DISCIPLINA EA701
Introdução aos Sistemas Embarcados

ROTEIRO 7: Comunicação Serial Assíncrona
Protocolo RS-232 e Mecanismo de Sincronização *Start-Stop*, Codificação Lógica, Codificação em Sinais Físicos, UART, FIFO, Processamento de *Strings*, USART3/UART4

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 setembro de 2024

INTRODUÇÃO **2**
PROJETOS-EXEMPLO **3**
Projeto de Interface serial assíncrona com terminal usando polling 3
Projeto de Interface serial assíncrona usando interrupções 7
Projeto de interfaces seriais assíncronas com sinais replicados 12
CLIs - Interfaces de Linha de Comando 17
FUNDAMENTOS TEÓRICOS **21**
REPRESENTAÇÃO DOS ESTADOS BINÁRIOS EM SINAIS FÍSICOS 21
CÓDIGO ASCII: DO BIT AO SIGNIFICADO 23
CÓDIGOS DETECTORES E/OU CORRETORES DE ERROS 24
TIPOS DE TRANSMISSÃO DE SINAIS 26
HANDSHAKING 29
RS-232: PROTOCOLO DE COMUNICAÇÃO SERIAL ASSÍNCRONA 30
CIRCUITOS DEDICADOS: UART e USART 32
ESTRUTURA DE DADOS: FILA 36
UMA APLICAÇÃO: TERMINAIS SERIAIS 37
STM32H7A3: USART/UART **38**
Transmissão 40
Recepção 42
Registradores de estado e interrupções 43
Sinais de relógio 46
Alocação de pinos 47
Bufferização de dados 48
PROCESSAMENTO DE STRINGS EM C **48**

INTRODUÇÃO

A comunicação serial é um dos métodos mais amplamente utilizados para interconectar sistemas ou suas partes, permitindo a transmissão de dados entre microcontroladores e dispositivos periféricos. Ao contrário da comunicação paralela, onde múltiplos *bits* são transmitidos simultaneamente, a comunicação serial envia os *bits* um de cada vez, um após o outro. Embora o envio de um *bit* por vez possa parecer mais lento, a comunicação serial oferece várias vantagens, como a redução no número de pinos e fios necessários, maior confiabilidade devido à simplicidade na detecção de erros e flexibilidade em termos de adaptação a diferentes taxas de transmissão e protocolos de comunicação.

Este roteiro explora a comunicação serial assíncrona, uma técnica que dispensa um sinal de *clock* compartilhado entre os dispositivos. Nessa modalidade, a sincronização é alcançada através da concordância prévia sobre a taxa de transmissão (*baud rate*) e o formato dos dados. Para implementar essa comunicação, utilizaremos os periféricos UART4 (do inglês *Universal Asynchronous Receiver/Transmitter*) e USART3 (do inglês *Universal Synchronous/Asynchronous Receiver/Transmitter*) do microcontrolador STM32H7A3.

Embora o USART3 ofereça a flexibilidade de operar tanto em modo síncrono (com *clock* externo) quanto assíncrono (similar ao UART4), neste contexto, o empregaremos predominantemente em modo assíncrono. A comunicação assíncrona via USART3 será realizada em modo *full-duplex*, como a comunicação via UART4, permitindo a transmissão e recepção simultânea de dados pelas linhas **TX** (transmissão) e **RX** (recepção). Para garantir a comunicação bidirecional correta entre os dispositivos, as linhas **TX** e **RX** devem ser conectadas de forma cruzada: **a linha TX de um dispositivo conectada à linha RX do outro, e vice-versa**.

PROJETOS-EXEMPLO

Para demonstrar a aplicação prática dos conceitos de comunicação serial assíncrona, apresentaremos quatro projetos específicos: (1) **Comunicação da MCU com um Terminal**, onde exploraremos como configurar e utilizar o USART3 para comunicação com um terminal de computador aplicando a técnica *polling* para gestão de sincronia entre a recepção e transmisssão; (2) **Redução do tempo de espera na comunicação**, onde otimizaremos a comunicação para minimizar o tempo de espera desnecessário aplicando a técnica de interrupção suportada pelo *hardware*; (3) **Visualização dos sinais físicos de transmissão**, que nos permitirá compreender como os dados são transmitidos e como a informação pode ser codificada e decodificada na comunicação; e (4) **Implementação de uma interface por linha de comando**, que ilustrará como criar uma interface de usuário baseada em linha de comando para interagir com a MCU de maneira eficaz.

Ao longo desses projetos, abordaremos aspectos práticos da implementação e configuração da USART3, fornecendo uma compreensão aprofundada das técnicas e considerações necessárias para uma comunicação serial assíncrona bem-sucedida.

Projeto de Interface serial assíncrona com terminal usando *polling*

Você já se perguntou como os caracteres digitados em um teclado podem ser exibidos no terminal do seu computador? Imagine a tarefa de configurar um sistema onde um terminal serial ecoe os caracteres digitados em um teclado, ambos controlados por um microcontrolador. Programar um microcontrolador para realizar essa tarefa envolve uma série de etapas importantes, desde a configuração das conexões e a programação até a resolução de problemas de comunicação e sincronização.

Neste projeto, aprenderemos a criar um sistema de “eco” utilizando a placa NUCLEO. O sistema de “eco” simplesmente repete tudo o que você digita no teclado, exibindo os caracteres no terminal do computador. Para isso, utilizaremos três ferramentas principais: (1) o ST-LINK, integrado à placa NUCLEO, responsável por programar, depurar o microcontrolador STM32H7A3 e emular um dispositivo USB CDC (do inglês *Communication Device Class*), o que faz com que o computador reconheça o canal USB como uma porta serial (COM), permitindo a comunicação entre o microcontrolador e o

terminal; (2) o terminal do STM32CubeIDE, que permite visualizar os dados enviados e recebidos pelo microcontrolador, facilitando a depuração e o monitoramento da comunicação; e (3) o módulo USART3, presente no microcontrolador STM32H7A3, responsável por receber os caracteres digitados no teclado e enviá-los de volta para o terminal do computador. O funcionamento é simples: ao digitar um caractere no teclado, ele é enviado para o microcontrolador através da porta serial virtual criada pelo ST-LINK. O módulo USART3 do microcontrolador recebe o caractere e o envia de volta para o terminal do computador, também através da porta serial virtual. O terminal do STM32CubeIDE exibe o caractere recebido, criando o efeito de "eco".

Table 12. USART3 pins

Pin name	Function	Virtual COM port (Default configuration)	ARDUINO® D0 and D1	ST morpho connection
PD8	USART3 TX	SB103 OFF, **SB16 ON** and SB15 OFF	SB103 OFF, SB16 OFF and **SB15 ON**	**SB103 ON**, SB16 OFF, SB15 OFF
PD9	USART3 RX	SB104 OFF, **SB17 ON** and SB94 OFF	SB104 OFF, SB17 OFF and **SB94 ON**	**SB104 ON**, SB17 OFF and SB94 OFF

A interface ST-LINK é, de fato, uma ferramenta de depuração e programação utilizada nas placas NUCLEO-144. Embora sua principal função seja a depuração e programação do microcontrolador, ela também pode ser configurada para atuar como uma interface de comunicação serial. Nesse caso, a ST-LINK emula uma porta COM virtual no computador por meio do driver VCP (do inglês *Virtual COM Port*), permitindo que dados enviados do terminal serial no computador sejam transmitidos para a USART3 do microcontrolador. Dessa forma, os dados digitados no terminal do computador podem ser recebidos pelo microcontrolador, possibilitando a comunicação serial assíncrona entre o computador e o microcontrolador. Neste contexto específico, a interface ST-LINK é conhecida coloquialmente como um **relé de comunicação serial**, pois, como um bloco de circuito, possui duas interfaces seriais e roteia os sinais entre elas.

Este projeto demonstra de forma prática como utilizar a comunicação serial para trocar informações entre um microcontrolador e um computador. Essa técnica é fundamental para diversas aplicações, como depuração e monitoramento de sistemas embarcados, comunicação com sensores e outros dispositivos, e criação de interfaces de usuário simples. Ao final deste projeto, você terá aprendido a configurar e utilizar o ST-LINK, o terminal do STM32CubeIDE e o módulo USART3 para implementar a comunicação serial, adquirindo habilidades valiosas para o desenvolvimento de sistemas embarcados.

1. Crie um projeto novo usando o *Cube*, com o nome "Eco_Polling", com a opção **"Initialize all peripherals with their default!" desabilitada**. Ative o módulo *Debug* como "Serial Wire".

2. Entre na aba "Clock Configuration" e modifique a frequência do *clock* do sistema para **96MHz**, seguindo o mesmo procedimento mostrado no Roteiro 6.

3.Volte para a seção "Pinout & Configuration" e verifique que, na categoria "Connectivity", o módulo "USART3" está desativado. Em vez de habilitá-lo e configurá-lo pela interface gráfica, optaremos por realizar essa configuração diretamente através de código, utilizando a interface CMSIS.

4. Veja no “Pinout View” que os pinos PD8 e PD9 já são reservados para TX e RX do módulo USART3 (preenchidos em cor amarela). Quando os pinos estão reservados, o *Cube* já os configura para a função alternativa adequada. Assim, não será necessário configurá-los para a função alternativa da USART3 em nosso código.

5. Gere o código de inicialização pelo *Cube* e abra o arquivo “main.c”. Declare no escopo /* USER CODE BEGIN 1 */ a variável local que usaremos na função

```
char c;
```

Na função `MX_GPIO_Init` gerada, consegue localizar o bloco de instruções que configura os pinos PD8 e PD9 para as funções alternativas do USART3?

6. Vamos ativar e configurar o módulo USART3 para operar com as seguintes configurações: ***baud rate* de 9600, caracteres de 8 *bits*, 1 *bit* de parada e sem paridade**. Realizaremos essa configuração no bloco de código /* USER CODE BEGIN 2 */ com as seguintes atribuições:

```
RCC->APB1LENR |= RCC_APB1LENR_USART3EN;  // Ativa o clock para USART3

 // Configura USART3
USART3->CR1 &= ~(USART_CR1_PCE_Msk |   // Desativa o controle de paridade (sem
paridade)
          USART_CR1_OVER8_Msk |    // Configura oversampling para 16x (OVER8 = 0)
          USART_CR1_M0_Msk |       // Configura tamanho de caractere (word length
= 8 bits)
          USART_CR1_M1_Msk);
USART3->BRR = 0x2710;   // Configura o baud rate para 9600 (considerando um
clock de 96 MHz)
USART3->CR2 &= ~USART_CR2_STOP_Msk; // Configura bits de parada (STOP = 1 bit)
USART3->PRESC = ~USART_PRESC_PRESCALER_Msk;   //Configurar prescaler (DIV = 1)
USART3->CR1 |= USART_CR1_UE; // Habilita o USART3

USART3->CR1  |=  USART_CR1_TE  |  USART_CR1_RE; // Habilita o transmissor e o
receptor
while (!(USART3->ISR & USART_ISR_REACK)); //aguarda a efetivacao de RX
while (!(USART3->ISR & USART_ISR_TEACK)); //aguarda a efetivacao de TX
```

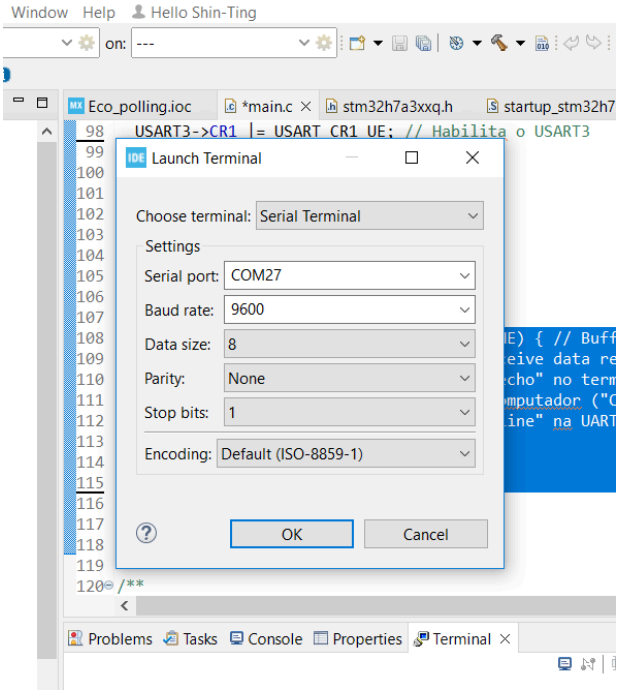
Observe que foi seguida a ordem de configuração de transmissão e de recepção descritas no Manual de Referência. No STM32H7A3, o registrador `USART3_BRR`, configurado com o valor 0x2710, atua como um divisor de frequência, resultando em uma taxa de transmissão de 9600 *bauds*. Essa relação é dada por: 9600 *bauds* = 96.000.000 / `USART3_BRR`.

7. Adicione no escopo /* USER CODE BEGIN 3 */ instruções de ler o valor correspondente ao caractere digitado no registrador de recepção `USART3->RDR` e transferí-lo para o registrador de transmissão `USART3->TDR`, se o *bit* de estado de recepção `USART_ISR_RXNE` estiver em “1”.

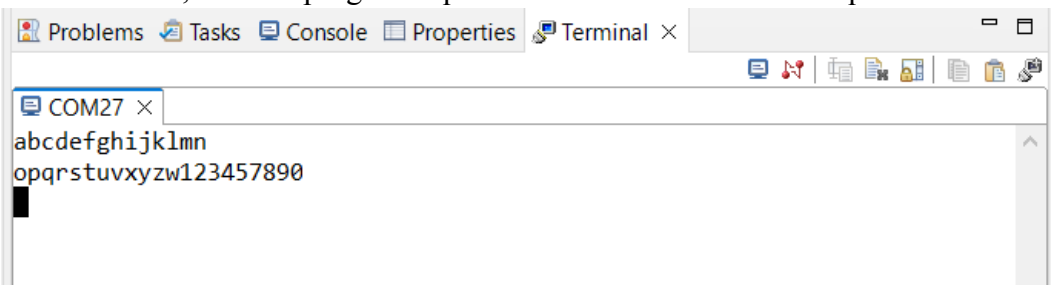
```
if(USART3->ISR & USART_ISR_RXNE_RXFNE) { // Buffer nao esta vazio
  c = (uint8_t)USART3->RDR; // Receive data register
  while (!(USART3->ISR & USART_ISR_TXE_TXFNF)){} // aguarda TX vazio
  USART3->TDR = c;
  if(c == 0x0D) {    "Enter" no computador ("Carriage Return")
    while (!(USART3->ISR & USART_ISR_TXE_TXFNF)){}
    USART3->TDR = 0x0A;    // Adiciona "New Line" na UART
  }
}
```

8. Construa (“Build”) o código executável e transfira o código para o microcontrolador no modo *Debug*.

9. Para garantir uma comunicação adequada com o Terminal integrado no STM32CubeIDE, os parâmetros de comunicação serial do Terminal devem coincidir exatamente com os configurados para o módulo USART3. Vá até “Window > Show View > Terminal” e ative o “view” do Terminal. O ícone do Terminal aparecerá na interface. Abra o Terminal clicando em “Open a Terminal**”** e configure os parâmetros de comunicação para corresponder aos valores definidos para o módulo USART3. Quando houver várias portas seriais ativadas, consulte o Gerenciador de Dispositivos para identificar a porta à qual o “STMicroelectronics ST-Link Virtual COM Port” está conectado.



10. Continue (“Resume”) a execução e digite no Terminal uma sequência de caracteres que serão mostrados no Terminal. Note que o terminal não “ecoa” automaticamente os caracteres digitados no mesmo, mas é o programa que devolve ao terminal tudo o que recebe do mesmo.



11. Vamos analisar agora o fluxo de controle implementado. Pause o programa e **comente** as seguintes instruções. Termine e execute o código atualizado (“Terminate and Relaunch”).

```
//while (!(USART3->ISR & USART_ISR_TXE_TXFNF)){} // aguarda TX vazio
//USART3->TDR = c;
//if(c == 0x0D) {     "Enter" no computador ("Carriage Return")
//while (!(USART3->ISR & USART_ISR_TXE_TXFNF)){}
//USART3->TDR = 0x0A;     // Adiciona "New Line" na UART
```

```
//}
```

12. Continue ("Resume") a execução e digite novamente a mesma sequência de caracteres no Terminal. Observe o que acontece. Quais foram as alterações no comportamento do sistema e o que você acredita que causou essas mudanças?

13. Pause e restaure a versão original do programa, **descomentando** as instruções. Regere (“Terminate and Relaunch”) o código executável. Execute o programa para certificar que foi restaurado o comportamento do programa.

14. Pause e faça as seguintes alterações: modifique o valor do *baud rate* para 19200 e altere os *bits* `USART3_CR2_STOP` para 2 *stop bits*. Qual é o valor configurado no registrador `USART3_BRR`? Após cada alteração, retome a execução clicando em "Resume" e digite alguns caracteres no Terminal. Observe o comportamento resultante e tente fornecer uma explicação para as mudanças observadas.

15. Repita o procedimento anterior, mas desta vez, ajuste simultaneamente os parâmetros de comunicação do Terminal para que correspondam aos novos valores configurados. Observe o comportamento resultante e forneça uma explicação para as mudanças observadas.

Projeto de Interface serial assíncrona usando interrupções

O *polling*, método de controle de fluxo em comunicação serial assíncrona, apresenta desvantagens significativas. Nessa técnica, o processador verifica continuamente o estado do periférico, aguardando condições específicas como o receptor cheio (exemplificado por `if(USART3->ISR & USART_ISR_RXNE_RXFNE) {}`) ou o transmissor vazio (`while (!(USART3->ISR & USART_ISR_TXE_TXFNF)){}`, ambos do projeto anterior). Essa abordagem resulta em:

- **ineficiência:** O processador desperdiça ciclos em verificações constantes, impedindo a execução de outras tarefas.
- **baixa responsividade:** A ocupação do processador com polling atrasa respostas a outras demandas do sistema.
- **alto consumo de energia:** Especialmente em sistemas embarcados, o polling constante aumenta o consumo.
- **complexidade e dificuldade de manutenção:** O código se torna complexo, especialmente com múltiplas condições e fluxos de controle.

Vimos no Roteiro 3 que uma alternativa ao *polling* é o uso de interrupções. Você consegue imaginar como um módulo USART pode ser configurado para gerar uma interrupção quando um evento específico ocorre, como a chegada de um caractere no registrador de recepção ou quando o registrador de transmissão fica vazio? Desafios surgem ao fazer essa transição. Como você controlaria o fluxo de execução do sistema para evitar que interrupções mais críticas sejam retardadas por outras menos urgentes? E como garantir que as transferências de dados sejam coordenadas com o fluxo de execução, mesmo com a natureza assíncrona dos eventos de interrupção? Vamos explorar como os módulos USART podem facilitar a superação desses desafios na configuração e no gerenciamento das interrupções em

transmissões assíncronas de dados, particularmente na comunicação entre um microcontrolador e o terminal serial integrado no *Cube*.

1. Crie um novo projeto chamado "Eco_Int", seguindo os passos 1 a 4 do projeto anterior, exceto pelo passo 2 em que definimos a frequência do relógio do sistema em **64MHz**.

2. Para garantir que o *baud rate* seja mantido em 9600 com um *prescaler* de 1, devemos configurar o registrador USART3->BRR com o valor 64.000.000/9600=6666,67 arredondando para o próximo valor inteiro superior, que é 6667. Em hexadecimal, isso corresponde a 0x1A0B. Assim, configuraremos o módulo USART3 com esse valor para que opere com um ***baud rate* de 9600**, **caracteres de 8 *bits***, **2 *bits* de parada** e ***bit* de paridade par**. Além disso, será necessário definir a máscara da interrupção "RXNE" (*Receiver Buffer Not Empty*), permitindo o recebimento de caracteres no canal RX. Para realizar essa configuração, insira no escopo /* USER CODE BEGIN 2 */ o seguinte bloco de códigos:

```
	RCC->APB1LENR |= RCC_APB1LENR_USART3EN;  // Ativa o clock para USART3
	// Configura USART3
	USART3->CR1 &= ~(USART_CR1_OVER8_Msk |  // Config. oversampling para 16x
(OVER8 = 0)
                 USART_CR1_M1_Msk);
	USART3->CR1 |= USART_CR1_M0_Msk; // Configura tamanho de caractere (word
length = 9 bits)
	USART3->BRR = 0x1A0B;  // Configura o baud rate para 9600 (considerando um
clock de 64 MHz)
	USART3->PRESC &= ~USART_PRESC_PRESCALER_Msk; //Configure prescaler (divisor
= 1)
	USART3->CR2 &= ~USART_CR2_STOP_Msk; // Configura bits de parada (STOP = 2
bits)
	USART3->CR2 |= USART_CR2_STOP_1;
	USART3->CR1 |= USART_CR1_PCE_Msk;  // Ativa o controle de paridade (com
paridade)
	USART3->CR1 &= ~USART_CR1_PS_Msk;  // Paridade Par
	USART3->CR1 |= USART_CR1_UE; // Habilita o USART3
	USART3->CR1 |= (USART_CR1_TE |   // Habilita o transmissor
                USART_CR1_RE |  // Habilita o receptor
                USART_CR1_RXNEIE_RXFNEIE);  //Habilita a interrupcao de RX
	while (!(USART3->ISR & USART_ISR_REACK)); //aguarda a efetivacao de RX
	while (!(USART3->ISR & USART_ISR_TEACK)); //aguarda a efetivacao de TX
```

Adicionalmente, deve-se habilitar a linha de solicitação de interrupção IRQ39, à qual o periférico USART3 está associado, e definir o nível de prioridade da interrupção. A macro correspondente a IRQ39 está definida no arquivo `Drivers\CMSIS\Device\ST\STM32H7xx\Include\stm32h7a3xxq.h`. Para este exemplo, vamos definir o nível de prioridade como 2. No escopo /* USER CODE BEGIN 2 */, insira as seguintes linhas de código:

```
// Configura NVIC
NVIC_SetPriority(USART3_IRQn, 2); // Define a prioridade da interrupção
NVIC_EnableIRQ(USART3_IRQn); // Habilita a interrupção USART3_IRQn
```

3. Agora, vamos implementar a ISR para a interrupção do USART3. Para isso, abra o arquivo "stm32h7xx_it.c" e crie a ISR do zero. Como não configuramos as interrupções no STM32CubeMX, não temos um *handler* previamente definido. Para descobrir o nome da ISR que deve ser utilizado, acesse a pasta "Core – Startup" e abra o arquivo

“startup_stm32h7a3zitxq.s”. Na seção “g_pfnVectors”, localize o vetor correspondente ao USART3 para identificar o nome da ISR a ser utilizada.

De volta ao arquivo “stm32h7xx_it.c” vá até o escopo `/* USER CODE BEGIN 1 */` e insira a implementação da ISR conforme o nome identificado.

```
void USART3_IRQHandler(void) {
    static uint8_t c;
    if ((USART3->ISR & USART_ISR_RXNE_RXFNE)) { //Verifica se há dados recebidos
      c = USART3->RDR; // Lê o caractere recebido
      USART3->CR1 |= USART_CR1_TXEIE_TXFNFIE_Msk; // Habilita interrupcao de TX
    } else if (USART3->ISR & USART_ISR_TXE_TXFNF) {
      if(c != 0x0D) { // "Enter" no computador ("Carriage Return")
       USART3->CR1 &= ~USART_CR1_TXEIE_TXFNFIE_Msk;//Desabilita interrupcao TX
      }
      USART3->TDR = c;
      if (c == 0x0D) {
           c = 0x0A; // Adiciona "New Line" na UART
      }
    }
}
```

É declarada uma variável local `c`, mas estática, o que significa que seu valor é preservado entre as chamadas da ISR. Esta variável é usada para armazenar o caractere a ser transmitido.

O primeiro bloco “`if`” processa os eventos de **interrupção de recepção**. O código checa a *flag* de recepção `USART3_ISR_RXNE`, usando a expressão “`USART3->ISR & USART3_ISR_RXNE_RXFNE`”. Isso indica se há um caractere disponível para leitura. O caractere recebido é lido do registrador de dados de recepção (`USART3->RDR`) e armazenado na variável `c`. O *hardware* limpa automaticamente o *bit* `USART3_ISR_RXNE` na leitura. A interrupção de transmissão é então habilitada com “`USART3->CR1 |= USART_CR1_TXEIE_TXFNFIE_Msk`”, permitindo que a ISR também trate a transmissão quando o registrador `USART3->TDR` estiver vazio.

O segundo bloco “`else if`” lida com os eventos de **interrupção de transmissão**. O código utiliza a expressão “`USART3->ISR & USART_ISR_TXE_TXFNF`” para verificar se a *flag* de transmissão `USART_ISR_TXE_TXFNF` está setada em ‘1’, indicando que o registrador `USART->TDR` está pronto para receber um novo caractere. Se o caractere armazenado não for o código ASCII de “*Carriage Return*” (0x0D=‘\r’), a interrupção de transmissão `USART_CR1_TXEIE` é desabilitada. O caractere armazenado é então transmitido através do registrador de dados de transmissão `USART->TDR`. Se o caractere recebido for um 0x0D, um caractere de controle adicional, “*New Line*” (0x0A= ‘\n’), é transmitido para garantir que a transmissão inclua uma quebra de linha, evitando sobreposições de linhas na saída.

4. Realize o *Build* e transfira o código executável para o microcontrolador no modo *Debug*. Configure o Terminal Serial com os seguintes parâmetros: ***baud rate* de 9600**, **8 *bits* de dados**, **2 *bits* de parada** e **paridade par**.

5. Continue (“Resume”) a execução do programa. Digite caracteres **(apenas do código ASCII de 7 *bits*)** no terminal e observe o comportamento do programa em comparação com o projeto anterior.

6. Vamos analisar o fluxo de controle implementado. Primeiro, pause o programa e inverta a ordem das instruções de configuração, ajustando o valor da taxa de transmissão (*baud rate*) antes de definir o tamanho da palavra, não seguindo a recomendação do fabricante. Em seguida, gere novamente o código executável (“Terminate and Relaunch”) e retome a execução do programa (“Resume”). Após isso, digite alguns caracteres no Terminal e observe o comportamento resultante. Como você explica o efeito causado pela troca na ordem das instruções de configuração?

```
//USART3->CR1 &= ~(USART_CR1_OVER8_Msk |
//                 USART_CR1_M1_Msk);
//USART3->CR1 |= USART_CR1_M0_Msk;
USART3->BRR = 0x1A0B;
USART3->CR1 &= ~(USART_CR1_OVER8_Msk |
                 USART_CR1_M1_Msk);
USART3->CR1 |= USART_CR1_M0_Msk;
USART3->PRESC &= ~USART_PRESC_PRESCALER_Msk; //Configure prescaler (divisor = 1)
```

7. Pause o programa e comente as duas linhas da ISR. Em seguida, regere o código executável ("Terminate and Relaunch"). Execute o programa novamente e observe o comportamento. Como você explica o que foi observado?

```
// USART3->CR1 |= USART_CR1_TXEIE_TXFNFIE_Msk;
} else if (USART3->ISR & USART_ISR_TXE_TXFNF) {
    if(c != 0x0D) { // "Enter" no computador ("Carriage Return")
//USART3->CR1 &= ~USART_CR1_TXEIE_TXFNFIE_Msk;
```

6. Pause o programa novamente e descomente a primeira linha da ISR que foi comentada. Em seguida, regere o código executável (“Terminate and Relaunch”). Execute o programa e observe o comportamento.

```
            USART3->CR1 |= USART_CR1_TXEIE_TXFNFIE_Msk;
      } else if (USART3->ISR & USART_ISR_TXE_TXFNF) {
          if(c != 0x0D) { // "Enter" no computador ("Carriage Return")
//            USART3->CR1 &= ~USART_CR1_TXEIE_TXFNFIE_Msk;
```

Você consegue explicar por quê é necessário habilitar e desabilitar a interrupção associada ao evento de transmissão (TXE) em vez de simplesmente habilitá-la uma única vez, como foi feito com a interrupção de recepção (`USART_CR1_RXNEIE_RXFNEIE`)?

7. Pause o programa novamente, descomente a linha da ISR que foi comentada e comente as linhas que transmitem o caractere de controle “\n” para o terminal. Em seguida, regere o código executável (“Terminate and Relaunch”). Execute o programa e observe o comportamento. Explique a função da transmissão do caractere “\n” adicional para o terminal e como isso afeta a exibição dos dados.

```
      } else if (USART3->ISR & USART_ISR_TXE_TXFNF) {
//          if(c != 0x0D) { // "Enter" no computador ("Carriage Return")
            USART3->CR1 &= ~USART_CR1_TXEIE_TXFNFIE_Msk;
//          }
            USART3->TDR = c;
//          if (c == 0x0D) {
//                c = 0x0A; // Adiciona "New Line" na UART
//          }
      }
```

8. Pause novamente o programa e descomente as linhas que transmitem o caractere de controle “\n” para o terminal, e inverte a ordem de tratamento dos eventos de interrupção por

recepção `RXNEIE` e por transmissão `TXEIE`. Em seguida, regere o código executável (“Terminate and Relaunch”). Execute o programa e observe o comportamento. Explique como a ordem de tratamento dos eventos afeta a exibição dos dados.

```
if (USART3->ISR & USART_ISR_TXE_TXFNF) {
	if(c != 0x0D) { // "Enter" no computador ("Carriage Return")
		USART3->CR1 &= ~USART_CR1_TXEIE_TXFNFIE_Msk;
	}
	USART3->TDR = c;
	if (c == 0x0D) {
		c = 0x0A; // Adiciona "New Line" na UART
	}
} else if ((USART3->ISR & USART_ISR_RXNE_RXFNE)) {
	c = USART3->RDR; // Lê o caractere recebido
	USART3->CR1 |= USART_CR1_TXEIE_TXFNFIE_Msk;
}
```

9. Restaure a versão original do progrma, regere o código executável e verifique se retornou ao comportamento origina.

10. É possível que você tenha observado que, em certas situações, um caractere digitado é replicado no eco mesmo habilitando e desabilitando interrupções de transmissão. Para solucionar essa ocorrência e otimizar a sequência de execução entre a recepção e a transmissão de dados, implementaremos uma *flag* adicional na rotina de serviço de interrupção (ISR). Para corrigir a duplicação do caractere ecoado, interrompa a execução do programa, substitua as instruções atuais da rotina de serviço de interrupção (ISR) pelo conjunto de instruções revisado abaixo, recompile o código executável ("Terminate and Relaunch") e, em seguida, retome a execução do programa ("Resume").

```
static uint8_t c;
static uint8_t flag=0; //RX
if ((USART3->ISR & USART_ISR_RXNE_RXFNE) && !flag) {
	c = USART3->RDR; // Lê o caractere recebido
	flag = 1;
	USART3->CR1 |= USART_CR1_TXEIE_TXFNFIE_Msk;
} else if (USART3->ISR & USART_ISR_TXE_TXFNF && flag) {
	if(c != 0x0D) { // "Enter" no computador ("Carriage Return")
		USART3->CR1 &= ~USART_CR1_TXEIE_TXFNFIE_Msk;
		flag = 0;
	}
	USART3->TDR = c;
	if (c == 0x0D) {
		c = 0x0A; // Adiciona "New Line" na UART
	}
}
```

Reflita: qual é o impacto da *flag* no fluxo de dados da comunicação serial? Como essa variável de controle transforma o sequenciamento das operações de recepção e transmissão, promovendo maior eficiência? Não se preocupe se a resposta não for imediata; discutiremos este conceito em breve..

11. Com base em sua exploração, esboce algumas diretrizes essenciais para o controle do fluxo de execução, considerando a necessidade de coordenar as transferências de dados com o fluxo do programa, mesmo diante da natureza assíncrona dos eventos de interrupção. É muito importante que consideremos como a ocorrência de interrupções pode impactar a execução e a integridade dos dados. Devemos garantir que as transferências de dados sejam corretamente sincronizadas e que o fluxo de execução do programa permaneça estável, apesar

das interrupções assíncronas. Esta análise meticulosa ajudará a evitar problemas de sincronização e garantir uma operação confiável e eficiente do sistema.

Projeto de interfaces seriais assíncronas com sinais replicados

Já imaginou como os caracteres digitados no teclado são encapsulados, codificados, em níveis de tensão e transições dos sinais, e transmitidos via uma única linha de comunicação? Como os conceitos teóricos, como taxa de *baud* (em inglês *baud rate*), níveis de tensão e temporização dos sinais envolvidos em um protocolo de comunicação, se manifestam na prática? Nos projetos anteriores, a análise desses sinais poderia ser conduzida pelo acesso direto aos pinos PD8 (TX) e PD9 (RX), bastando conectar neles as pontas de prova de um analisador lógico. No entanto, o NUCLEO-144 nos apresenta um desafio: esses pinos não estão acessíveis através dos conectores Zio.

Diante dessa limitação, surge a pergunta: como podemos contornar essa barreira e visualizar os sinais transmitidos? Existem alternativas para desvendar os segredos da comunicação serial assíncrona? Este projeto não apenas responde a essas perguntas, mas também oferece uma experiência de aprendizado enriquecedora. Você aprenderá a utilizar o módulo UART4, cujo canal TX é acessível através do pino PA0 do conector Zio CN10 Zio, para gerar os sinais seriais de um mesmo caractere digitado no teclado e visualizá-los através de um analisador lógico.

1. Crie um projeto novo usando o *Cube*, com o nome “Serial_View”, com a opção **“Initialize all peripherals with their default!” desabilitada**. Ative o módulo *Debug* como “Serial Wire”. Desta vez vamos configurar o *clock* em **96MHz** no painel “Clock Configuration”.

2. Nos dois projetos anteriores, os pinos PD8 e PD9 estão reservados para a USART3. Porém, podemos ver no painel “Pinout & Configuration” que os pinos PA0 e PA1 não estão previamente reservados para a UART4. Poderíamos reservar as funções neste painel, porém vamos fazer isto em nosso código, configurando a função alternativa 8 (UART4_TX) para o pino PA0. Gere o código e abra o arquivo “main,c”. No escopo `/* USER CODE BEGIN 2 */`, escreva o código:

```
// Habilitar o clock para o GPIOA
RCC->AHB4ENR |= RCC_AHB4ENR_GPIOAEN;
// Configurar o pino correspondente ao UART4_TX (PA0)
GPIOA->MODER &= ~(GPIO_MODER_MODE0_Msk); // Limpar modos
GPIOA->MODER |= GPIO_MODER_MODE0_1; // Modo alternativo
// Selecionar a função alternativa para UART4_TX (AF8)
GPIOA->AFR[0] &= ~(GPIO_AFRL_AFSEL0); // AF8 para PA0
GPIOA->AFR[0] |= (GPIO_AFRL_AFSEL0_3); // AF8 para PA0
```

3. Vamos usar a mesma configuração de USART3 do primeiro projeto (sem paridade, 1 *stop bit*), com o BRR ajustado para o *clock* de 96MHz), incluindo as interrupções. Vamos adicionar a configuração de UART4 e suas interrupções. A forma de configurar a UART4 é exatamente a mesma da USART3. No escopo `/* USER CODE BEGIN 2 */`, adicione o código ao final das instruções já existentes:

```
RCC->APB1LENR |= RCC_APB1LENR_USART3EN | RCC_APB1LENR_UART4EN; // Ativa o clock para USART3 e UART4

// Config USART3
USART3->CR1 &= ~(USART_CR1_OVER8_Msk |  // Config. 16x (OVER8 = 0)
      USART_CR1_M1_Msk | USART_CR1_M0_Msk);
USART3->BRR = 0x2710;  // Configura o baud rate para 9600 (considerando um clock de 96 MHz)
USART3->PRESC &= ~USART_PRESC_PRESCALER_Msk; //Configure prescaler (divisor = 1)
USART3->CR2 &= ~(USART_CR2_STOP_Msk | USART_CR2_STOP_1); // Configura bits de parada (STOP = 1 bit)
USART3->CR1 &= ~USART_CR1_PCE_Msk;  // Destiva o controle de paridade (com paridade)
USART3->CR1 |= (USART_CR1_TE |   // Habilita o transmissor
            USART_CR1_RE |  // Habilita o receptor
            USART_CR1_RXNEIE_RXFNEIE);  //Habilita a interrupcao de RX
USART3->CR1 |= USART_CR1_UE; // Habilita o USART3

while (!(USART3->ISR & USART_ISR_REACK)); //aguarda a efetivacao de RX
while (!(USART3->ISR & USART_ISR_TEACK)); //aguarda a efetivacao de TX

// Config UART4
UART4->CR1  &=  ~(USART_CR1_OVER8_Msk  |   // Config. oversampling para 16x (OVER8 = 0)
                  USART_CR1_M1_Msk | USART_CR1_M0_Msk);
UART4->BRR = 0x2710;  // Configura o baud rate para 9600
UART4->PRESC &= ~USART_PRESC_PRESCALER_Msk; //Configure prescaler (divisor = 1)
UART4->CR2 &= ~(USART_CR2_STOP_Msk | USART_CR2_STOP_1); // Configura bits de parada (STOP = 1 bit)
UART4->CR1 &= ~USART_CR1_PCE_Msk;  // Desativa contr. de paridade UART4->CR1 |= USART_CR1_TE;   // Habilita o transmissor
UART4->CR1 |= USART_CR1_UE; // Habilita o UART4

while (!(UART4->ISR & USART_ISR_TEACK)); //aguarda a efetivacao de TX
```

4. Vamos configurar o NVIC colocando prioridade 2 na USART3, como no primeiro projeto::

```
// Configura NVIC
  NVIC_SetPriority(USART3_IRQn, 2); // Define a prioridade da interrupção
  NVIC_EnableIRQ(USART3_IRQn); // Habilita a interrupção USART3_IRQn
```

5. Agora vamos implementar as ISRs. Vamos copiar a ISR da USART3 do projeto anterior e modificá-la para usar a verificação de *flag* de *buffer* de transmissão disponível em vez da interrupção de transmissão, e adicionar o envio do caractere recebido à UART4, Na área `/* USER CODE BEGIN 1 */` do arquivo “stm32h7xx_it.c”, escreva a definição das duas ISRs:

```
void USART3_IRQHandler(void) {
  uint8_t c;
 if ((USART3->ISR & USART_ISR_RXNE_RXFNE)) { // há dados recebidos?
       c = USART3->RDR; // Lê o caractere recebido
       while(!(USART3->ISR & USART_ISR_TXE_TXFNF)){} // Espera Tx livre
       USART3->TDR = c; // Eco
       while(!(UART4->ISR & USART_ISR_TXE_TXFNF)){} // Espera Tx livre
       UART4->TDR = c; // Rele para UART4
       if (c == 0x0D) {
             c = 0x0A; // Adiciona "New Line" na UART
             while(!(USART3->ISR & USART_ISR_TXE_TXFNF)){}// Espera Tx livre
             USART3->TDR = c; // Eco
```

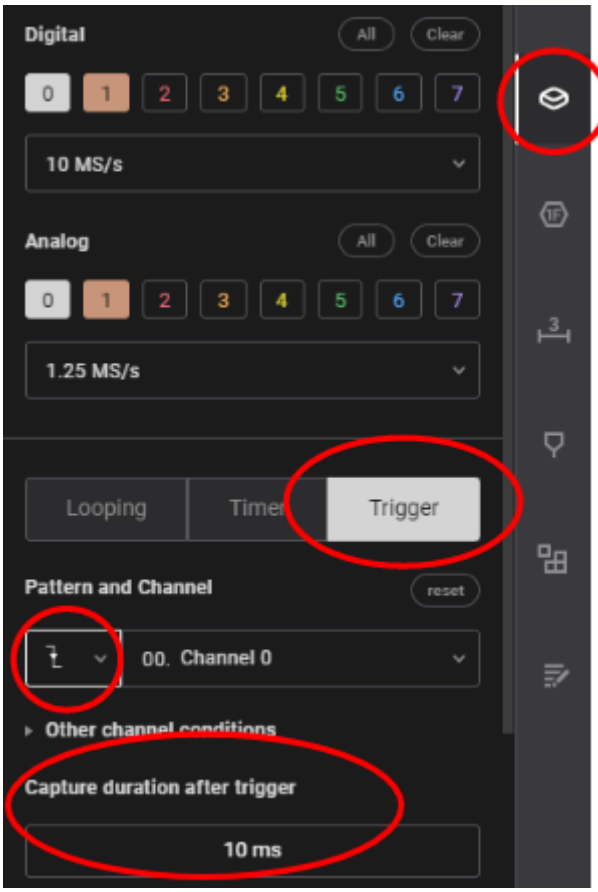
```
            while(!(UART4->ISR & USART_ISR_TXE_TXFNF)){}// Espera Tx livre
            UART4->TDR = c; // Rele para UART4
        }
    }
}
```

Note que na ISR, o caractere recebido é retransmitido por *polling*, sem o uso da interrupção de transmissão. Aqui, testa-se o *flag* "TXE" (*Transmitter Buffer Empty*).

6. Construa ("Build") o código executável e transfira-o para o microcontrolador no modo *Debug*. Na perspectiva *Debug*, abra o terminal serial do IDE e configure-o com a taxa de transmissão de 9600 *bauds*, 8 *bits* de dados, sem paridade e 1 *bit* de parada. Continue ("Resume") a execução do programa e experimente enviar caracteres digitados no teclado para o terminal dentro da perspectiva de *Debug*.

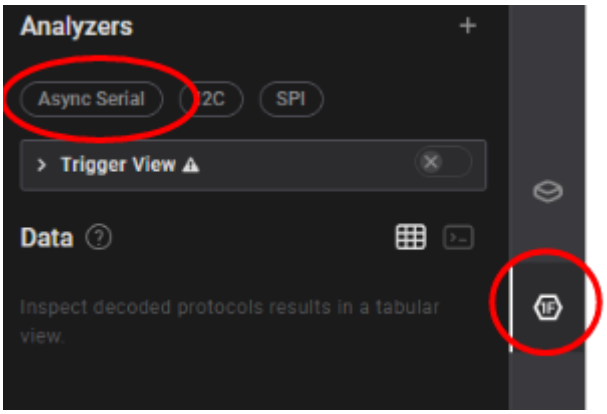
7. Vamos explorar melhor os sinais gerados pelo módulo USART3/UART4. Pegue o *jumper* e conecte o pino no terminal 29 de CN10 (PA0 - Tx de UART4). Conecte o canal 0 do analisador lógico à outra ponta do *jumper*, e o terra do analisador ao pino mais à direita do conector H11 (como foi feito no roteiro anterior).

8. No aplicativo Logic, na barra à direita, selecione o primeiro ícone (*Devices*). Selecione a opção *Trigger* e logo abaixo mantenha o *Channel 0* mas selecione a borda de descida. Por fim, configure o *Capture duration after trigger* para 10ms. A figura abaixo mostra esta configuração:

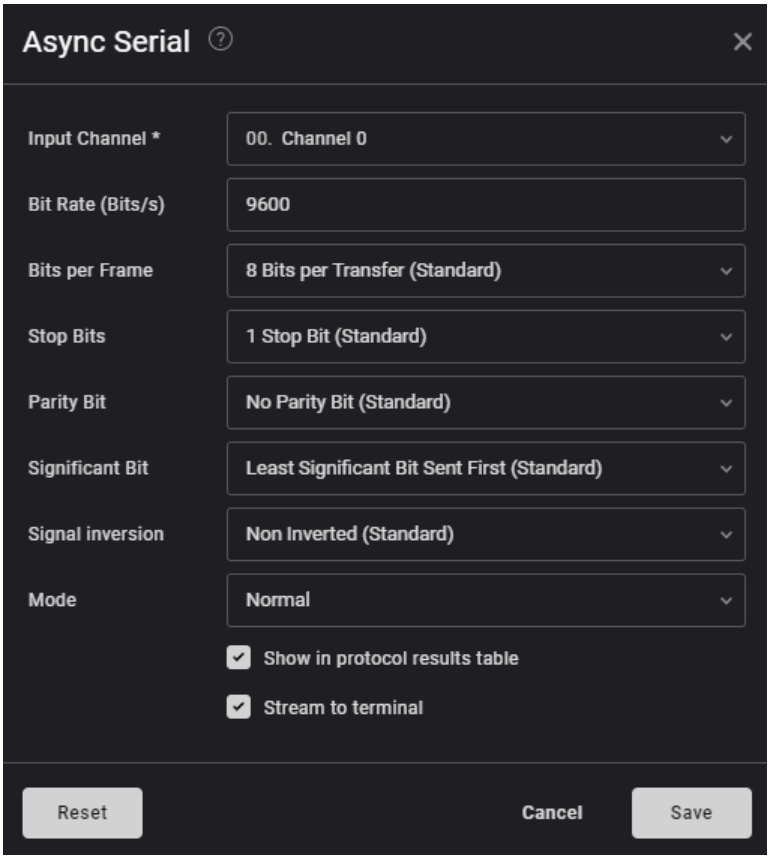


9. Com o programa `Serial_View` sendo executado na placa NUCLEO, inicie a aquisição no Logic. Note que, nesta configuração, ele aguarda uma borda de descida (a linha serial permanece em nível alto enquanto não é usada e o *Start Bit* inicia na primeira borda de descida) para iniciar a aquisição por 10ms. Digite um caractere no teclado. Este será capturado pela USART3 e, em seguida, retransmitido tanto para o terminal, através do pino PD8 (USART3), quanto para o analisador lógico, através do pino PA0 (UART4). O Logic irá detectar e capturar os *bits* transmitidos.

10. Dê um *zoom* no canal, para que o conjunto de dados recebido ocupe a maior parte da tela. Na barra da direita, clique no segundo ícone de cima para baixo (*Analyzers*). No topo do painel que se abre, clique no botão *Async Serial*.



11. Na janela que se abre, configure os parâmetros da comunicação serial a ser analisada, conforme o que foi configurado na UART4, selecionando o canal 0 para atribuir o analisador. Clique no botão “Save”.



12. Agora o analisador irá interpretar todos os *bytes* que chegarem no canal 0 considerando o formato serial assíncrono com os parâmetros selecionados. Pode-se ver que acima da forma de onda aparece uma barra com o valor interpretado para o *byte*, em formato ASCII (o formato pode ser mudado clicando com o botão direito sobre a barra). Para cada *bit* de dados, será colocado um ponto no meio da forma de onda. Note que os pontos não são colocados no *start bit* nem no *stop bit*. **Perceba o nível lógico que a UART mantém em seu Tx enquanto não há transmissão (*Idle*).**

Como os dois estados binários ‘0’ e ‘1’ são representados em sinais físicos? Qual é a codificação aplicada no periférico UART na conversão dos códigos binários em sinais físicos? Se essas questões parecem sem sentido para você, vamos ver mais adiante as discussões envolvidas em torno deste tema.

13. Dobre o tempo de captura. Reinicie a aquisição e agora aperte ENTER no terminal. Note que os dois caracteres “\r\n” são capturados pelo analisador.

14. Meça o tempo de duração de um *bit* e compare com o valor teórico para a *baud rate* de 9600 *bits* por segundo.

15. Após analisar as formas de onda, quais semelhanças você consegue identificar entre elas? Como será que cada caractere digitado é transformado em uma sequência de *bits* para ser transmitido serialmente de forma mais confiável possível? Você sabia que esse formato de transmissão, conhecido como mecanismo de sincronização *Start-Stop* (base do protocolo RS-232), é uma das "linguagens" mais utilizadas na comunicação serial assíncrona entre dispositivos digitais?

CLIs - Interfaces de Linha de Comando

Já pensou em transformar as interfaces de linha de comando (em inglês *Command-Line Interface* – CLIs) dos projetos anteriores em um sistema de controle remoto para os periféricos do microcontrolador? Como você estruturaria esse controle de forma programática? Neste quarto projeto, vamos estabelecer a base para um sistema que recebe dados pela porta serial conectada a um Terminal e utiliza essas informações para gerenciar os periféricos da placa de expansão.

O objetivo é aprendermos a estruturar o código de forma eficiente, utilizando o comando `switch` para associar cada comando a um bloco de código específico. Dessa forma, os comandos digitados no Terminal são enviados para o microcontrolador, e o processador não apenas ativa a tarefa correspondente, mas também envia uma mensagem textual para o terminal indicando a tarefa que foi ativada. E a melhor parte? Você terá a chance de implementar o controle dos periféricos com base nesse princípio, aplicando o que aprendeu para operar os dispositivos que exploramos no Roteiro 6. Prepare-se para um desafio estimulante que vai expandir suas habilidades e despertar sua criatividade!

1. Reproduza os passos de 1 a 6 do projeto "Serial_View", mas em um novo projeto chamado "Serial_CLIs".

2. Agora, vamos implementar a ISR no arquivo "stm32h7xx_it.c", especificamente no escopo `/* USER CODE BEGIN 1 */`. Primeiro, incluímos o arquivo de cabeçalho necessário para acessar os protótipos das funções nativas de processamento de *strings* em C.

```
#include "string.h"
```

Em seguida, vamos declarar as variáveis globais

```
uint8_t flagRX4 = 0; // flag de estado de uma nova recepcao em USART3
uint8_t TX3_ind;     // índice do elemento do buffer TX3 em acesso
char RX4;            // valor recebido
char TX3[80];        // buffer de mensagem textual da tarefa ativa
```

Então, implementaremos a ISR associada ao periférico USART3.

```
void USART3_IRQHandler(void) {
  if ((USART3->ISR & USART_ISR_RXNE_RXFNE)) { //há dados recebidos?
        RX4 = USART3->RDR; // Lê o caractere recebido
        while(!(USART3->ISR & USART_ISR_TXE_TXFNF)){}//Espera Tx livre
```

```
            USART3->TDR = RX4; // Eco
            if (RX4 == 0x0D) {
             while(!(USART3->ISR & USART_ISR_TXE_TXFNF)){}//Espera Tx livre
             USART3->TDR = 0x0A; // "Line Feed"
            }
            flagRX4 = 1;
    } else
    if (USART3->ISR & USART_ISR_TXE_TXFNF) {
            USART3->TDR = TX3 [TX3_ind++];
            if (TX3[TX3_ind] == '\0') {
                  USART3->CR1 &= ~USART_CR1_TXEIE_TXFNFIE_Msk;
            }
    }
}
```

3. Vamos agora implementar o controle da ativação de tarefas com base no comando (na forma de um caractere) digitado e armazenado na variável `RX4`. Esse controle será centralizado na função `main` definida no arquivo "main.c". Para garantir uma melhor encapsulação e modularidade do código, evitamos o uso do qualificador `extern` no "main.c" para acessar variáveis declaradas em "stm32h7xx_it.c". Em vez disso, definimos funções em "stm32h7xx_it.c" que permitem que outras partes do código acessem essas variáveis de maneira controlada. Vamos adicionar entre a seção de declaração das variáveis globais e a definição das ISRs no arquivo "stm32h7xx_it.c" as seguintes funções:

```
uint8_t le_flagRX4() {
   return flagRX4;
}
void limpa_flagRX4() {
   flagRX4 = 0;
}
char le_RX4() {
   return RX4;
}
void reseta_TX3_ind (){
   TX3_ind = 0;
}
void carrega_TX3 (char *buffer) {
   strcpy (TX3, buffer);
}
```

4. Dispondo dessas funções, precisamos declarar seus protótipos no escopo `/* USER CODE BEGIN PFP */` do arquivo "main.c". Vá à seção e insira os seguintes protótipos:

```
uint8_t le_flagRX4();
void limpa_flagRX4();
char le_RX4();
void reseta_TX3_ind ();
void carrega_TX3 (char *buffer);
```

5. Com isso, estamos prontos para programar o fluxo de controle central de ativação das tarefas com base nos caracteres digitados pelo usuário. O comando `switch` em C é um comando apropriado para selecionar entre múltiplas alternativas com base no valor de uma expressão/variável. Ele oferece uma estrutura para fazer escolhas entre diferentes blocos de código, proporcionando uma maneira organizada de lidar com múltiplas condições, como mostra o seguinte trecho de códigos que implementaremos no escopo /* USER CODE BEGIN 3 */

```
if (le_flagRX4()) {
	limpa_flagRX4();
	switch (le_RX4()) {
	case '0':
		carrega_TX3 ("\nApaga as 3 cores do LED RGB.\r\n");
		break;
	case 'R':
	case 'r':
		carrega_TX3 ("\nAcende LED vermelho.\r\n");
		break;
	case 'G':
	case 'g':
		carrega_TX3 ("\nAcende LED verde.\r\n");
		break;
	case 'B':
	case 'b':
		carrega_TX3 ("\nAcende LED azul.\r\n");
		break;
	case 'H':
	case 'h':
		carrega_TX3 ("\nMotor girando no sentido horario.\r\n");
		break;
	case 'A':
	case 'a':
		carrega_TX3 ("\nMotor girando no sentido anti-horario.\r\n");
		break;
	case 'P':
	case 'p':
		carrega_TX3 ("\nMotor parado.\r\n");
		break;
	case '+':
		carrega_TX3 ("\nIncremento na potencia do motor em 5%.\r\n");
		break;
	case '-':
		carrega_TX3 ("\nDecremento na potencia do motor em 5%.\r\n");
		break;
	}
	reseta_TX3_ind();
	USART3->CR1 |= USART_CR1_TXEIE_TXFNFIE_Msk;
}
```

6. Faça “Build” e transfira o código executável para microcontrolador no modo *Debug*. Na perspectiva *Debug*, abra o terminal serial do IDE e configure-o com a taxa de transmissão de 9600 *bauds*, 8 *bits* de dados, sem paridade e 1 *bit* de parada.

7. Continue (“Resume”) a execução. Digite no Terminal uma das letras programadas como alternativas no comando `switch` e observe o que acontece no Terminal Serial integrado ao IDE.

8. Vamos analisar o fluxo de controle subjacente. Pause e substitua os textos nas chamadas da função `carrega_TX3` ou substitua as letras nas alternativas para diferentes cases, como substituir “-” por “#”, regere o código (“Terminate e Relauch”) e verifique as alterações no comportamento do sistema. Mais especificamente, observe o que acontece com as mensagens no Terminal quando você altera os textos na chamada da função `carrega_TX3`, e o que acontece com as respostas do sistema quando você digita s teclas que foram substituídas. Qual função o comando *switch* está implementando?

9. Pause e restaure a versão original. Regere (“Terminate and Relaunch”) o código executável para verificar se o comportamento do sistema foi restaurado. Pause o programa, comente a última linha da função main. Em seguida, regere o código executável (“Terminate and Relaunch”). Execute o programa e observe o comportamento. Observando o novo comportamento do sistema, você consegue esboçar uma explicação para o que está acontecendo? Se não, não se preocupe, vamos explorar isso juntos mais adiante.

```
194             case '-':
195                 carrega_TX3 ("\nDecremento na potencia do motor em 5%.\r\n");
196                 break;
197             }
198             reseta_TX3_ind();
199 //          USART3->CR1 |= USART_CR1_TXEIE_TXFNFIE_Msk;
200     }
```

10. Pause o programa, descomente a linha da função main e comenta a linha da rotina `USART3_IRQHandler`. Em seguida, regere o código executável (“Terminate and Relaunch”). Execute o programa e observe o comportamento. E agora? Pelo novo comportamento do sistema, consegue imaginar a função dessa linha de desabilitação de interrupção de transmissão quando o registrador estiver vazio.

```
256     if (USART3->ISR & USART_ISR_TXE_TXFNF) {
257         USART3->TDR = TX3 [TX3_ind++];
258         if (TX3[TX3_ind] == '\0') {
259 //          USART3->CR1 &= ~USART_CR1_TXEIE_TXFNFIE_Msk;
260         }
261     }
```

11. Se você ainda não conseguiu formular uma resposta plausível para as perguntas anteriores relacionadas a este projeto, restaure a versão original do programa e adicione um *breakpoint* na linha 258 e outro na linha 259. Continue (“Resume”) a execução e, em cada parada, monitore o envio dos caracteres carregados no vetor `TX3` antes da habilitação de `TXEIE` na

função `main`, até alcançar a instrução que desabilita `TXEIE`. Isso enfatiza a utilização de *hardware* para o controle de fluxo de envio em vez de depender de laços de espera em *software*, e ressalta a importância de planejar e coordenar as interrupções no fluxo de controle para assegurar que o comportamento do sistema esteja alinhado com as expectativas.

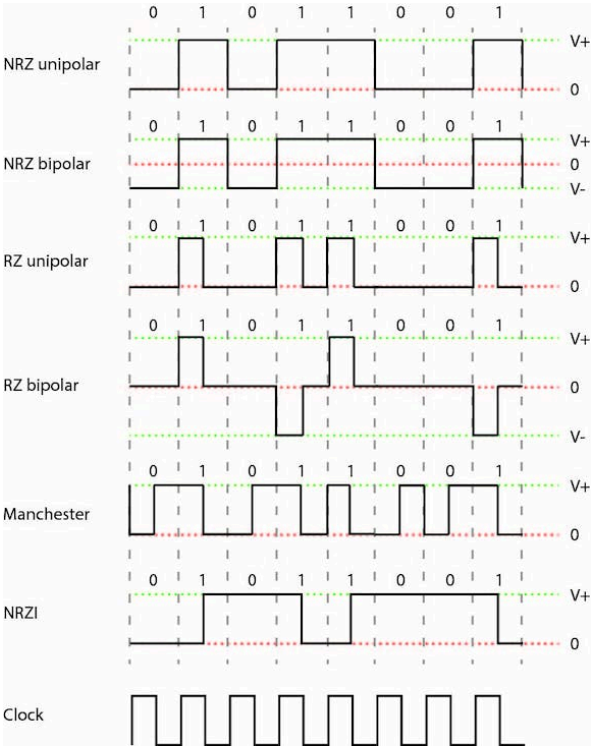
FUNDAMENTOS TEÓRICOS

A comunicação digital de dados evoluiu de forma notável, impulsionada por inovações que superaram desafios desde a representação de *bits* em sinais físicos até a implementação de *hardwares* dedicados em microcontroladores. Esses recursos, combinados a estruturas de dados, permitem a codificação e decodificação eficiente de informações, viabilizando transmissões rápidas e confiáveis por uma variedade de meios físicos.

REPRESENTAÇÃO DOS ESTADOS BINÁRIOS EM SINAIS FÍSICOS

A comunicação digital de dados enfrenta desafios desde os seus primeiros passos. Isso porque os dados digitais, em sua forma binária, não podem ser transmitidos diretamente por meios físicos, como fios ou ondas eletromagnéticas. Para que isso seja possível, é necessário converter esses dados em sinais físicos apropriados. Um dos primeiros desafios foi encontrar uma maneira eficaz de representar os *bits* binários por meio de sinais elétricos. A solução inicial consistiu em usar dois níveis distintos de tensão: um para representar o “0” e outro para o “1”. Apesar de funcional, essa abordagem básica logo se mostrou limitada.

A simples conversão binária não garantia uma transmissão confiável. Sinais elétricos estão sujeitos a interferências e ruídos, o que pode comprometer a integridade dos dados transmitidos. Por isso, surgiram diversas técnicas de codificação, que vão além da simples representação binária. Essas técnicas transformam os códigos binários em formas de sinais mais robustas, com o objetivo de reduzir a ocorrência de erros, otimizar o uso da largura de banda, e facilitar a sincronização entre transmissor e receptor. A figura a seguir ilustra algumas dessas técnicas, demonstrando diferentes formas de codificação aplicadas a um mesmo conjunto de *bits*.



A codificação **Retorno a Zero** (**RZ**) é uma técnica em que o sinal retorna a um nível de referência, geralmente zero, *dentro do período de cada bit transmitido*. Durante a transmissão de um *bit* "1", o sinal fica em um nível alto por metade do tempo do *bit* e retorna a zero na outra metade. Isso ajuda na sincronização devido às frequentes transições de sinal. No entanto, o RZ é menos eficiente em termos de largura de banda, pois requer o dobro da largura de banda que codificações NRZ. Esta **codificação** é **unipolar** quando só envolve 2 níveis de tensão (0 e positivo). Quando o *bit* "1" é representado por pulsos de tensão alternados, ou seja, um *bit* "1" é transmitido como um pulso positivo, o próximo *bit* "1" como um pulso negativo, e assim por diante, dizemos que é **codificação bipolar**, porque utiliza 3 níveis de tensão (negativo, 0 e positivo).

Por outro lado, a codificação **Não Retorno a Zero** (**NRZ**) mantém o sinal em um nível alto ou baixo durante toda a *duração do bit*, sem retornar a zero entre os *bits*. Essa abordagem é mais eficiente em termos de largura de banda, pois não requer mudanças constantes de sinal, mas pode enfrentar desafios relacionados à sincronização e à detecção de erros devido à falta de transições. A falta de transições pode levar a problemas de deriva de *clock* no receptor, dificultando a sincronização. A **codificação** é **unipolar**, quando o *bit* "1" e *bit* "0" assumem, respectivamente, níveis de *tensão positivo* e 0. E é **bipolar**, quando eles assumem, respectivamente, níveis de tensão positivo e negativo. Uma variante é a **codificação AMI** (do inglês *Alternate Mark Inversion*) em que o *bit* "1" é representado por pulsos de tensão alternados como na codificação RZ bipolar. A alternância de pulsos de tensão ajuda a evitar o acúmulo de tensão DC na linha de transmissão.

A codificação **Retorno a Zero Invertido** (**RZI**) é semelhante ao RZ, mas utiliza níveis de sinal invertidos para representar os *bits*. Assim como o RZ, o RZI também facilita a

sincronização através de transições visíveis, mas a largura de banda necessária é semelhante à do RZ. De forma análoga, a **codificação Não Retorno a Zero Invertido (NRZI)** usa níveis de tensão invertidos de NRZ para representar os *bits*.

A codificação **Manchester**, por sua vez, representa cada *bit* com uma transição de nível no meio do intervalo do *bit*. Para um *bit* "1", há uma transição de baixo para alto, enquanto para um *bit* "0", há uma transição de alto para baixo. Como há sempre uma transição no meio de cada *bit*, o receptor pode usar essas transições para sincronizar seu relógio com o do transmissor, eliminando a necessidade de um sinal de relógio separado. Por outro lado, essa técnica dobre a largura de banda necessária em comparação com o NRZ.

A codificação **Diferença de Manchester** é similar à Manchester, mas a transição ocorre no início do intervalo do *bit*, e o *bit* é representado com base na mudança de nível entre os intervalos de *bit*. Isso ajuda a manter a sincronização e a reduzir a interferência de corrente contínua, embora também exija uma largura de banda maior.

CÓDIGO ASCII: DO *BIT* AO SIGNIFICADO

Após entender como os computadores representam *bits* fisicamente, surge uma pergunta essencial: como essas sequências de '0's e '1's se transformam em algo com significado para nós, como letras, números ou símbolos? A resposta está na **codificação lógica**, um conjunto de regras que o *software* utiliza para interpretar essas sequências de *bits*. Ou seja, se no nível físico o sistema entende "presença ou ausência de sinal", "nível alto ou baixo de tensão", a codificação lógica estabelece um acordo que nos permite atribuir um significado não-ambíguo a uma sequência de *bits*. Para isso, são definidas **tabelas de correspondência** entre sequências de *bits* e seus respectivos significados. A mais famosa delas, e base para praticamente todos os sistemas digitais modernos, é o código ASCII, que representa caracteres alfanuméricos.

O **código ASCII** (do inglês *American Standard Code for Information Interchange*) desempenha um papel essencial na comunicação serial, pois estabelece um padrão universal para a representação e troca de caracteres entre dispositivos digitais. Como visto no Roteiro 4, o ASCII define uma tabela composta por **128 caracteres**, codificados em **7 *bits***, que incluem letras maiúsculas e minúsculas, dígitos, sinais de pontuação e também caracteres de controle. Os **caracteres de controle** são símbolos não renderizáveis, ou seja, não possuem representação visual direta. Eles fazem parte de um conjunto reservado para comandos que influenciam o comportamento do sistema ou a disposição dos caracteres visíveis. De modo geral, os códigos ASCII de 0 a 31 (hexadecimal 0x00 a 0x1F) são considerados caracteres de controle. Quando inseridos em uma *string*, esses códigos podem alterar o fluxo da apresentação textual ou acionar funções específicas no dispositivo.

Alguns exemplos ilustrativos incluem 0x07 (*bell*), que é o caractere de controle que faz o dispositivo emitir um som, 0x08 (*backspace*), que retrocede para sobrescrever o último caractere renderizado, 0x0A (*line feed* ou *new line*), que marca o fim de uma linha, e 0x0D (*carriage return*), que move o cursor de volta para a primeira coluna. Para incorporar esses caracteres de controle dentro de *strings* em C, utilizam-se as chamadas **sequências de escape** compostas por uma barra invertida ('\') seguida de uma letra ou de uma combinação de

dígitos. As sequências de escape para *bell*, *backspace*, *new line* e *carriage return* são, respectivamente, “\a”, “\b”, “\n” e “\r”.

Há pares de caracteres ASCII que têm funções complementares ou associadas no contexto de controle, especialmente em comunicações seriais e sistemas de terminal. Seguem-se alguns desses pares e suas relações:

- CR (‘\r’, ASCII 13, 0xD) e LF (‘\n’, ASCII 10, 0xA): São um par clássico de controle de cursor/linha em terminais e arquivos de texto. O par. avançar para a próxima linha com o cursor posicionado no início da linha.
- XON (-, ASCII 17, 0x11) e XOFF (-. ASCII 19, 0x13): O par atua como comandos de “pausa” e “continua”, permitindo o controle de fluxo por *software*.
- STX (-, ASCII 2, 0x2) e ETX (-, ASCII 3, 0x3): O para atua como delimitadores de mensagens. Muito utilizados em protocolos de comunicação baseados em pacotes, como Modbus ASCII e protocolos industriais.
- BEL (‘\a’, *Bell*, ASCII 7, 0x7) e BS (‘\b’, *Backspace*, ASCII 8, 0x8): São ambos relacionados a *feedback* do terminal e edição de entrada.

Com o tempo, a necessidade de representar um conjunto mais amplo de símbolos levou ao desenvolvimento de **extensões do padrão ASCII**. Uma das mais conhecidas é o **ISO 8859-1**, também chamado de **Latin-1** ou **ASCII estendido**, que utiliza 8 *bits* por caractere, permitindo até 256 combinações distintas. Essa expansão inclui letras acentuadas, símbolos gráficos e caracteres adicionais úteis em idiomas europeus. Tais padronizações garantem a compatibilidade e a correta interpretação dos dados textuais em sistemas distintos, independentemente de variações de *hardware* ou *software*.

As **strings**, vetores de caracteres codificados em ASCII e finalizados com o caractere nulo (\0), assumem um papel central nesse cenário. Elas possibilitam a organização e o envio de mensagens textuais, comandos e informações de forma clara e estruturada. Como a maioria das linguagens de programação oferece suporte nativo para manipulação de *strings* (concatenação, busca, formatação, entre outros), o trabalho com dados textuais em sistemas embarcados se torna eficiente e intuitivo. Além disso, durante o desenvolvimento e a depuração de sistemas embarcados, o uso de *strings* ASCII para mensagens de *status*, *logs* e diagnósticos permite uma comunicação transparente e legível por humanos. Isso facilita significativamente a análise de funcionamento do sistema e a identificação de falhas, contribuindo para um processo de desenvolvimento mais ágil e confiável.

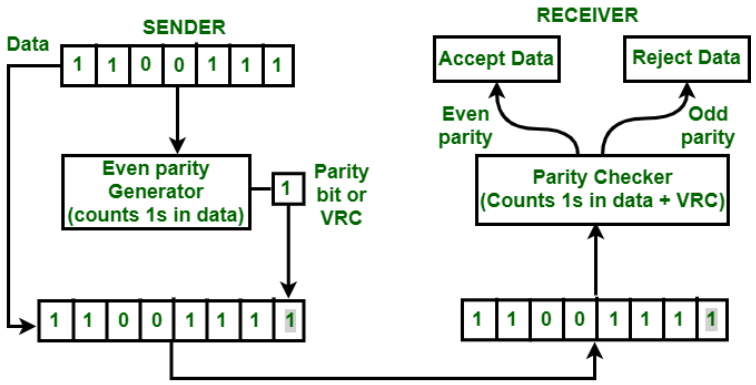
CÓDIGOS DETECTORES E/OU CORRETORES DE ERROS

Apesar das técnicas avançadas de codificação de sinais, a transmissão de dados ainda enfrenta um desafio crítico: ruídos e interferências. Esses obstáculos podem distorcer os sinais, comprometendo a interpretação correta dos *bits* pelo receptor. É crucial entender que tanto o padrão ASCII de 7 *bits* quanto suas extensões de 8 *bits* focam na **representação simbólica** dos dados, sem garantir a **integridade da transmissão**. Em ambientes com ruídos ou interferências, os códigos ASCII são vulneráveis à corrupção.

Para superar essa limitação, recorremos aos **códigos detectores e corretores de erros**. Esses códigos introduzem redundância nos dados, permitindo identificar e, em muitos casos, corrigir falhas na transmissão. Além de proteger a integridade dos dados, alguns desses

códigos auxiliam na sincronização entre transmissor e receptor e no controle do fluxo de informações, elementos cruciais para uma comunicação digital eficiente e confiável. Essa confiabilidade se torna ainda mais crítica em sistemas embarcados, especialmente em aplicações sensíveis como automação industrial, controle de aviação e dispositivos médicos. Nesses cenários, a precisão da comunicação é vital, pois erros podem ter consequências graves. A codificação de erros, portanto, complementa as técnicas de codificação de sinais físicos, garantindo a robustez e a precisão dos dados em ambientes de comunicação sujeitos a ruídos e interferências.

Uma técnica simples para detectar erros em transmissões seriais com baixa taxa de erro é a paridade. A **paridade** de uma palavra, ou de um grupo de *bits*, se refere à paridade da quantidade de *bits* "1" contidos na palavra. Ela é **par** se essa quantidade for par e **ímpar** caso contrário. O processo consiste em adicionar um *bit* (de paridade) à palavra antes de transmití-la, indicando se o número de *bits* "1" na palavra é par ("0") ou ímpar ("1"). No receptor, a quantidade de *bits* "1" é novamente contada e comparada com o *bit* de paridade recebido. Se houver diferença entre eles, é caracterizado com um erro na transmissão.



Uma maneira eficaz de determinar a paridade de uma palavra é através do **algoritmo XOR** (ou-exclusivo). Este algoritmo realiza uma operação lógica "ou-exclusivo" (XOR) *bit* a *bit* entre os *bits* da palavra. Se o número de *bits* "1" na palavra for par, o resultado da operação XOR será 0, indicando uma paridade par. Por outro lado, se o número de *bits* "1" for ímpar, o resultado será 1, indicando uma paridade ímpar. O algoritmo XOR é altamente eficiente, pois não requer a contagem direta dos *bits* "1" na palavra, e pode ser facilmente implementado usando operações *bit* a *bit* simples. Uma otimização adicional pode ser alcançada ao dividir os *bits* de um inteiro recursivamente em duas metades e aplicar repetidamente a operação XOR em cada metade até que reste apenas 1 *bit*. Este e outros algoritmos para o cálculo da paridade de uma palavra são detalhadamente explorados nesta referência.

Como um código corretor de erros, apresentamos o **código de Hamming** (7,4) que adiciona 3 *bits* de paridade a uma palavra de dados de 4 *bits*, resultando em uma palavra de código de 7 *bits*, p1 p2 d1 p3 d2 d3 d4, onde p1, p2 e p3 são os *bits* de paridade e d1, d2, d3 e d4 são os *bits* de dados. Esses *bits* de paridade (par) são posicionados em posições específicas (1, 2 e 4) e calculados de forma que

- **p1:** verifica as posições onde o *bit* menos significativo é 1 (1, 3, 5 e 7).
- **p2:** verifica as posições onde o segundo *bit* é 1 (2, 3, 6 e 7).
- **p3:** verifica as posições onde o terceiro *bit* é 1 (4, 5, 6 e 7).

Para cada *bit* de paridade **pi**, atribui-se um valor ‘0’ (se a paridade estiver correta) ou ‘1’ (se estiver incorreta). Esses valores são chamados síndrome *bit* i, que são usadas para formar um **número binário de 3 *bits***, **p3p2p1**. Este número é denominado **síndrome** ou **posição do erro**. O valor da síndrome indica diretamente a **posição do *bit* que contém o erro** (de 1 a 7). O receptor pode usar essa síndrome para identificar e corrigir o *bit* errado, restaurando os dados originais.

Para aplicações mais exigentes, técnicas de codificação como **4B/5B** e **8B/10B** são empregadas. Essas técnicas introduzem uma forma de codificação mais avançada, onde 4 *bits* de dados são codificados em 5 *bits* de sinal, e 8 *bits* de dados em 10 *bits* de sinal, respectivamente. Elas não apenas detectam erros, mas também ajudam a manter a sincronização e a integridade dos dados ao adicionar redundância e garantir transições de sinal adequadas. A codificação e decodificação dessas técnicas são facilitadas por tabelas de busca (em inglês, *lookup tables*), que permitem uma implementação eficiente e rápida.

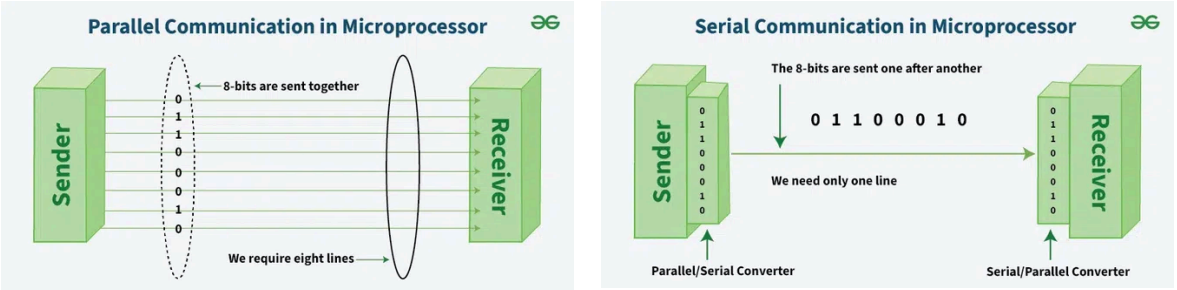
Além dessas técnicas, a **codificação de redundância cíclica** (em inglês *Cyclic Redundancy Check* – CRC) é amplamente utilizada para detecção de erros em comunicações digitais. O CRC calcula um código de verificação, ou *checksum*, a partir dos dados transmitidos, que é anexado à mensagem. O receptor recalcula o CRC com base nos dados recebidos e compara o resultado com o *checksum* recebido. Se os valores coincidirem, os dados são considerados íntegros. Caso contrário, um erro é detectado. O CRC é eficaz na detecção de erros em rajadas (em inglês, *burst errors*), onde múltiplos *bits* consecutivos são corrompidos. A implementação do CRC geralmente envolve operações de divisão polinomial em *hardware* ou *software*.

TIPOS DE TRANSMISSÃO DE SINAIS

No contexto dos sistemas de comunicação e processamento de sinais, após a etapa de codificação, seja para representação binária, compressão ou aplicação de códigos de detecção e correção de erros, os dados precisam ser fisicamente transportados entre os dispositivos envolvidos na comunicação. Essa transmissão se dá por meio de canais físicos, os quais podem ser constituídos por condutores metálicos, trilhas em circuitos impressos, fibras ópticas ou até mesmo o ar, no caso de ondas eletromagnéticas. Podemos analisar o comportamento dessa transmissão sob dois aspectos: a quantidade de canais físicos envolvidos e o modo de duplexidade da comunicação.

Quanto à quantidade de canais físicos envolvidos, distinguem-se a transmissão paralela a serial. A **transmissão paralela** é caracterizada pelo envio simultâneo de múltiplos *bits* através de múltiplas vias físicas. Cada *bit* ocupa um condutor ou trilha distinta, permitindo a transferência de palavras inteiras em uma única operação de *clock*. Essa abordagem é comum em barramentos internos de computadores, como o antigo padrão IDE (Integrated Drive Electronics), e em interfaces de dados em FPGAs, como barramentos de dados entre blocos lógicos. Apesar de permitir altas taxas de transferência em curtas distâncias, essa técnica apresenta limitações significativas em aplicações de médio e longo alcance, devido a problemas como ruído entre canais (*crosstalk*), interferência eletromagnética, defasagem temporal (*skew*) e o elevado custo físico dos cabos.

Por sua vez, a **transmissão serial** utiliza apenas um canal físico para transmitir os *bits* de forma sequencial, *bit* a *bit*. Embora, teoricamente, possa parecer mais lenta, a transmissão serial é amplamente adotada em sistemas modernos justamente pela possibilidade de alcançar altas taxas de transmissão com alta integridade do sinal, mesmo em longas distâncias. Esse método apresenta ainda vantagens, como a simplificação do *design* dos sistemas de comunicação e a redução da complexidade em termos de cabeamento e conexões. Exemplos de aplicações incluem comunicação ponto a ponto em sistemas embarcados e em microcontroladores e sensores. Em sistemas industriais, a comunicação serial também é amplamente utilizada em barramentos de campo.

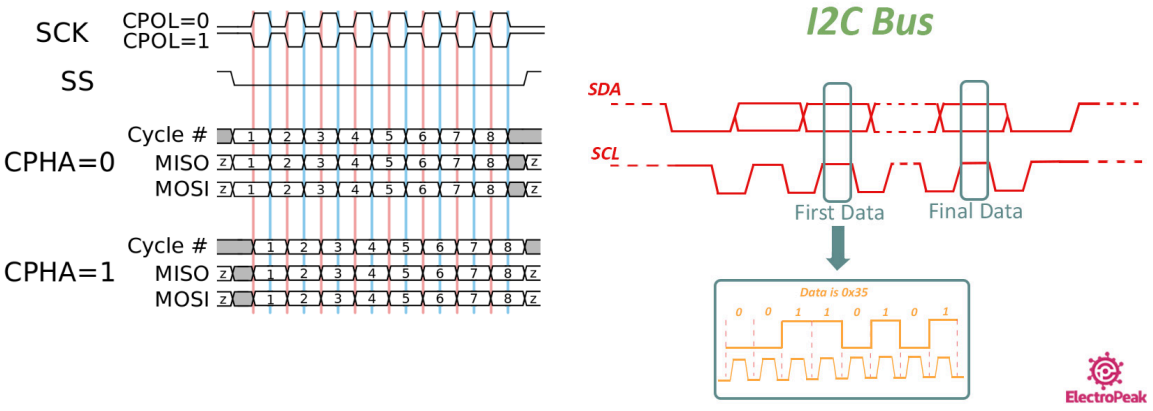


A comunicação serial, apesar de ser mais comum em longas distâncias por exigir menos recursos físicos, apresenta um desafio importante: o **sincronismo**. Como os *bits* são enviados um após o outro, de forma sequencial, o receptor precisa saber exatamente quando cada *bit* começa e termina para interpretá-los corretamente. Essa exigência de precisão no tempo exige mecanismos que garantam a sincronização entre transmissor e receptor.

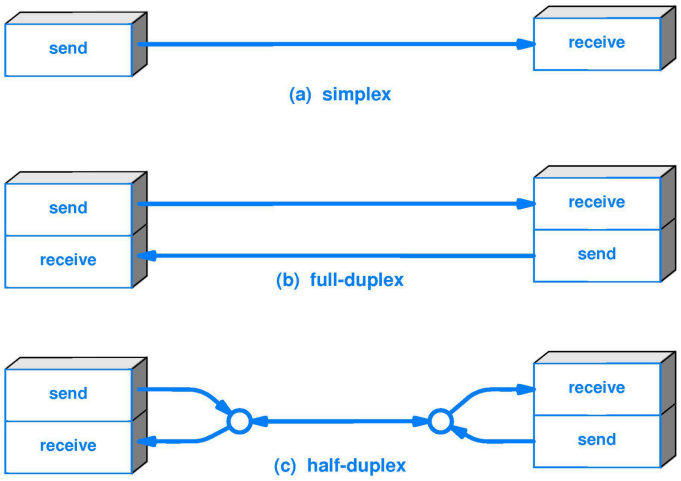
Para superar esses desafios, a comunicação serial utiliza diversas técnicas de sincronização, como sincronização assíncrona e sincronização síncrona. Na **sincronização assíncrona**, cada caractere ou bloco de dados é enviado com *bits* de início (em inglês, *start bit*) e parada (em inglês, *stop bit*), que permitem ao receptor sincronizar-se com cada unidade de dados individualmente. Na **sincronização síncrona**, o transmissor e o receptor utilizam um sinal de "relógio" compartilhado para sincronizar a transmissão e a recepção dos *bits*. Essa abordagem oferece maior precisão, mas exige um canal de sincronização dedicado ou técnicas de sincronização embutidas nos dados, como a codificação Manchester.

Existem diferentes métodos de implementação da comunicação síncrona, sendo um dos mais comuns o **modelo mestre-escravo** (em inglês, *master-slave*), onde o dispositivo mestre é responsável por gerar e controlar o sinal de *clock*. O dispositivo escravo segue o sinal de *clock* do mestre, garantindo que a troca de dados ocorra de forma sincronizada com as transições do *clock*. Os parâmetros de sincronização entre o transmissor e o receptor podem variar. Em alguns sistemas, essa sincronização envolve dois parâmetros específicos do sinal de *clock*: a polaridade (em inglês, *Clock Polarity* - CPOL) e a fase (em inglês, *Clock Phase* - CPHA). A **polaridade** define se o sinal de *clock* é ativo-alto ou ativo-baixo (ou seja, se o nível de tensão do *clock* é alto ou baixo durante o ciclo de inatividade), enquanto a **fase**

especifica qual borda do *clock* (subida ou descida) os dados são amostrados. A combinação dos estados de CPOL e CPHA define quatro modos distintos de operação no transmissor através dos quais o receptor sabe exatamente quando amostrar os dados transmitidos pelo mestre, mantendo a sincronização. Em outros, a sincronização é feita de maneira mais simples, utilizando apenas as transições do sinal de *clock*, sem a necessidade de ajustes complexos de polaridade ou fase. O dispositivo mestre gera e controla o sinal de *clock*, e todos os dados são trocados em conformidade com as transições desse sinal de *clock*, geralmente nas bordas de subida. Aprofundaremos essas discussões acerca comunicação serial síncrona no Roteiro 10.



Além do tipo de transmissão física, o **modo de duplexidade** define como os dados são trocados entre o transmissor e o receptor. Distinguem-se 3 modos de duplexidade. Em um **sistema simplex**, ou *single-wire*, a comunicação ocorre em apenas um sentido: o transmissor envia e o receptor apenas recebe, sem possibilidade de resposta. Esse modelo é utilizado em sistemas de telemetria unidirecional ou em transmissões de *broadcast*, como em sistemas de rádio FM ou TV analógica, onde o foco está na distribuição massiva da informação.



Na comunicação ***half-duplex***, a troca de dados ocorre em ambos os sentidos, mas nunca simultaneamente. O canal é compartilhado alternadamente, de acordo com o controle de

acesso. Esse tipo de comunicação é comum em sistemas de rádio comunicador (como *walkie-talkies*), mas também em redes industriais como o padrão RS-485, que permite comunicação multiponto, com alternância coordenada entre os dispositivos transmissores e receptores.

Por fim, na comunicação ***full-duplex***, os dispositivos podem transmitir e receber dados simultaneamente, seja por meio de dois canais físicos independentes ou por técnicas de multiplexação no mesmo canal. Essa forma de comunicação é amplamente utilizada em redes Ethernet modernas, telefonia digital e protocolos de comunicação em tempo real, como UART full-duplex. Em redes industriais mais recentes e sistemas de controle distribuído, a comunicação *full-duplex* permite latências mais baixas e melhor desempenho em aplicações críticas.

A tabela a seguir sintetiza os 5 tipos de transmissão e suas relações.

Tipos de transmissão	Canais físicos	Direção de comunicação
Paralela	Múltiplos fios/canais	Pode ser simplex. *half-duplex*, *full-duplex*.
Serial	Um único fio/canal	Pode ser simplex. *half-duplex*, *full-duplex*.
Simplex	Um ou mais fios/canais	Só vai do transmissor para receptor.
Half-duplex	Um ou mais fios/canais	Vai e volta, mas não ao mesmo tempo.
Full-duplex	Um ou mais fios/canais	Vai e volta ao mesmo tempo.

HANDSHAKING

Em comunicações digitais, a troca eficiente de dados exige mais do que apenas um caminho e direção definidos. Os dispositivos precisam sincronizar seus envios e recebimentos para evitar sobrecarga e perda de informações. É aqui que entra o mecanismo de *handshaking* para o controle de fluxo e sincronização. O ***handshaking*** garante que os dispositivos se comuniquem de forma coordenada, trocando sinais de confirmação para assegurar que os dados foram recebidos corretamente e que ambos estão prontos para continuar a troca. Existem duas abordagens principais: *handshaking* por *hardware* e por *software*.

Na abordagem de ***handshaking* por *hardware***, sinais físicos dedicados controlam a comunicação. Os sinais RTS (do inglês *Request To Send*) e CTS (do inglês *Clear To Send*) são exemplos comuns. O transmissor ativa o sinal RTS para solicitar permissão para enviar dados. O receptor, por sua vez, ativa o sinal CTS para indicar que está pronto para receber. Somente após receber o CTS, o transmissor inicia a transmissão, evitando sobrecarga no *buffer* do receptor e garantindo uma comunicação organizada.

Em contraste, o ***handshaking* por *software*** utiliza caracteres de controle especiais, como XON e XOFF, para gerenciar o fluxo de dados. O receptor envia XON (Transmissão Permitida) para indicar que pode receber mais dados e XOFF (Transmissão Interrompida) para solicitar uma pausa na transmissão até que seu *buffer* esteja disponível. Essa abordagem dispensa sinais físicos adicionais, utilizando o próprio canal de dados para o controle de fluxo.

Um exemplo clássico de aplicação de *handshaking* por *software* é encontrado em sistemas embarcados onde **uma única rotina de interrupção (ISR)** trata tanto eventos de **recepção (RXE)** quanto de **transmissão (TXE)**. Suponha que a transmissão só deva ocorrer após a conclusão do processo de recepção. Se a *flag* de transmissão for habilitada prematuramente, a ISR pode ser desviada para o código de transmissão antes que os dados recebidos tenham sido processados, violando a ordem correta da comunicação.

Para evitar esse tipo de conflito, o sistema pode empregar uma **variável de controle**, como uma ***flag* lógica definida por *software***, que indique se a recepção foi completamente processada, complementando a ***flag* de estado RXE**, que apenas sinaliza a chegada de novos dados. Dessa forma, mesmo que a interrupção de transmissão seja acionada (por meio da ***flag* de estado TXE**, que indica que o periférico está pronto para transmitir), a rotina de serviço de interrupção (ISR) verifica primeiro essa variável de controle antes de iniciar a transmissão. Esse mecanismo funciona como um ***handshak*e por *software*** entre as fases de recepção e transmissão, assegurando que o **sequenciamento** das operações ocorra de maneira correta, segura e previsível.

RS-232: PROTOCOLO DE COMUNICAÇÃO SERIAL ASSÍNCRONA

Até aqui, foram apresentados os conceitos fundamentais que envolvem a comunicação entre dispositivos, incluindo os diferentes tipos de transmissão, os mecanismos de controle e as técnicas utilizadas para garantir a integridade e a ordem dos dados. No entanto, para que dois dispositivos possam realmente trocar informações de forma eficiente, não basta apenas saber transmitir e receber sinais: é necessário que ambos sigam as mesmas regras de operação, ou seja, que "falem a mesma língua". Essas regras definem aspectos como o formato dos dados, a velocidade de transmissão, os sinais de controle, o início e o fim de uma comunicação, entre outros detalhes essenciais para o entendimento mútuo. Esse conjunto de regras é conhecido como **protocolo de comunicação**.

Dentre os diversos tipos de protocolos de comunicação serial, destaca-se o padrão RS-232 (do inglês *Recommended Standard* 232) para **comunicações seriais assíncronas**. O **padrão RS-232**, desenvolvido pela *Electronic Industries Alliance* (EIA), é um protocolo de comunicação serial amplamente utilizado para a transmissão de dados entre dispositivos. Introduzido na década de 1960, o RS-232 define os aspectos elétricos e mecânicos para a comunicação serial entre um computador e periféricos, como modems e impressoras. Ele especifica a forma como os sinais de dados são representados e como os dispositivos devem se comunicar.

No contexto do **RS-232**, os dados são codificados utilizando dois estados elétricos principais: **Mark** e **Space**. Esses estados representam níveis de tensão específicos, usados para indicar valores binários durante a transmissão:

- **Mark (Marca):** corresponde a um nível de tensão negativo, geralmente entre -3V e -25V, e representa o *bit* binário "1".
- **Space (Espaço):** corresponde a um nível de tensão positivo, geralmente entre +3V e +25V, e representa o *bit* binário "0".

Além da codificação dos dados, o padrão RS-232 também especifica os conectores físicos utilizados na comunicação. Os mais comuns são:

- **DB9:** conector de 9 pinos, amplamente utilizado em portas seriais padrão de computadores e periféricos.
- **DB25:** conector de 25 pinos, comum em sistemas mais antigos e ainda presente em aplicações industriais ou legadas.

Esses conectores garantem a conexão física entre os dispositivos e a transmissão adequada dos sinais elétricos, permitindo que os estados **Mark** e **Space** sejam corretamente interpretados durante a comunicação.

Tipicamente, o RS-232 utiliza um esquema de transmissão assíncrona baseado em *bits* de início (*start*) e de parada (*stop*) para delimitar a transmissão de dados por caractere, operando da seguinte forma:

1. ***Start Bit*:** Antes do envio de cada *byte* de dados, um *bit* de início (em inglês, *start bit*) é transmitido. O *start bit* é um sinal de espaço (nível de tensão positivo), que sinaliza o início da transmissão de um novo *byte*.
2. ***Data Bits*, ou quadros de dados** (em inglês, *data frames*)**:** Após o *start bit*, os *bits* de dados são enviados, codificados como estados Mark e Space. Normalmente, um *byte* de dados é composto por 8 *bits*, mas o número de *bits* pode variar dependendo da configuração do sistema.
3. ***Stop Bit*:** Após os *bits* de dados, um ou mais *bits* de parada (*stop bits*) são transmitidos. Os *stop bits* são sinais de marca (nível de tensão negativo) que indicam o fim da transmissão do *byte* de dados. Eles garantem que o receptor possa identificar a conclusão do *byte* e se preparar para o próximo *byte*.

É comum adicionar um *bit* de paridade ao quadro de dados para realizar uma verificação simples de erros durante a transmissão, ajudando a identificar alterações acidentais nos dados. Vale destacar que, embora o mecanismo de sincronização *Start-Stop* seja amplamente utilizado com o RS-232 em transmissões assíncronas, especialmente em portas seriais conhecidas como portas COM de computadores, o padrão RS-232 não está restrito a esse método específico.

O RS-232 define principalmente as características elétricas, lógicas e mecânicas da interface, sendo suficientemente flexível para ser utilizado com diferentes esquemas de comunicação, incluindo protocolos síncronos, modulações específicas ou até transmissões personalizadas, dependendo das exigências da aplicação e do controle implementado por *software*. Essa flexibilidade torna o RS-232 uma escolha versátil para diversas aplicações industriais, laboratoriais e legadas, mesmo diante do surgimento de tecnologias de comunicação mais modernas.

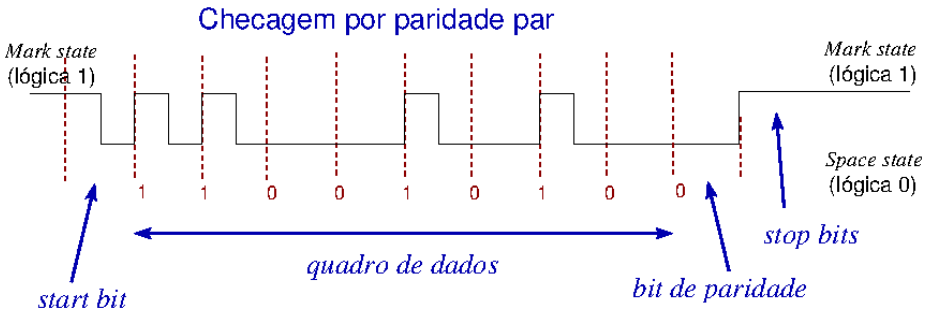


Figura 1: Transmissão por pacote numa comunicação serial assíncrona.

Apesar da sua popularidade, o RS-232 possui várias limitações significativas, incluindo uma restrição a distâncias curtas, geralmente até 15 metros, e suporte apenas para comunicação ponto-a-ponto entre dois dispositivos. Sua suscetibilidade a interferências eletromagnéticas e ruídos, combinada com uma redução na taxa de transmissão em distâncias maiores, limita sua eficácia em ambientes industriais e em longas distâncias. Protocolos como RS-485 e comunicação via USB são frequentemente escolhidos para superar essas restrições e atender a necessidades mais complexas de comunicação serial.

CIRCUITOS DEDICADOS: UART e USART

Agora que compreendemos como os dados binários são representados fisicamente, como eles podem ser transmitidos por diferentes meios e direções, e como protocolos como o RS-232 organizam essa comunicação, é hora de abordar como isso é realizado dentro de um microcontrolador na prática. Em sistemas embarcados modernos, como os baseados na família STM32, a conversão entre dados digitais, como caracteres ASCII, e sinais físicos, níveis de tensão na linha TX/RX, não é feita por *software*, mas sim por módulos de *hardware* dedicados integrados ao microcontrolador: os periféricos UART e USART. Esses módulos são responsáveis por codificar automaticamente no transmissor os dados digitais (como *bytes* armazenados na memória) em sinais elétricos conforme o protocolo escolhido (como RS-232), e também por decodificar os sinais no receptor em dados binários compreensíveis pelo processador.

UART (do inglês *Universal Asynchronous Receiver/Transmitter*) e **USART** (do inglês *Universal Synchronous/Asynchronous Receiver/Transmitter*) são circuitos dedicados de interface para comunicação serial, amplamente utilizados em sistemas embarcados e microcontroladores. Esses periféricos são responsáveis por converter dados paralelos, manipulados internamente pelo processador, em dados seriais transmitidos por uma ou duas linhas físicas, e vice-versa, seguindo formatos compatíveis com diversos protocolos de comunicação. Pode-se integrar esses sinais seriais com o padrão elétrico RS-232 utilizando um transceptor como o MAX3232, que adapta os níveis de tensão TTL (utilizados pelas UARTs/USARTs internas) aos níveis especificados pelo RS-232. Essa adaptação é comum em comunicações assíncronas que utilizam UARTs em conjunto com dispositivos ou interfaces compatíveis com RS-232, como terminais seriais.

Uma das principais semelhanças entre UART e USART é que ambos operam na **transmissão de dados seriais**, ou seja, enviam e recebem informações ***bit* a *bit***, através de uma única linha de comunicação por direção. Em ambos os casos, os dados são organizados em quadros (*frames*) compostos por *bits* de dados, *bit* de paridade (opcional) e *bits* de parada (em inglês *stop bit*), o que permite estruturar a comunicação e contribuir para a detecção de erros. Além disso, UARTs e USARTs permitem configurar diferentes **taxas de transmissão**, conhecidas como *baud rates*, que correspondem à quantidade de símbolos por segundo.

Em termos de implementação, tanto UART quanto USART são disponibilizados como blocos de *hardware* integrados aos microcontroladores, e seu funcionamento é controlado via registradores especiais, que permitem configurar o modo de operação, *baud rate*, formato do *frame* e ativar interrupções, entre outras funcionalidades. Para a comunicação serial, ambos utilizam os pinos TX (*transmit*) e RX (*receive*). O pino **TX** é responsável por transmitir os dados gerados pelo dispositivo para outro equipamento conectado, enquanto o pino **RX** recebe os dados provenientes de um transmissor externo, convertendo-os em um formato que pode ser interpretado pelo microcontrolador. Em outras palavras, TX envia e RX recebe. A integridade dos sinais é assegurada pelo protocolo de comunicação utilizado, que estabelece a estrutura e a temporização dos dados transmitidos. Em determinadas situações, como na comunicação com módulos Bluetooth H06, pode ser necessário utilizar um resistor de *pull-up* no pino RX para evitar que o sinal fique em estado flutuante (indeterminado) quando não há transmissão ativa. A necessidade desse resistor depende das características do *hardware* envolvido e da lógica de sinal adotada no circuito.

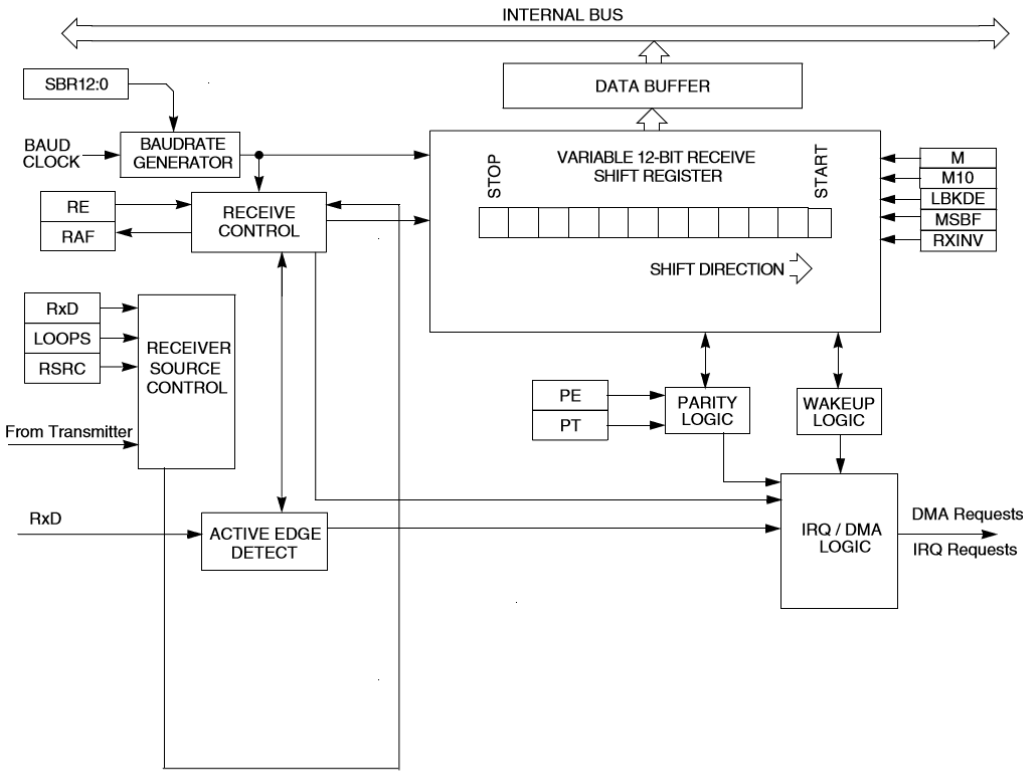


Figure 39-2. UART receiver block diagram

O processo de recepção (RX) inicia-se com a chegada de sinais seriais de um dispositivo externo. Esses sinais representam os dados como uma sequência de *bits* transmitidos em série. O módulo USART/UART identifica o início e o fim de cada quadro de dados por meio de sinais de *start* e *stop*. Após o recebimento, o módulo converte a sequência de *bits* do formato serial para paralelo, utilizando um registrador de deslocamento, e armazena os dados em um *buffer*/registrador interno (DATA BUFFER), conforme ilustrado no diagrama de blocos extraído do manual de referência do microcontrolador Kinetis KL25Z. O processador, então, acessa e processa esses dados conforme necessário.

Por outro lado, o canal TX inicia sua operação ao receber dados paralelos do processador (UART_D) e prepará-los para a transmissão serial. Este processo envolve a adição de *bits* de *start*, *stop* e, se necessário, *bits* de paridade, assegurando a integridade dos dados. A sequência de *bits* paralelos é então convertida para o formato serial e transmitida *bit* a *bit* pelo canal TX, conforme ilustrado no diagrama de blocos extraído do manual de referência do microcontrolador Kinetis KL25Z. Durante a transmissão, o módulo USART/UART ajusta a taxa de envio de acordo com a configuração da taxa de *baud*, garantindo a correta ordem e velocidade dos *bits*.

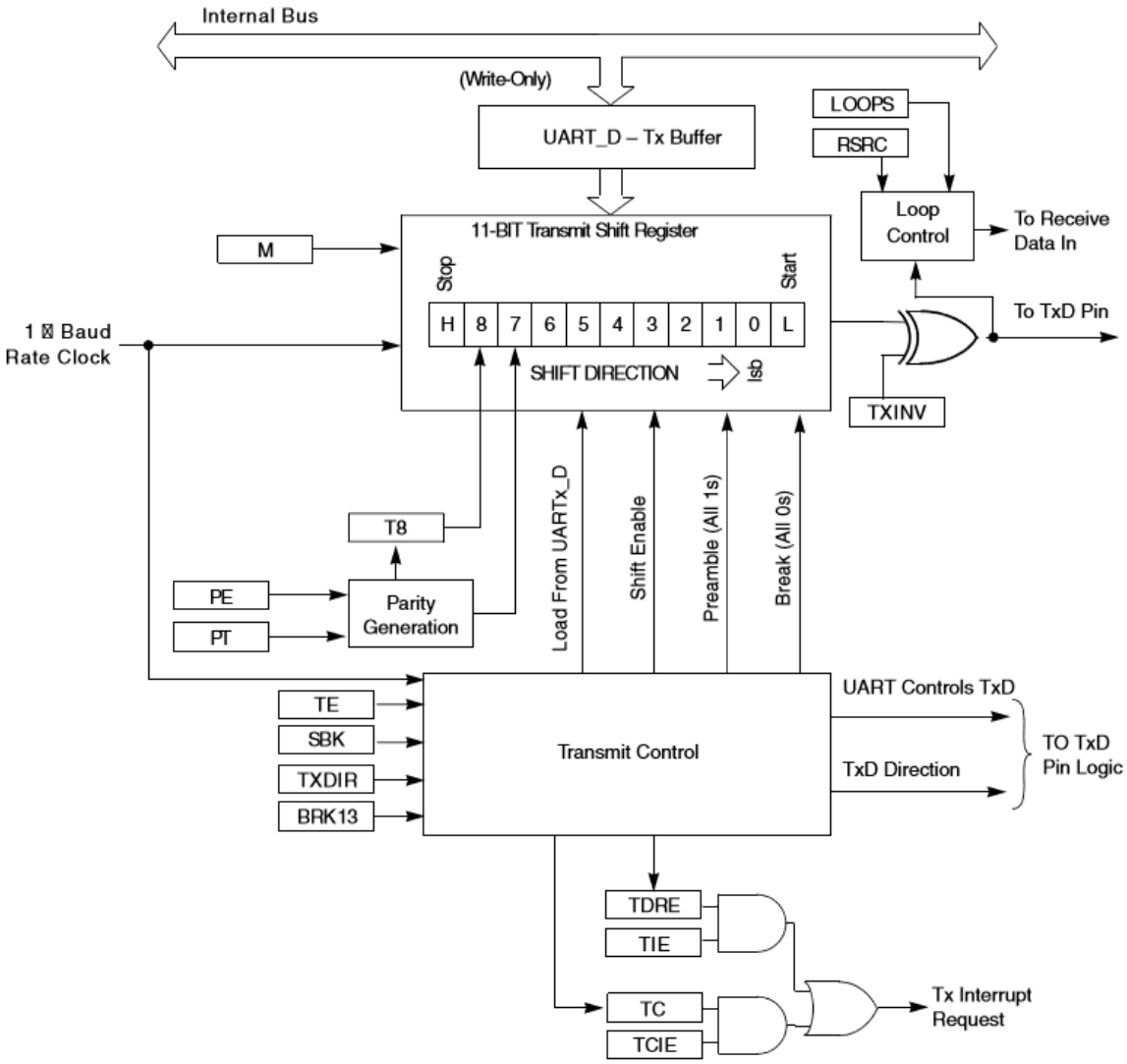


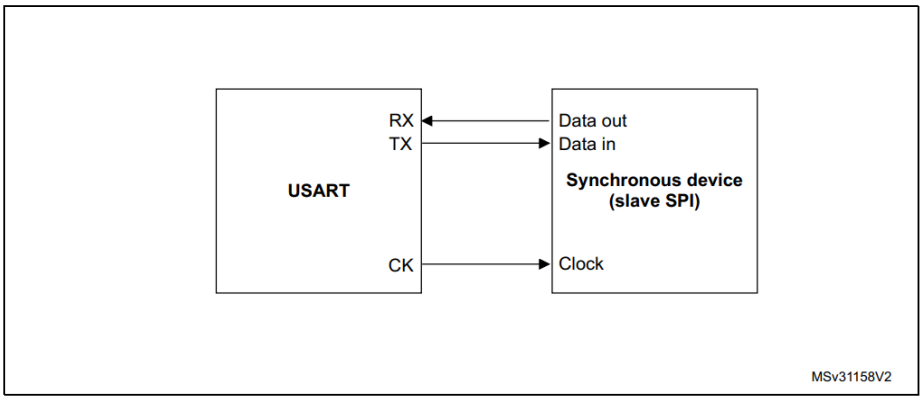
Figure 39-1. UART transmitter block diagram

Para coordenar a operação dos canais RX e TX e evitar conflitos no acesso ao processador, são empregadas ***flags* de estado**. Essas *flags* indicam o *status* dos *buffers* de recepção e transmissão, permitindo que o sistema controle o fluxo de dados de forma eficiente. Por exemplo, uma ***flag* de recepção** pode sinalizar que os dados foram recebidos e estão prontos para processamento, enquanto uma ***flag* de transmissão** pode indicar que o *buffer* de transmissão está vazio e pronto para receber novos dados. Essas ***flags*** permitem que o sistema gerencie o fluxo de dados de maneira ordenada e sem sobrecarregar o processador. Elas garantem que a recepção e a transmissão ocorram de forma sincronizada e controlada.

A principal diferença entre UART e USART reside no gerenciamento da comunicação. A UART opera exclusivamente em modo assíncrono, dispensando um sinal de *clock* compartilhado. Em vez disso, a sincronização é alcançada por meio de *start* e *stop bits* que delimitam a transmissão de cada caractere. Essa abordagem simplifica a implementação, mas pode comprometer a velocidade e a precisão, especialmente em longas distâncias ou altas taxas de dados. Por outro lado, a USART oferece maior versatilidade ao suportar tanto o modo síncrono quanto o assíncrono. No modo síncrono, a comunicação é sincronizada por um sinal de *clock* (CK) compartilhado, resultando em maior velocidade e precisão. Isso é particularmente vantajoso em aplicações que exigem alta taxa de transferência ou

sincronização precisa. No entanto, o modo síncrono exige configuração adicional e é mais complexo de implementar em comparação com o modo assíncrono da UART.

Figure 554. USART example of synchronous master transmission

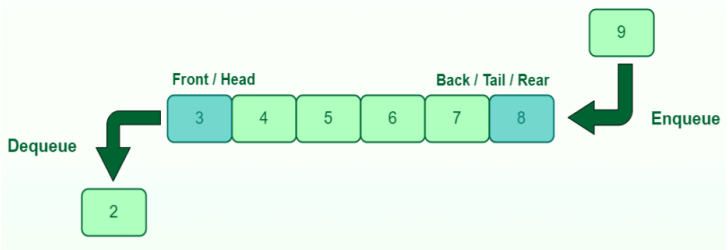


ESTRUTURA DE DADOS: FILA

Embora os periféricos UART/USART permitem a troca eficiente de dados entre dispositivos, a diferença de velocidade de transmissão entre os dispositivos em sistemas reais pode causar atrasos, perda de dados ou até conflitos de acesso ao processador. Para mitigar esses problemas, utilizamos estruturas de dados como FIFO (do inglês *First In, First Out*), ou filas, em conjunto com UART/USART. Essas filas atuam como *buffers*, armazenando temporariamente os dados recebidos ou a serem transmitidos, permitindo o processamento mesmo em cenários de taxas de transmissão e recepção dessincronizadas. Em uma fila, os dados são inseridos em uma extremidade, chamada de **cauda**, e removidos na extremidade oposta, conhecida como **início** ou **frente**.

Um exemplo comum de fila é qualquer fila de consumidores que aguardam para acessar um recurso: o primeiro consumidor a chegar é o primeiro a ser atendido. A principal diferença entre filas e pilhas, que vimos no Roteiro 3, está na forma como os itens são removidos. Em uma pilha, o item removido é o mais recentemente adicionado, seguindo o princípio "*Last In, First Out*" (LIFO). Em contraste, em uma fila, o item removido é o que foi adicionado há mais tempo, seguindo o princípio FIFO. Isso garante que a ordem de entrada dos elementos seja respeitada e que cada elemento seja processado na sequência em que chegou.

Uma fila pode ser implementada de várias maneiras, incluindo listas encadeadas e vetores. Em uma implementação baseada em vetores, a fila é geralmente representada por dois índices: um para a frente e outro para a cauda. Quando um novo elemento é inserido, ele é colocado na posição indicada pelo índice da cauda, e o índice da cauda é incrementado. Quando um elemento é removido, ele é retirado da posição indicada pelo índice da frente, e o índice da frente é incrementado, como ilustra a figura a seguir. Em uma lista encadeada, a fila é composta por uma série de nós, onde cada nó aponta para o próximo na fila, e a inserção e remoção são realizadas nas extremidades apropriadas da lista.



Queue Data Structure

As operações básicas para manipulação dos elementos armazenados em uma fila incluem:

- **enqueue()** é responsável por inserir um elemento no final da fila, ou seja, na cauda.
- **dequeue()** remove e retorna o elemento que está na frente da fila, garantindo que o item que foi inserido há mais tempo seja o primeiro a ser retirado.
- **front()** permite acessar o elemento que está na frente da fila sem removê-lo, fornecendo uma visão do próximo item a ser processado.
- **rear()** retorna o elemento na extremidade traseira sem removê-lo, permitindo verificar o último item inserido.
- **isEmpty()** indica se a fila está vazia, ou seja, se não há elementos presentes.
- **isFull()** informa se a fila está cheia e, portanto, não pode acomodar mais elementos.
- **size()** fornece o número total de elementos atualmente presentes na fila, oferecendo uma visão do seu tamanho.

Existem várias referências disponíveis na *internet* para a implementação de filas utilizando diferentes estruturas de dados, como uma implementação usando um vetor ou arranjo, onde as operações básicas são realizadas nas extremidades de um vetor, e uma implementação com listas ligadas no *site* Geeksforgeeks.

UMA APLICAÇÃO: TERMINAIS SERIAIS

Terminais seriais são interfaces de comunicação que permitem a troca de dados entre dispositivos, utilizando uma conexão serial. Essa abordagem é amplamente utilizada em sistemas embarcados, dispositivos de rede e na comunicação entre computadores e equipamentos periféricos. Nos sistemas Windows, um dos terminais seriais mais populares é o PuTTY, que suporta várias conexões, incluindo SSH e Telnet, além de permitir comunicação serial. Outra opção bastante utilizada é o Tera Term, conhecido por sua simplicidade e funcionalidades úteis, como a gravação de sessões. No iOS, o aplicativo Get Console oferece suporte a conexões seriais, facilitando o acesso físico ao console serial de equipamentos de rede e outros dispositivos e a gestão de dispositivos em rede. No Linux, o Minicom é um emulador de terminal muito popular que permite comunicação serial e possui uma interface amigável, adequada para usuários mais avançados. O screen é outra ferramenta versátil que pode ser utilizada para comunicação serial e gerenciamento de sessões de terminal, sendo frequentemente empregada em *scripts* e por usuários que preferem uma **interface de linha de comando** (em inglês *Command Line Interface* – CLI). Para aqueles

que preferem uma abordagem visual, o Cutecom oferece uma interface gráfica que facilita a comunicação serial. A escolha do terminal serial ideal depende do sistema operacional e das necessidades específicas do usuário, mas todos esses aplicativos fornecem uma variedade de funcionalidades, como a configuração de parâmetros de comunicação, que são essenciais para garantir a troca correta de dados.

Embora os terminais seriais sejam essenciais em muitos sistemas de comunicação de dados, a **falta de padronização na codificação** pode causar problemas de interoperabilidade. Dispositivos e aplicativos podem usar diferentes formatos de codificação, como variantes do ASCII ou padrões mais complexos, como UTF-8, o que pode gerar erros na transmissão e interpretação dos dados. Quando a codificação de um aplicativo não é conhecida, é recomendado utilizar um formato simples e amplamente reconhecido, como o ASCII de 7 *bits*. O ASCII é uma base sólida, pois muitas codificações modernas são compatíveis com seus caracteres. Isso garante que os caracteres básicos, como letras, números e símbolos, sejam corretamente interpretados, reduzindo o risco de erros.

STM32H7A3: USART/UART

Para comunicação serial assíncrona, o STM32H7A3 é equipado com 5 USART, 5 UART e 1 LPUART (do inglês *Low-Power Universal Asynchronous Receiver Transmitter*). A USART integrado é um periférico altamente versátil, que permite a comunicação serial com dispositivos externos através de diversos protocolos. Diferente do USART, a UART não suporta comunicação síncrona nem o modo de comunicação SmartCard[1]. Portanto, enquanto descrevemos a USART de forma abrangente, é importante notar que a UART apenas carece dessas duas funcionalidades adicionais.

A USART oferece suporte para **comunicação assíncrona full-duplex** e opera no formato padrão **NRZ** (Não Retorno a Zero). É projetado para ser eficiente, utilizando DMA (Acesso Direto à Memória) para transferir grandes volumes de dados com mínima carga no processador. Adicionalmente, a USART possui duas estruturas FIFOs internas, uma para transmissão e outra para recepção, que armazenam dados temporariamente e melhoram a eficiência da comunicação. Além de suportar tanto comunicação síncrona quanto assíncrona, a USART inclui modos especializados de comunicação serial como **LIN** (do inglês, *Local Interconnection Network*), **Smartcard** (T=0 e T=1), **IrDA** (do inglês, *Infrared Data Association*) **SIR** (do inglês, *Serial Infrared*) e **Modbus** (RTU, do inglês *Remote Terminal Unit*, e ASCII). O periférico é ainda capaz de realizar comunicação **half-duplex** em configuração *single-wire*, com os pinos TX e RX conectados internamente. Também é compatível com comunicação multiprocessadora, permitindo que vários dispositivos compartilhem o mesmo barramento serial, facilitando a comunicação em sistemas mais complexos.

Para a **geração da taxa de *baud* e amostragem**, a USART possui um gerador de taxa de *baud* programável que abrange uma ampla gama de velocidades de comunicação, permitindo uma interface flexível com dispositivos que exigem diferentes taxas. Além disso, o periférico emprega **técnicas de superamostragem** (em inglês, *oversampling*) por 8 ou 16 vezes para melhorar a tolerância a variações no *clock* e aumentar a confiabilidade da comunicação.

[1] O modo de comunicação SmartCard no STM32H7A3 refere-se a uma funcionalidade específica dos microcontroladores da série STM32 que permite a comunicação com dispositivos SmartCard.

Quanto à **detecção e controle de erros**, a USART está equipado com mecanismos para verificar paridade, detectar sinais de *break* e identificar erros de enquadramento (em inglês, *framing error*), assegurando a integridade dos dados durante a transmissão. O periférico também suporta controle de fluxo de *hardware* através dos sinais de *handshaking* CTS (do inglês, *Clear to Send*) e RTS (do inglês, *Request to Send*), o que facilita a negociação do fluxo de dados com dispositivos externos e ajuda a prevenir a perda de dados.

O periférico é também equipado com avançados recursos de **gerenciamento de energia**, incluindo a capacidade de operar com um *clock* de baixo consumo e ativar a MCU a partir do modo de baixo consumo quando necessário. Além disso, a USART permite a inversão de dados binários para compatibilidade com diferentes padrões de sinalização e suporta a **detecção automática de taxa de *baud***, simplificando a configuração da comunicação.

O **bloco de interface DMA** (do inglês *Direct Memory Access*) permite a comunicação contínua usando DMA para transmissão e recepção de dados. Os *bits* `USART_CR3_DMAT` e `USART_CR3_DMAR` no registrador `USART_CR3` habilitam o modo DMA para transmissão e recepção, respectivamente.

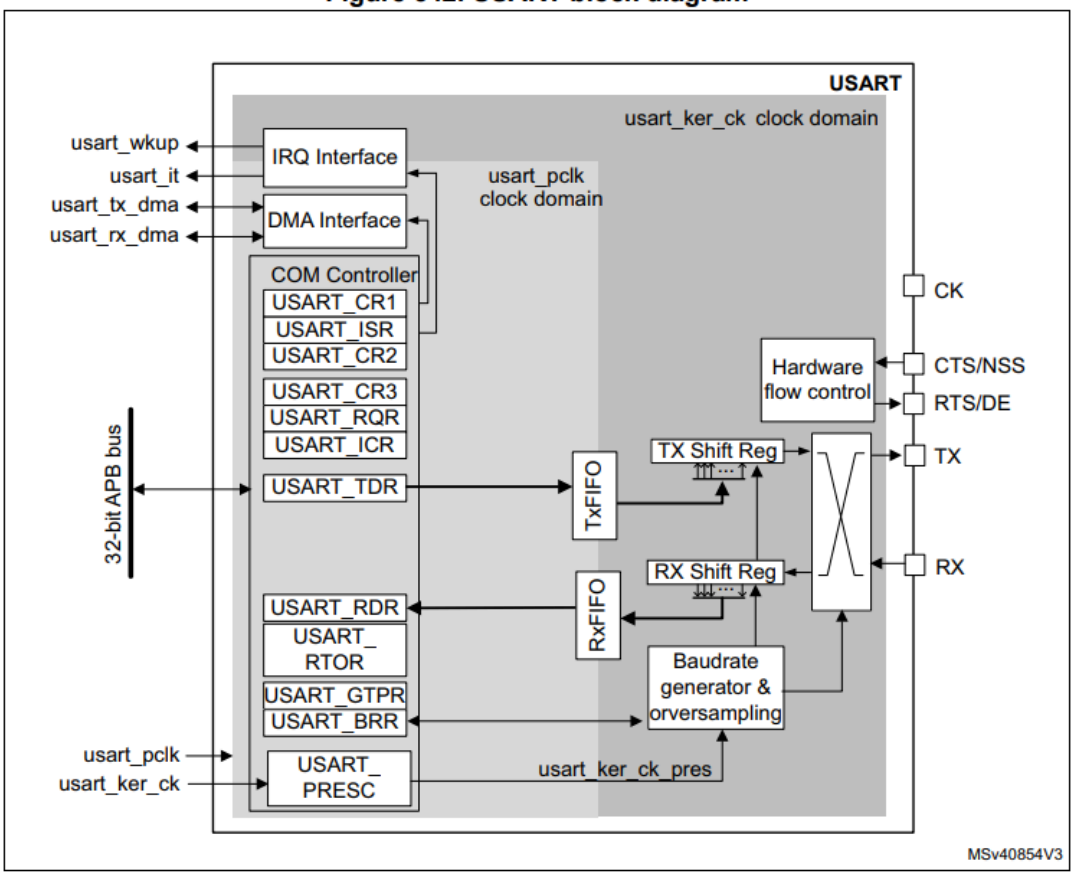
O diagrama de blocos da USART, mostrado na figura, ilustra detalhadamente os componentes e o fluxo de dados envolvidos na transmissão e recepção de dados seriais assíncronos. A **habilitação dos canais** de recepção (Rx) e transmissão (Tx) numa USART é independente, o que significa que cada canal pode ser ativado e desativado separadamente. A **habilitação do transmissor** (Tx) é controlada pelo *bit* `USART_CR1_TE` (*Transmitter Enable*) no registrador USART_CR1. Quando o *bit* `USART_CR1_TE` é configurado para 1, o transmissor é habilitado, enquanto a configuração para 0 desabilita o transmissor. Da mesma forma, a **habilitação do receptor** (Rx) é controlada pelo *bit* `USART_CR1_RE` (*Receiver Enable*) no mesmo registrador. Configurando `USART_CR1_RE` para 1, o receptor é habilitado, e para 0, o receptor é desabilitado.

Para assegurar que as configurações de ativação de recepção e transmissão no periférico USART foram efetivamente aplicadas, é recomendável verificar os *bits* `USART_ISR_REACK` e `USART_ISR_TEACK` antes de colocar o dispositivo em um modo de baixo consumo, como o modo de espera ou o modo de parada, para garantir que todas as operações de recepção e transmissão estejam concluídas e evitar a perda de dados. Da mesma forma, após uma reinicialização do USART, seja por *software* ou *hardware*, a verificação desses *bits* assegura que o periférico esteja completamente inicializado e pronto para operar. Além disso, após modificar os *bits* `USART_CR1_TE` e `USART_CR1_RE`, é crucial aguardar até que os *bits* `USART_ISR_REACK` e `USART_ISR_TEACK` sejam definidos para garantir que a mudança de configuração tenha sido aplicada corretamente. Em aplicações onde a precisão temporal é crítica, a checagem desses *bits* pode ser utilizada para sincronizar as ações do *software* com os eventos da USART, assegurando a precisão na comunicação.

Transmissão

A **transmissão** de dados começa com a escrita de informações no registrador USART_TDR, que serve como interface paralela entre o barramento interno e o registrador de deslocamento de saída. Se a verificação de paridade estiver ativada, o *bit* mais significativo (MSB) no USART_TDR é substituído pelo *bit* de paridade calculado pelo periférico.

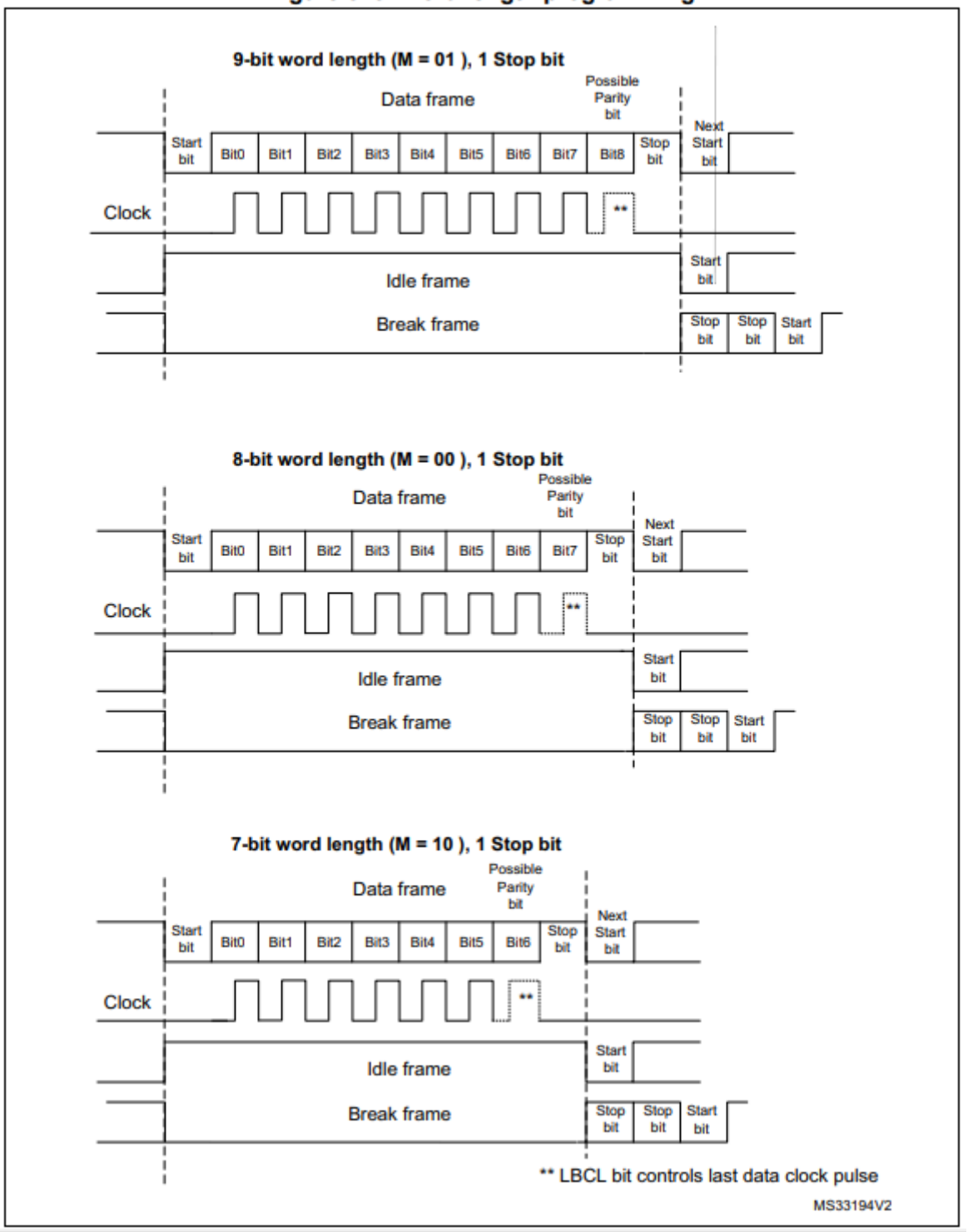
Figure 542. USART block diagram



O controlador de comunicação (COM) lida aspectos importantes da transmissão, como o **tamanho da palavra**, a **paridade** e o **número de *stop bits***, com base nas configurações do registrador USART_CR1. Especificamente, o tamanho da palavra é determinado pelos *bits* USART_CR1_M0 e USART_CR1_M1, a paridade é configurada com o *bit* USART_CR1_PCE para habilitação e o *bit* USART_CR1_PS para seleção do tipo de paridade, enquanto o número de *stop bits* é definido pelos *bits* USART_CR1_STOP[1:0].

A definição se um USART opera em modo assíncrono ou síncrono é feita através da configuração do *bit* USART_CR2_CLKEN. No **modo assíncrono** (USART_CR2_CLKEN=0), não há um sinal de *clock* dedicado para sincronizar a transmissão e recepção de dados. A temporização é baseada em um acordo prévio entre o transmissor e o receptor sobre a taxa de transmissão (*baud rate*). O início de cada caractere é sinalizado por um *bit* de *start*, e o final por um ou mais *bits* de *stop*, como mostra a figura a seguir.

Figure 543. Word length programming



Note que, além dos **quadros de dados** (em inglês, *data frame*), que são usados para transmitir informações efetivas em um sistema de comunicação serial, existem outros dois tipos de quadros que desempenham papéis importantes na gestão da linha de transmissão: o **quadro ocioso** (em inglês, *idle frame*) e o **quadro de quebra** (em inglês, *break frame*). O ***idle frame*** é um período em que a linha de transmissão permanece inativa, geralmente mantida em um estado lógico alto (“1”). Este quadro indica que não há dados sendo transmitidos no momento. O *idle frame* inclui os *bits* de *stop*, e sua detecção pode ser usada

para gerar uma interrupção no sistema, sinalizando ao *software* que a linha está pronta para uma nova transmissão ou recepção de dados.

Por outro lado, o ***break frame*** é um sinal de sinalização que força a linha de transmissão a um estado lógico baixo (0) por um período fixo. Esse quadro é utilizado para indicar uma condição excepcional ou para chamar a atenção do receptor. Normalmente, o *break frame* é mais longo do que um caractere de dados padrão e é seguido por um ou mais *bits* de *stop*. No receptor, um *break frame* é interpretado como um erro de enquadramento, conhecido como *framing error*. Além disso, o *break frame* pode ser utilizado em conjunto com o padrão RS-485 para sinalizar o término da transmissão. O comprimento do *break frame* pode ser configurado através do *bit* `USART_CR1_M` no registrador `USART_CR1` e, em alguns casos, como no modo LIN (*Local Interconnect Network*), é utilizado para sincronizar os nós na rede. A detecção de um *break frame* também pode gerar uma interrupção, permitindo que o sistema responda adequadamente a esse evento.

No **modo síncrono** (`USART_CR2_CLKEN`=1**)**, há um sinal de *clock* dedicado (CK) para sincronizar a transmissão e recepção de dados. O pino CK pode ser configurado como entrada ou saída, dependendo se a USART atua como *slave* ou *master*, respectivamente. A polaridade (CPOL) e a fase (CPHA) do sinal de *clock* podem ser configuradas através do registrador `USART_CR2` para garantir a compatibilidade com diferentes dispositivos. O modo síncrono **não requer *bits* de *start* e *stop***. Nenhum pulso de *clock* é enviado para o pino CK durante esses *bits*, pois a sincronização é garantida pelo sinal de *clock* compartilhado entre o transmissor e o receptor.

A ordem de transmissão dos *bits* (*LSB first* ou *MSB first*) é definida no registrador de controle `USART_CR2` do periférico USART. Especificamente, o *bit* 19 deste registrador é o `MSBFIRST` (do inglês *Most significant bit first*)

- Se o *bit* MSBFIRST estiver em 0, os dados são transmitidos/recebidos com o *bit* 0 primeiro, seguindo o *bit* de *start* (esta é a configuração padrão).
- Se o *bit* MSBFIRST estiver em 1, os dados são transmitidos/recebidos com o *bit* mais significativo (MSB) primeiro, seguindo o *bit* de start.

Recepção

Na **recepção**, os dados seriais no pino RX (*Receive Data Input*) são amostrados pelo bloco de superamostragem. Em seguida, o bloco "RX Shift Reg" (*Receive Shift Register*) converte o fluxo serial de *bits* em dados paralelos, que são armazenados no registrador `USART_RDR`. Este registrador fornece a interface paralela entre o registrador de deslocamento de entrada e o barramento interno, e também armazena o *bit* de paridade recebido quando a verificação de paridade está ativada. O controlador de comunicação (COM) verifica a integridade dos dados recebidos examinando possíveis erros de paridade (indicado pelo *bit* `USART_ISR_PE`),

erros de enquadramento (indicado pelo *bit* USART_ISR_FE) e erros de ruído (indicado pelo *bit* USART_ISR_NE), todos pertencentes ao registrador de estado USART_ISR. Além disso, o bloco "Hardware Flow Control" gerencia o sinal CTS, que indica quando o dispositivo está pronto para receber novos dados.

No modo RS-485, o sinal USART_CR1_DE* (*Driver Enable*) controla a ativação do transmissor externo, e sua ativação é gerenciada pelo bloco **"**Hardware Flow Control**"**. Este bloco utiliza os sinais de *handshaking* CTS (do inglês, *Clear To Send*) e RTS (do inglês, *Request To Send*) para gerenciar o fluxo de dados entre os dispositivos. A habilitação do controle de fluxo RTS e CTS é configurada, respectivamente, pelos *bits* USART_CR3_RTSE e USART_CR3_CTSE no registrador USART_CR3.

Registradores de estado e interrupções

O registrador USART_ISR (*USART Interrupt and Status Register*) contém vários *bits* de estado que indicam o **estado atual do transmissor e do receptor**. O *bit* USART_ISR_RXNE (*Read Data Register Not Empty*) no USART_ISR indica se o receptor recebeu um novo dado. Quando USART_ISR_RXNE está em "1", significa que há dados disponíveis para leitura no registrador USART_RDR; quando USART_ISR_RXNE está em 0, o registrador de dados está vazio. O *bit* USART_ISR_TXE (*Transmit Data Register Empty*) indica se o transmissor está pronto para transmitir um novo dado. Quando USART_ISR_TXE está em 1, o registrador USART_TDR está vazio e pronto para receber um novo dado para transmissão; quando USART_ISR_TXE está em 0, o registrador de dados de transmissão ainda contém dados a serem transmitidos. O *bit* USART_ISR_TC (*Transmission Complete*) no USART_ISR indica que a transmissão do último dado foi concluída e a linha de transmissão está inativa (em inglês, *idle*). Quando USART_ISR_TC está em 1, a transmissão está completa, e quando USART_ISR_TC está em 0, a transmissão ainda está em andamento. A figura a seguir associa os eventos de interrupção aos diferentes *bits* de habilitação de interrupção, *bits* de estado e técnicas para limpá-los.

Table 396. USART interrupt requests

Interrupt vector	Interrupt event	Event flag	Enable Control bit	Interrupt clear method	Exit from Sleep mode	Exit from Stop(1) modes	Exit from Standby mode
USART or UART	Transmit data register empty	TXE	TXEIE	Write TDR	Yes	No	No
	Transmit FIFO not Full	TXFNF	TXFNFIE	TXFIFO full		No	
	Transmit FIFO Empty	TXFE	TXFEIE	Write TDR or write 1 in TXFRQ		Yes	
	Transmit FIFO threshold reached	TXFT	TXFTIE	Write TDR		Yes	
	CTS interrupt	CTSIF	CTSIE	Write 1 in CTSCF		No	
	Transmission Complete	TC	TCIE	Write TDR or write 1 in TCCF		No	
	Transmission Complete Before Guard Time	TCBGT	TCBGTIE	Write TDR or write 1 in TCBGT		No	
USART or UART	Receive data register not empty (data ready to be read)	RXNE	RXNEIE	Read RDR or write 1 in RXFRQ	Yes	Yes	No
	Receive FIFO Not Empty	RXFNE	RXFNEIE	Read RDR until RXFIFO empty or write 1 in RXFRQ		Yes	
	Receive FIFO Full	RXFF(2)	RXFFIE	Read RDR		Yes	
	Receive FIFO threshold reached	RXFT	RXFTIE	Read RDR		Yes	
	Overrun error detected	ORE	RXNEIE/ RXFNEIE	Write 1 in ORECF		No	
	Idle line detected	IDLE	IDLEIE	Write 1 in IDLECF		No	
	Parity error	PE	PEIE	Write 1 in PECF		No	
	LIN break	LBDF	LBDIE	Write 1 in LBDCF		No	
	Noise error in multibuffer communication	NE	EIE	Write 1 in NFCF		No	
	Overrun error in multibuffer communication	ORE(3)		Write 1 in ORECF		No	
	Framing Error in multibuffer communication	FE		Write 1 in FECF		No	
	Character match	CMF	CMIE	Write 1 in CMCF		No	
	Receiver timeout	RTOF	RTOFIE	Write 1 in RTOCCF		No	
	End of Block	EOBF	EOBIE	Write 1 in EOBCF		No	
	Wake-up from low-power mode	WUF	WUFIE	Write 1 in WUC		Yes	
	SPI slave underrun error	UDR	EIE	Write 1 in UDRCF		No	

1. The USART can wake up the device from Stop mode only if the peripheral instance supports the wake-up from Stop mode feature. Refer to *Section 53.4: USART implementation* for the list of supported Stop modes.

O **bloco de interface IRQ** gerencia as interrupções geradas pelo USART, como interrupções de recepção de dados, transmissão de dados e erros. Os registradores `USART_CR1`, `USART_CR2` e `USART_CR3` contêm *bits* para configurar e habilitar essas interrupções associadas à USART. Para gerenciamento das interrupções, deve-se configurar o NVIC. Isso envolve ativar as interrupções apropriadas nos registradores `NVIC_ISERm` do NVIC, ajustar suas prioridades nos registradores `NVIC_IPRn` e assegurar que as interrupções sejam corretamente tratadas pelo sistema por meio das rotinas de serviço de serviço (ISR) implementadas. Apesar da USART/UART poder gerar múltiplos eventos de interrupção, há apenas uma única linha de interrupção (IRQ) associada a cada módulo, como mostra a figura

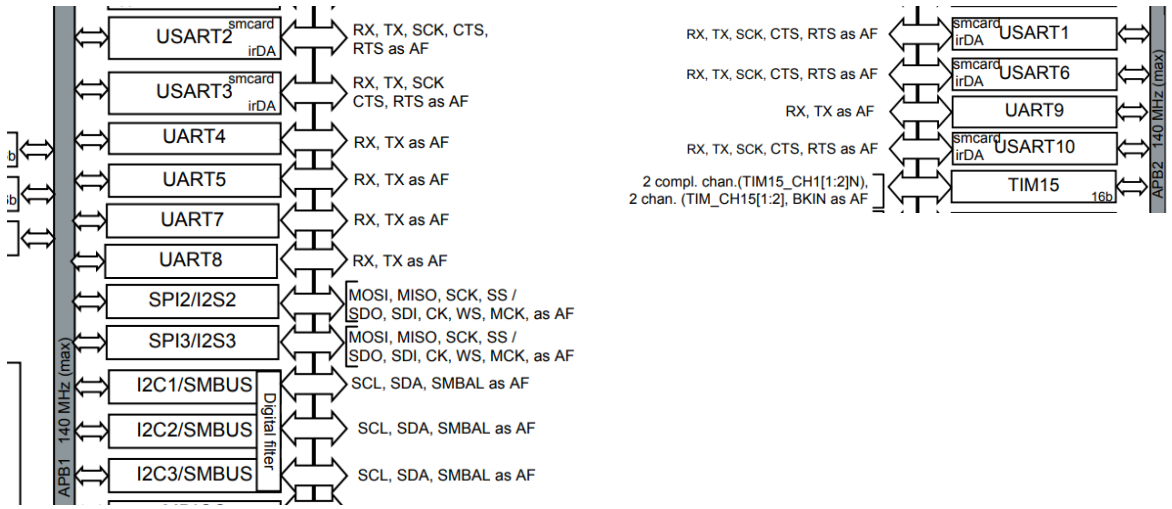
a seguir. Isso significa que quando ocorre uma interrupção, o processador é notificado através dessa **única IRQ**, e o controlador de interrupções precisa identificar qual evento específico causou a interrupção. Portanto, dentro da rotina de serviço de interrupção (ISR), é necessário verificar as bandeiras (em inglês, *flags*) ou registradores do módulo UART/USART para determinar qual evento acionou a interrupção. Com base nessa verificação, a ISR pode então processar o evento corretamente, seja para lidar com a transferência de dados ou para tratar de erros.

uart4_gbl_it exti_uart4_wkup	59	52	UART4	UART4 global interrupt	0x0000 0110
uart5_gbl_it exti_uart5_wkup	60	53	UART5	UART5 global interrupt	0x0000 0114
uart7_gbl_it exti_uart7_wkup	89	82	UART7	UART7 global interrupt	0x0000 0188
uart8_gbl_it exti_uart8_wkup	90	83	UART8	UART8 global interrupt	0x0000 018C
uart9_it exti_uart9_wkup	147	140	UART9	UART9 global interrupt	0x0000 0270
usart1_gbl_it exti_usart1_wkup	44	37	USART1	USART1 global interrupt	0x0000 00D4
usart2_gbl_it exti_usart2_wkup	45	38	USART2	USART2 global interrupt	0x0000 00D8
usart3_gbl_it exti_usart3_wkup	46	39	USART3	USART3 global interrupt	0x0000 00DC
usart6_gbl_it exti_usart6_wkup	78	71	USART6	USART6 global interrupt USART6 wakeup interrupt	0x0000 015C
usart10_it exti_usart10_wkup	148	141	USART10	USART10 global interrupt	0x0000 0274

Por exemplo, para habilitar a linha de requisição de interrupção (IRQ) associada à interrupção por recepção de dados (RXNE) de USART3, deve-se ajustar o *bit* `USART3_CR1_RXNEIE` no registro de controle `USART3_CR1`, habilitar a IRQ39 no NVIC definindo o *bit* (39&0x1f) no registrador `NVIC_ISERm`, onde m = 39 >> 5 = 1, e ajustando a prioridade no *byte* (39&0x3) do registrador `NVIC_IPRn`, onde n = 39 >> 2 = 9.

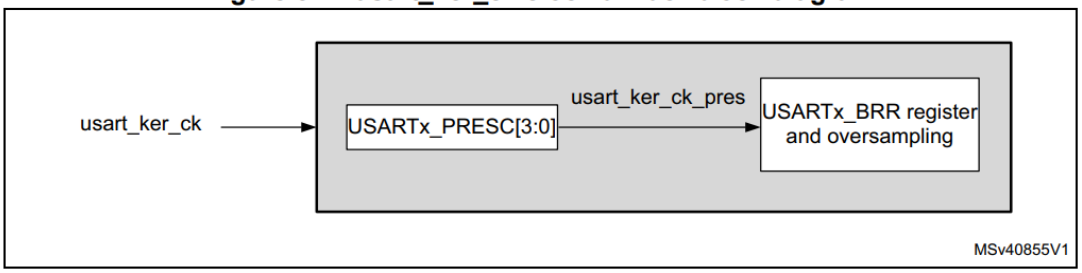
Sinais de relógio

O diagrama mostra ainda **dois domínios de *clock***: `usart_pclk` e `usart_ker_ck`. O `usart_pclk` é o *clock* da interface do barramento periférico (APB1 e APB2), responsável por alimentar acessos aos registradores da USART, como mostra o recorte da figura no *Datasheet*. O `usart_ker_ck` é a fonte de *clock* para a própria USART, controlando seu funcionamento interno. O *baud rate* da USART é gerado com base no valor deste *clock*. Este *clock* é independente do `usart_pclk`. Em situações onde o recurso de duplo domínio de *clock* e a ativação a partir de modos de baixo consumo são suportados, a fonte de *clock* `usart_ker_ck` pode ser configurada no módulo RCC. Caso contrário, o `usart_ker_ck` será o mesmo que o `usart_pclk`.



A frequência do sinal de relógio `usart_ker_ck` pode ser dividida por um fator programável, definido no registrador `USART_PRESC`. Os valores possíveis para o *prescaler* variam de 1 (sem divisão) a 256.

Figure 547. usart_ker_ck clock divider block diagram



O bloco "`Baud Rate Generator & Oversampling`" é responsável por gerar o sinal de *clock* para a transmissão e recepção dos dados. A taxa de transmissão é ajustada programando o registrador `USART_BRR` (*Baud Rate Register)*, e o método de amostragem (em inglês, *oversampling*) por 8 ou 16 é selecionado pelo *bit* `USART_CR1_OVER`. Durante a transmissão, o bloco "TX Shift Reg" (*Transmit Shift Register*) converte os dados paralelos do

USART_TDR em um fluxo serial de *bits*, que é então transmitido através do pino TX (*Transmit Data Output*).
A taxa de transmissão/recepção numa USART é determinada pela combinação do *prescaler* do *clock* (USARTx_PRESC) e do valor do registrador de taxa de transmissão (USARTx_BRR). A fórmula para calcular esta taxa varia de acordo com o modo de operação da USART, seja *oversampling* por 8 ou 16. No caso do *oversampling* por 16, por exemplo, para uma taxa de transmissão desejada de 9600 *bauds* e uma frequência do *clock* principal (usart_ker_ck_pres) de 8 MHz, o cálculo é feito da seguinte forma:

$$USARTDIV = \frac{usart\_ker\_ck\_pres}{Tx/Rx\ \ baud} = \frac{8000000}{9600} \approx 833{,}33$$

Arredondando USARTDIV para o inteiro maior mais próximo, obtemos 834, que em hexadecimal é 0x342. Assim, o campo USARTx_BRR_USARTDIV do registrador USARTx_BRR deve ser configurado para 0x342. Para o caso de *oversampling* por 8, o divisor pode ser obtido com a equação:

$$USARTDIV = \frac{2 \times usart\_ker\_ck\_pres}{Tx/Rx\ \ baud}$$

O microcontrolador, para garantir a correta recepção dos dados seriais, amostra o sinal recebido várias vezes dentro de cada baud. No caso do STM32H7A3, ele faz isso 8 ou 16 vezes. Essa sobreamostragem ajuda a reduzir erros causados por ruídos e variações no sinal. Ela é feita internamente. É transparente para o dispositivo que se comunica com o microcontrolador.

Cabe ressaltar aqui que **existe uma ordem recomendada para configurar os registradores USART**, especialmente o registrador de taxa de transmissão USART_BRR, para garantir uma inicialização e operação corretas tanto na transmissão quanto na recepção. Este registrador deve ser configurado com o módulo USART desabilitado, depois da configuração do tamanho de cada caractere e superamostragem e antes da configuração da quantidade de *bits* de parada.

Alocação de pinos

A atribuição de pinos para os canais TX e RX de cada módulo USARTx ou UARTx não é definida de forma fixa. Em vez disso, o projetista pode configurar os pinos a serem utilizados consultando as tabelas de funções alternativas (AF) para cada porta GPIO. Por exemplo, a Tabela 8 mostra que o pino PA9 pode ser configurado como USART1_TX utilizando a função alternativa AF7. Cada módulo USART/UART possui um conjunto específico de funções alternativas, permitindo flexibilidade na escolha dos pinos a serem usados para comunicação serial.

Bufferização de dados

Uma das características importantes na UART é o modo FIFO, que utiliza *buffers* para o envio e recebimento de dados, tornando a comunicação mais eficiente. A UART possui dois FIFOs: um para transmissão, conhecido como TXFIFO, e outro para recepção, denominado RXFIFO. O TXFIFO armazena os dados a serem transmitidos, permitindo que o processador envie um bloco de dados para a UART e continue com outras tarefas enquanto a UART processa a transmissão. Por sua vez, o RXFIFO armazena os dados recebidos, possibilitando que o processador leia um bloco de dados da UART de uma só vez.

Para habilitar o modo FIFO, é necessário configurar o *bit* `USART_CR1_FIFOEN` no registrador `USART_CR1`. Os *bits* `USART_CR1_M1` e `USART_CR1_M0` definem o tamanho da palavra de dados (7, 8 ou 9 *bits*). Cada FIFO possui um tamanho fixo, que varia conforme a implementação da USART; por exemplo, no STM32H7A3_7B3, os FIFOs têm 16 posições. Além disso, é possível configurar os níveis de preenchimento dos FIFOs, conhecidos como limiares (em inglês, *threshold*), que acionam interrupções. Esses limiares são configurados nos campos `USART_CR3_RXFTCFG` e `USART_CR3_TXFTCFG` do registrador `USART_CR3`, permitindo que o processador seja notificado quando um determinado número de *bytes* foi transmitido ou recebido. O *bit* `USART_CR3_TXFTIE` habilita a interrupção quando o TXFIFO atinge o limiar configurado, enquanto o `USART_CR3_RXFTIE` faz o mesmo para o RXFIFO. Além disso, os *bits* `USART_CR3_DMAT` e `USART_CR3_DMAR` habilitam o DMA para transmissão e recepção, respectivamente. Por fim, no registrador `USART_ISR`, o *bit* `USART_ISR_TXFE` indica que o TXFIFO está vazio, o `USART_ISR_RXFNE` sinaliza que o RXFIFO não está vazio e o `USART_ISR_RXFF` mostra que o RXFIFO está cheio.

PROCESSAMENTO DE *STRINGS* EM C

Na comunicação serial assíncrona, as mensagens trocadas entre microcontroladores, sensores, módulos Bluetooth, GPS ou terminais seriais são, quase sempre, sequências de *bytes* ASCII, representando letras, números, comandos e respostas. Isso significa que, na prática, grande parte da comunicação se resume ao envio e à recepção de *strings*. Por isso, aprender a manipular *strings* em C, saber como armazená-las, percorrê-las, compará-las, extraí-las, formatá-las e convertê-las, não é só uma habilidade indispensável mas uma necessidade prática para quem trabalha com comunicação serial.

A linguagem C oferece um conjunto robusto de funções nativas para trabalhar com *strings*. ***Strings*** em C são representadas como arranjos/vetores de caracteres (tipo de dado `char`), onde cada elemento do vetor é um caractere da *string*. O terminador nulo '\0' é utilizado para marcar o fim da *string*. O mecanismo de sincronização *Start-Stop* é assíncrono e não possui um delimitador específico para o início e o fim das mensagens além dos próprios *bits* de controle. Ao utilizar *strings* em C, a convenção do terminador nulo ajuda a definir claramente o final das mensagens recebidas, tornando a análise e o processamento mais diretos. O

terminador nulo tem também um impacto significativo nas funções nativas de manipulação de *strings* em C. Sem esse terminador, as funções de *string* podem continuar lendo além do final da *string*, resultando em comportamentos indefinidos e possíveis falhas de segurança. Algumas das funções mais comuns e suas interações com o terminador nulo incluem:

`strlen(const char *str)`: Calcula o comprimento de uma *string*, parando na primeira ocorrência do terminador nulo. Se o terminador não estiver presente, pode levar a leitura de memória inválida.

`strcpy(char *dest, const char *src)`: Copia uma *string* de `src` para `dest`, incluindo o terminador nulo. Se a `src` não estiver corretamente terminada, a cópia pode ser truncada ou levar a corrupção de memória.

`strcat(char *dest, const char *src)`: Anexa a *string* `src` ao final da *string* `dest`, adicionando o terminador nulo ao final da nova string concatenada. A ausência de um terminador nulo em `src` pode causar anexações incorretas.

`strcmp(const char *str1, const char *str2)`: Compara duas *strings* e retorna um valor que indica sua relação lexicográfica. O terminador nulo é utilizado para determinar o final das *strings* a serem comparadas.

`strchr(const char *str, int c)`: Localiza a primeira ocorrência do caractere `c` em `str`, retornando um ponteiro para essa posição ou `NULL` se o caractere não for encontrado. O terminador nulo define o final da busca.

`strstr(const char *haystack, const char *needle)`: Localiza a primeira ocorrência da substring `needle` em `haystack`. A ausência do terminador nulo na *substring* pode levar a falhas na busca.

Os terminais são comumente associados a uma interface de linha de comando (*command-line interface*, CLI), devido à sua eficácia e flexibilidade na interação com sistemas operacionais. A CLI é independente de plataforma e demanda menos recursos do sistema. Uma linha de comando consiste em um comando seguido de seus argumentos, separados por espaços, e é executada quando o usuário pressiona “enter”. Para implementar uma interface de linha de comando para um Terminal serial, é necessário processar os caracteres digitados pelo usuário em linhas distintas.

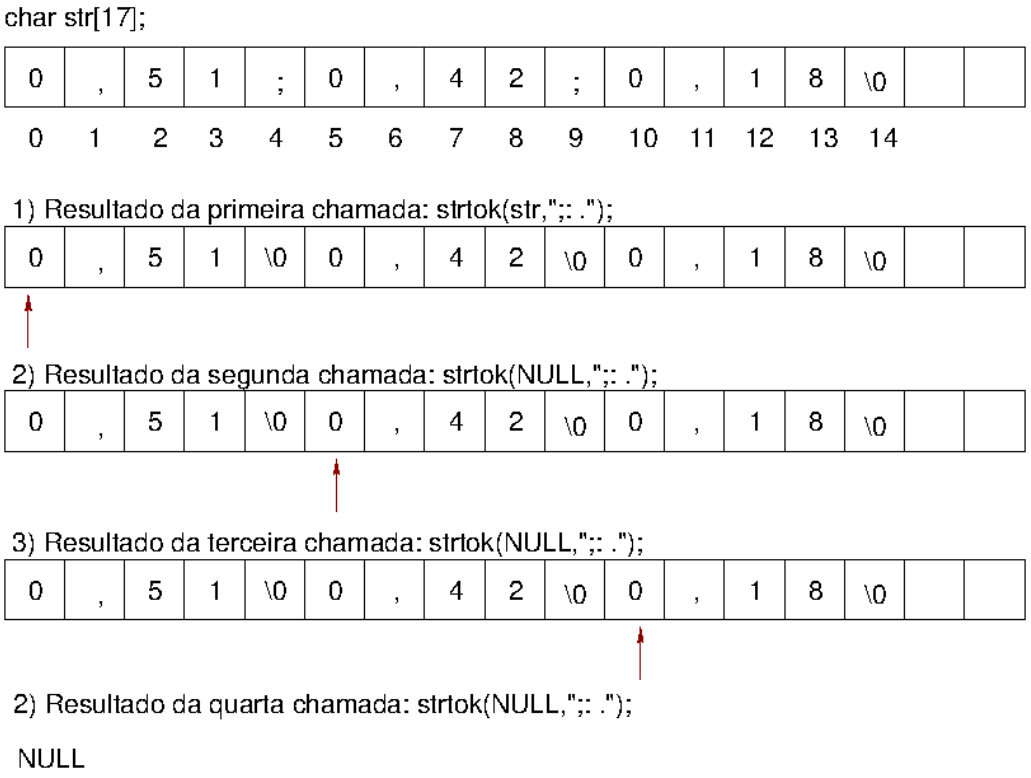
Cada pacote recebido por um módulo USART/UART contém um quadro de dados com 7 a 9 bits. Em C, esse quadro é geralmente representado pelo tipo de dado `char`, que suporta até 8 *bits*. Quando uma sequência de quadros de dados, correspondendo a uma linha de caracteres no terminal, é recebida pelo microcontrolador, ela pode ser armazenada como uma *string*. Para que essa sequência seja tratada corretamente como uma *string* em C, é necessário substituir o caractere de controle ‘\r’ (carriage return), correspondente à tecla “Enter”, pelo terminador nulo (‘\0’).

Antes de processar um comando presente em uma linha recebida, é necessário extrair os tokens ou “unidades de informação”, que incluem o comando em si e seus argumentos. Esses tokens são geralmente separados por delimitadores, como vírgulas (`,`), pontos (`.`), ponto e vírgula (`;`) ou espaços em branco ( ).

A função **`strtok`** da biblioteca padrão de C é uma ferramenta eficaz para extrair tokens de uma linha de caracteres recebida no terminal. A função `strtok` opera com dois parâmetros: o primeiro é a própria linha de caracteres (`str`), e o segundo é uma *string* contendo os delimitadores que separam os tokens (`lista_delimitadores`).

Na primeira chamada, `strtok` percorre a linha de caracteres, substituindo os delimitadores encontrados por terminadores nulos (‘\0’), que indicam o fim de cada token. Em seguida, retorna o endereço da primeira *sub-string* encontrada. Nas chamadas subsequentes, quando o primeiro argumento é passado como `NULL`, a função continua a busca pela próxima *sub-string*, retornando o seu endereço inicial. Esse processo se repete até que `strtok` retorne um ponteiro `NULL`, indicando que todos os tokens foram extraídos ou que não há mais caracteres para analisar na *string* original.

O resultado final é uma série de endereços correspondentes às *sub-strings* que compõem a linha de caracteres original.



É importante observar que a função `strtok` modifica o conteúdo da string original (`str`) durante sua execução, substituindo delimitadores por terminadores nulos (`'\0'`). Portanto, se houver a necessidade de preservar o conteúdo original da *string*, é aconselhável trabalhar com uma cópia da *string* em vez da original. Dessa forma, podemos evitar alterações indesejadas nos dados originais e garante a integridade das informações contidas na *string* original.