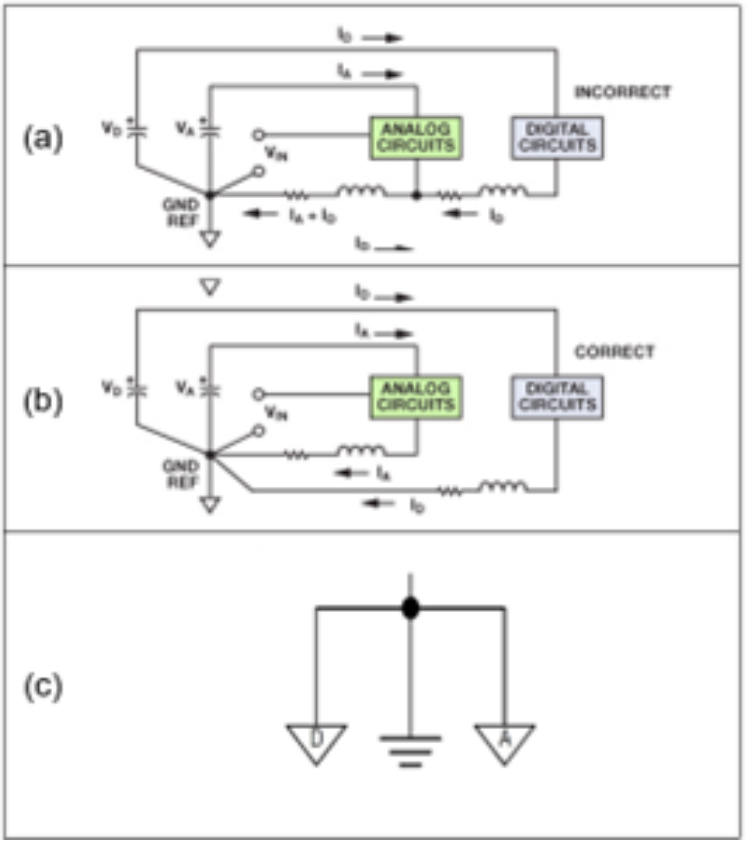
Para resolver essa questão, uma abordagem comum é manter **aterramentos separados** para as partes analógicas (em inglês, *analog grounding* – AGND) e digitais (em inglês, *digital grounding* — DGND, ou simplesmente GND). Isso ajuda a evitar que os sinais digitais, com suas rápidas transições e picos de corrente, interfiram nos sinais analógicos, mais sensíveis a variações de tensão. No entanto, a separação dos aterramentos pode causar problemas, especialmente quando há diferenças de potencial entre os pontos de aterramento. Quando os aterramentos analógico e digital são conectados de forma inadequada, isso pode criar laços de terra (em inglês, *ground loops*). Esses **laços de terra** ocorrem quando existem múltiplos caminhos para a corrente de retorno, que podem ter diferentes potenciais de terra. Isso permite que a corrente digital flua pelo caminho de retorno do circuito analógico, gerando ruídos e interferências que afetam a qualidade do sinal analógico. Esse fenômeno é especialmente prejudicial em sistemas de alta precisão, onde os sinais analógicos são muito sensíveis.



(Fonte: WICI).

Uma possível solução para minimizar o impacto desses problemas é utilizar **filtros entre os aterramentos analógico e digital**, de modo a permitir que ambos compartilhem um ponto de terra comum como ilustra a seguinte Figura (a). Essa abordagem pode reduzir a formação de

laços de terra, pois o filtro ajuda a isolar as correntes de alta frequência dos sinais digitais, impedindo que elas afetem os sinais analógicos. No entanto, mesmo com essa estratégia, o *design* do aterramento deve ser feito com cuidado para evitar a introdução de novos caminhos de interferência.



(Fonte: AnalogDevice).

A prática mais comum e recomendada é garantir que os **aterramentos analógicos e digitais se conectem em um único ponto de referência**, como mostra a Figura (b), o que evita a circulação de correntes parasitas e a formação de laços de terra. Esse ponto de conexão deve ser cuidadosamente escolhido no projeto, levando em conta a distribuição de corrente no sistema. Dessa forma, todos os sinais e circuitos analógicos devem se referir a AGND, enquanto os sinais e circuitos digitais devem se referir a DGND. A conexão desses dois pontos deve ser feita de maneira estratégica, geralmente em um único local, para garantir a integridade dos sinais e minimizar a interferência.

Em sistemas bem projetados, o uso de filtros entre AGND e DGND pode ser eficaz, mas a principal solução é garantir que ambos os aterramentos se conectem a um ponto único de referência, bem planejado, evitando **problemas como os laços de terra e a contaminação de sinais**.

Além dos cuidados com o aterramento, a proximidade inadequada entre os planos analógico e digital em um sistema eletrônico pode resultar em *crosstalk*. O ***crosstalk*** é uma interferência

indesejada entre os sinais, que pode causar contaminação dos sinais analógicos com ruídos digitais, ou vice-versa. Esse fenômeno é especialmente problemático em sistemas críticos, onde até pequenas variações nos sinais podem comprometer o funcionamento do sistema. Em sistemas de alta precisão, como em instrumentos de medição, processamento de áudio ou vídeo, esse tipo de interferência pode resultar em falhas de funcionamento, distorções nos dados ou perda de desempenho.

Para mitigar esse risco, a gestão cuidadosa do *layout* da placa de circuito impresso (PCB) é essencial. Uma das principais práticas para reduzir o *crosstalk* e melhorar a compatibilidade eletromagnética (em inglês, *eletromagnetic compatibility* – EMC) é **separar fisicamente os planos de aterramento dos domínios analógico e digital**, criando duas zonas dedicadas: uma zona de silêncio analógica e uma zona de ruído digital. A **zona de silêncio analógica** é a área da PCB onde os componentes e as trilhas de sinais analógicos sensíveis são cuidadosamente posicionados para minimizar a exposição a fontes de ruído. Por outro lado, a **zona de ruído digital** concentra os componentes digitais, que são inerentemente fontes de ruído de alta frequência. Ao adotar essa segregação espacial, o objetivo é confinar as emissões de ruído digital dentro de sua própria zona e proteger a integridade dos sinais analógicos na zona de silêncio.

Em muitas placas de circuito impresso, essa segregação é feita alocando camadas de cobre dedicadas para cada domínio, com a camada de aterramento analógico (AGND) separada da digital (DGND). Essa estratégia reduz a resistência e a indutância nos caminhos de retorno, proporcionando um desacoplamento mais eficaz e minimizando o impacto das correntes de alta frequência associadas aos sinais digitais. Além disso, outra técnica eficaz é a **blindagem** dos componentes digitais, que ajuda a isolar os sinais digitais de alta frequência e impedir que eles se espalhem para os circuitos analógicos. Isso pode ser feito utilizando blindagem metálica ou ilhas de cobre nas camadas da PCB, criando uma barreira física que impede a propagação do campo eletromagnético. A utilização de vias de aterramento, distribuídas ao longo da PCB, também contribui para a dissipação do ruído e a diminuição da interferência entre os sinais.

A **separação de vias e trilhas** também é uma prática recomendada para garantir que os sinais digitais e analógicos percorram caminhos distintos na PCB. Manter os sinais digitais longe das trilhas de sinais analógicos ajuda a reduzir o risco de indução de ruídos. Quando necessário, as trilhas digitais podem ser colocadas em camadas internas da PCB, enquanto as trilhas analógicas podem ser mantidas nas camadas externas, mais protegidas de possíveis fontes de interferência. O uso de **materiais de alta qualidade** para a construção da PCB, como placas de cobre com baixa resistência e camadas de isolantes com baixa constante dielétrica, pode também contribuir para a redução do acoplamento capacitivo entre os planos e sinais.

Além do aterramento e do acoplamento eletromagnético entre os domínios analógicos e digitais, o **desacoplamento eficaz** das fontes de alimentação pode reduzir a propagação de ruídos. Capacitores de desacoplamento devem ser estrategicamente colocados perto dos pinos de alimentação de circuitos digitais, filtrando as flutuações de tensão de alta frequência. O uso

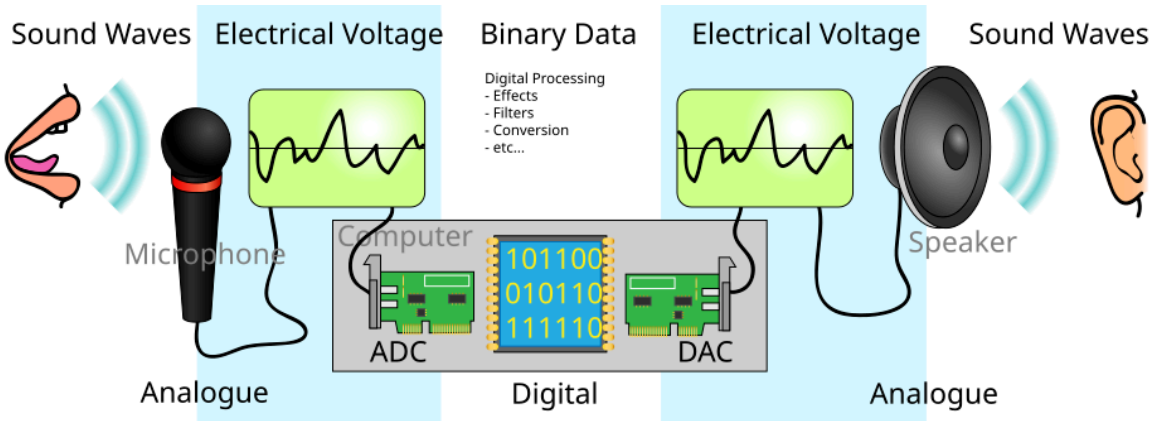
de capacitores de diferentes valores (geralmente de 0,1 µF a 10 µF) garante que as variações de frequências diferentes sejam adequadamente filtradas.

Em sistemas onde a conformidade com as regulamentações EMC é essencial, garantir que o *design* minimize os caminhos de emissão de ruídos eletromagnéticos e a susceptibilidade do sistema a interferências externas é fundamental. Para isso, o uso de **gabinetes blindados** e **filtros de linha** também pode ser necessário, especialmente em ambientes industriais ou em dispositivos que interagem com outras tecnologias de comunicação de alta frequência.

INTEGRAÇÃO DE DOMÍNIOS ANALÓGICOS E DIGITAIS

Após a compreensão das quantidades analógicas e digitais e da coexistência desses dois domínios em sistemas físicos, vamos avançar para a análise da **integração entre os dois mundos**. Embora circuitos analógicos e digitais sejam frequentemente abordados como áreas separadas e com características próprias, a verdadeira inovação e eficiência surgem quando esses domínios se integram. Em muitas aplicações práticas, especialmente nas modernas tecnologias de sistemas embarcados, o tratamento e o processamento de sinais analógicos e digitais precisam ser realizados de maneira intercambiável. Isso significa que, enquanto o mundo digital pode oferecer recursos poderosos de processamento, controle e precisão, os sinais analógicos interagem com o mundo físico, como em sensores, atuadores e sistemas de áudio.

Por exemplo, sensores de temperatura, pressão ou umidade geram sinais analógicos, mas essas informações precisam ser processadas digitalmente para serem manipuladas em sistemas de controle, como microcontroladores ou sistemas de controle de *feedback*. Além disso, um sistema de áudio digital pode gerar som com qualidade superior, mas, no final do processo, o sinal precisa ser convertido de volta em analógico para ser reproduzido em alto-falantes.



(Fonte: Wikimedia Commons).

Assim, a **integração** entre sinais analógicos e digitais não é apenas desejável, mas uma necessidade para o funcionamento adequado de muitos sistemas modernos. Essa integração não só possibilita a troca de informações entre os dois, mas também maximiza as vantagens de cada domínio, criando sistemas híbridos que combinam o melhor de ambos. O grande desafio está em fazer com que esses dois domínios se comuniquem de maneira eficiente, sem comprometer a qualidade do sinal e a integridade dos dados processados. A verdadeira inovação nos sistemas modernos não está apenas na coexistência dos domínios analógico e digital, mas nessa integração.

No cerne dessa integração estão os **conversores ADC** (do inglês *Analog-to-Digital Converter*) e **DAC** (do inglês *Digital-to-Analog Converter*). Esses componentes atuam como ponte entre os domínios analógico e digital, permitindo que os sinais se movam entre os dois mundos e que aproveitemos o melhor de ambos os mundos, com alta qualidade, baixo custo e grande eficiência. Os ADCs são responsáveis por converter sinais analógicos para formatos digitais, enquanto os DACs fazem a conversão inversa, levando os dados digitais de volta para o domínio analógico.

Os conversores ADC e DAC demandam um posicionamento estratégico na placa de circuito impresso (PCB), fortemente influenciado pelo domínio com o qual interagem mais diretamente e por sua inerente sensibilidade a ruídos. Recomenda-se que os ADCs sejam alocados o mais próximo possível da fonte do sinal analógico a ser amostrado, assim como os DACs devem estar adjacentes ao destino do sinal analógico gerado, minimizando o comprimento das trilhas analógicas e, consequentemente, a probabilidade de captação de ruídos e perdas de sinal.

No que tange ao aterramento, o ideal é que ADCs e DACs residam na fronteira entre a zona de silêncio analógica e a zona de ruído digital, com conexões diretas e dedicadas tanto ao plano de terra analógico (AGND) quanto ao digital (DGND). O ponto de conexão entre AGND e DGND, o ponto único de referência, deve preferencialmente situar-se próximo aos pinos de terra do próprio conversor, evitando que correntes de retorno digitais transitem pelo plano de terra analógico e vice-versa. Em algumas configurações, pode ser vantajoso posicionar o componente de modo que seus pinos analógicos predominem sobre o AGND e os digitais sobre o DGND, com a junção dos terras imediatamente abaixo ou nas proximidades.

Em relação ao isolamento de ruído digital, embora a interação com o domínio digital seja necessária, ADCs, especialmente os de alta resolução, são particularmente vulneráveis a ruídos digitais, tornando imperativo manter os pinos de entrada analógica e os circuitos analógicos associados estritamente dentro da zona de silêncio analógica, minimizar o cruzamento de trilhas digitais sobre a área analógica e vice-versa, e empregar blindagem, quando necessário, para um isolamento ainda maior. Por fim, tanto ADCs quanto DACs requerem uma alimentação estável e com baixo ruído, demandando a colocação de capacitores de desacoplamento o mais próximo possível de seus pinos de alimentação, em ambos os lados (analógico e digital), para filtrar ruídos de alta frequência, sendo que sistemas

mais sensíveis podem se beneficiar de uma fonte de alimentação separada e regulada para a seção analógica.

Em cenários onde uma conexão de terra dedicada pode não estar imediatamente disponível ou em configurações experimentais simplificadas, uma alternativa extremamente cautelosa e com limitações significativas seria utilizar um pino do microcontrolador configurado como saída *open-drain* e forçado ao nível lógico baixo como um ponto de referência de terra. Isso ocorre porque, ao definir a saída em nível baixo, o transistor interno (tipicamente um MOSFET de canal N) é ativado, estabelecendo uma conexão direta do pino ao terra interno do microcontrolador, comportando-se, nesse estado, como uma via de baixa impedância para o terra.

Contudo, é fundamental estar plenamente ciente das implicações e restrições dessa prática. Primeiramente, a capacidade de corrente (em inglê, *sink current*) do pino *open-drain* é limitada. Exceder essa corrente máxima pode danificar permanentemente o microcontrolador. Em segundo lugar, a impedância dessa "conexão de terra" não é ideal e pode ser mais alta do que um terra dedicado, o que pode afetar a precisão de medições analógicas, especialmente em circuitos sensíveis. Além disso, o pino só atuará como terra enquanto estiver ativamente configurado como saída em nível baixo pelo *software*. Qualquer alteração na configuração do pino ou no seu nível lógico resultará na perda da referência de terra.

Portanto, embora essa técnica possa oferecer uma solução temporária ou em situações muito específicas de baixíssima corrente, ela não é recomendada como uma prática de projeto robusta ou para aplicações com requisitos de precisão ou corrente mais elevados. A utilização de uma linha de terra dedicada e bem planejada é sempre a abordagem preferível para garantir a integridade dos sinais analógicos e a estabilidade do sistema.

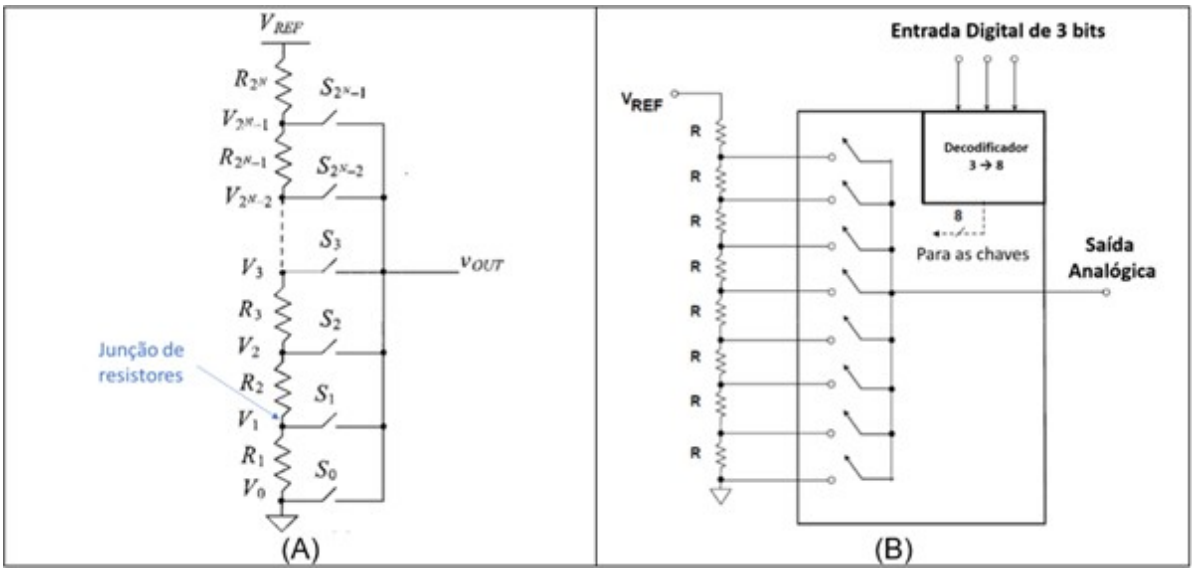
ARQUITETURAS DE CONVERSORES DIGITAL-ANALÓGICOS

Um **conversor digital-analógico** (em inglês, *Digital-to-Analog Converter* – DAC) 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_{fs}), determinada pela fonte de alimentação. A tecnologia base da maioria dos DACs é a manipulação e combinação de correntes ou tensões controladas digitalmente para gerar uma saída analógica proporcional ao código digital de entrada. A utilização de resistores para ponderar as diferentes contribuições dos *bits* digitais e somá-las para gerar a saída analógica é uma forma comum de implementar essa manipulação e combinação. 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. Outras tecnologias incluem a modulação por largura de pulso (PWM) seguida por filtragem.

Portanto, a implementação de um DAC pode ser feita com diversas arquiteturas, cada uma com características únicas para atender a requisitos específicos. Vamos apresentara **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. Os mais encontrados no mercado são os de arquitetura R-2R, como o DAC0808 e o AD7524.

Resistor *String* DAC

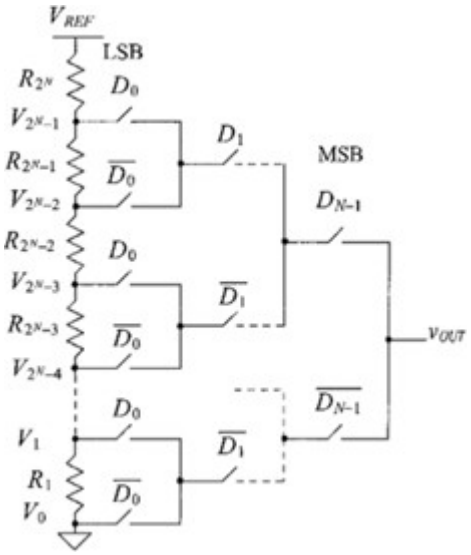
A arquitetura mais elementar de conversão Digital-Analógica é ilustrada na figura (A) a seguir e é denominada Resistor *String* DAC, ou simplesmente *String* DAC. Essa configuração consiste em uma cadeia de 2^N resistores (R_j) de igual valor e 2^N chaves idênticas (S_i), as quais conectam as diversas 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.



(Fonte:).

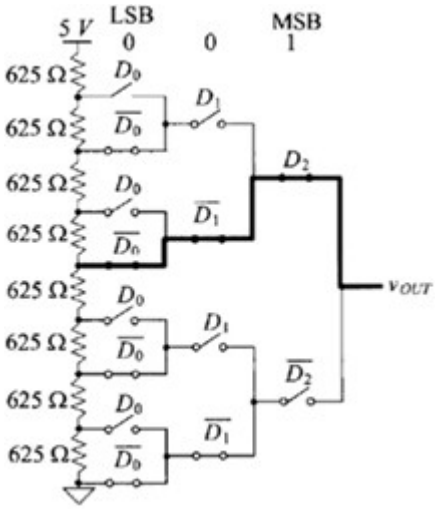
Uma grande vantagem da arquitetura de resistores em cadeia é a garantia de monotonicidade em sua saída. Contudo, um desafio associado a conversores com essa construção é a conexão permanente de 2^N-1 chaves desligadas e uma chave ligada à saída. Em casos de resoluções mais elevadas, as chaves inativas 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.



(Fonte: SlideShare).

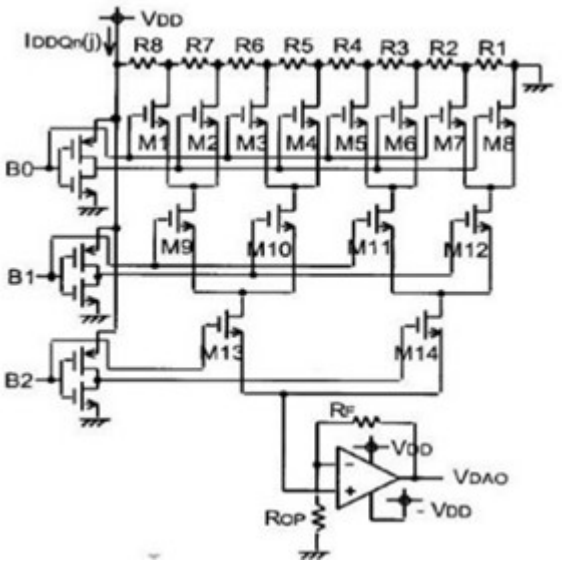
Para ilustrar o funcionamento de um *string*-DAC com chaves em árvore, considere o exemplo de um conversor de três *bits* mostrado na seguinte figura. Para um valor de entrada binária igual a 0b100, as chaves estarão ligadas como mostrado com linha grossa. Para este valor de entrada, a saída será igual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



(Fonte: SlideShare).

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

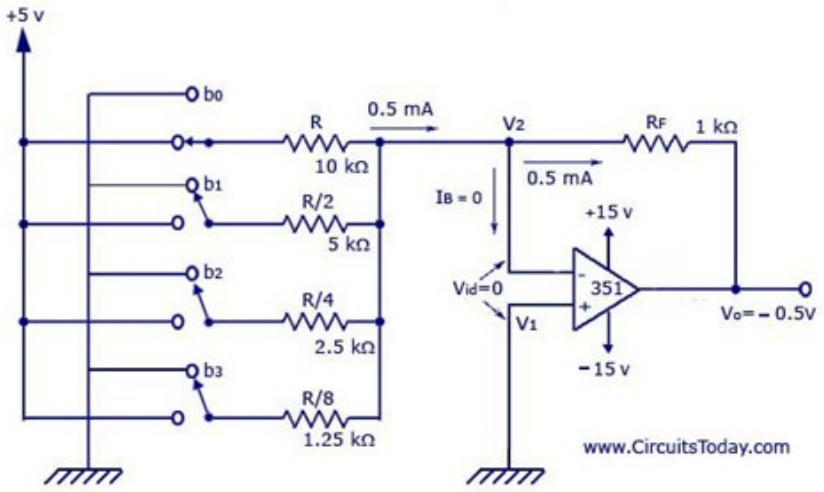
é substituir os resistores por transistores MOS, como ilustrado na figura que se segue. Essa abordagem visa superar as limitações associadas à dissipação de potência, proporcionando uma solução mais eficiente em termos de área e consumo de energia.



(Fonte: SlideServe).

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.



(Fonte: CircuitsToday).

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.

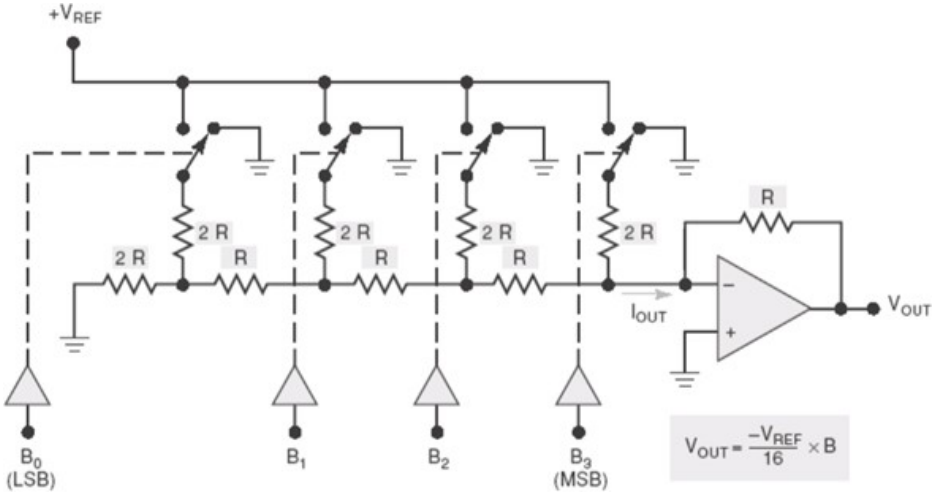
$$A_{fs} = K(2^n - 1) \rightarrow K = \frac{A_{fs}}{(2^n - 1)}$$

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. A corrente através de cada ramo do DAC é influenciada pelos valores dos resistores. Para o *bit* mais significativo (MSB), a corrente é proporcional a (Vref / 2R), para o próximo é (Vref / 4R), e assim por diante. Se os resistores nos ramos forem diferentes, essas correntes serão ligeiramente diferentes dos valores ideais e a tensão de saída real, que corresponde à soma dessas correntes multiplicada por $-R_F$, será diferente da tensão de saída ideal. Se os valores dos resistores se desviarem significativamente dos seus valores ideais, a relação entre o código digital de entrada e a tensão de saída analógica pode se tornar não linear, levando à **não monotonicidade** em certas transições de *bits*.

O principal problema do esquema paralelo usando resistores como pesos de ponderação reside no 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

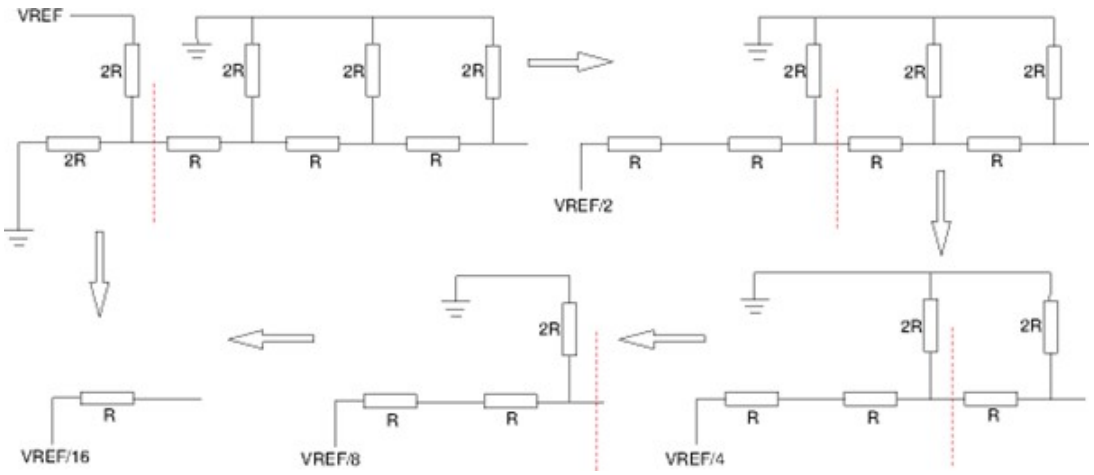
Um circuito alternativo ao circuito DAC com resistores ponderados é a **rede R/2R** que envolve somente dois valores de resistências R e 2R, como mostra a figura que se segue.



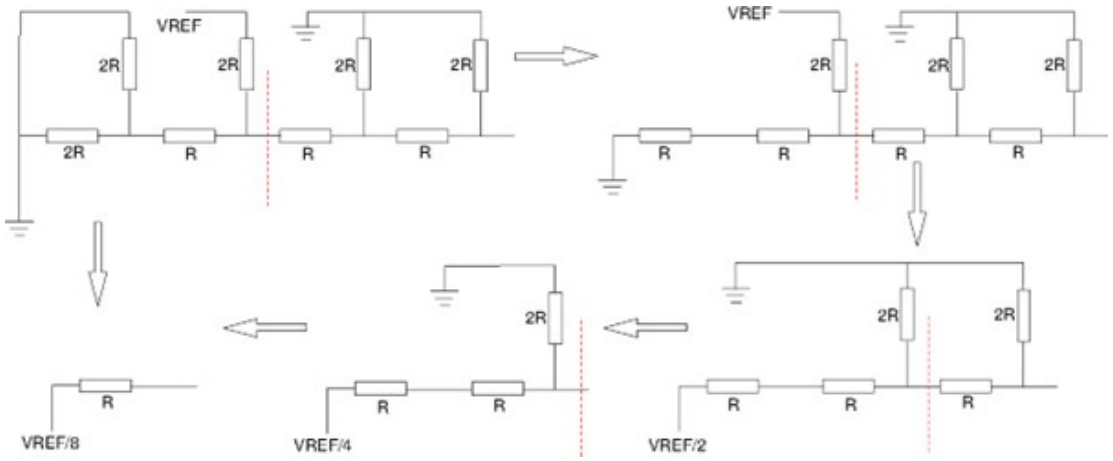
(Fonte: Tocci et al. Sistemas Digitais: Princípios e Aplicações. 10a. edição).

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.

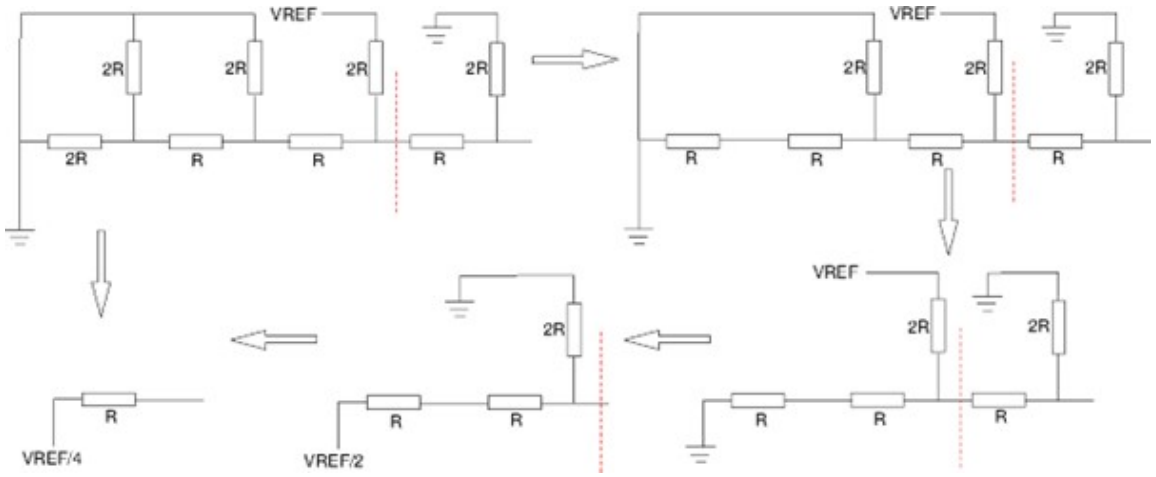
Para o *bit* b_0=1, o circuito equivalente de Thévenin é uma tensão *VREF*/16 com um resistor série equivalente *R*, conforme mostra o processo de redução esquematizado.



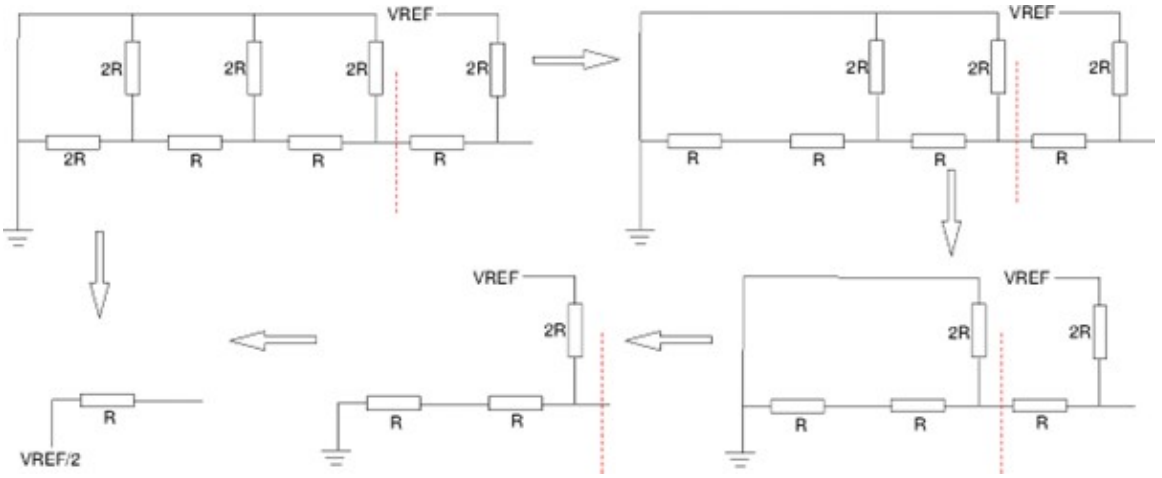
Para o *bit* b_1=1, o circuito equivalente de Thévenin é uma tensão *VREF*/8 com um resistor série equivalente *R*, conforme mostra o processo de redução a seguir.



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.

É importante notar que, embora existam 2^N níveis de quantização, indexados de 0 a 2^N-1 na representação binária, por exemplo $b_3b_2b_1b_0$ para N=4, nossa consideração inicial sugere implicitamente uma contagem de 1 a 2^N. Para refletir corretamente que o valor máximo é alcançado no último índice correspondente a 2^N-1 passos a partir do zero, subtraímos 1 do número total de níveis ao calcular o valor máximo de tensão de saída, ou seja,

$$V_{OUT} = \left(b_0 + 2b_1 + 4b_2 + 8b_3 + \ldots + 2^{(N-1)}b_{n-1}\right) \times \frac{VREF}{2^N - 1}.$$

Note que, com N=0, o conversor não consegue definir uma saída. Isso ocorre porque 0 *bits* implicam a ausência de níveis distintos, impossibilitando qualquer conversão significativa.

Assim como nos DACs com resistores ponderados, a precisão dos resistores na rede R/2R é fundamental para garantir a correta ponderação da contribuição de cada *bit* à tensão de saída analógica. Desvios significativos nos valores dos resistores podem causar não linearidade na relação entre o código digital de entrada e a tensão de saída analógica, resultando em **não monotonicidade** em certas transições de *bits*. Transições onde múltiplos *bits* mudam simultaneamente (como de 0111 para 1000) são especialmente sensíveis a problemas de não monotonicidade devido a ponderações incorretas dos *bits*.

Além dos CIs DAC0808 e o AD7524, um outro 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).

ARQUITETURAS DE CONVERSORES ANALÓGICO-DIGITAIS

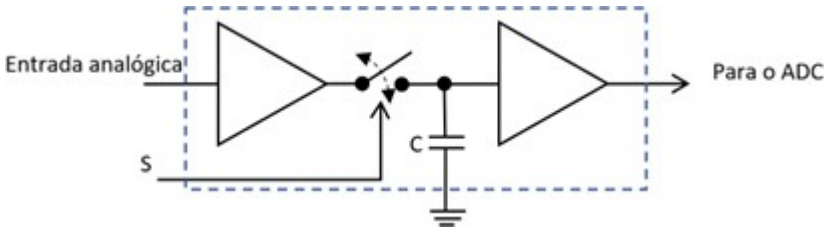
Um **conversor analógico-digital** (em inglês, *Analog-to- Digital Converter* – ADC) é um circuito projetado para transformar um sinal analógico, geralmente expresso como corrente ou

tensão, em um sinal digital representado por códigos binários. Esse processo envolve três etapas principais: amostragem, que captura instantâneos do sinal em intervalos de tempo definidos; quantização, onde esses instantâneos são convertidos em valores discretos por meio de comparações com níveis de referência; e codificação, que transforma esses valores discretos em códigos binários representáveis. A iniciação de qualquer conversão nos ADCs integrados em microcontroladores é feita por **disparos** (em inglês, *triggers*), implementados **via *software* ou *hardware***. No caso do *software*, um acesso de escrita a um *bit* de registrador geralmente dispara a conversão. Já o disparo por *hardware* demanda a seleção e a ativação de uma fonte de *trigger*, tipicamente um temporizador, previamente configurada.

Embora os conversores ADCs estejam disponíveis comercialmente, compreender as técnicas empregadas em sua implementação é fundamental para compreender os parâmetros envolvidos na especificação do fabricante. Isso facilita a seleção apropriada do ADC para uma aplicação específica. Antes de apresentarmos algumas arquiteturas de ADC e suas tecnologias fundamentais, é pertinente apresentar um módulo auxiliar crucial para seu funcionamento, o **circuito de amostragem e retenção** (S/H), bem como a **metodologia para converter o código binário** resultante de volta ao valor na grandeza física original.

Circuito 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 "congela" 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*.

Conversão de códigos binários para valores em unidades físicas

A saída de um ADC é um **código binário**, uma representação digital do valor da grandeza física amostrada no instante da conversão. Para muitas aplicações, essa representação digital é suficiente para processamento posterior, armazenamento ou transmissão. No entanto, em cenários onde é necessário realizar operações diretamente na **unidade física original** (como comparar a temperatura medida com um valor de referência em graus Celsius, ou a pressão

com um limite em Pascal), uma etapa adicional de **conversão** é necessária. Essa conversão transforma o código binário de volta para a unidade física significativa para a aplicação.

O resultado de uma conversão ADC, tipicamente armazenado em um registrador como `ADCx_DR`, é um código binário de N *bits* que representa o valor da tensão analógica amostrada. Em um conversor ADC devidamente calibrado, assume-se uma relação linear entre a faixa de tensão de fundo de escala, delimitada por uma tensão de referência inferior (VREFL) e uma tensão de referência superior (VREFH), e a faixa de valores digitais correspondente, que varia de 0 a 2^N-1. Assim, para obter o valor da tensão amostrado, *Tensão_amostrada*, a partir do código binário lido em `ADCx_DR`, podemos aplicar uma transformação linear, frequentemente implementada por meio de uma regra de três, utilizando operações em ponto flutuante:

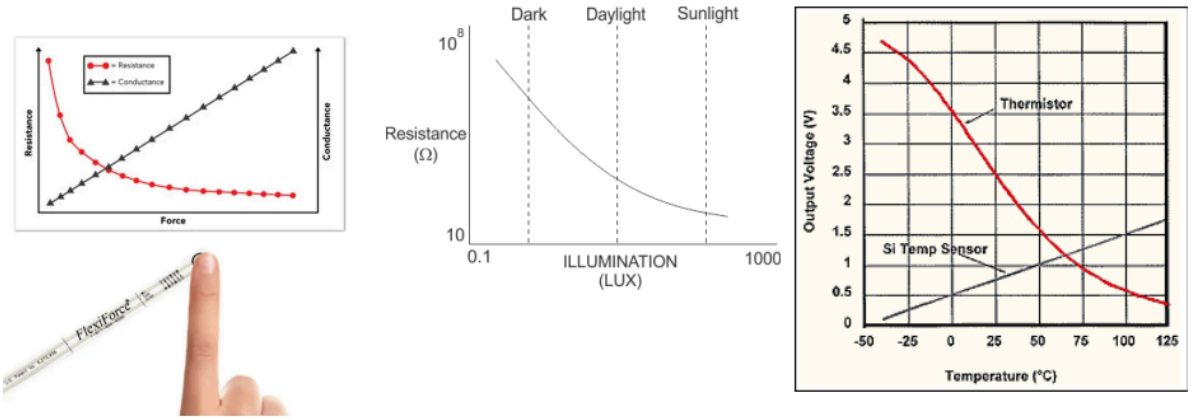
(Tensão amostrada-VREFL) → [ADCx_DR]

(VREFH – VREFL) → 2^N - 1

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

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

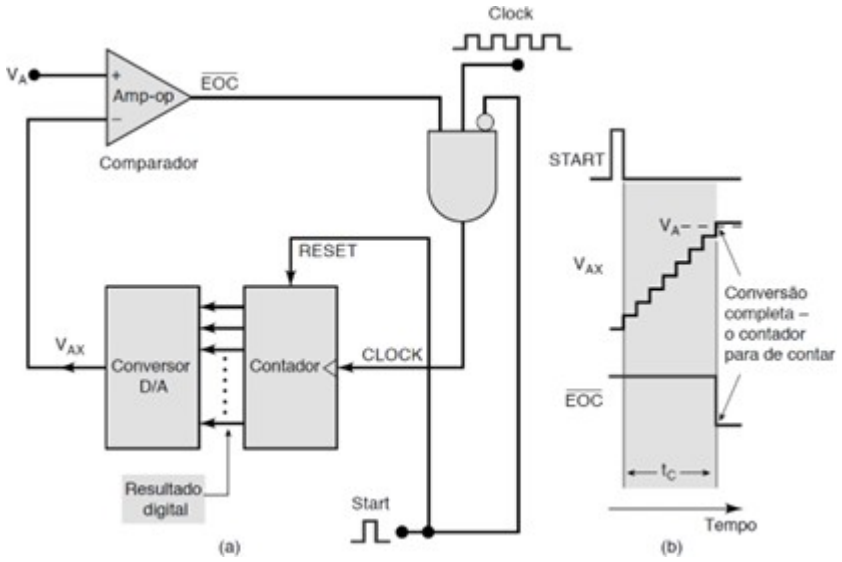
Se for necessário obter os valores originais das grandezas físicas dos sinais amostrados, é essencial realizar um pós-processamento das amostras de tensão recuperadas, *Tensão_amostrada*, convertendo-as para os valores nas grandezas físicas originais. Isso geralmente requer consulta aos *datasheets* 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).



ADC de rampa digital (contador-rampa)

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.



(Fonte: Tocci et al. Sistemas Digitais: Princípios e Aplicações. 10a. edição).

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:

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

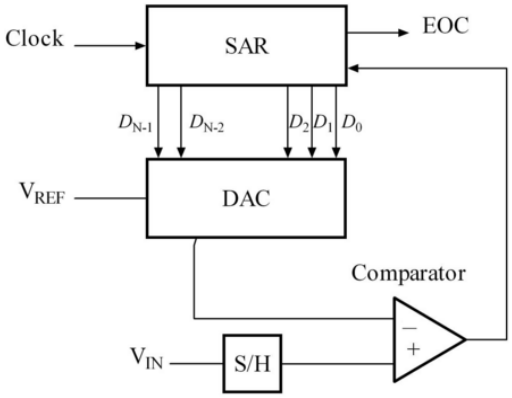
onde $T_{relógio}$ e n denotam, respectivamente, o período do relógio e a quantidade de *bits* do contador. Essa característica torna essa arquitetura uma escolha menos comum em aplicações comerciais. Além disso, o tempo de conversão varia bastante com o valor da tensão de entrada. Geralmente, o tempo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

Tipicamente, a transição para o próximo nível digital ocorre quando a tensão de entrada analógica atinge ou ultrapassa o limite superior do nível de quantização atual. Em outras palavras, um determinado código digital de saída representa uma faixa de tensão de entrada analógica. A mudança para o código digital seguinte ocorre no ponto de transição entre essas faixas.

É importante notar que, em algumas análises teóricas e na definição do erro de quantização, o valor digital é frequentemente associado ao **ponto médio** do intervalo de quantização. Nesse caso, o código 0 representaria o valor 1/2ΔV, o código 1 representaria o valor 3/2ΔV, e assim por diante. No entanto, para determinar o código digital *resultante* de uma dada tensão de entrada, a transição de um nível para o próximo ocorre no **limite do intervalo**.

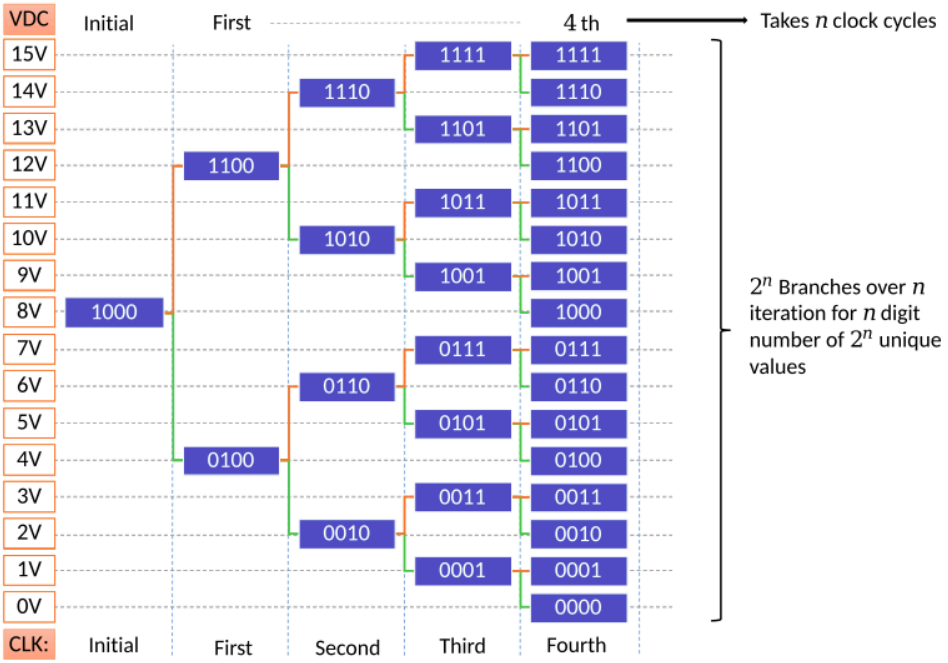
ADC com registrador de aproximações sucessivas

A arquitetura de **conversor de aproximações sucessivas** (SAR) é uma das mais 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.



(Fonte: Wikipedia).

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.



(Fonte: Wikimedia Commons).

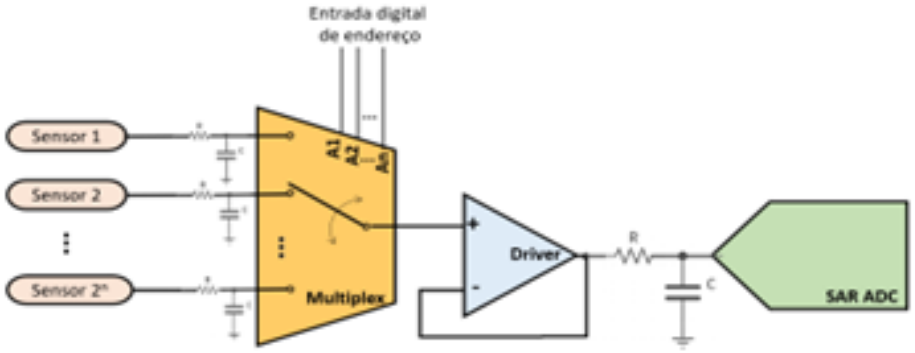
Exemplos de circuitos integrados que implementam a arquitetura SAR incluem o ADC0804 e o ADS8866, ambos da Texas Instruments.

Aprimoramento das tecnologias de ADC

A tecnologia de conversão analógico-digital (ADC) evoluiu para atender às demandas de sistemas complexos em termos de ruído, precisão e flexibilidade, mas a base clássica (amostragem, quantização e codificação) continua essencial para a compreensão das inovações práticas que surgiram. Para ilustrar essa evolução, apresentamos quatro funcionalidades presentes nos conversores ADC modernos.

ADCs tradicionais utilizam medição em modo de entrada simples (referenciada ao terra). Em contraste, ADCs modernos frequentemente empregam **entrada diferencial**, medindo a diferença de tensão entre dois sinais (positivo e negativo). Essa abordagem elimina ruídos comuns que afetam ambos os sinais igualmente, melhorando a qualidade do sinal em ambientes com interferência eletromagnética. A medição diferencial oferece, portanto, melhor desempenho em cenários ruidosos.

Tipicamente, um microcontrolador dispõe de um a dois conversores ADC e suporta vários canais de entrada analógica multiplexáveis. A figura a seguir ilustra vários canais de sinais analógicos para um mesmo conversor ADC. 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. Para este cenário, a conversão em ADCs tradicionais segue uma sequência fixa de canais com tempo e prioridade definidos.



Para sistemas complexos que exigem respostas rápidas a eventos prioritários, ADCs modernos incorporam **canais "injetados"**. Estes canais interrompem a sequência regular para realizar conversões urgentes, acionadas instantaneamente por eventos externos. Essa funcionalidade é crucial para monitorar eventos específicos ou obter leituras rápidas de forma assíncrona, conferindo maior flexibilidade e capacidade de resposta.

Enquanto ADCs tradicionais realizam uma única amostragem por conversão, a **superamostragem** (em inglês, *oversampling*) moderno efetua múltiplas amostras do mesmo sinal por conversão. Essa técnica aprimora a qualidade do sinal e minimiza erros e ruído sem aumentar a resolução do ADC. Ao amostrar várias vezes, o *oversampling* melhora a relação sinal-ruído (SNR) e aumenta a resolução efetiva do sinal, sendo útil para ADCs com resolução limitada que precisam gerar sinais digitais de alta qualidade.

A leitura contínua de sinais em ADCs tradicionais contrasta com o *watchdog* analógico presente em ADCs modernos. Esse recurso monitora continuamente o valor da conversão de uma entrada analógica e gera uma interrupção caso o valor exceda um limite predefinido (superior ou inferior). Essa funcionalidade é crucial para sistemas que demandam monitoramento em tempo real, como dispositivos de segurança ou controle de processos, permitindo ações imediatas em caso de valores fora do esperado.

Outros tipos de ADC

O ponto crucial no projeto de um conversor ADC reside no equilíbrio entre seus diversos parâmetros de desempenho. Para otimizar esse compromisso, surgiram diferentes abordagens tecnológicas, resultando em uma variedade de arquiteturas de ADCs que vão além da comparação com rampa temporal (usada no ADC de rampa) e da comparação por aproximação sucessiva (presente no ADC com SAR). Não existe uma arquitetura ideal para todas as aplicações, pois cada uma oferece vantagens e desvantagens em termos de velocidade de conversão (taxa de amostragem), resolução, precisão, linearidade, consumo de potência, custo e complexidade de implementação. As diferentes arquiteturas de ADC evoluíram para atender às necessidades de uma ampla gama de aplicações, desde sistemas de aquisição de dados de alta velocidade até sensores de baixo consumo de energia. Entre as arquiteturas notáveis, destacam-se o ADC de Dupla Rampa, ADC de Carga-Equilíbrio, ADC Flash e o ADC Delta-Sigma (ΔΣ).

O **ADC de Dupla Rampa** (em inglês, *Dual Slope*) opera em duas fases: primeiro, a tensão de entrada é integrada por um tempo fixo, carregando um capacitor proporcionalmente ao seu valor; em seguida, uma tensão de referência de polaridade oposta é integrada, descarregando o capacitor até zero. O tempo necessário para essa descarga é medido por um contador e é diretamente proporcional à tensão de entrada. A contagem resultante fornece a representação digital do sinal analógico. Esse tipo de conversor é conhecido por sua alta precisão e excelente imunidade a ruídos, embora apresente velocidade de conversão mais baixa. É amplamente utilizado em instrumentos de medição como multímetros digitais.

O **ADC de Carga-Equilíbrio** (em inglês, *Charge-Balancing*), também chamado de ADC de Conversão Tensão-Frequência, converte a tensão de entrada em uma frequência de pulsos. Isso é feito por meio da integração da tensão de entrada e da descarga periódica e controlada de carga em um integrador, equilibrando a carga acumulada. A frequência dos pulsos gerados é proporcional à tensão de entrada e é quantificada por um contador digital. Essa técnica oferece alta resolução, boa linearidade e estabilidade, sendo especialmente adequada para aplicações que exigem precisão e conversão confiável ao longo do tempo, como sistemas de aquisição de dados e sensores com saída em tensão-frequência.

O **ADC *Flash*** (Comparador Paralelo) opera através da **comparação simultânea** da entrada analógica com um conjunto de níveis de referência, utilizando um banco de comparadores cuja saída é diretamente convertida no código digital. Essa abordagem permite conversões de alta velocidade, tornando o ADC *Flash* ideal para aplicações que exigem taxas de amostragem elevadas, como em sistemas de vídeo e comunicação de alta frequência. No entanto, a resolução desse tipo de ADC é geralmente limitada pelo número de comparadores necessários, que cresce exponencialmente com o aumento de *bits*.

Empregando uma abordagem fundamentalmente diferente em sua tecnologia de comparação, o **ADC Delta-Sigma** ($\Delta\Sigma$) se distingue por oferecer alta precisão e resolução, características que o tornam particularmente adequado para aplicações sensíveis à qualidade do sinal digital, como sistemas de áudio de alta fidelidade e a medição de sinais de pequena magnitude em instrumentação de precisão. Ele utiliza um modulador que realiza uma **comparação contínua** e em alta velocidade entre o sinal analógico de entrada e uma versão quantizada de sua própria saída (geralmente de 1 *bit*). Essa comparação contínua, combinada com técnicas mais complexas de modulação, superamostragem (amostragem a uma taxa muito maior que a de Nyquist) e modelagem de ruído (reformatando o espectro do ruído de quantização para frequências fora da banda de interesse), permite converter o sinal analógico em um formato digital com grande exatidão, mesmo para sinais com pequenas variações. Devido a essas características, o ADC Delta-Sigma é frequentemente empregado em cenários que exigem alta fidelidade e baixo ruído. Este Roteiro não aprofundará nas complexidades dessas técnicas de modulação, superamostragem e modelagem de ruído, mas é importante reconhecer seu papel fundamental no alto desempenho dos ADCs Delta-Sigma.

ESPECIFICAÇÕES E MÉTRICAS DE DESEMPENHO

Após conhecermos as diversas arquiteturas de DACs e ADCs, direcionamos nossa atenção para as especificações e métricas de desempenho que caracterizam esses componentes.

Embora o projeto interno detalhado de um circuito conversor possa não ser o foco imediato no desenvolvimento de sistemas embarcados, dada a sua integração na maioria dos microcontroladores e a ampla disponibilidade de circuitos integrados com variados encapsulamentos para integração simplificada, a compreensão dos parâmetros fornecidos nas folhas de dados dos fabricantes é de suma importância para o engenheiro projetista. Esses parâmetros são essenciais para a seleção do componente mais adequado a uma aplicação específica, garantindo o desempenho esperado do sistema. Essa relevância é ainda mais acentuada pela íntima ligação entre os transdutores (sensores e atuadores) e os conversores ADC e DAC, onde uma escolha inadequada pode comprometer a eficácia de toda a cadeia de sensoriamento ou atuação. A seguir, apresentaremos os parâmetros comumente encontrados nas folhas de dados dos fabricantes:

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 ao 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: O **erro de *offset*** se refere ao valor da **saída** do conversor (analógica para DAC, digital para ADC) quando a **entrada** correspondente deveria ser zero (todos os bits de entrada em "0" para DAC, tensão de entrada zero para ADC). Para um DAC, um erro de offset não corrigido resulta em uma tensão de saída diferente de zero quando a entrada digital é zero, adicionando um valor constante e incorreto à saída em todas as condições de entrada. Para um ADC, um erro de offset significa que uma tensão de entrada de zero volts pode resultar em um código digital de saída diferente de zero. O **erro de ganho**,

por sua vez, refere-se ao desvio na inclinação da curva de transferência do conversor em relação à inclinação ideal. Para um DAC, ele se manifesta como a diferença entre a tensão de saída real no fundo de escala e a tensão de saída esperada para o código digital de entrada máximo. Para um ADC, ele se manifesta como a diferença entre a tensão de entrada real necessária para atingir o código digital de fundo de escala e a tensão de entrada esperada.

Muitos DACs e ADCs oferecem ajustes de *offset* e ganho para mitigar esses problemas. O ajuste de *offset* garante que a saída seja o mais próxima de zero possível quando a entrada correspondente é zero. O ajuste de ganho assegura que a saída atinja seu valor máximo (ou código máximo para ADC) quando a entrada atinge a condição esperada para o fundo de escala. Essa calibragem é essencial para garantir que os erros de *offset* e ganho permaneçam dentro de limites aceitáveis ao longo do tempo, assegurando a precisão das conversões.

Monotonicidade: embora seja uma característica importante para ambos os tipos de conversores, DACs e ADCs, o termo é primariamente associado aos DACs, especialmente em aplicações de controle e geração de forma de onda. Em DACs, é uma especificação que garante uma relação de saída analógica sempre crescente (ou constante) com uma entrada digital crescente, relevante para a estabilidade e o comportamento previsível em aplicações como controle.

Tempo de estabilização (em inglês, *Settling Time*): Este termo é primariamente associado a DACs. 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 (em inglês, *Conversion Time*): este termo é primariamente associado a ADCs. Ele se refere ao tempo total necessário para o ADC realizar o processo completo de conversão de um sinal analógico de entrada em um valor digital de saída. Este processo envolve amostragem, quantização e codificação do sinal analógico. 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.

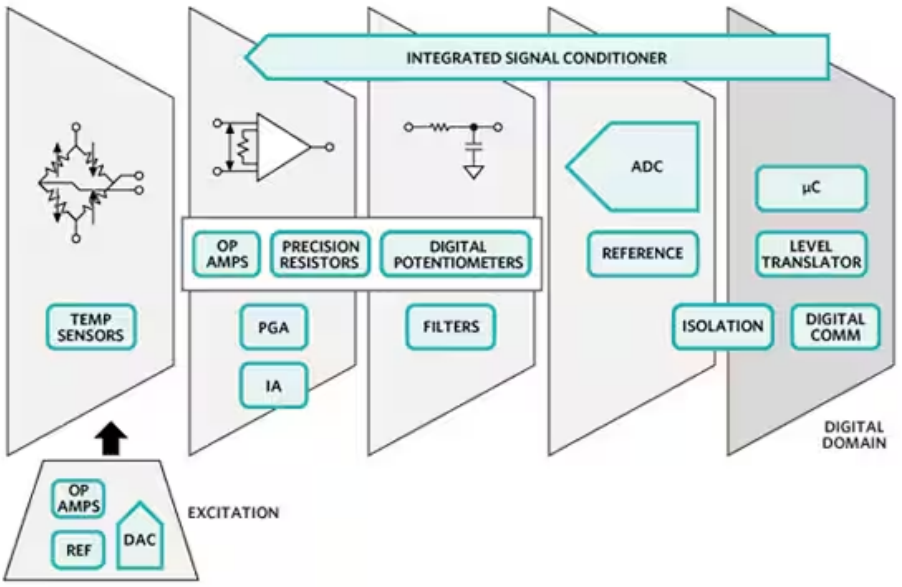
Consumo de Energia: É um parâmetro importante, especialmente em sistemas embarcados, onde a eficiência energética é fundamental. Um baixo consumo de energia pode prolongar a vida útil da bateria e reduzir os custos operacionais. Muitos conversores AD e DA oferecem modos de operação de baixo consumo que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

ARQUITETURA DE INTERFACE ANALÓGICA EM MICROCONTROLADORES

Os sinais sobre os quais os conversores atuam são sinais elétricos. No entanto, os sinais que encontramos no mundo real não são todos elétricos. Podemos ter sinais de diferentes naturezas, como luminosidade (luz), temperatura, pressão, som, movimento, entre outros. Para que esses sinais possam ser processados por circuitos eletrônicos, como microcontroladores, precisamos convertê-los para sinais elétricos. Essa conversão é realizada por **transdutores**, dispositivos que transformam uma forma de energia em outra. No contexto da eletrônica, os **sensores** são um tipo específico de transdutor que converte grandezas físicas do ambiente em sinais elétricos, geralmente na forma de tensões ou correntes, que podem ser manipuladas e interpretadas por circuitos eletrônicos.

Da mesma forma, para que os circuitos eletrônicos, como microcontroladores e processadores, possam interagir com o mundo físico e produzir efeitos, eles precisam converter sinais elétricos de volta para outras formas de energia. Essa função é desempenhada por **atuadores**, que são transdutores que convertem sinais elétricos (tensões ou correntes) em ações físicas. Exemplos de atuadores incluem motores (que convertem energia elétrica em movimento mecânico), alto-falantes (que convertem sinais elétricos em ondas sonoras), LEDs (que convertem eletricidade em luz), resistores de aquecimento (que convertem eletricidade em calor) e válvulas solenoides (que convertem eletricidade em controle de fluxo). Assim, o conceito de transdutor abrange tanto a captura de informações do mundo real (através de sensores) quanto a atuação sobre ele (através de atuadores).

Os transdutores permitem que os microcontroladores se limitem a lidar com sinais elétricos em sua interação com o mundo exterior. Após a obtenção de um sinal elétrico, este geralmente precisa passar por diversos componentes, denominados **condicionadores de sinais**, para torná-lo apropriado para processamento pelo ADC.



(Fonte: Digikey).

Um dos blocos essenciais nos condicionadores de sinais é o **amplificador operacional (OP-AMP)**, amplamente utilizado para aumentar a amplitude do sinal analógico de entrada. Sua função principal é elevar o nível de sinais fracos através dos **resistores de precisão** em sua malha de realimentação, garantindo um ganho adequado enquanto se preservam características importantes como largura de banda e resposta em frequência dentro dos limites desejados. Além da amplificação, os OP-AMPs também oferecem a flexibilidade de inverter ou não a polaridade do sinal.

Outro componente importante são os **filtros**, empregados para modificar o conteúdo de frequência de um sinal analógico ou selecionar faixas específicas. Na interface analógica, filtros passa-baixa (para atenuar altas frequências), passa-alta (para atenuar baixas frequências) e passa-banda (para selecionar uma faixa de frequências) são comuns para remover ruídos, eliminar componentes indesejados ou isolar sinais de interesse. **Potenciômetros digitais** podem ser integrados aos circuitos de filtragem para permitir o ajuste dinâmico das frequências de corte ou de outras características do filtro, variando os valores de resistência na rede do filtro sob controle digital.

Além da amplificação e filtragem, os **condicionadores de sinal** desempenham um papel crucial no ajuste e otimização de outras características do sinal analógico, como linearização, compensação de offset, atenuação, nivelamento de ganho e calibração. Esses processos garantem que o sinal analógico esteja em conformidade com os requisitos de precisão e qualidade necessários para a subsequente conversão digital.

É importante notar que os conversores **ADC** modernos frequentemente **integram circuitos de condicionamento de sinal** dentro do próprio *chip*, o que simplifica o projeto de sistemas embarcados para os desenvolvedores. Isso permite que os engenheiros se concentrem na configuração do sistema por meio de **registradores**, em vez de precisarem montar circuitos

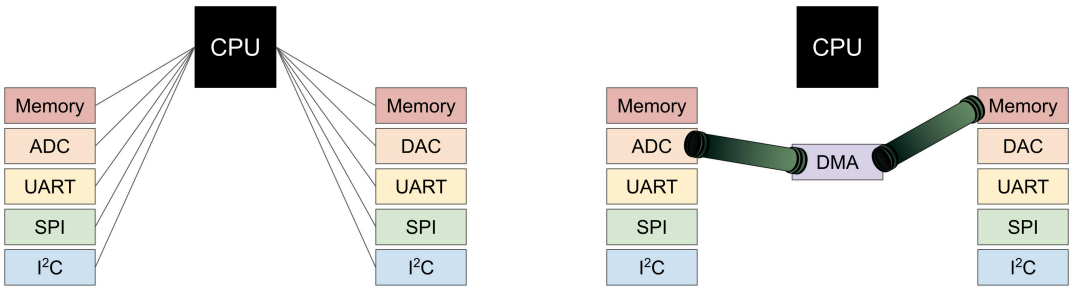
externos complexos. Da mesma forma que os condicionadores de sinal preparam os sinais analógicos para a entrada do ADC, um pós-processamento adequado é frequentemente necessário para ajustar os sinais de saída do **DAC**, garantindo que tenham a amplitude e características corretas para controlar dispositivos externos. A natureza desse pós-processamento varia de acordo com a aplicação e o tipo de dispositivo que está sendo controlado ou acionado, geralmente exigindo circuitos adicionais que não são tipicamente integrados ao microcontrolador.

Por fim, muitos ADCs modernos incorporam **multiplexadores (MUX)** para selecionar qual sinal analógico de entrada será convertido e, em alguns casos (para DACs), **demultiplexadores (DEMUX)** para direcionar o sinal analógico de saída para diferentes destinos. Essa capacidade de seleção de canais é particularmente útil em sistemas que interagem com múltiplos sensores ou outras fontes/destinos de sinais analógicos.

ACESSO DIRETO À MEMÓRIA

À medida que a complexidade dos sistemas embarcados aumenta, surgem desafios na gestão do fluxo de dados entre periféricos (como ADCs e DACs) e a memória do sistema, especialmente em altas velocidades ou com grandes volumes de dados. A manipulação direta e contínua desses dados pela Unidade Central de Processamento (CPU) pode criar gargalos de desempenho, limitando a capacidade de resposta do sistema. Imagine um sistema que precisa monitorar continuamente múltiplos sensores através de um ADC ou gerar formas de onda complexas através de um DAC em altas velocidades. Se cada amostra convertida ou cada ponto de dados a ser gerado exigisse a intervenção direta do nosso processador principal, rapidamente nos depararíamos com um gargalo de desempenho, limitando a capacidade de resposta e a eficiência do sistema.

Para superar essa limitação, os microcontroladores modernos incorporam *hardware* dedicado, como o Controlador de **Acesso Direto à Memória** (em inglês, *Direct Memory Access* – DMA) capaz de gerenciar a transferência de dados entre periféricos e a memória de forma autônoma, sem a necessidade de intervenção contínua da CPU durante a transferência, como ilustra a figura que se segue.



(Fonte: DigiKey).

O conceito DMA 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 entendemos hoje. Nessa 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 otimiza 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.

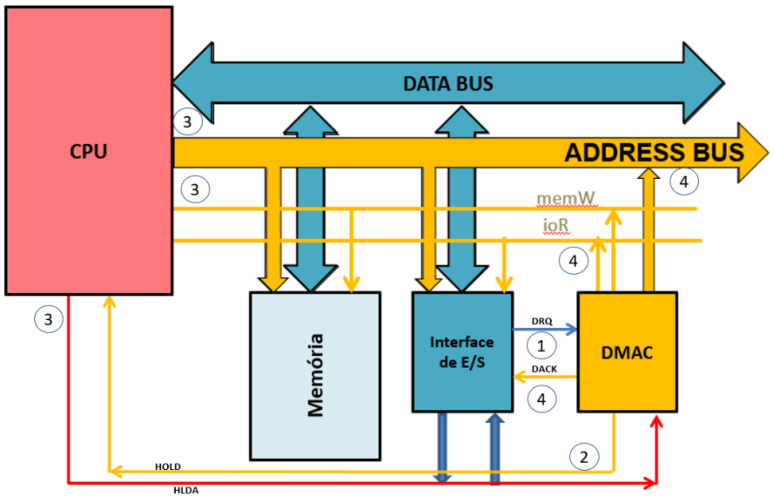
A primeira, conhecida como **sistema de acesso direto à memória de primeira parte** (em inglês, *first-party DMA*), baseia-se na arbitragem do uso de um recurso compartilhado, o barramento, entre a memória principal e os periféricos que podem se comunicar diretamente. Nesse caso, o periférico que ganha 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. Quando um periférico atua como mestre de barramento, ele assume o controle do barramento do sistema durante a transferência de dados, mas a CPU é ainda responsável pela configuração inicial da transferência. Isso pode fazer com que a CPU experimente uma sobrecarga devido ao tempo de espera forçado, à interrupção do fluxo de execução e ao envolvimento no gerenciamento da transferência. Essa sobrecarga reduz a eficiência da CPU e pode fazer com que o sistema pareça mais lento. Um controlador DMA dedicado, por outro lado, opera de forma independente da CPU, 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

Na abordagem de DMA de terceira parte, o periférico negocia o acesso ao barramento com o DMAC quando ele deseja realizar uma transferência direta de dados para ou da memória principal. O DMAC pode operar no mesmo barramento do sistema que a CPU e outros dispositivos, ou pode usar barramentos separados, dependendo da arquitetura do sistema. Em muitos sistemas, o DMAC **compartilha o mesmo barramento** do sistema com a CPU e outros periféricos. Nesse cenário, o controlador precisa solicitar o barramento e aguardar a concessão para realizar a transferência. Um árbitro de barramento gerencia as solicitações e concede o acesso ao barramento a um dispositivo por vez. Alguns sistemas possuem **barramentos separados** para DMA. Por exemplo, pode haver um barramento de alta velocidade dedicado para transferências entre o DMAC e a memória principal. Isso pode melhorar ainda mais o desempenho, pois o DMA não compete com a CPU pelo mesmo barramento.

Quando o DMAC compartilha o mesmo barramento com a CPU, ele negocia diretamente com o árbitro do barramento para gerenciar o tráfego de dados, dispensando a intervenção da CPU. A figura ilustra esse processo, detalhando as cinco etapas básicas de acesso ao barramento para transferências diretas:



1. **Requisição de DMA** pelo periférico, enviando o sinal DRQ ao DMAC.
2. **Solicitação de liberação dos barramentos** da CPU (HOLD) pelo DMAC.
3. **Concessão do Barramento** 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. **Transferência de dados** quando 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 gerencia o processo de transferência, gerando 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).

Quando a contagem de *bytes*/palavras atinge zero (ou outra condição de término é satisfeita), a transferência é concluída. O DMAC então sinaliza o fim da transferência ao periférico e, opcionalmente, à CPU (geralmente através de um sinal de interrupção). Após a conclusão da transferência, o DMAC libera o controle dos barramentos, devolvendo-o à CPU. A CPU pode então retomar suas operações normais.

Antes de iniciar a transferência, a CPU (ou outro mestre do sistema) configura o controlador DMA com as informações necessárias sobre a transferência, incluindo:

- **endereço de origem:** O endereço inicial na memória ou no periférico de onde os dados serão lidos.
- **endereço de destino:** O endereço inicial na memória ou no periférico para onde os dados serão escritos.
- **contagem de *bytes*/palavras:** O número de unidades de dados a serem transferidas.
- **direção da transferência:** Leitura da memória para o periférico ou escrita na memória a partir do periférico (ou memória para memória).
- **modo de transferência:** Define a maneira como os dados são transferidos, podendo ser em bloco contínuo, unidade por unidade, ou outras modalidades.
- **incremento/decremento de endereço:** Se os endereços de origem e/ou destino devem ser incrementados ou decrementados após cada transferência.

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.

Em microcontroladores modernos com múltiplos canais DMA ativos, pode-se atribuir **prioridades relativas entre os canais**, permitindo que transferências consideradas mais críticas tenham preferência no acesso ao barramento de dados. O circuito de arbitragem integrado ao controlador DMA (**arbitrator**) é responsável por decidir qual canal DMA ativo terá acesso ao barramento a cada ciclo, em situações onde mais de um canal solicita transferência simultaneamente. Essa arbitragem gerencia a competição por recursos compartilhados, como a memória principal ou periféricos, e pode seguir diferentes p**olíticas de arbitragem**, cada uma com diferentes compromissos em termos de latência, justiça e eficiência do barramento, incluindo:

- **Arbitragem por Pedido de Transferência de Canal Único** (em inglês, *Single-Channel Request*): Este é um modo simples onde apenas **um canal de DMA** pode ser atribuído para cada **linha de requisição**. Ou seja, cada linha de requisição (periférico) só pode ser mapeada para um único canal de DMA por vez.
- **Arbitragem por Prioridade** (em inglês, *Priority-Based Arbitration*): Esta política de arbitragem atribui prioridades aos canais de DMA, o que permite que o controlador de DMA selecione o canal com maior prioridade quando várias linhas de requisição competem pelo acesso ao barramento.

- **Arbitragem por *Round-Robin*** (em inglês, *Round-Robin Arbitration*): Nesta política, os canais de DMA são atendidos de forma cíclica. Se várias linhas de requisição de DMA estiverem ativas ao mesmo tempo, o controlador de DMA atende a cada uma delas de forma sequencial, "rotacionando" entre os canais disponíveis.
- **Arbitragem por Prioridade com *Round-Robin*** (em inglês, *Priority and Round-Robin Hybrid*): Esta política é um híbrido entre a arbitragem por prioridade e o *round-robin*, onde o controlador de DMA dá prioridade a certos canais, mas, em vez de atender o canal com maior prioridade continuamente, ele também faz uma rotação regular para garantir que os canais de prioridade mais baixa também sejam atendidos de forma justa.
- **Arbitragem Automática de Canal** (em inglês, *Automatic Channel Arbitration*): Há microcontroladores que realizam a arbitragem automática de canais, onde a alocação do canal DMA é feita com base no tipo de periférico e na disponibilidade do barramento de dados pelo DMAC, sem intervenção do *software*.

Por fim, os modos de transferência 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 DMAC

Os modos de operação de um controlador DMA se referem às diferentes configurações que determinam como ele gerencia as transferências de dados entre a memória e os periféricos, ou entre diferentes áreas da memória. Cada modo possui características específicas que o tornam mais adequado a determinadas necessidades de transferência. Seguem-se alguns dos modos de operação mais comuns, amplamente suportados pelos microcontroladores modernos como STM32H7.

Transferência por Rajada (em inglês, *Burst Transfer Mode*): Nesse modo, o controlador DMA executa uma sequência contínua de transferências de blocos de dados (uma "rajada") para um único endereço de destino ou a partir de um único endereço de origem, sem a necessidade de acionar individualmente cada operação. O número de transferências (tamanho da rajada) é geralmente configurável nos registradores do DMAC. Tamanhos de rajada maiores podem aumentar a taxa de transferência, pois reduzem a sobrecarga da fase de arbitragem do barramento para cada dado individual. Isso é especialmente eficiente para leituras ou escritas contínuas. No entanto, durante uma rajada de transferências, o controlador DMA assume o controle exclusivo do barramento do sistema, impedindo temporariamente a CPU de realizar outras operações. Essa monopolização garante a eficiência das transferências, mas pode suspender momentaneamente a comunicação entre a CPU e os periféricos.

Roubo de Ciclo (em inglês, *Cycle Stealing Mode*): Esse é o modo de operação fundamental do DMA. Nesse modo, o DMA "rouba" ciclos de barramento da CPU de forma pontual para realizar suas transferências. Em vez de assumir o controle total do barramento, como no modo de rajada, o DMA interrompe a CPU apenas durante os ciclos estritamente necessários para a transferência de dados. Essa abordagem permite que a CPU continue executando suas tarefas

com interrupções mínimas, sendo ideal para sistemas em que a CPU deve permanecer responsiva a eventos e interações frequentes com periféricos.

Transferência Intercalada (em inglês, *Interleaving Transfer Mode*) ou **Transferência Transparente** (em inglês, *Transparent Transfer Mode*): Neste modo, o DMA realiza transferências de forma intercalada com as operações da CPU, de modo que ambas compartilham o barramento de maneira alternada. A CPU nunca é completamente interrompida, pois o DMA só transfere dados nos momentos em que o barramento não está sendo utilizado pela CPU. Isso preserva a execução contínua dos programas da CPU e evita interferências perceptíveis em seu desempenho. No entanto, essa técnica exige lógica adicional no *hardware* para identificar quando o barramento está disponível, o que pode aumentar a complexidade do sistema. Além disso, as transferências são mais lentas do que nos outros modos, embora esse seja, em muitos casos, o modo mais eficiente em termos de desempenho geral do sistema.

Esses modos de operação são de fato técnicas de baixo nível, implementadas diretamente no *hardware* do controlador DMAC. Eles ditam como o DMAC acessa o barramento do sistema e gerencia a transferência de dados sem a intervenção constante da CPU. Esses modos são geralmente transparentes para o desenvolvedor de *software*, que não precisa se preocupar com os detalhes de como cada *byte* ou bloco é transferido no nível do *hardware*. Os modos configuráveis oferecidos pelos microcontroladores são abstrações de mais alto nível, construídas sobre esses modos de *hardware* fundamentais. Eles oferecem mais flexibilidade e conveniência para o programador, permitindo configurar transferências de dados de maneiras mais adequadas às necessidades da aplicação. Entre os modos de transferência avançados figuram:

Modo de Transferência Normal (em inglês, *Normal Transfer Mode*): É o modo mais simples e direto, no qual o DMA transfere um número finito de dados de uma origem para um destino (como de memória para periférico ou vice-versa). Após completar a transferência, o canal DMA é automaticamente desabilitado. É ideal para tarefas únicas e pontuais.

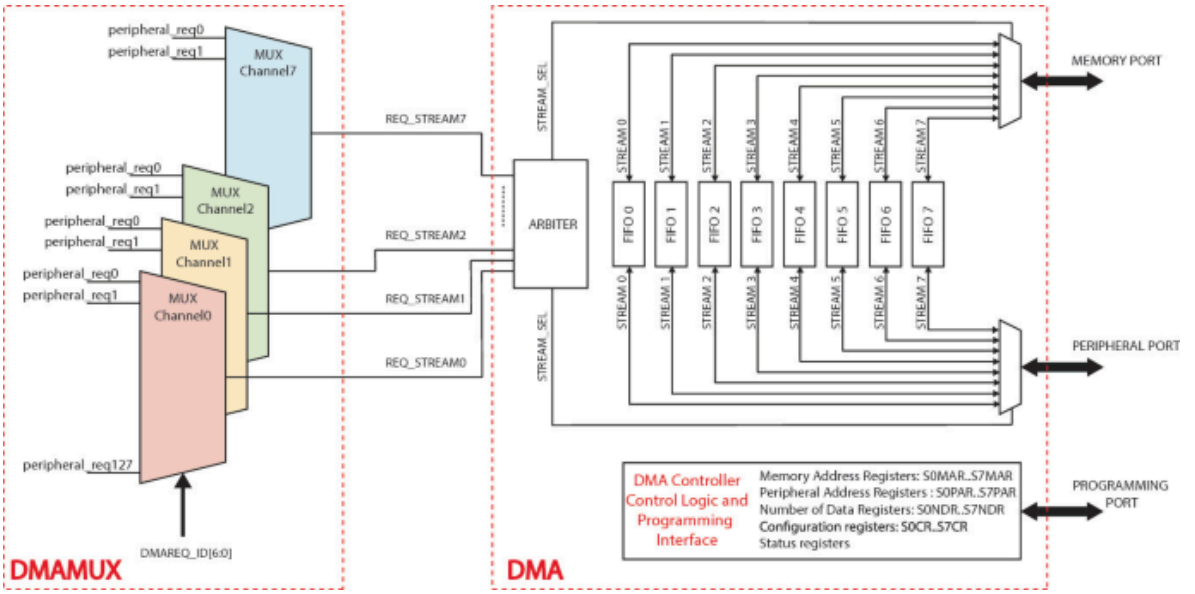
Modo de Transferência Circular (em inglês, *Circular Transfer Mode*): Este modo permite que o DMA continue transferindo dados de maneira contínua e cíclica. Após atingir o final do *buffer* de memória, o DMA reinicia automaticamente a transferência a partir do início do buffer, permitindo operação contínua e sem intervenção da CPU. É amplamente utilizado em aplicações de aquisição contínua de dados, como em leituras periódicas de ADCs ou recepção de dados seriais.

Modo de Transferência em Bloco (em inglês, *Block Transfer Mode*)**:** Tecnicamente, esse modo se refere à realização de transferências agrupadas, utilizando rajadas (*bursts*) sucessivas para mover um bloco completo de dados sem a intervenção do software. Isso permite otimizar o uso do barramento e reduzir o overhead de controle. Embora as transferências permaneçam sequenciais, elas ocorrem em pacotes, o que melhora significativamente o desempenho em memórias rápidas ou em periféricos que possuem *buffers* internos.

Roteador de requisições de DMA

Um controlador DMA tradicional gerencia múltiplos canais, cada um dedicado a uma fonte ou destino específico. Embora eficiente para transferências bem definidas, essa abordagem da ligação entre canais DMA e periféricos ser fixa no projeto do *hardware* dificulta a adaptação dinâmica a diferentes cenários de operação ou a alocação otimizada de recursos de transferência para tarefas específicas. O **roteador de linhas de requisições DMA**, ou **multiplexador para DMA** (DMAMUX), surge para mitigar essas limitações, atuando como uma camada intermediária entre os periféricos que solicitam transferência e os canais DMA disponíveis. A principal função desse circuito eletrônico é rotear os sinais de requisição gerados por periféricos para os canais DMA configurados, permitindo maior flexibilidade na associação entre fontes de requisição e canais de transferência. Isso otimiza o uso dos recursos do controlador DMA, especialmente em microcontroladores com múltiplos periféricos e canais disponíveis. Uma política básica de multiplexação envolve, conforme ilustrado na figura:

- **identificação da requisição:** Cada requisição gerada por um periférico, `peripheral_reqn`, atua como um sinal de *trigger*, solicitando ao canal DMA correspondente que inicie uma transferência. As informações sobre endereços de origem e destino, tamanho da transferência, e prioridade relativa do canal devem ser previamente configuradas nos registradores do canal DMA, via *software*.
- **mapeamento para canais DMA:** Cada canal DMA possui uma entrada correspondente no DMAMUX. Essa entrada é configurada via registrador, indicando qual periférico será a fonte de requisição para aquele canal. O DMAMUX apenas encaminha o sinal de *trigger* ao canal DMA apropriado.



(Fonte: Patricio Bulić)

Estrutura de dados: *buffer* circular

Imagine um sistema embarcado monitorando um sensor de temperatura através de um ADC. A cada leitura do sensor, o ADC converte o valor analógico em um dado digital e transfere este dado para um endereço da memória do microcontrolador diretamente via DMA. A cada nova conversão do ADC, um novo valor digital está pronto para ser transferido para o mesmo endereço de memória. Se o processador (CPU) não concluir o processamento do dado anterior no momento em que o DMAC tenta escrever o novo dado, o dado anterior na memória seria sobrescrito pelo novo valor. Em outras palavras, se o processamento do dado anterior levar um pouco mais de tempo do que o intervalo entre as conversões do ADC, dados importantes seriam perdidos. Ao reservarmos um bloco de memória, ou um vetor/*buffer* linear com mais de um dado, para transferência, o DMAC pode escrever o novo dado num endereço subsequente do dado anterior.

Mesmo com um *buffer* linear (não circular), surge outra limitação: ao ser completamente preenchido pelo DMAC, a transferência de dados cessaria. A continuidade da aquisição exigiria que o processador fosse alertado sobre o *buffer* cheio, processasse seu conteúdo e, então, reconfigurasse o DMA para sobrescrever o *buffer* desde o início ou utilizar um novo. Nesse contexto, a integração de ***buffers* circulares** com o DMAC oferece uma solução elegante e eficiente. 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.

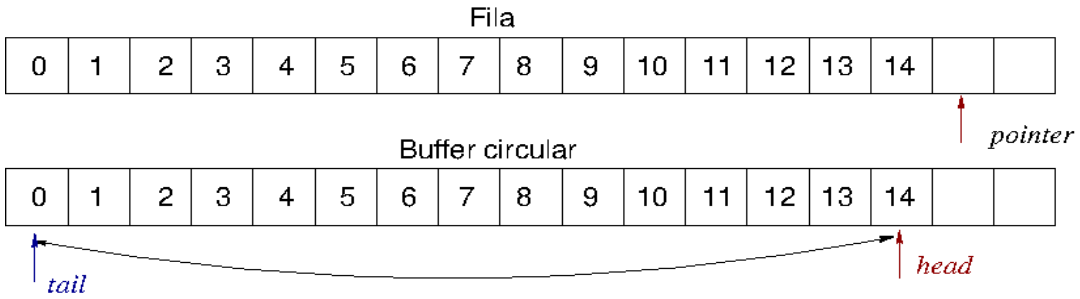


Figura: Fila e *buffer* circular.

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;
```

Imagine agora o DMA trabalhando com um *buffer* circular. Quando o ADC gera um novo dado, o DMA o escreve na próxima posição disponível no *buffer*. Ao invés de parar quando atinge o final do *buffer* linear, o DMA **automaticamente retorna ao início**, sobrescrevendo os dados mais antigos – dados esses que, idealmente, já teriam sido lidos e processados pelo processador.

Essa abordagem **elimina o problema da sobrescrita imediata e da perda de dados** que vimos na transferência direta sem *buffer*. O *buffer* circular atua como um "armazém" temporário, dando ao processador mais tempo para consumir os dados antes que sejam sobrescritos. Além disso, a natureza circular do *buffer*, gerenciada automaticamente pelo *hardware* do DMA (em muitos microcontroladores), **resolve o problema da descontinuidade e da gestão complexa** inerentes ao uso de *buffers* lineares sem reinicialização manual. O DMA opera de forma contínua, e o processador pode ler os dados do *buffer* em seu próprio ritmo, sem a necessidade de interromper a aquisição ou reconfigurar constantemente o DMA.

No entanto, em aplicações como a conversão digital-analógica (DA), podem ocorrer problemas se a CPU não conseguir atualizar os dados na mesma velocidade com que o DMAC os consome. Nesse cenário, o DMA pode sobrescrever dados ainda não processados ao reiniciar o ciclo de escrita no *buffer* circular, especialmente se todo o *buffer* já tiver sido preenchido. Essa situação caracteriza um transbordo ou sobrecarga (em inglês, *overrun*) do *buffer*. Embora o modo circular amplie a janela de tempo para a CPU acessar ou atualizar os dados, ele não impede que a sobrescrita ocorra caso o processamento não acompanhe o ritmo do DMA. O DMAC apenas executa as transferências conforme configurado, sem mecanismos de *hardware* para detectar ou evitar essa condição.

A prevenção de perda de dados deve, portanto, ser implementada em *software*, por meio de estratégias como:

- **Uso de interrupções de "meia transferência" e "transferência completa"**: permitem que o *software* seja notificado em tempo real quando metade ou todo o *buffer* for preenchido, possibilitando a leitura ou substituição dos dados antes que sejam sobrescritos.

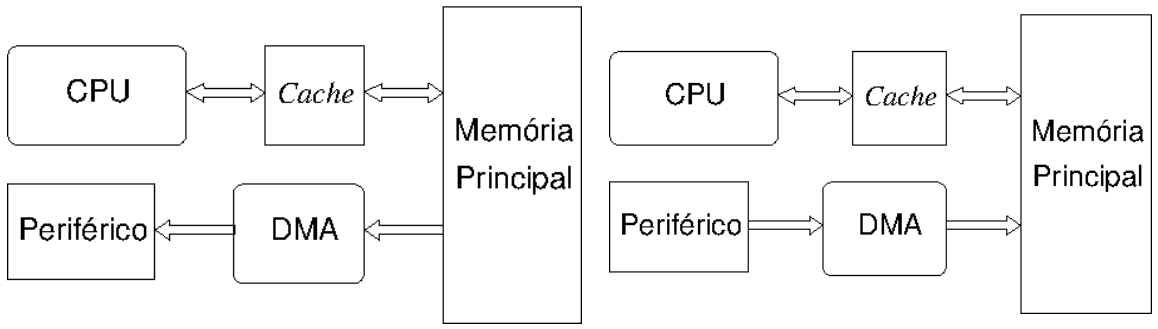
- **Ponteiros de leitura** (*tail*) **e escrita** (*head*): implementam controle mais fino sobre o acesso ao *buffer*, permitindo que o *software* acompanhe onde o DMA está escrevendo e onde a CPU deve ler, evitando colisões de acesso.

- ***Buffers* maiores ou duplos** (em inglês, *double buffering*): aumentam o tempo disponível para o processamento ao permitir que, enquanto um *buffer* é preenchido pelo DMA, o outro seja processado pela CPU, trocando-se os papéis após cada ciclo.

Esse cenário ilustra a importância do *software* no gerenciamento da transferência direta de dados entre a CPU e os periféricos via DMAC com *buffer* circular, complementando as limitações do *hardware*.

Coerência de *cache*

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* eventualmente 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.

Outra solução simples é declarar a região de memória usada com DMA como não cacheável. Isso evita o problema, mas sacrifica desempenho.

SENSORES ANALÓGICOS

Construindo sobre o conhecimento adquirido com sensores (botões, *keypads* de membrana) e atuadores digitais (LEDs, motores), este Roteiro representa uma progressão para o domínio dos sensores analógicos. Apresentaremos três exemplos de interface simples, escolhidos especificamente para ilustrar e explorar os conceitos de conversão Analógico-Digital (AD). Vamos examinar três sensores distintos: um joystick, um potenciômetro e um sensor de temperatura. Cada um desses dispositivos converte uma ação física específica, o movimento de uma alavanca, a rotação de um eixo e a variação de calor, respectivamente, em um sinal elétrico analógico correspondente. Ao analisarmos o princípio de funcionamento de cada sensor, entenderemos como diferentes fenômenos físicos podem ser capturados e representados eletricamente, facilitando sua leitura e interpretação por microcontroladores e outros circuitos eletrônicos.

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.

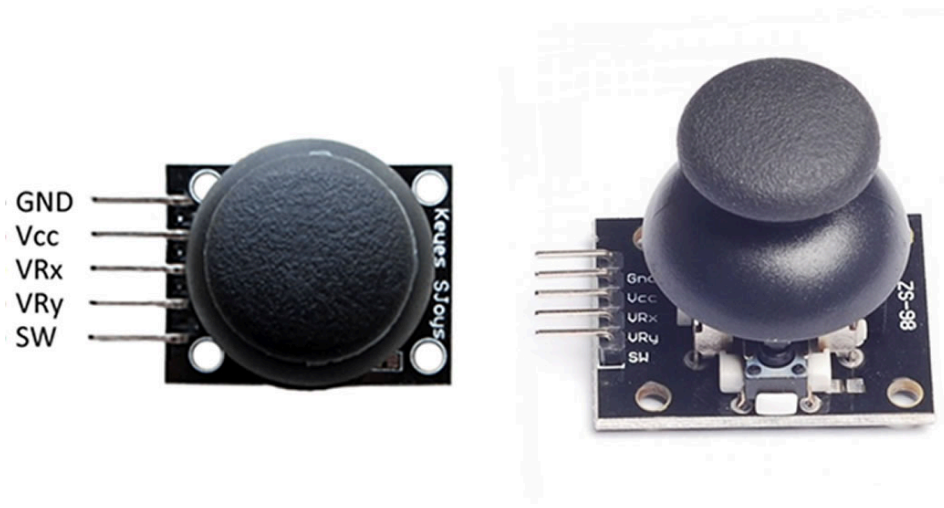
O módulo de *joystick* analógico normalmente tem cinco pinos principais:

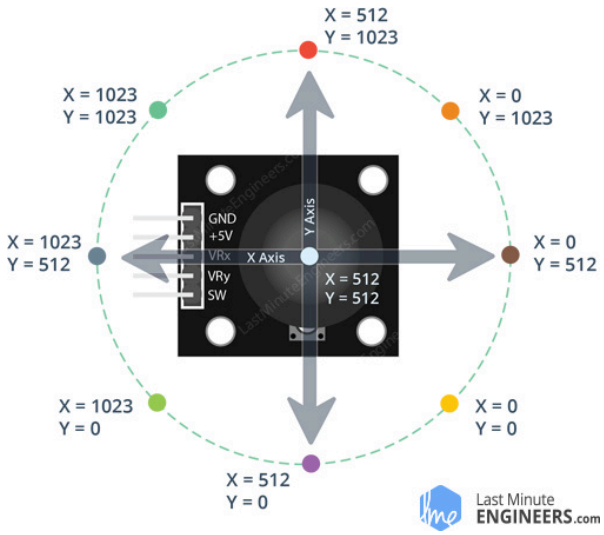
- VCC: Alimentação. Muitas vezes marcado como “+5V”, mas pode ser qualquer valor de tensão, definindo o valor máximo de tensão. No projeto deste roteiro, usamos 3.3V, pois este é o valor máximo aceito pelo ADC.
- GND: Terra
- VRx: Saída analógica do eixo X
- VRy: Saída analógica do eixo Y
- SW: Saída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

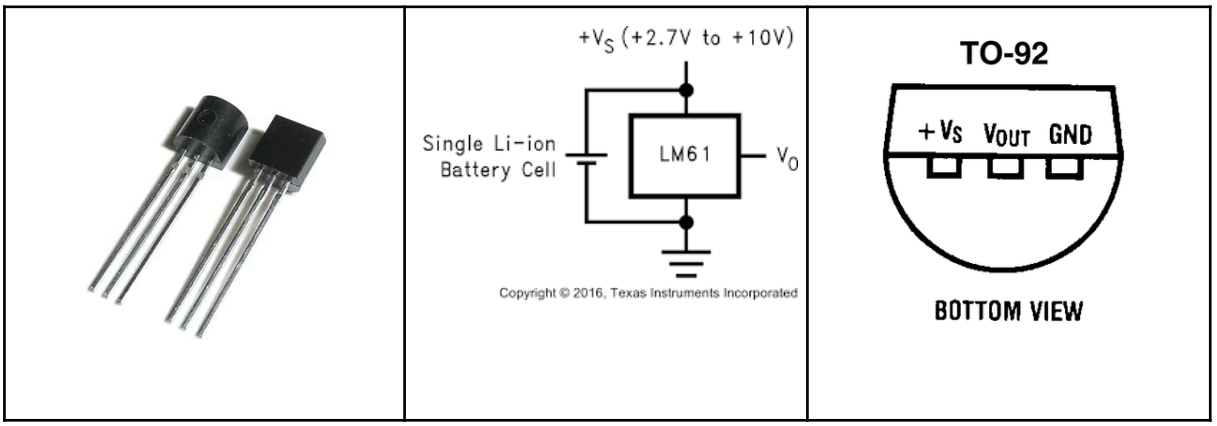
Fonte: [Squids]

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**.



(Fonte: Texas Instruments)

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).

ATUADORES ANALÓGICOS

No contexto de sistemas embarcados, os **atuadores analógicos** são dispositivos que convertem sinais elétricos contínuos em ações físicas observáveis no ambiente, como movimento, som ou variações luminosas. Esses atuadores permitem que sistemas digitais interajam com o mundo real de maneira precisa e controlada, atuando como a "voz" ou a "força" dos microcontroladores em diversas aplicações, desde automação residencial até sistemas médicos e automotivos. Diferentemente dos atuadores digitais, que operam apenas em estados discretos (ligado/desligado), os atuadores analógicos respondem a uma **variação contínua de entrada**. Isso possibilita ajustes finos e controle proporcional da atuação, como controlar a velocidade de um motor, a intensidade de uma luz LED ou a frequência de um som. Vamos ilustrar a operação de um atuador através de um *buzzer* ou um disco piezoelétrico.

Disco Piezoelétrico

Um disco piezoelétrico é um componente fino e circular que utiliza o efeito piezoelétrico para converter sinais elétricos em vibrações mecânicas, gerando ondas sonoras. Ele é comumente utilizado em alarmes, campainhas, pequenos alto-falantes e sensores de vibração.

Ao aplicar uma tensão alternada nos terminais do disco, o material piezoelétrico se deforma ligeiramente, expandindo e contraindo em resposta à variação do sinal elétrico. Essas deformações, quando ocorrem em frequências audíveis, produzem vibrações no ar circundante, resultando na emissão de som. A faixa de tensão de operação para excitar um disco piezoelétrico pode variar significativamente dependendo do modelo e da aplicação, mas geralmente situa-se entre alguns volts (V) de pico a pico até dezenas de volts. A corrente consumida também é relativamente baixa, tipicamente na ordem de miliamperes (mA), especialmente em frequências de ressonância onde a eficiência da conversão é maior.

As saídas típicas de um disco piezoelétrico são sinais sonoros, cuja intensidade (volume) e frequência dependem diretamente da amplitude e da frequência do sinal elétrico aplicado. Esses fatores também são influenciados pelas características físicas do disco, como diâmetro, espessura e material, e pela forma como ele está montado. Para aumentar o volume do som

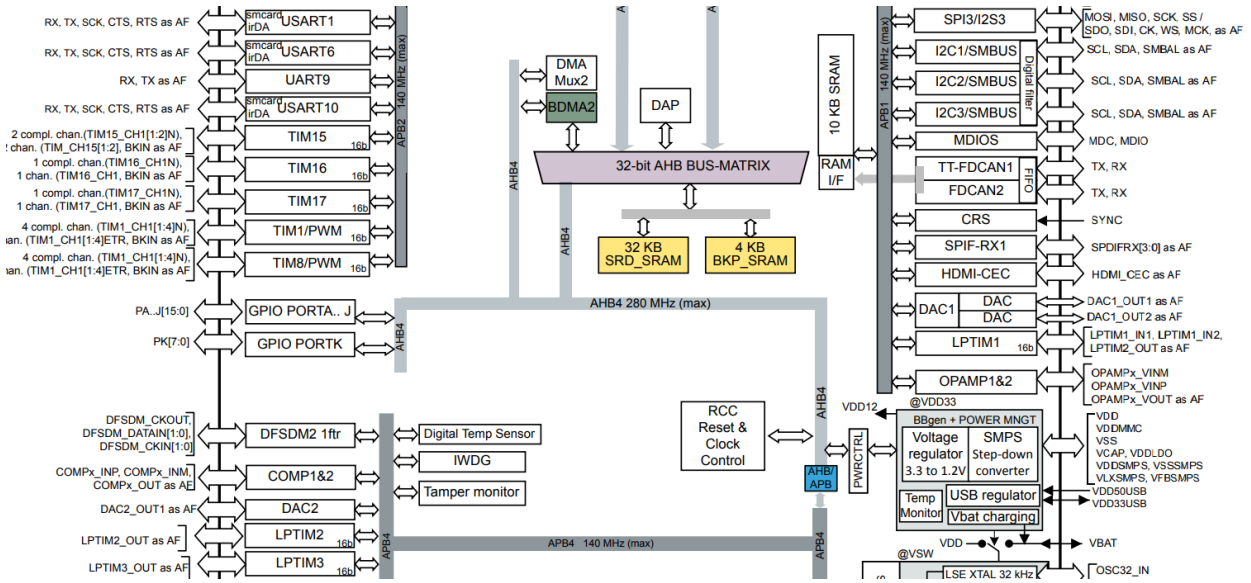
gerado, o disco piezoelétrico utilizado neste Roteiro foi acoplado a uma cavidade ressonante confeccionada com um tubo de PVC, que atua como amplificador acústico, como mostra na seguinte figura.

STM32H7A3

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

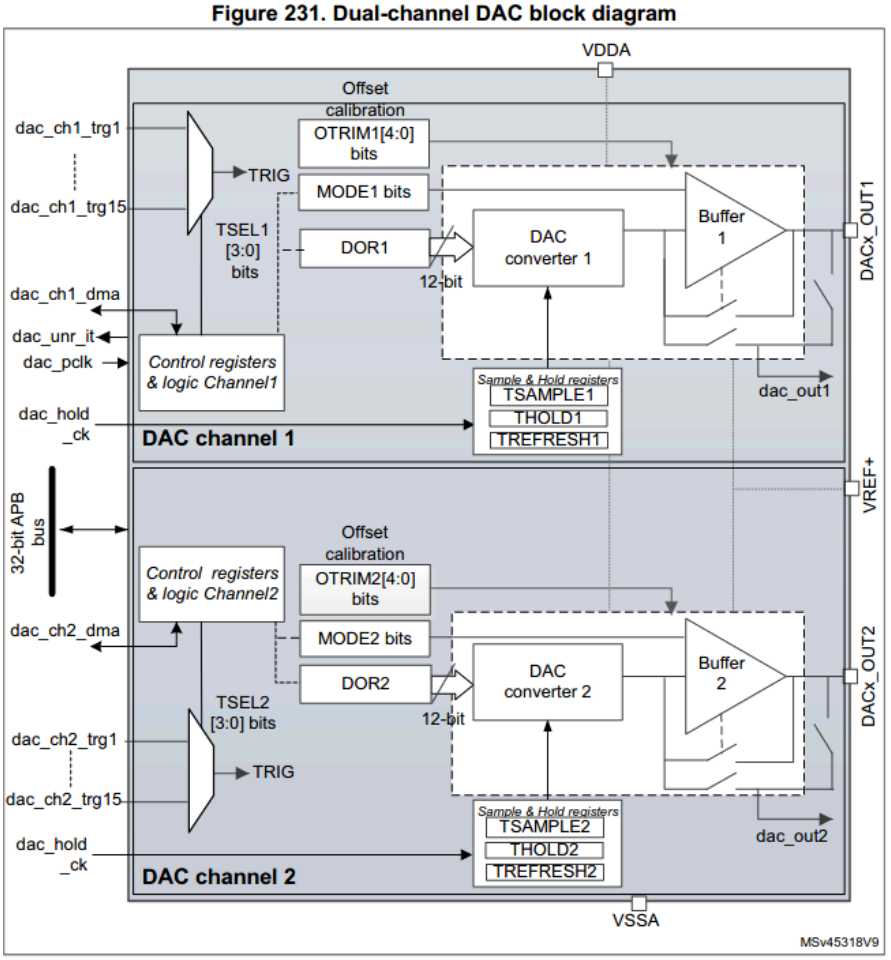
DAC

O STM32H77A3 possui duas unidades conversoras digital-analógico (DAC), denominadas DAC1 e DAC2. Essas conversoras utilizam a **arquitetura R/2R** para realizar a conversão digital-para-analógica, aproveitando as vantagens desta configuração em termos de simplicidade e precisão. 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*.



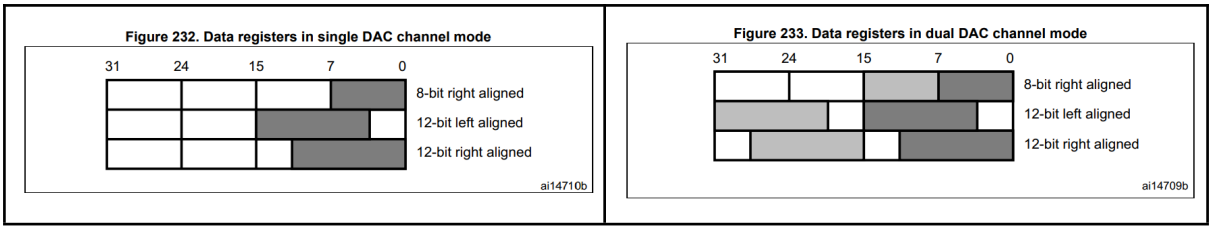
O diagrama de blocos, retirado do Manual de Referência, ilustra claramente a funcionalidade do DAC2 com os blocos distintos “DAC channel 1” e “DAC channel 2”. Cada canal possui seu próprio caminho de entrada de dados e registradores de controle para os pinos de saída, `DACn_OUT1` e `DACn_OUT2`. Essa 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 quantidade de *bits* (8 *bits* ou 12 *bits*) dos dados), 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]).

Os dados digitais a serem convertidos para uma saída analógica são escritos pelo *software* nos registradores de retenção de dados (DAC_DHRx) correspondentes a cada canal do DAC, como DAC_DHR12R1 ou DAC_DHR12L1. Estes registradores DAC_DHRx estão mapeados na memória e possuem endereços específicos. A conversão digital-analógica e a atualização da tensão na saída do DAC ocorrem com base no valor digital presente no registrador de saída de dados (DAC_DORx) para o canal correspondente, como DAC_DOR1. A transferência dos dados do registrador DHRx para o registrador DORx é controlada pelo *bit* DAC_CR_TENx (*Trigger Enable*) para cada canal, por exemplo, *bit* DAC_CR_TEN1 para o Canal 1). Existem duas configurações principais para este controle:

1. Quando o *trigger* está desabilitado (DAC_CR_TENx = 0): Se o *bit* DAC_CR_TENx = 0, o mecanismo de *trigger* está desativado para o canal DAC correspondente. Neste modo, qualquer escrita de um novo valor digital no registrador DAC_DHRx causa automaticamente a transferência deste dado para o registrador DAC_DORx. A transferência ocorre um ciclo de *clock* dac_pclk após a escrita no DAC_DHRx.
2. Quando o *trigger* está habilitado (DAC_CR_TENx = 1): Se o *bit* DAC_CR_TENx = 1, o mecanismo de *trigger* está habilitado. Neste modo, escrever um valor no registrador DAC_DHRx apenas armazena o dado a ser convertido. A transferência deste dado do DAC_DHRx para o DAC_DORx não ocorre imediatamente. Em vez disso, a transferência só será iniciada quando ocorrer um evento de *trigger* configurado.
    a. Os *bits* DAC_CR_TSELx[3:0] selecionam a fonte deste evento de *trigger* entre 16 possibilidades, mostradas no Manual de Referência. Estas fontes podem ser tanto um *trigger* por *software* quanto diversos *triggers* por *hardware*. A interface do DAC detecta uma borda de subida na fonte de

disparo selecionada para iniciar a conversão. Os *bits* DAC_CR_TSELx[3:0] não podem ser alterados enquanto o *bit* DAC_CR_ENx estiver ativado para o canal correspondente

b. Se o *trigger* por *software* for selecionado (via DAC_CR_TSELx[3:0]) enquanto o *trigger* estiver habilitado (DAC_CR_TENx=1), a transferência de DAC_DHRx para DAC_DORx é iniciada definindo o *bit* DAC_SWTRGR_SWTRIGx, por exemplo, DAC_SWTRGR_SWTRIG1 para o Canal 1. Este *bit* DAC_SWTRGR_SWTRIGx é resetado por *hardware* assim que a conversão é iniciada, que ocorre após a transferência para DAC_DORx. Quando o *trigger* por *software* é selecionado e habilitado, a transferência de DAC_DHRx para DAC_DORx leva apenas um ciclo de *clock* dac_pclk.

c. Se um *trigger* por *hardware* for selecionado (via DAC_CR_TSELx[3:0]) enquanto o *trigger* estiver habilitado (DAC_CR_TENx=1), a transferência do último dado armazenado no DAC_DHRx para o DAC_DORx ocorre automaticamente quando o evento de *hardware* configurado acontece. Neste caso específico (*trigger* de *hardware* com DAC_CR_TENx=1), a transferência de DAC_DHRx para DAC_DORx leva três ciclos de *clock* dac_pclk após a ocorrência do *trigger*. O código Y da fonte de disparo a ser configurado em DAC_CR_TSELx corresponde ao número Y no nome do sinal dac_chx_trgY.

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 DAC utiliza DMA para transferir dados para o registro DAC_DHRx, um problema associado a *triggers* de *hardware* de alta frequência é o evento de subcarga (em inglês, *underrun*) do DMA. O pedido de DMA para o DAC não é enfileirado (em inglês, *queued*). Se um segundo *trigger* externo chega antes que o reconhecimento (em inglês, *acknowledgement*) para o primeiro *trigger* externo seja emitido (indicando que o DMA transferiu o dado para o DAC_DHRx), a *flag* DAC_SR_DMAUDRx é setada em '1'. Isso reporta a condição de erro de que o próximo dado não estava pronto no registrador DAC_DHRx no momento do *trigger*.

Este evento está intrinsecamente ligado ao uso de DMA. Para amenizar um novo *underrun* DMA no DAC, deve-se modificar a frequência de conversão do *trigger* DMA ou aliviar a carga de trabalho do DMA. Reduzir a frequência dos *triggers* dá mais tempo para o DMA concluir a transferência de dados antes da próxima conversão. Aliviar a carga de trabalho do DMA pode envolver otimizar outras operações de DMA concorrentes ou garantir que o DMA tenha prioridade suficiente para atender ao DAC a tempo. A *flag* `DAC_SR_DMAUDRx` é definida por *hardware* e é limpa escrevendo '1' nela.

Quando o registrador `DAC_DORm` é carregado com o conteúdo do `DAC_DHRx`, a tensão de saída analógica de 12 *bits* 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.

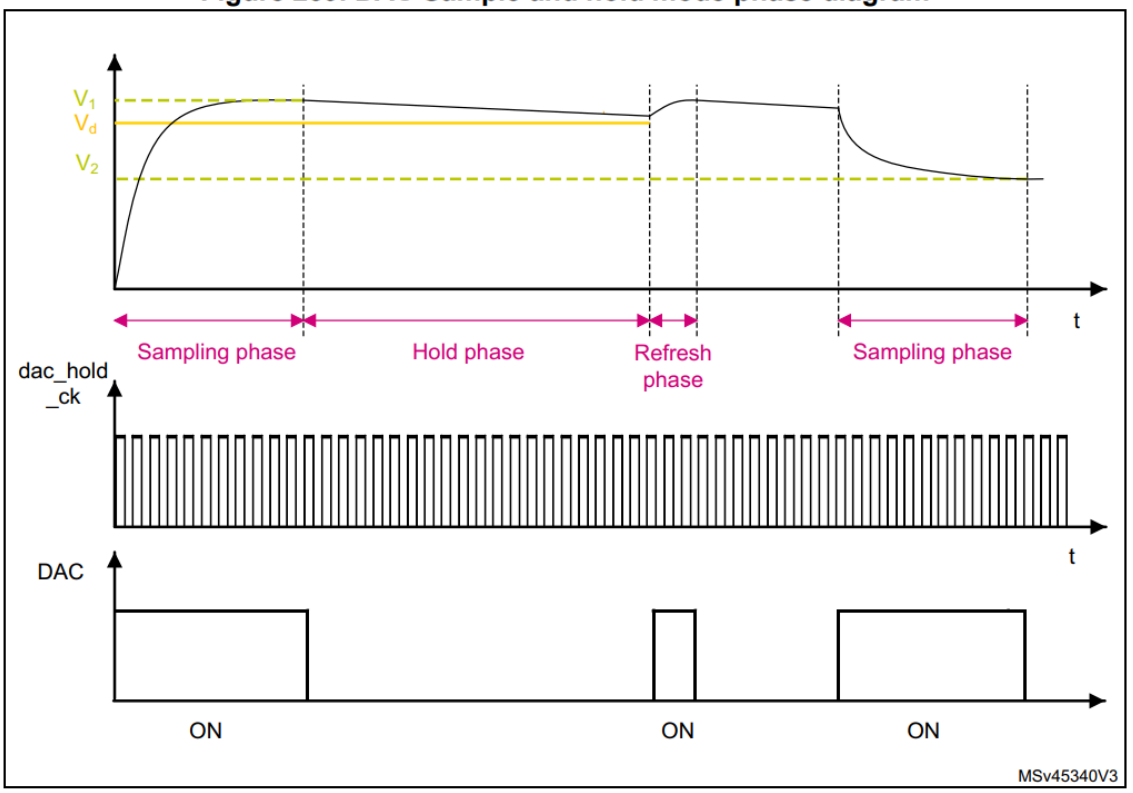
Table 222. Channel output modes summary

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.

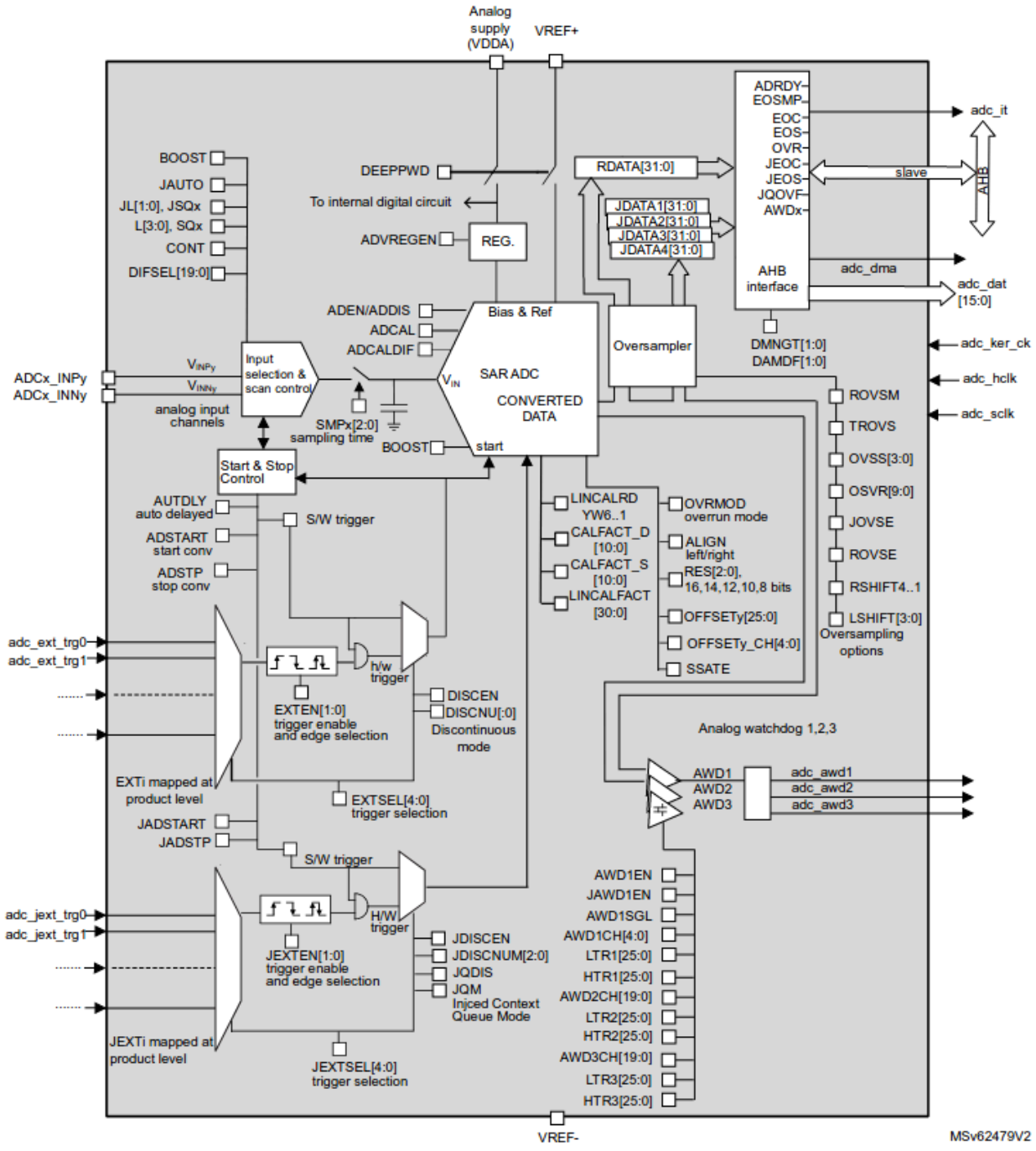
Entretanto, os dados devem ser escritos no DAC_DHRx antes do primeiro evento de disparo, pois a transferência ocorre antes do pedido de DMA. Um aspecto importante a considerar é a subcarga de DMA: como os pedidos de DMA do DAC não são enfileirados, um segundo disparo externo que chegue antes do reconhecimento 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

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. Este módulo, de 16 *bits*, incorpora a **arquitetura SAR** (do inglês *Successive Approximation Register*), sendo projetado para aplicações que demandam medições de alta precisão e elevadas taxas de dados. O diagrama de blocos detalhado, disponível no Manual de Referência, oferece uma visão abrangente da arquitetura do módulo ADC, ilustrando seus diversos blocos funcionais e suas interconexões



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.

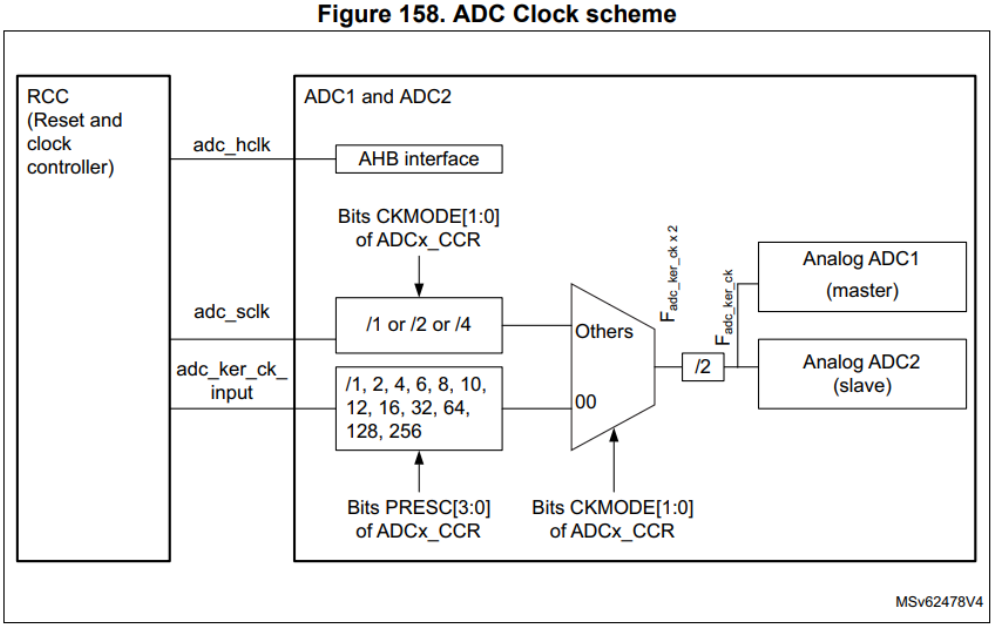
A configuração do ADC envolve diversos parâmetros, que podem ser ajustados através dos registradores do ADC.

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 Conversão: A seleção do modo de conversão é feita por meio dos *bits* ADC_CFGR_CONT e ADC_CFGR_DISCEN. O ADC do STM32H7A opera em três modos principais: **modo único** (`ADCx_CFGR_CONT` == 0 e `ADCx_CFGR_DISCEN` == 1), que realiza uma conversão a cada disparo; **modo contínuo** (`ADCx_CFGR_CONT` == 1 e `ADCx_CFGR_DISCEN` == 0), que executa conversões ininterruptamente; e **modo descontínuo**, que realiza grupos de conversões com intervalos entre eles. O multiplexador de

entrada seleciona rapidamente entre diferentes canais analógicos (VINP[i]), conforme as sequências dos canais especificadas nos registradores ADCn_SQRm e ADCn_JSQR.

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.

Pré-Seleção de Canal: Cada ADC do STM32H7A possui até 20 canais multiplexados, que podem ser selecionados através do registrador ADCn_PCSEL. Os *switches* analógicos nos pinos de I/O não são tão rápidos quanto o multiplexador de entrada do ADC que seleciona as entradas com base na sequência predefinida nos registradores ADCn_SQRm e ADCn_JSQR. Por isso, antes de o MUX selecionar um canal, o *bit* correspondente no registrador ADC_PCSEL precisa ser habilitado. Esse processo garante que o caminho analógico entre o pino de entrada e o MUX esteja estabilizado, evitando problemas de sinal causados pela lentidão nos switches analógicos dos pinos de I/O.

Os pinos correspondentes a esses canais são definidos pelo *hardware*, e ao escolher um canal, é importante garantir que o pino esteja configurado como entrada analógica no registrador GPIOx_MODER. Após identificar o pino, consulte a última coluna da Tabela 7 no *Datasheet* para encontrar o canal correspondente e, em seguida, habilite o canal desejado no ADCn_PCSEL. Por exemplo, o pino PC4 pode ser mapeado para a entrada positiva (INP) do canal 4 nos módulos ADC1 e ADC2 quando configurado em modo diferencial. Nesse caso, além de configurar o pino como entrada analógica, é necessário habilitar o *bit* 4 do ADCn_PCSEL para que o **PC4** funcione corretamente como a entrada do canal 4.

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

Superamostragem: O mecanismo de superamostragem no ADC do STM32H7A permite aumentar a precisão da conversão analógica ao realizar várias amostras de um sinal e calcular a média dessas amostras. Isso ajuda a reduzir o ruído e a melhorar a qualidade da medição. A superamostragem é configurada por meio do campo ADCn_CFGR2_OVSR para selecionar a taxa de superamostragem. O valor de ADC_CFGR2_OVSR define o número de amostras a serem realizadas antes de calcular a média (por exemplo, 1x, 2x, 4x, 8x, etc.). Em conjunto com o registrador ADCn_SMPR, que configura o tempo de amostragem de cada canal, pode-se configurar o ADC para capturar os sinais com uma precisão maior em diferentes condições de entrada. Para desabilitar a superamostragem, o *bit* de habilitação correspondente, ADCn_CFGR2_OVRSE, deve estar em 0. Se a superamostragem estiver desabilitada, efetivamente a taxa de amostragem é 1x no sentido de que cada conversão resulta em uma única amostra utilizada para o resultado.

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:

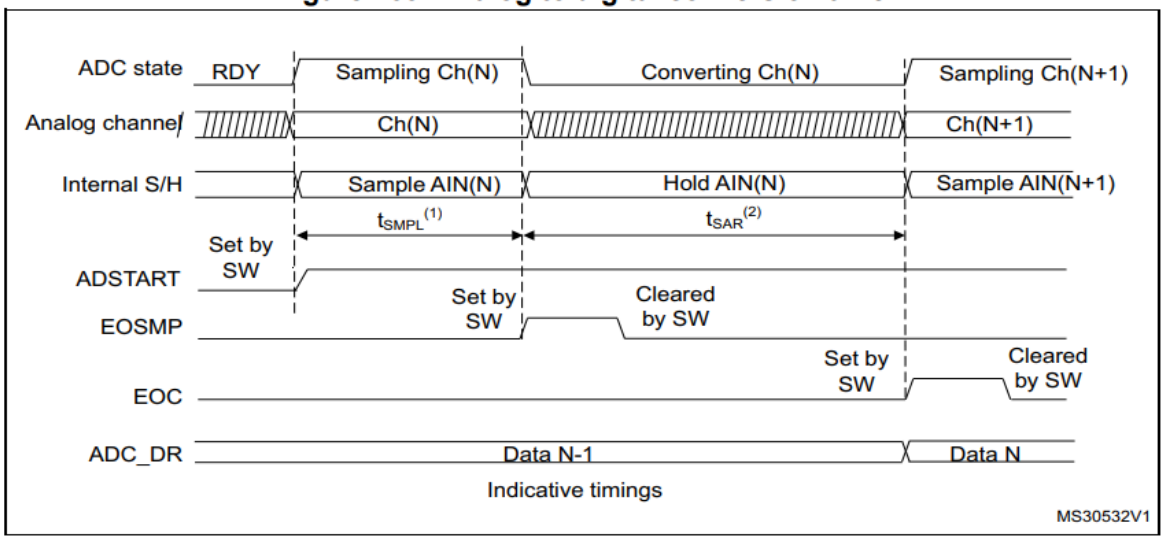
$$\text{Converted value} = \frac{\text{ADC\_Full\_Scale}}{2} \times \left[1 + \frac{V_{INP} - V_{INN}}{V_{REF+}}\right]$$

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.

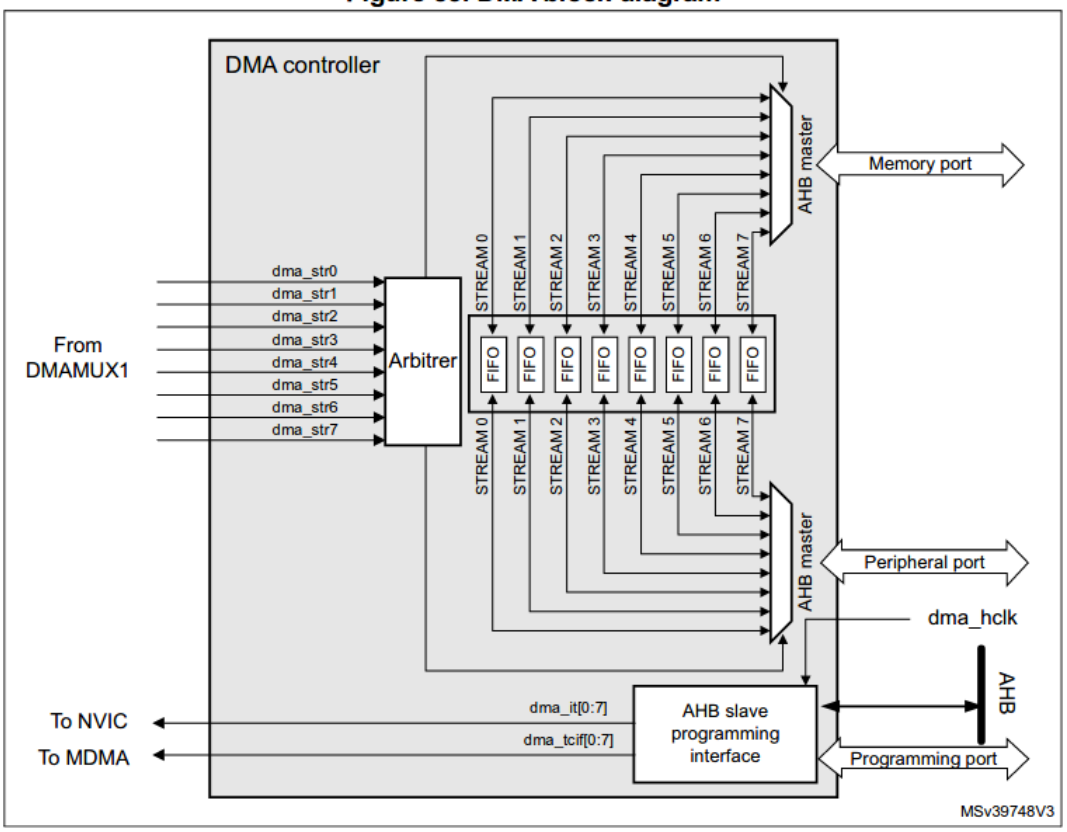
Adicionalmente, o ADC pode gerar interrupções para sinalizar eventos importantes, como a conclusão de uma conversão ou a ocorrência de um erro. Essas interrupções podem ser habilitadas e configuradas através dos bits do registrador ADCn_IER, além dos *bits* correspondentes à IRQ18 nos registradores do 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

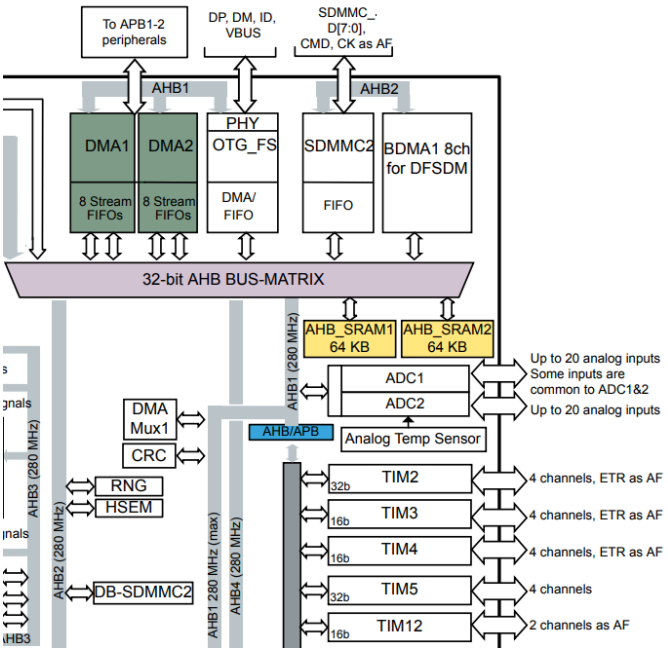
DMA

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 DMAC conta com duas portas mestres AHB, AHB *Memory Port* e AHB *Peripheral Port*, permitindo transferências simultâneas entre:

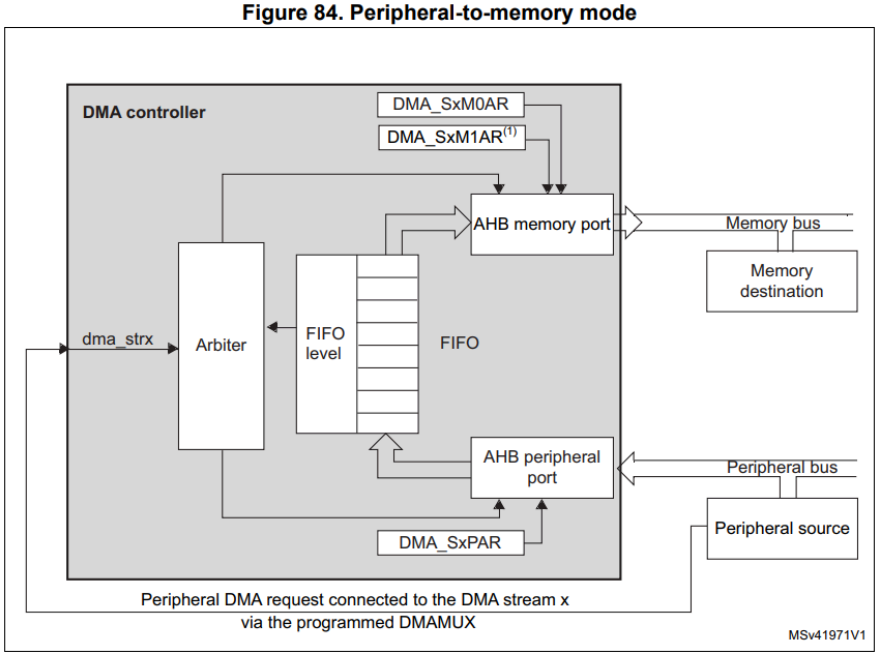
- **Periférico para Memória:** O DMAC lê dados do periférico e os grava na memória.
- **Memória para Periférico:** O DMAC lê dados da memória e os grava no periférico.
- **Memória para Memória:** O DMAC copia dados de uma área da memória para outra.

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 DMAC, 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

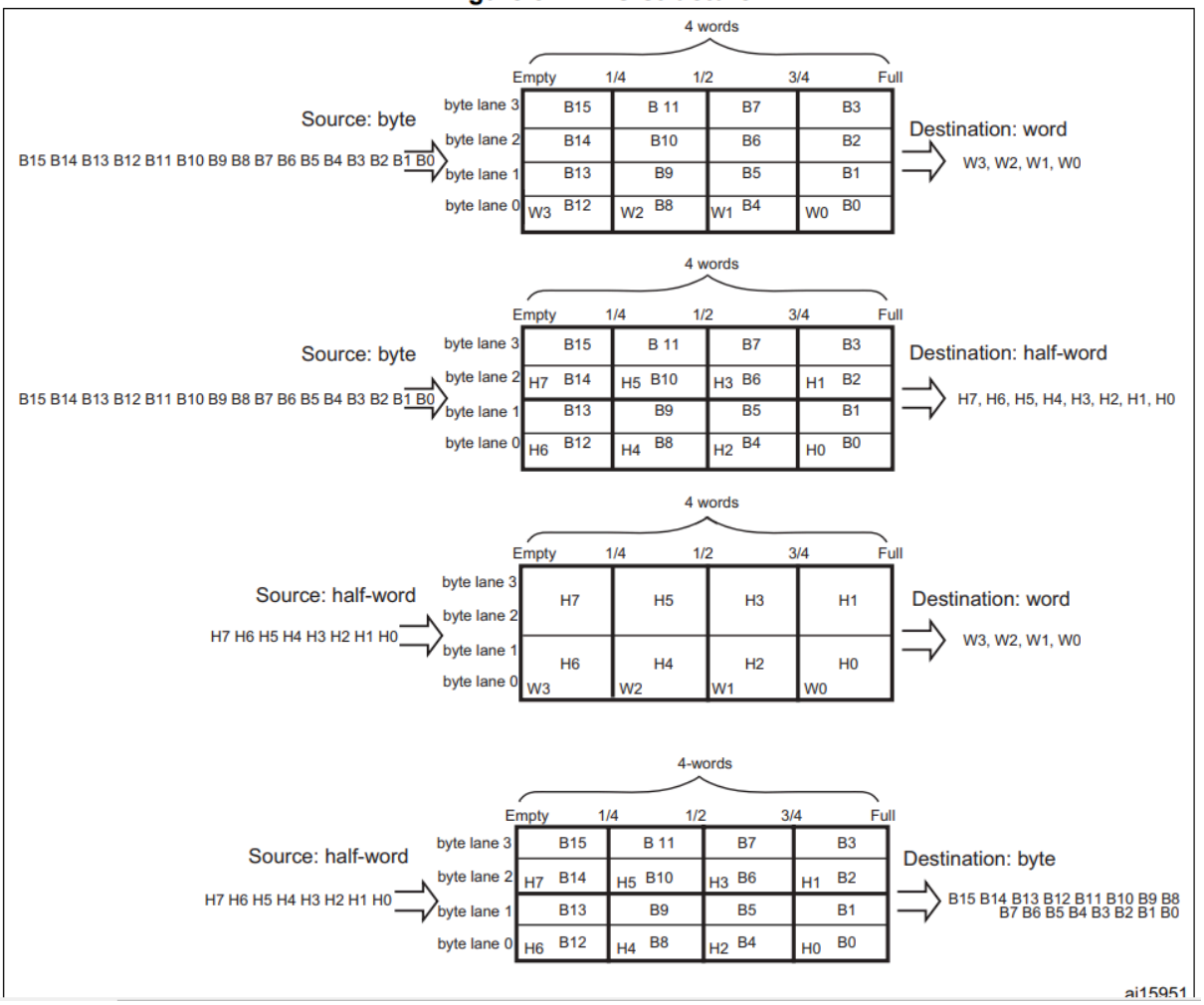


1. For double-buffer mode.

Quando há mais de um *stream* habilitado, um arbitador gerencia os pedidos de *stream* (canal) 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



As **filas**, que seguem o princípio FIFO (do inglês *First-In, First-Out*), são estruturas utilizadas para armazenar dados temporariamente, organizando-os na ordem em que são recebidos, antes de serem transmitidos 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.

DMAMUX

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. O conjunto de sinais de controle necessários para o protocolo de solicitação e reconhecimento de DMA é conhecido como **linha de solicitação de DMA** (em inglês, *DMA request line*).

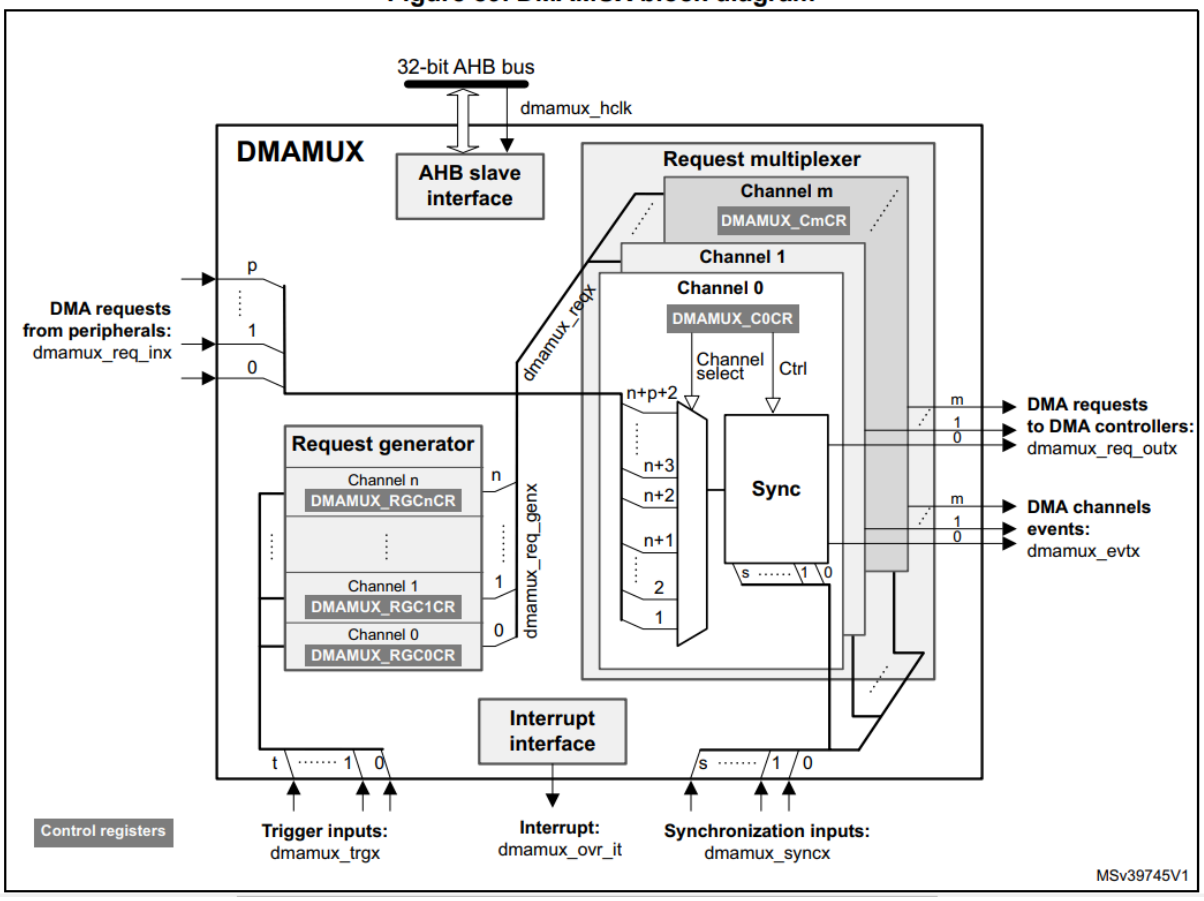
Para otimizar a comunicação entre os periféricos e o controlador de DMA, utiliza-se um multiplexador de solicitações de DMA, conhecido como DMAMUX (do inglês DMA *Request Line Multiplexer*). DMAMUX "multiplexa" a escolha entre várias fontes possíveis, permitindo que diferentes eventos possam ser conectados ao mesmo canal do DMA. Isso dá flexibilidade e reutilização do mesmo canal de DMA para múltiplas fontes de eventos.

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.

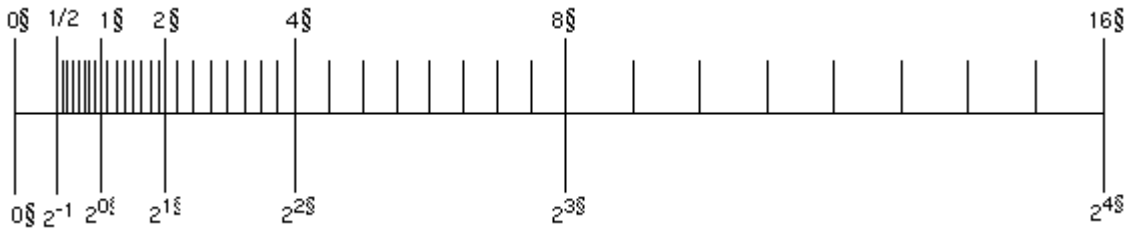
O sinal `dmamux_evtx` indica um evento gerado pelo canal x do DMAMUX. Esses eventos podem ser usados para sincronizar ou disparar ações em outros componentes do sistema. Um evento é, por exemplo, gerado quando o contador de solicitações DMA do canal, configurado pelos *bits* `DMAMUX_CxCR_NBREQ[4:0]`, 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*.

PROGRAMAÇÃO EM C: VALOR DECIMAL EM VETOR DE CARACTERES ASCII

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 ASCII (*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 (",").