DISCIPLINA EA701
Introdução aos Sistemas Embarcados

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

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

FEEC / UNICAMP

Revisado em setembro de 2024

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.

Neste contexto, exploraremos neste roteiro a comunicação serial assíncrona, uma variante da comunicação serial que não requer um sinal de relógio compartilhado entre os sistemas. Em vez disso, os dispositivos devem ter seus relógios internos sincronizados, e a comunicação é realizada com uma taxa de transmissão e formato de sinal predefinidos. Utilizaremos o periférico USART3 do microcontrolador STM32H7A3ZIT6-Q. Este módulo permite a comunicação *full-duplex*, significando que dados podem ser transmitidos e recebidos

simultaneamente através das linhas TX e RX, que devem ser conectadas cruzadamente entre os dispositivos. **A linha TX de um lado deve ser 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) **Comunicação com um Celular**, demonstrando como conectar e comunicar o microcontrolador com um dispositivo móvel via Bluetooth ou outro módulo serial; 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 e UART4, 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, vamos explorar como utilizar o ST-LINK, presente na placa NUCLEO, juntamente com o Terminal integrado ao STM32CubeIDE e o módulo USART3 disponível no microcontrolador STM32H7A3, para realizar essa tarefa de forma eficiente e direta. Além de ser responsável pela depuração e programação do microcontrolador através de interfaces como SWD ou JTAG, o ST-LINK emula uma porta serial via USB no computador, permitindo que a comunicação entre um terminal serial do computador e o módulo USART3 do microcontrolador seja realizada através da porta COM emulada (**o número da porta deve ser verificado em cada bancada de trabalho**). O Terminal integrado permite monitorar a troca de dados, e o módulo USART3 gerencia a transmissão dos caracteres do teclado para o terminal. Com a combinação desses recursos, podemos simplificar o processo e implementar a funcionalidade de eco de forma prática e eficaz.

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

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 ativá-lo e configurá-lo através do editor gráfico, optamos por fazê-lo diretamente através das instruções que escreveremos, baseadas na 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 UART 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;
```

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
```

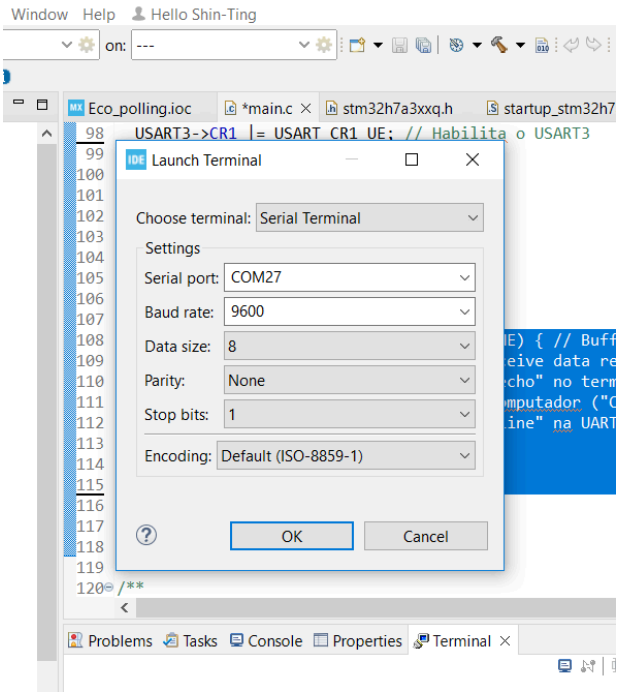
Observe que foi seguida a ordem de configuração de transmissão e de recepção descritas no Manual de Referência.

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".

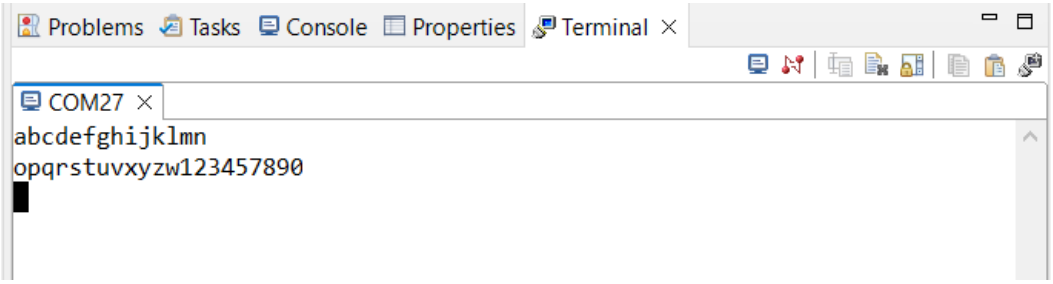
```
if(USART3->ISR & USART_ISR_RXNE_RXFNE) { // Buffer nao esta vazio
  c = (uint8_t)USART3->RDR; // Receive data register
  while (!(USART3->ISR & USART_ISR_TXE_TXFNF)){} // aguarda TX vazio
  USART3->TDR = c;
  if(c == 0x0D) {     "Enter" no computador ("Carriage Return")
    while (!(USART3->ISR & 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*. 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 uso de *polling* para controlar o fluxo de transferência de caracteres em uma comunicação serial assíncrona pode apresentar várias desvantagens. No polling, o processador verifica continuamente o estado do periférico para saber se uma condição específica foi atendida, como se o registrador receptor está cheio (com a instrução “`if(USART3->ISR & USART_ISR_RXNE_RXFNE) {}`” no projeto anterior) ou se o registrador de transmissão está vazio (com a instrução “`while (!(USART3->ISR & USART_ISR_TXE_TXFNF)){}`” no projeto anterior). Esse método pode levar a um uso ineficiente dos ciclos do processador, pois ele fica ocupado com essas verificações constantes e não pode realizar outras tarefas. Além disso, o *polling* reduz a responsividade do sistema, já que o processador está envolvido com essas verificações e não pode responder prontamente a outras demandas. Outro ponto crítico é o aumento do consumo de energia, especialmente em sistemas embarcados que necessitam de operações eficientes em termos de consumo. Além disso, o código baseado em *polling* pode

se tornar complexo e difícil de manter, particularmente quando há múltiplas condições e fluxos de controle envolvidos.

Vimos no Roteiro 3 que uma alternativa ao *polling* é o uso de interrupções. Você consegue imaginar como um módulo USART/UART 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/UART 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 valor inteiro mais próximo, 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. 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
//Defina a prioridade nos bits [7:5] do byte de prioridade do NVIC_IPRn,
//onde n = 39 >> 2
NVIC->IP[39]=(uint8_t)((0x2U << (8U-__NVIC_PRIO_BITS)) &
(uint32_t)0xFFUL);
//Define o bit 39%32=(39&0x1F) do NVIC_ISERm, onde m = 39/32 = 39>>5.
NVIC->ISER[(((uint32_t)39) >> 5UL)] = (uint32_t)(1UL << (((uint32_t)39)
& 0x1FUL));
```

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
de 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 que 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 definida, indicando que o registrador `USART->TDR` está pronto para receber um novo caractere. Se o caractere armazenado não

for o código ASCII para "*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 alguns tenham notado que, em certos momentos, o caractere digitado é ecoado mais de uma vez. Para resolver esse problema e melhorar o sequenciamento da execução entre recepção e transmissão, vamos adicionar uma *flag* adicional na ISR. Para implementar essa correção, primeiro, pause o programa. Em seguida, substitua as instruções atuais da ISR pelas instruções revisadas fornecidas. Após fazer essa alteração, regere novamente o código executável (“Terminate and Relaunch”) e retome (“Resume”) a execução do programa.

```
	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
		}
```

```
}
```

Você consegue identificar por quê a utilização de uma variável de controle, como `flag`, é essencial para aprimorar o sequenciamento entre as operações de recepção e transmissão?

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 módulo *bluetooth* e interrupções

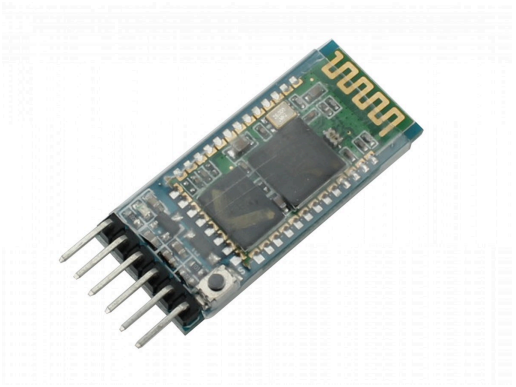
Já imaginou controlar o microcontrolador diretamente pelo seu celular? Com essa abordagem, você poderá ajustar a cor do LED RGB e o motor, como vimos no Roteiro 6, apenas enviando comandos do celular, substituindo a necessidade de pressionar um botão azul. Para possibilitar esse controle, utilizaremos um relé de comunicação serial.

Um **relé de comunicação serial** é um bloco de circuito que possui duas interfaces seriais. Ele redireciona dados da interface “A” para a interface “B” e vice-versa. Nos dois projetos anteriores deste Roteiro, já utilizamos um desses relés. A interface ST-LINK, por exemplo, emula um dispositivo USB CDC (Communication Device Class), o que faz com que o computador reconheça o canal USB como uma porta serial (COM). No lado oposto do ST-LINK, uma UART está conectada diretamente à USART3 do microcontrolador, permitindo que os dados digitados no terminal do computador sejam recebidos pela USART3.

Neste projeto, vamos incorporar dois novos relés. O primeiro é um módulo dedicado que redireciona dados entre uma UART e um rádio Bluetooth. Quando o rádio está pareado com um dispositivo, ele aparece como uma “Porta Serial *Bluetooth*”. O segundo relé será implementado em *software* no microcontrolador, permitindo que o que é digitado no terminal do computador seja transmitido para o dispositivo *Bluetooth* e vice-versa. Além disso, vamos focar apenas em interrupções para a recepção de caracteres. Para a transmissão, verificaremos se o *buffer* está disponível. Como a velocidade de comunicação nos dois lados do relé do microcontrolador é a mesma, o risco de perda de caracteres é praticamente nulo.

Para analisar os resultados do projeto integralmente, é necessário estabelecer a conexão de um celular com o módulo Bluetooth HC06 instalado na placa Auxiliar. Este módulo está configurado para uma *baud rate* de 9600, 8 *bits* de dados, sem paridade, e 1 *stop bit*. Primeiramente, verifique se o seu celular consegue se conectar ao módulo HC06. Para isso, mantenha a placa NUCLEO desconectada da USB e remova temporariamente o módulo

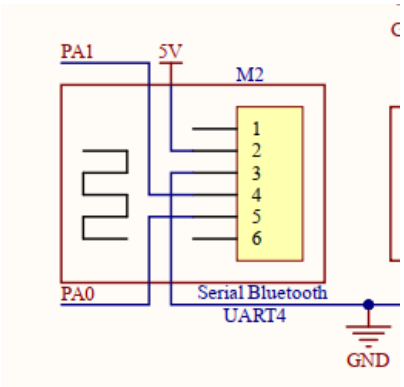
HC06 da placa. Anote o nome do módulo que está na etiqueta na parte inferior antes de recolocá-lo no lugar. Em seguida, reconecte a placa NUCLEO à USB do computador e ative o Bluetooth do seu celular seguindo as instruções fornecidas pelo suporte do Google (para iPhone, siga o procedimento do suporte da Apple).



Certifique-se de que o nome exclusivo do módulo HC06 está visível na lista de "Dispositivos pareados". Se o módulo estiver na lista, despareie-o e aguarde que ele reapareça na lista de "Dispositivos disponíveis". Em seguida, repareie o módulo clicando no nome exclusivo do HC06. Um "Pedido de Pareamento Bluetooth" será exibido, solicitando a inserção de um PIN. Utilize o PIN "1234" (ou "0000" se o primeiro não funcionar). Se você encontrar dificuldades para parear o dispositivo com o HC06, siga os passos abaixo, ignorando as seções destacadas em laranja. Nesse caso, habilitaremos apenas o canal TX de UART4 para que possamos visualizar os sinais do protocolo Start-Stop gerados por esse canal no analisador lógico. Opcionalmente, você pode implementar todas as instruções, mas algumas delas poderão não funcionar corretamente devido à ausência de comunicação entre o HC06 e o celular.

Se você conseguiu parear seu dispositivo Bluetooth com o módulo HC06, o próximo passo é utilizar um aplicativo de terminal serial em seu celular ou *tablet*. Para dispositivos Android, recomendamos o aplicativo "Serial Bluetooth Terminal", que foi testado com sucesso. Abra o aplicativo e conecte-se ao dispositivo "Bluetooth Classic" com o nome exclusivo do HC06 na lista de "Devices". Para iPhone, o aplicativo "Bluetooth Serial Controller" é uma boa opção para conectar e enviar comandos para dispositivos "Bluetooth Classic". Embora seja amplamente utilizado em projetos de *hardware*, o seu uso não foi testado por nós (se você testou, compartilhe por favor conosco as suas experiências de uso).

1. Crie um projeto novo usando o *Cube*, com o nome "Serial_Relay", 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. O módulo *Bluetooth* que iremos usar está ligado aos pinos PA0 e PA1 usando a UART4, porém podemos ver no painel "Pinout & Configuration" que estes pinos não estão previamente reservados. Poderíamos reservar as funções neste painel, porém vamos fazer isto em nosso código. Gere o código e abra o arquivo "main,c".

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. O módulo *Bluetooth* está configurado para uma *baud rate* de 9600,8 *bits* de dados, sem paridade, e 1 *stop bit*. No escopo `/* USER CODE BEGIN 2 */`, escreva o código:

```
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
            USART_CR1_RE |  // Habilita o receptor
                  USART_CR1_RXNEIE_RXFNEIE);  //Habilita a interrup. de RX
UART4->CR1 |= USART_CR1_UE; // Habilita o UART4

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

4. Vamos configurar o NVIC colocando prioridade 1 na UART4 e prioridade 2 na USART3. A prioridade maior para o módulo *Bluetooth* foi decidida porque o terminal *Bluetooth* envia uma sequência de caracteres de uma vez, enquanto que o terminal do computador envia cada caractere digitado. O vetor de interrupção atribuído a UART4 é o 52.

```
// Configura NVIC
//Seta a prioridade nos bits [7:5] do byte de prioridade no registrador NVIC_IPRm
NVIC->IP[39]=(uint8_t)((0x2U << (8U-__NVIC_PRIO_BITS)) & (uint32_t)0xFFUL);
//Setar o bit 39%32=(39&0x1F) do registrador NVIC_ISERn, onde n = 39/32 = 39>>5.
NVIC->ISER[(((uint32_t)39) >> 5UL)] = (uint32_t)(1UL << (((uint32_t)39) & 0x1FUL));
// Idem para o vetor 52
NVIC->IP[52]    =    (uint8_t)((0x1U    <<    (8U    -    __NVIC_PRIO_BITS))    & (uint32_t)0xFFUL);
NVIC->ISER[(((uint32_t)52) >> 5UL)] = (uint32_t)(1UL << (((uint32_t)52) & 0x1FUL));
```

5. Em muitos fóruns de sistemas embarcados, como stackexchange, Particle Community e mcuoneclipse, há a informação de que o pino Tx de vários módulos *Bluetooth* não é capaz de sustentar o nível lógico alto, e assim não envia os níveis lógicos corretos para o pino Rx da UART, resultando em caracteres diferentes dos transmitidos. Assim, seria necessário conectar um resistor de *pull-up* externo, mas nosso microcontrolador possui o recurso de resistores de *pull-up* internos, programados pelo registrador GPIOx_PUPDR. Vamos habilitar o GPIOA para que possamos realizar a configuração de PA0 e PA1 em modo alternativo, bem como ativar o resistor de *pull-up* em PA1. Adicione ainda no escopo `/* USER CODE BEGIN 2 */` o seguinte código:

```
RCC->AHB4ENR |= RCC_AHB4ENR_GPIOAEN; // Habilita configs no PORT A

// Config PA0 e PA1 como UART4_TX e UART4_RX (Alternate Function 8)
// zera os bits MODER0[1:0] e MODER1[1:0]
GPIOA->MODER &= ~(GPIO_MODER_MODE0_Msk | GPIO_MODER_MODE1_Msk);
// Seta bits MODER0[1] e MODER1[1] (10 - alternate function)
GPIOA->MODER |= GPIO_MODER_MODE0_1 | GPIO_MODER_MODE1_1;
// zera os bits AFR0[3:0] e AFR1[3:0]
GPIOA->AFR[0] &= ~(GPIO_AFRL_AFSEL0_Msk | GPIO_AFRL_AFSEL1_Msk);
// seta bits AFR0[3] e AFR1[3] (1000 - AF8)
GPIOA->AFR[0] |= GPIO_AFRL_AFSEL0_3 | GPIO_AFRL_AFSEL1_3;

// Pull-up em PA1
GPIOA->PUPDR &= ~GPIO_PUPDR_PUPD1_Msk; // zera os bits PUPDR1{1:0]
GPIOA->PUPDR |= GPIO_PUPDR_PUPD1_0; // seta o bit PUPDR1[0]
```

6. Agora vamos implementar as ISRs. Primeiro, 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,

Depois, vamos usar a mesma ISR para a UART4, cujo nome `UART4_IRQHandler` pode ser obtido no arquivo "startup_stm32h7a3xxq.s" ao fazermos uma busca por "UART4" Na `UART4_IRQHandler` apenas enviamos o caractere recebido para a USART3, sem o eco. 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
       }
  }
}

void UART4_IRQHandler(void) {
       uint8_t c;
       if ((UART4->ISR & USART_ISR_RXNE_RXFNE)) { // há dados recebidos?
             c = UART4->RDR; // Lê o caractere recebido
             while(!(USART3->ISR & USART_ISR_TXE_TXFNF)){}// Espera Tx livre
             USART3->TDR = c; // Rele para USART3
       }
}
```

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

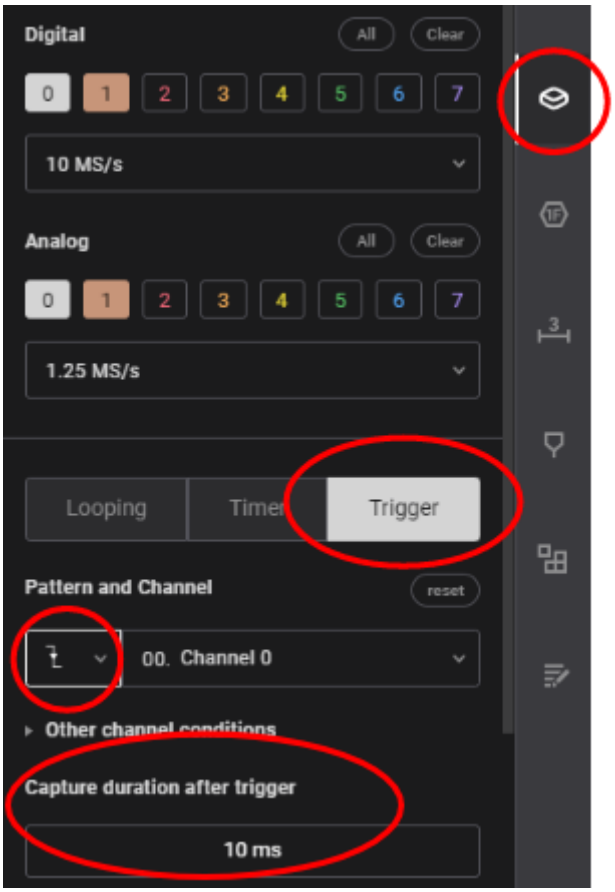
7. Construa ("Build") o código executável e transfira-o para o microcontrolador no modo *Debug*. Na perspectiva *Debug*, abra o terminal serial do IDE e configure-o com a taxa de transmissão de 9600 bps, 8 *bits* de dados, sem paridade e 1 *bit* de parada. Em seguida, abra o aplicativo "Serial Bluetooth Terminal", selecione o módulo desejado e estabeleça a conexão. O LED do módulo deverá parar de piscar e permanecer aceso, indicando que a conexão foi estabelecida com sucesso.

8. Continue ("Resume") a execução do programa e experimente enviar mensagens do terminal dentro da perspectiva de *Debug* para o app no celular e vice-versa. Se for usado o app sugerido, nas suas configurações ele já adiciona a sequência "\r\n" a cada envio de caracteres (CR + LF). Se usar outro programa, pode ser necessário configurar esta condição.

9. Vamos explorar melhor os sinais gerados pelo módulo USART3/UART4. Comente todas as linhas "`while(!(USART3->ISR & USART_ISR_TXE_TXFNF)){}`" (e o mesmo para a UART4). Rode novamente o programa e veja que, como as velocidades de todas as interfaces seriais é a mesma, dificilmente o *buffer* de transmissão estará ocupado quando o programa enviar um novo caractere.

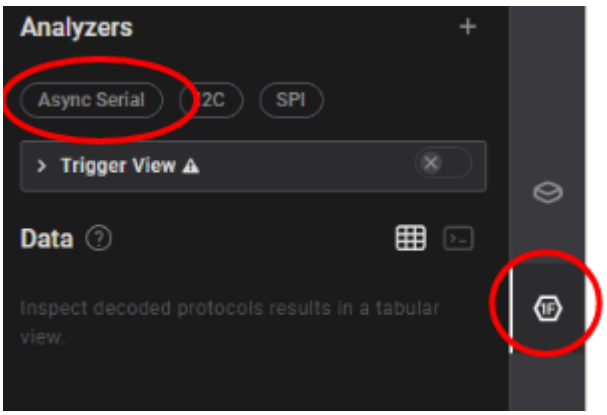
10. Pegue o *jumper* e conecte o pino no terminal 29 de CN10 (PA0 - Tx de UART4). Conecte o canal 0 do analisador lógico à outra ponta do *jumper*, e o terra do analisador ao pino mais à direita do conector H11 (como foi feito no roteiro anterior).

11. 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:

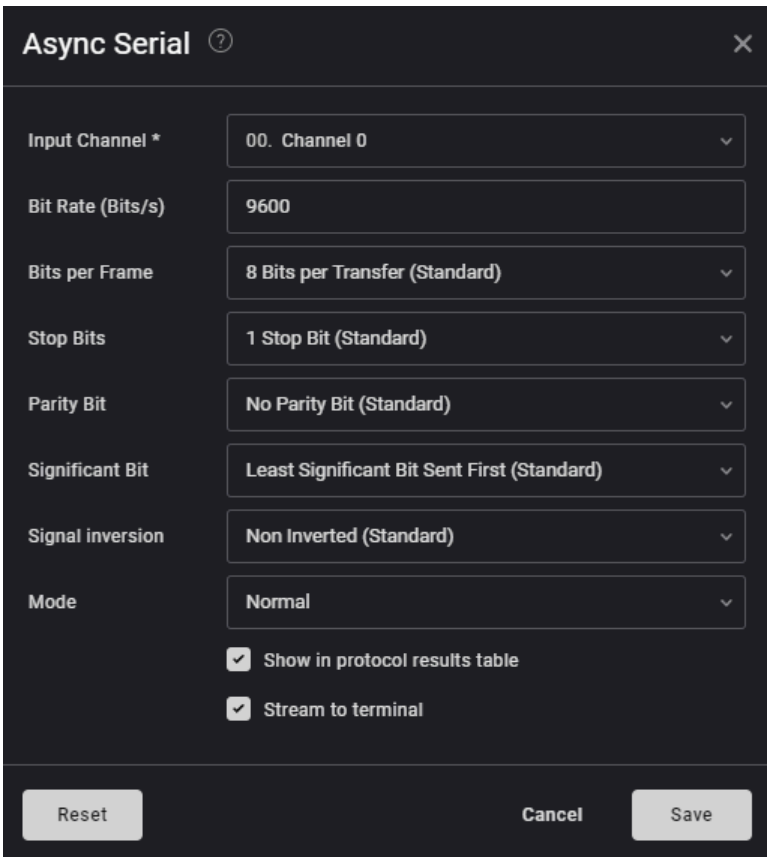


12. Com o programa de relé sendo executado na placa NUCLEO, inicie a aquisição no Logic. Note que, nesta configuração, ele aguarda uma borda de descida (a linha serial permanece em nível alto enquanto não é usada e o *Start Bit* inicia na primeira borda de descida) para iniciar a aquisição por 10ms. No terminal, envie um caractere qualquer. O caractere será recebido pela USART3 e retransmitindo pela UART4, através do pino PA1. O Logic irá detectar e capturar os *bits* transmitidos.

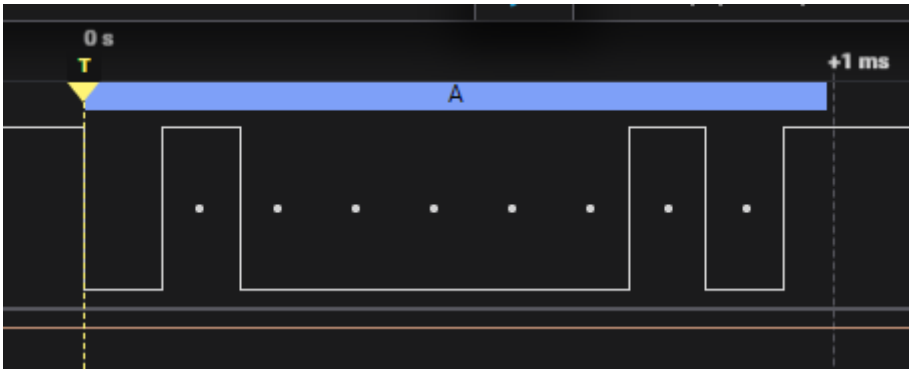
13. Dê um *zoom* no canal, para que o conjunto de dados recebido ocupe a maior parte da tela. Na barra da direita, clique no segundo ícone de cima para baixo (*Analyzers*). No topo do painel que se abre, clique no botão *Async Serial*.



14. 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”.



15. 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*).**



16. Reinicie a aquisição e agora aperte ENTER no terminal. Note que os dois caracteres "\r\n" são capturados pelo analisador.

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

Uso de dados seriais para controle de sistemas

Já pensou em transformar o relé de comunicação serial do projeto anterior 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 ao módulo HC06 e utiliza essas informações para gerenciar os periféricos da placa de expansão. Se você não conseguiu parear seu dispositivo Bluetooth com o módulo HC06, usaremos o Terminal Serial integrado no *Cube* para implementar esse controle.

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 celular ou no Terminal são enviados para o microcontrolador via *Bluetooth*, 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 remoto dos periféricos com base nessa fundação, 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_Relay", mas em um novo projeto chamado "Serial_Control".

2. Agora, vamos implementar as duas ISRs 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 do recepção de dados no RX do UART4
```

```
uint8_t TX3_ind;      // índice do elemento do buffer TX3 em acesso
char RX4;             // dado recebido no RX do UART4
char TX3[80];         // buffer de mensagem textual da tarefa ativa
```

Finalmente, implementaremos as ISRs. A ISR associada ao módulo UART4, que se comunica com o HC06, é responsável apenas por receber os comandos (em um caractere) enviados pelo celular/*tablet*. O aplicativo se encarrega automaticamente de fazer o eco dos caracteres digitados.

```
void UART4_IRQHandler(void) {
 if ((UART4->ISR & USART_ISR_RXNE_RXFNE)){//há dados recebidos?
       RX4 = UART4->RDR; // Lê o caractere recebido
       flagRX4 = 1;
 }
}
```

A ISR associada ao módulo USART3 é, por sua vez, responsável por enviar apenas a mensagem textual gerada pelo processador, de acordo com o comando digitado pelo usuário.

```
void USART3_IRQHandler(void) {
  if (USART3->ISR & USART_ISR_TXE_TXFNF) {
        USART3->TDR = TX3 [TX3_ind++];  // transfere um caractere
        if (TX3[TX3_ind] == '\0') {  // chegou no '\0'?
              USART3->CR1 &= ~USART_CR1_TXEIE_TXFNFIE_Msk;
        }
  }
}
```

Caso você **não tenha conseguido parear seu dispositivo com o módulo HC06**, focaremos apenas na implementação da ISR do módulo USART3. Esta ISR será responsável por receber os comandos e enviar as tarefas a serem executadas..

```
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 bps, 8 *bits* de dados, sem paridade e 1 *bit* de parada. Em seguida, abra o aplicativo de terminal Bluetooth, selecione o módulo desejado e estabeleça a conexão. O LED do módulo deverá parar de piscar e permanecer aceso, indicando que a conexão foi estabelecida com sucesso. **Antes de prosseguir, certifique-se que o app no celular seja reconfigurado para NÃO enviar os caracteres “CR+LF” (ou seja, “\r\n”) em cada envio.** No app recomendado, isto é feito entrando no menu geral (3 linha horizontais no canto superior esquerdo), opção “Settings”, aba “Send”, primeira opção (“Newline”), mudando a opção para “None”.

7. Continue (“Resume”) a execução. Digite no app “Serial Bluetooth Terminal”, ou um app. equivalente, 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 na 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.

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 uma linha da função main. Em seguida, regere o código executável (“Terminate and Relaunch”). Execute o programa e observe o comportamento. Explique a função dessa linha de habilitação de interrupção de transmissão quando o registrador estiver vazio.

```
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         }
```

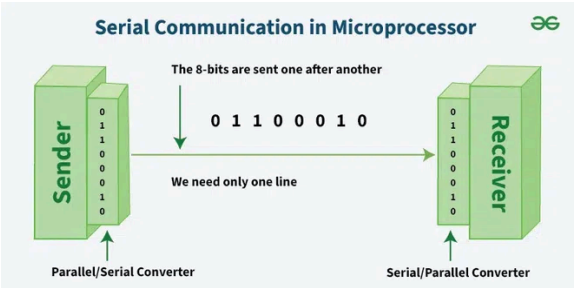
10. Pause o programa, descomente a linha da função main e comenta a linha da rotina `USART3_IRQHandler`. Em seguida, regere o código executável (“Terminate and Relaunch”). Execute o programa e observe o comportamento. Explique a função dessa linha de desabilitação de interrupção de transmissão quando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

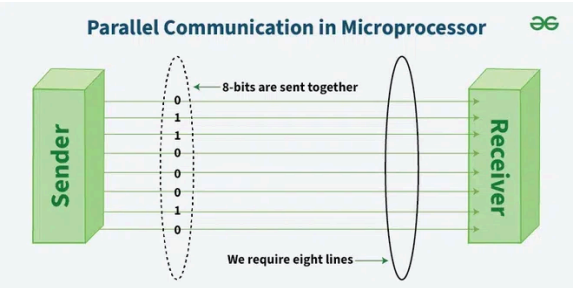
```
256     if (USART3->ISR & USART_ISR_TXE_TXFNF) {
257         USART3->TDR = TX3 [TX3_ind++];
258         if (TX3[TX3_ind] == '\0') {
259 //          USART3->CR1 &= ~USART_CR1_TXEIE_TXFNFIE_Msk;
260         }
261     }
```

11. Se você ainda não conseguiu formular uma resposta plausível para as perguntas anteriores, 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.

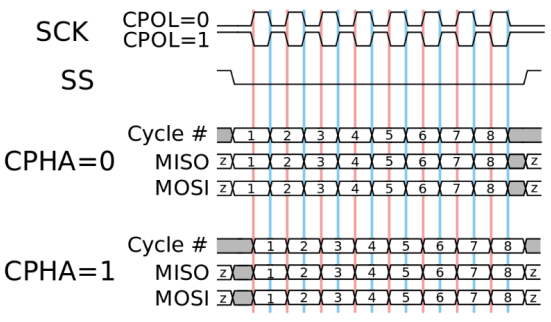
COMUNICAÇÃO SERIAL

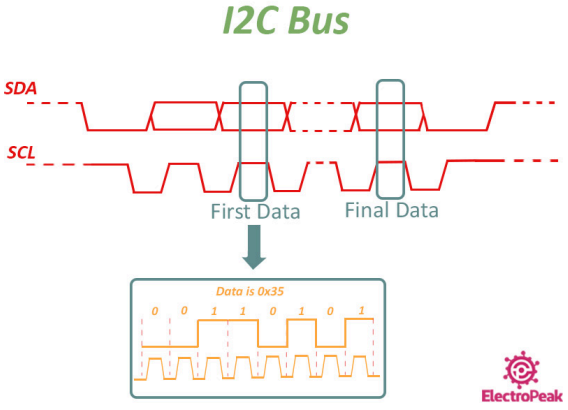
A **comunicação serial** é um método de transmissão de dados entre dispositivos eletrônicos que envia os *bits* sequencialmente, um de cada vez, através de um único canal ou fio. Diferente da **comunicação paralela**, onde múltiplos *bits* são transmitidos simultaneamente por vários canais, a comunicação serial transmite os dados em uma sequência linear. A abordagem serial simplifica o *design* dos sistemas de comunicação e pode ser mais eficiente em termos de cabeamento e conexões, especialmente em distâncias maiores ou em aplicações onde a redução da complexidade é desejável.





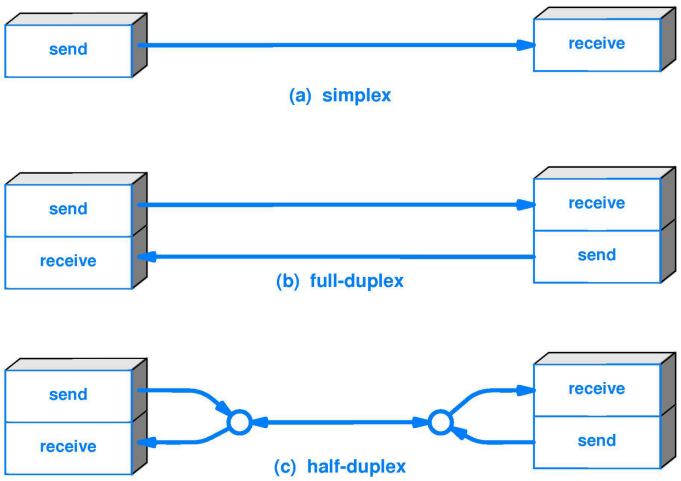
Existem dois principais tipos de comunicação serial: síncrona e assíncrona. Na **comunicação síncrona**, a sincronização entre o transmissor e o receptor é garantida por meio de um sinal de *clock* compartilhado. Este sinal de *clock* coordena a transferência de dados, garantindo que ambos os dispositivos, frequentemente designados como mestre (em inglês, *master*) e escravo (em inglês, *slave*), operem em sincronia. A precisão na comunicação síncrona é garantida pela configuração adequada da polaridade (CPOL) e da fase (CPHA) do sinal de *clock*. A **polaridade** define se o sinal de *clock* é ativo em nível alto ou baixo, enquanto a **fase** especifica em qual borda do *clock* (subida ou descida) os dados são amostrados. Além disso, outra técnica comum de sincronização é a geração e controle centralizados do sinal de *clock* por um dispositivo mestre. Neste método, 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.





Em contraste, na **comunicação assíncrona**, não há um sinal de *clock* compartilhado para sincronizar a transmissão e recepção de dados. Em vez disso, a comunicação assíncrona pode utilizar *bits* de *start* e *stop* para delimitar o início e o fim de cada caractere transmitido. Isso permite que o transmissor e o receptor sincronizem a leitura dos dados sem a necessidade de um *clock* contínuo. Este mecanismo de sincronização será explorado com mais detalhes posteriormente.

Além desses métodos, os protocolos de comunicação serial podem ser configurados para operar em diferentes modos de duplexidade. No modo **full-duplex**, é possível realizar a troca de dados bidirecional simultaneamente, o que significa que a transmissão e a recepção de dados ocorrem independentemente e ao mesmo tempo. Esse modo é ideal para sistemas que requerem comunicação contínua e eficiente. Em contraste, o modo **half-duplex** permite a comunicação bidirecional, mas de forma alternada, ou seja, os dados são enviados e recebidos em momentos diferentes. Outra variação é a comunicação ***single-wire***, também conhecida por simplex, que utiliza apenas um único fio para transmitir e receber dados, operando geralmente em modo half-duplex. Este método reduz o número de conexões necessárias, simplificando o *design* do sistema.



Para coordenar a troca de dados entre dispositivos e ajustar a transmissão conforme a capacidade do receptor e a disponibilidade de armazenamento (*buffer*), o mecanismo de *handshaking* é frequentemente utilizado no controle de fluxo. O ***handshaking*** é um processo de controle de comunicação no qual sinais de confirmação são trocados entre os dispositivos para garantir que os dados foram recebidos corretamente e que ambos estão prontos para continuar a troca de informações. Existem duas abordagens principais para o handshaking: por *hardware*, que utiliza sinais físicos para a coordenação, e por *software*, que emprega caracteres de controle e protocolos de comunicação.

No *handshaking* por *hardware*, sinais de controle são empregados para gerenciar a comunicação entre dispositivos. Dois exemplos comuns desses sinais são CTS (do inglês, *Clear To Send*) e RTS (do inglês, *Request To Send*). O sinal RTS é enviado pelo transmissor para indicar que deseja iniciar a transmissão de dados. Quando o transmissor está pronto para

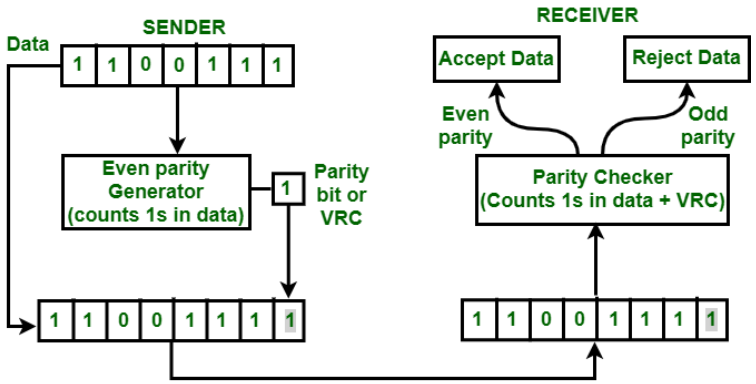
enviar os dados, ele ativa o sinal RTS para informar ao receptor que a transmissão está iminente. Por sua vez, o sinal CTS é enviado pelo receptor para informar ao transmissor que está pronto para receber os dados. O receptor ativa o sinal CTS quando está preparado para lidar com a entrada, confirmando que seu *buffer* não está cheio. Somente após o transmissor receber o sinal CTS é que ele inicia a transmissão dos dados. Esse método previne a sobrecarga do *buffer* do receptor e assegura uma comunicação eficiente e ordenada.

Em contraste, o *handshaking* por *software* utiliza sinais especiais gerados em *software* para controlar o fluxo de dados. Em vez de depender de sinais físicos como CTS e RTS, o controle de fluxo é realizado por meio de protocolos de comunicação que utilizam caracteres de controle especiais, como XON e XOFF. O XON (Transmissão Permitida) é enviado pelo receptor para indicar que está pronto para receber mais dados, enquanto o XOFF (Transmissão Interrompida) é enviado para informar que a transmissão deve ser interrompida temporariamente até que o *buffer* do receptor esteja novamente pronto para novos dados.

CÓDIGOS DETECTORES E/OU CORRETORES DE ERROS

Em sistemas embarcados, onde a integridade dos dados é crucial, especialmente em aplicações críticas como automação industrial, sistemas de controle de aviação e dispositivos médicos, garantir a correção e precisão dos dados transmitidos é essencial. Erros na comunicação podem ter consequências graves, portanto, a implementação de técnicas eficazes de detecção e correção de erros é vital. Códigos detectores e corretores de erro são fundamentais nesse contexto, pois ajudam a identificar e, em alguns casos, corrigir erros de transmissão antes que eles causem problemas significativos.

Uma técnica simples para detectar erros em transmissões seriais com baixa taxa de erro é a paridade. A **paridade** de uma palavra 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.



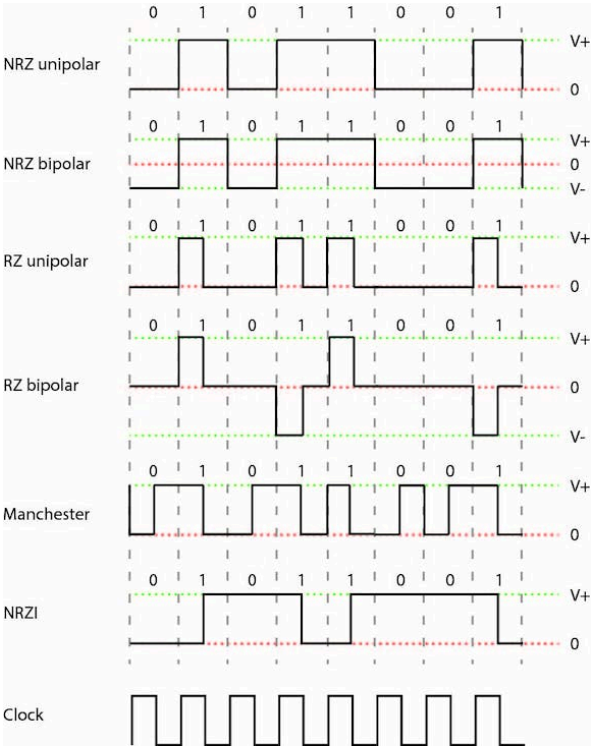
Uma maneira eficaz de determinar a paridade de uma palavra é através do **algoritmo XOR** (ou-exclusivo). Este algoritmo realiza uma operação lógica “ou-exclusivo” (XOR) *bit* a *bit*

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.

Para aplicações mais exigentes, técnicas de codificação como 4B/5B e 8B/10B são empregadas. Essas técnicas avançadas 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.

TÉCNICAS DE CODIFICAÇÃO EM SINAL FÍSICO

As técnicas de codificação de sinais desempenham um papel crucial na garantia da integridade dos dados, na otimização da largura de banda e na facilitação da sincronização entre transmissor e receptor. A figura a seguir ilustra diferentes formas de codificação de um mesmo código binário.



A codificação **Retorno a Zero** (**RZ**) é uma técnica em que o sinal retorna a um nível de referência, geralmente zero, entre cada *bit* transmitido. Durante a transmissão de um *bit* "1", o sinal fica em um nível alto por uma parte do tempo e retorna a zero antes do próximo *bit*. Isso ajuda na sincronização devido às frequentes transições de sinal, embora possa ser menos eficiente em termos de largura de banda, já que o sinal muda frequentemente.

Por outro lado, a codificação **Não Retorno a Zero** (**NRZ**) mantém o sinal em um nível alto ou baixo durante todo o intervalo de um *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 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.

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. Esta técnica é eficaz na sincronização devido às transições inclusas em cada *bit*, embora 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.

PROTOCOLOS DE COMUNICAÇÃO SERIAL ASSÍNCRONA: RS-232

Um **protocolo de comunicação** é um conjunto de regras e procedimentos que permitem a comunicação entre dispositivos em uma rede ou entre sistemas. Ele delinea a forma como os dados serão transmitidos, formatados, verificados e processados. Dentre os diversos tipos de protocolos de comunicação serial, destaca-se o padrão RS-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 usando dois estados principais conhecidos como **Mark** e **Space**. Estes estados representam os níveis de tensão usados para indicar valores binários e codificar *bits* de dados:

- **Mark (Marca):** Refere-se a um nível de tensão negativo, geralmente entre -3V e -25V. Em termos binários, o estado Mark corresponde a um "1".
- **Space (Espaço):** Refere-se a um nível de tensão positivo, geralmente entre +3V e +25V. Em termos binários, o estado Space corresponde a um "0".

Além dos aspectos de codificação, o padrão RS-232 também define os conectores físicos que facilitam a comunicação. Os conectores mais comuns incluem:

- **Conector DB9:** Um conector de 9 pinos, frequentemente utilizado em portas seriais padrão. É um dos conectores mais comuns para interfaces RS-232 em computadores e equipamentos periféricos.
- **Conector DB25:** Um conector de 25 pinos, que era amplamente utilizado em sistemas mais antigos e também para comunicação RS-232. Embora menos comum hoje em dia, ainda é encontrado em algumas aplicações industriais e legadas.

Esses conectores são projetados para garantir uma conexão física adequada e a transmissão dos sinais RS-232 entre dispositivos. Eles ajudam a assegurar que os sinais de *Mark* e *Space* sejam corretamente interpretados durante a comunicação.

Tipicamente, o RS-232 utiliza um protocolo de ***Start-Stop Bit*** para delimitar a transmissão de dados por caractere e opera da seguinte forma [1]:

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. Vale destacar que, embora o protocolo *Start-Stop* seja frequentemente utilizado com o RS-232 para a comunicação assíncrona em portas seriais (também conhecidas como portas COM) em computadores, o padrão RS-232 não está restrito a esse protocolo específico. O RS-232 é flexível e pode ser utilizado com diferentes métodos de comunicação, além do *Start-Stop*, para atender a diversas necessidades de transmissão de

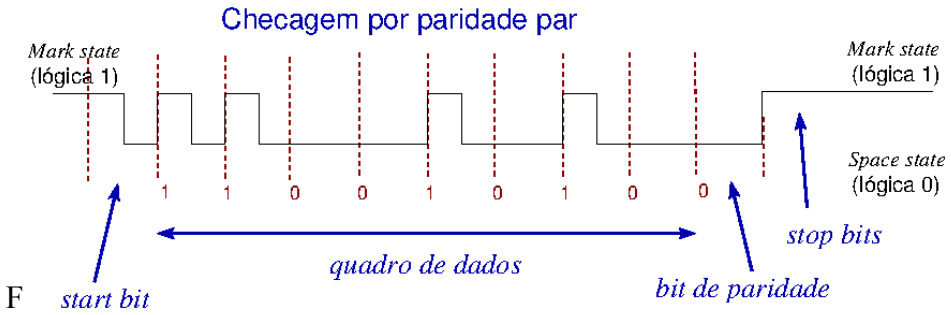


Figura 1: Transmissão por pacote numa comunicação serial assíncrona.

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.

UART e USART

UART (do inglês *Universal Asynchronous Receiver/Transmitter*) e **USART** (do inglês *Universal Synchronous/Asynchronous Receiver/Transmitter*) são interfaces de comunicação serial amplamente utilizadas em sistemas embarcados e microcontroladores para a transmissão e recepção de dados. O protocolo RS-232 estabelece as características físicas e elétricas da comunicação serial, incluindo os níveis de tensão utilizados (por exemplo, +12V para representar um *bit* lógico 1 e -12V para um *bit* lógico 0) e a configuração dos conectores, como DB9 e DB25. Por outro lado, o UART e o USART são circuitos que gerenciam a conversão entre dados paralelos e seriais e formatam esses dados de acordo com as especificações de diversos protocolos de comunicação. No entanto, o UART e o USART não definem os níveis de tensão nem a configuração física dos conectores. Para compatibilizar essas interfaces com o padrão RS-232, é comum usar um transceptor, como o MAX3232, que ajusta os sinais de acordo com as especificações do RS-232. Esta integração é frequentemente utilizada em comunicações assíncronas que envolvem um UART e o protocolo RS-232.

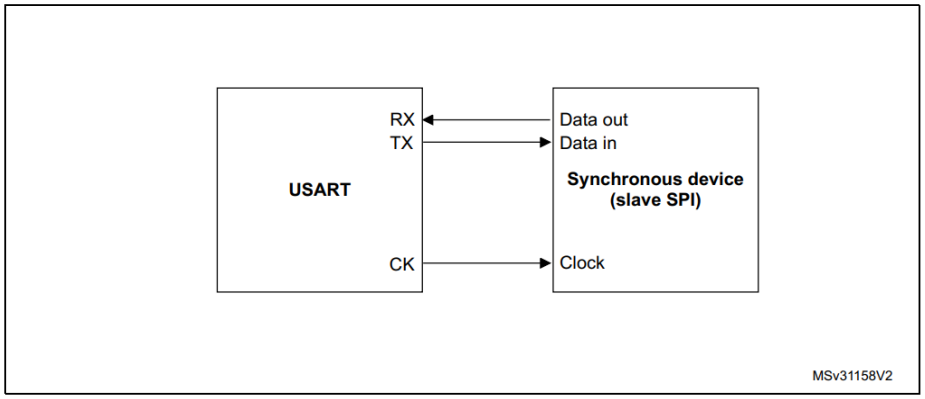
O UART e o USART são dois tipos de interfaces de comunicação serial com funcionalidades semelhantes e diferenças significativas. Ambos servem para converter dados entre formatos paralelos e seriais, permitindo a comunicação entre dispositivos. Uma das principais semelhanças entre UART e USART é que ambos operam na transmissão de dados seriais, ou seja, enviam e recebem dados um *bit* por vez através de uma única linha de comunicação. Ambos utilizam a configuração de *bits* de dados, paridade e *bits* de parada para garantir a

integridade dos dados transmitidos e podem ser configurados para diferentes taxas de transmissão, mais especificamente **taxas de mudanças do sinal por segundo** (em inglês, *baud rates*). Em termos de implementação, tanto o UART quanto o USART podem ser encontrados como blocos de *hardware* em microcontroladores, e suas funcionalidades podem ser acessadas e manipuladas por meio de registradores.

Tanto o UART quanto o USART utilizam pinos TX e RX para a comunicação serial. O pino TX (transmissor) é responsável por enviar dados do dispositivo transmissor para outro dispositivo receptor. O pino RX (receptor), por sua vez, recebe os dados enviados pelo transmissor e os converte em um formato que pode ser processado pelo dispositivo receptor. Em outras palavras, o TX é utilizado para transmitir informações, enquanto o RX é utilizado para recebê-las. Os sinais de comunicação são gerenciados por protocolos que garantem a integridade dos dados. Normalmente, o pino TX é controlado pelo dispositivo transmissor, que assegura que os sinais sejam corretamente definidos e enviados durante a operação normal. Em contraste, o pino RX aguarda os sinais de dados provenientes de um dispositivo transmissor. Em algumas situações, como em comunicações com o dispositivo Bluetooth H06, um **resistor de *pull-up*** pode ser necessário no pino RX para evitar que ele fique flutuante quando não há sinais presentes. No entanto, a necessidade de um resistor de *pull-up* depende da configuração específica do hardware e da lógica de comunicação utilizada.

A principal diferença entre o UART e USART está na forma como eles gerenciam a comunicação. O UART opera exclusivamente em **modo assíncrono**, o que significa que não há um sinal de *clock* compartilhado entre o transmissor e o receptor. Em vez disso, a sincronização é realizada através de *bits* de *start* e *stop* que marcam o início e o fim da transmissão de um **caractere**. Isso simplifica a implementação, mas pode limitar a velocidade e a precisão da comunicação, especialmente em longas distâncias ou altas taxas de dados. Em contraste, o USART é mais versátil, pois pode operar tanto em modo síncrono quanto assíncrono. No **modo síncrono**, o USART utiliza um sinal de *clock* `CK` compartilhado entre o transmissor e o receptor, o que permite uma comunicação mais rápida e precisa, pois os dados são transmitidos de acordo com um ritmo sincronizado. Isso é particularmente útil em aplicações que exigem alta taxa de transferência de dados ou onde a sincronização precisa é crucial. No entanto, o modo síncrono exige uma configuração adicional e pode ser mais complexo de implementar comparado ao modo assíncrono do UART.

Figure 554. USART example of synchronous master transmission



Em um módulo USART/UART, os canais RX (Recepção) e TX (Transmissão) são os principais componentes responsáveis pela comunicação serial assínsrona. O **canal RX** é responsável por receber dados do dispositivo externo, enquanto o **canal TX** é responsável por

enviar dados para o dispositivo externo. Consideramos que seja utilizado o protocolo *start-stop* na transferência assíncrona de dados.

O princípio de operação do canal RX começa com a **recepção** dos sinais seriais do dispositivo externo. Estes sinais representam os dados como uma sequência de *bits* transmitidos serialmente. O módulo USART/UART identifica os sinais de início e fim dos *bytes* de dados através de sinais de *start* e *stop*. Uma vez que os sinais são recebidos, o módulo converte a sequência de *bits* do formato serial para um formato paralelo por um registrador de deslocamento e armazena esses dados em um *buffer*/registrador interno (DATA BUFFER), como ilustra o diagrama de blocos recortado do manual de referência do microcontrolador Kinetis KL25Z. O processador então pode acessar e processar esses dados conforme necessário.

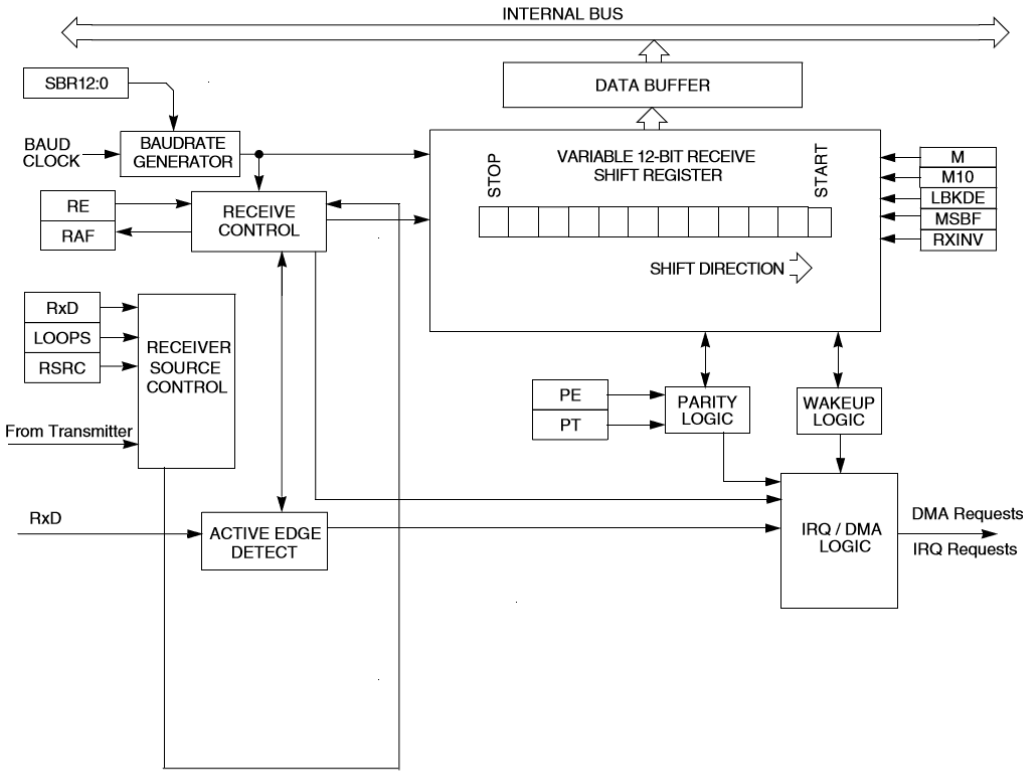


Figure 39-2. UART receiver block diagram

Por outro lado, o canal TX opera 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 para garantir a integridade dos dados. Os dados paralelos são então convertidos para uma forma serial e transmitidos *bit* a *bit* através do canal TX, como mostra o diagrama de blocos recortado do manual de referência do microcontrolador Kinetis KL25Z. Durante a transmissão, o módulo USART/UART controla a taxa de transmissão de acordo com a configuração da taxa de *baud*, garantindo que todos os *bits* sejam enviados corretamente e na ordem adequada.

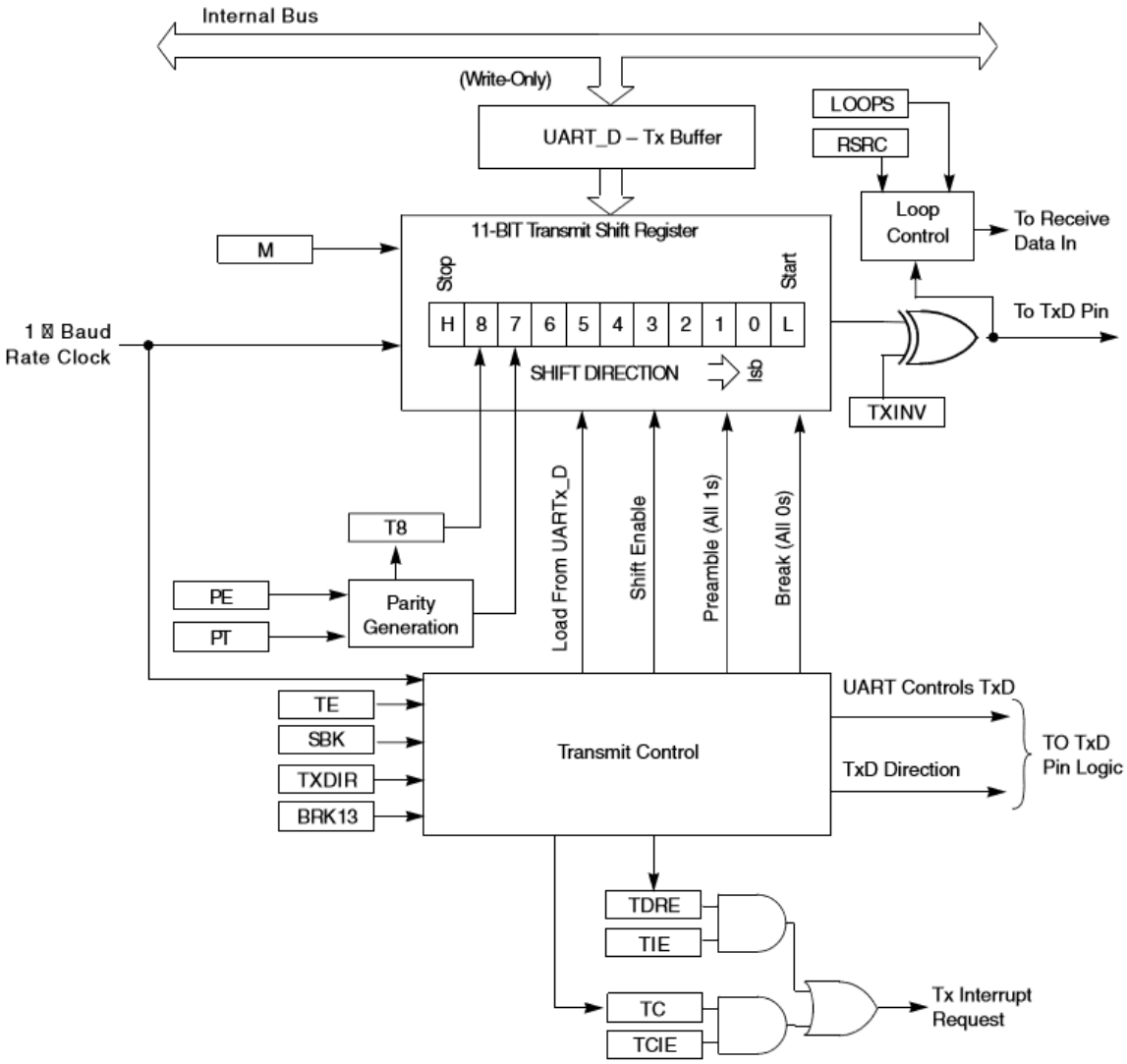


Figure 39-1. UART transmitter block diagram

Para coordenar a operação dos canais RX e TX e evitar conflitos no acesso ao processador, são utilizadas *flags* de estado. Essas *flags* são variáveis que indicam o estado dos *buffers* de recepção e transmissão. Por exemplo, uma *flag* de recepção pode ser usada para sinalizar quando os dados foram completamente recebidos e estão prontos para processamento, enquanto uma *flag* de transmissão pode indicar quando o *buffer* de transmissão está vazio e pronto para novos dados. Essas flags permitem que o sistema gerencie prograaticamente o fluxo de dados de forma ordenada, evitando a sobrecarga do processador e garantindo que a recepção e transmissão ocorram de forma sincronizada. Com as *flags* de estado, o código pode verificar o estado dos *buffers* antes de tentar enviar ou processar dados, assegurando que a comunicação entre os dispositivos seja sem erros.

CONTROLE DE CONCORRÊNCIA

Em uma comunicação serial que utiliza interrupções, garantir o sincronismo entre a recepção e a transmissão de dados é fundamental para evitar conflitos e preservar a integridade dos dados. Uma técnica eficiente para assegurar o sincronismo dos fluxos de controle

assincrônicos é o uso de *flags* de estado combinadas com variáveis de controle. *Flags* de estado, frequentemente controladas pelo *hardware*, são importantes na indicação do estado dos *buffers* de recepção e transmissão. Essas *flags* são essenciais para gerenciar o fluxo de dados e garantir que a comunicação entre o processador e os periféricos ocorra de maneira ordenada e sem erros. Elas permitem que o sistema detecte rapidamente quando um *buffer* está cheio ou vazio, possibilitando ações apropriadas, como a leitura de dados recebidos ou o envio de novos dados.

No entanto, em sistemas mais complexos, onde múltiplas tarefas e interrupções podem interagir simultaneamente, o uso exclusivo de *flags* de estado pode não ser suficiente. É necessário empregar variáveis de controle adicionais para lidar com problemas de concorrência e assegurar que o processador não se sobrecarregue ao acessar recursos compartilhados ao mesmo tempo. Sem um controle adequado, pode ocorrer uma situação onde duas ou mais operações competem pelo mesmo recurso, levando a resultados imprevisíveis ou até mesmo corrupção de dados.

Por exemplo, considere um sistema em que a mesma rotina de serviço de interrupção (ISR) gerencia tanto eventos de recepção (RXE) quanto de transmissão (TXE). Em um cenário onde a transmissão deve ocorrer somente após a recepção de dados, o fluxo de controle pode se tornar complicado. Se a *flag* de transmissão (TXE) for habilitada antes da conclusão da recepção, a ISR pode desviar o fluxo de controle para o código de transmissão imediatamente, mesmo que o sequenciamento correto exija que a transmissão só ocorra após o processamento da recepção.

Para evitar tal conflito, é fundamental implementar variáveis de controle que assegurem a ordem correta dos eventos. Por exemplo, você pode usar uma variável de controle para garantir que a transmissão só seja iniciada após a recepção ser completamente processada. Assim, mesmo que a ISR para TXE seja acionada, o sistema verificará a variável de controle para garantir que a recepção foi concluída e que é apropriado iniciar a transmissão.

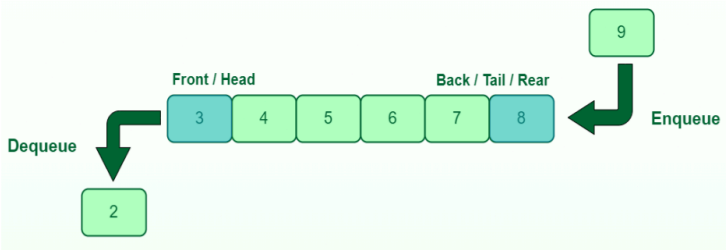
ESTRUTURA DE DADOS: FILA

Semelhante à pilha, que vimos no Roteiro 3, a **fila** é uma estrutura de dados linear que segue uma ordem específica para armazenar e manipular dados. A ordem adotada é a de "*First In, First Out*" (FIFO), ou seja, o primeiro elemento inserido na fila será o primeiro a ser removido. Pode-se visualizar uma fila como uma linha de pessoas aguardando para receber um serviço em uma sequência ordenada que começa pelo início da fila. Na estrutura de uma fila, as inserções de dados são realizadas em uma extremidade, chamada de **cauda**, enquanto as remoções ocorrem na outra extremidade, 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 pilhas e filas 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.

A estrutura FIFO é amplamente utilizada em várias aplicações de sistemas de computadores. No gerenciamento de tarefas, ela ajuda a garantir que as tarefas sejam processadas na ordem em que foram recebidas. Em *buffers* de entrada e saída, a FIFO é essencial para armazenar dados temporariamente durante a comunicação entre dispositivos ou entre diferentes partes de um sistema. Particularmente em periféricos USART/UART dos microcontroladores, a estrutura FIFO implementada em *hardware* permite a bufferização de dados, o que é fundamental para lidar com a diferença de velocidade entre o transmissor e o receptor. Isso ajuda a garantir uma comunicação ordenada e sem perdas, permitindo que os dados sejam armazenados temporariamente até que possam ser processados pelo receptor, mesmo que a taxa de transmissão e recepção não esteja sincronizada.

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.

CÓDIGO ASCII NA TRANSFERÊNCIA DE DADOS TEXTUAIS

O código ASCII é essencial na comunicação serial, pois estabelece um padrão para a representação e troca de caracteres entre dispositivos. Vimos no Roteiro 4 que o **código ASCII** define uma tabela de 128 **caracteres**, incluindo letras, números, sinais de pontuação e caracteres de controle, todos representados por 7 *bits*. Um **caractere de controle** é um caractere não renderizável, pertencente a um conjunto de códigos que representam símbolos de escrita com uma funções específicas universalmente reconhecidas. Todos os códigos abaixo de 32 (0x20) da tabela ASCII são considerados caracteres de controle. Ao serem inseridos numa *string*, esses caracteres podem alterar a disposição dos caracteres renderizáveis ou o comportamento do sistema. Por exemplo, 0x07 (*bell*) é o caractere de controle que faz o dispositivo emitir um som, 0x08 (*backspace*) retrocede para sobrescrever o último caractere renderizado, 0x0A (*line feed* ou *new line*) marca o fim de uma linha, e 0x0D (*carriage return*) move o cursor de volta para a primeira coluna. Para incluir os caracteres de controle junto com outros caracteres renderizáveis, usamos **sequências de escape** que consistem em uma barra invertida (‘\’) seguida de uma letra ou de uma combinação de dígitos. As sequência de escape para *bell*, *backspace*, *new line* e *carriage return* são, respectivamente, “\a”, “\b”, “\n” e “\r”.

Para acomodar uma gama mais ampla de caracteres, além dos definidos pelo ASCII de 7 *bits*, foram desenvolvidas várias extensões. Uma dessas extensões é o **ISO 8859-1**, também conhecido como **ISO Latin-1** ou **ASCII estendido**. Este padrão usa 8 *bits* para cada caractere, o que permite um total de 256 combinações possíveis (0 a 255). Essas padronizações são cruciais para garantir que diferentes sistemas possam interpretar corretamente os **dados textuais** enviados, independentemente das suas diferenças em *hardware* e *software*.

Observe que o ASCII de 7 *bits* e o ASCII estendido definem apenas a representação básica dos caracteres, sem considerar a integridade dos dados durante a transmissão, quando a integridade dos dados pode ser comprometida por ruído ou erros de transmissão. Um *bit* de paridade pode ser adicionado a cada caractere transmitido para verificar se os dados foram recebidos corretamente. Assim, se usar ASCII em uma comunicação serial com paridade, a tabela de ASCII de 7/8 *bits* será ampliada para incluir o *bit* de paridade, totalizando 8/9 *bits* por caractere transmitido.

As ***strings***, que são vetores de caracteres codificados em ASCII com terminador NULL (“\0”), desempenham um papel importante nesse contexto. Elas permitem a transmissão de mensagens, comandos e dados textuais de maneira clara e estruturada. Como a maioria dos sistemas e linguagens de programação oferecem suporte nativo para operações com *strings*, como concatenação, busca e formatação, trabalhar com *strings* ASCII torna-se uma tarefa direta e eficiente. Além disso, durante o desenvolvimento e a depuração de sistemas embarcados, as *strings* ASCII são usadas para enviar mensagens de estado e *logs* de erros em um formato legível por humanos. Isso facilita a análise e a solução de problemas, permitindo uma comunicação mais transparente e um diagnóstico mais rápido.

PROCESSAMENTO DE *STRINGS* EM C

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 protocolo *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

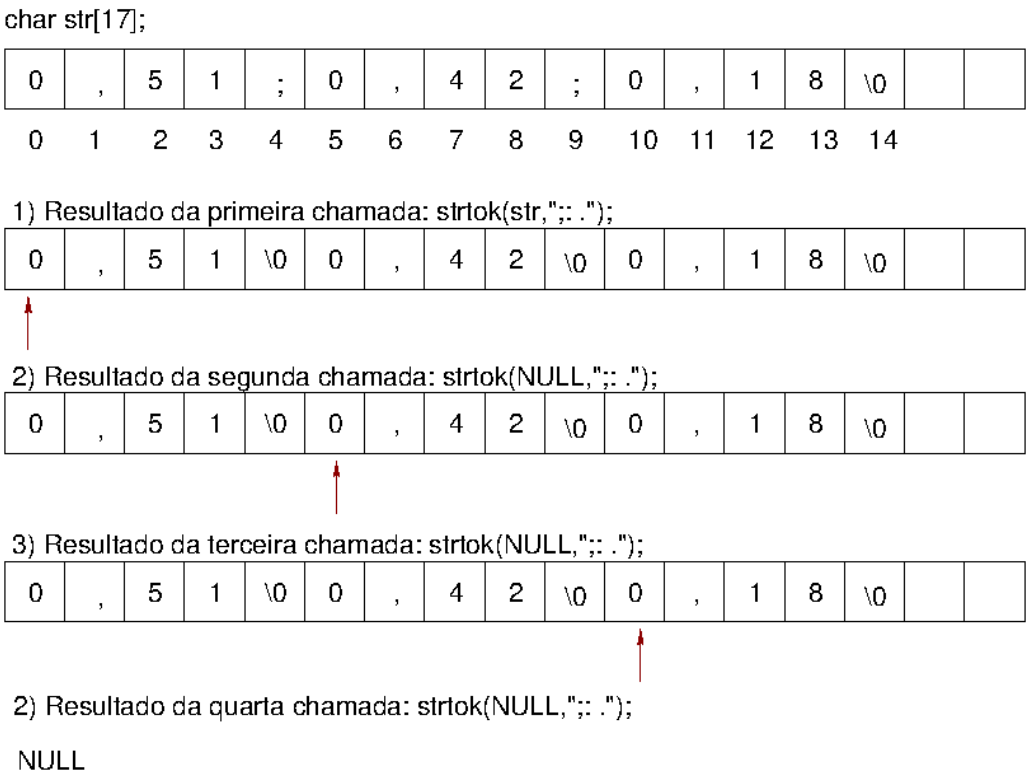
no terminal, é recebida pelo microcontrolador, ela pode ser armazenada como uma *string*. Para que essa sequência seja tratada corretamente como uma *string* em C, é necessário substituir o caractere de controle '\r' (carriage return), correspondente à tecla "Enter", pelo terminador nulo ('\0').

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.

APLICAÇÕES

A comunicação UART é amplamente utilizada em sistemas embarcados e dispositivos eletrônicos para estabelecer conexões simples e eficientes entre componentes. Esta seção explora duas aplicações comuns de módulos UART em dispositivos populares: a comunicação via Bluetooth e a interface com terminais seriais. Ambas as aplicações aproveitam a flexibilidade e a confiabilidade da UART para permitir a troca de dados de forma assíncrona, cada uma atendendo a requisitos específicos de conectividade e interação com o usuário.

Módulo Bluetooth

O Bluetooth é uma tecnologia de comunicação sem fio projetada para permitir a troca de dados entre dispositivos em distâncias curtas. Ele é uma aplicação particularmente notável do módulo USART/UART, oferecendo uma combinação de funcionalidade e versatilidade para uma ampla gama de aplicações. Operando na faixa de frequência de 2,4 GHz, o Bluetooth utiliza técnicas de modulação de rádio para transmitir e receber dados entre dispositivos. Ele permite que dispositivos se conectem e estabeleçam uma comunicação segura e confiável por meio do processo de **emparelhamento**. Uma das características mais interessantes do Bluetooth clássico é sua capacidade de emular uma interface serial RS-232 usando o *Serial Port Profile* (SPP). O SPP cria uma interface virtual de porta serial sobre a conexão Bluetooth, permitindo que dispositivos se comuniquem como se estivessem conectados por um cabo serial físico. Essa emulação proporciona uma forma de comunicação assíncrona e serial, tornando o Bluetooth funcionalmente semelhante a uma conexão serial tradicional.

Os módulos Bluetooth comerciais, como o HC05 e o HC06, são exemplos práticos de como essa tecnologia pode ser aplicada para conectar microcontroladores (MCUs) a dispositivos celulares. O módulo HC06, por exemplo, é frequentemente utilizado para comunicação serial sem fio, oferecendo uma interface simples para conectar uma MCU a um celular ou outro dispositivo Bluetooth. O módulo HC05 é uma opção similar, oferecendo recursos adicionais (como por exemplo modo *master*) e compatibilidade com o perfil SPP para emulação de portas seriais.

Tanto o HC05 quanto o HC06 permitem que o dispositivo sem fio pareado seja acessado através do SPP. Muitos programas e aplicativos utilizam este protocolo, inclusive para emular terminais seriais. No lado da MCU, esses módulos Bluetooth expõem os pinos TX e RX, que se conectam diretamente a uma interface UART. Dessa forma, a MCU percebe este canal de comunicação como uma linha serial assíncrona, simplificando a programação e a integração

do sistema. Esta abordagem permite que a MCU trate a comunicação Bluetooth de maneira semelhante à de uma porta serial tradicional, facilitando o desenvolvimento e a implementação de soluções sem fio.

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 (CLI, do inglês *Command Line Interface*). 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 uma troca de dados eficiente e confiável.

Embora a utilização de terminais seriais em comunicação de dados é fundamental em diversos sistemas, a **falta de padronização na codificação** pode acarretar uma série de problemas. Diferentes aplicativos e dispositivos podem empregar formatos de codificação variados, o que resulta em dificuldades de interoperabilidade. Essa diversidade de códigos, que inclui desde variantes do ASCII até codificações mais complexas como UTF-8, pode gerar confusões e erros na transmissão e interpretação de dados. Quando a codificação utilizada por um aplicativo não é conhecida, é recomendável optar por uma codificação mais básica e amplamente reconhecida, como o ASCII de 7 *bits*. O ASCII serve como uma base sólida, já que muitas codificações mais modernas foram desenvolvidas a partir dele e mantêm compatibilidade em relação aos seus caracteres. Ao adotar o ASCII, garante-se que a maioria dos caracteres básicos, como letras, números e símbolos comuns, será corretamente interpretada, minimizando assim o risco de perda ou corrupção de dados. Portanto, em um ambiente onde diferentes aplicativos de terminais seriais são empregados, a escolha da codificação correta é crucial. Sempre que houver incerteza, utilizar o ASCII como padrão inicial pode facilitar a comunicação e promover uma maior compatibilidade entre sistemas, permitindo uma transição mais suave para codificações mais complexas quando necessário.

STM32H7A3: USART/UART

Para comunicação serial assíncrona, o STM32H7A3ZIT6-Q é equipado com 5 USART, 5 UART e 1 LPUART (do inglês *Low-Power Universal Asynchronous Receiver Transmitter*). O USART integrado é um periférico altamente versátil, que permite a comunicação serial com dispositivos externos através de diversos protocolos. Diferente do USART, o UART não suporta comunicação síncrona nem o modo de comunicação SmartCard. Portanto, enquanto descrevemos o USART de forma abrangente, é importante notar que o UART apenas carece dessas duas funcionalidades adicionais.

O 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, o USART possui duas estruturas FIFOs (do inglês *First-In, First-Out*) 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, o 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**, o 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.

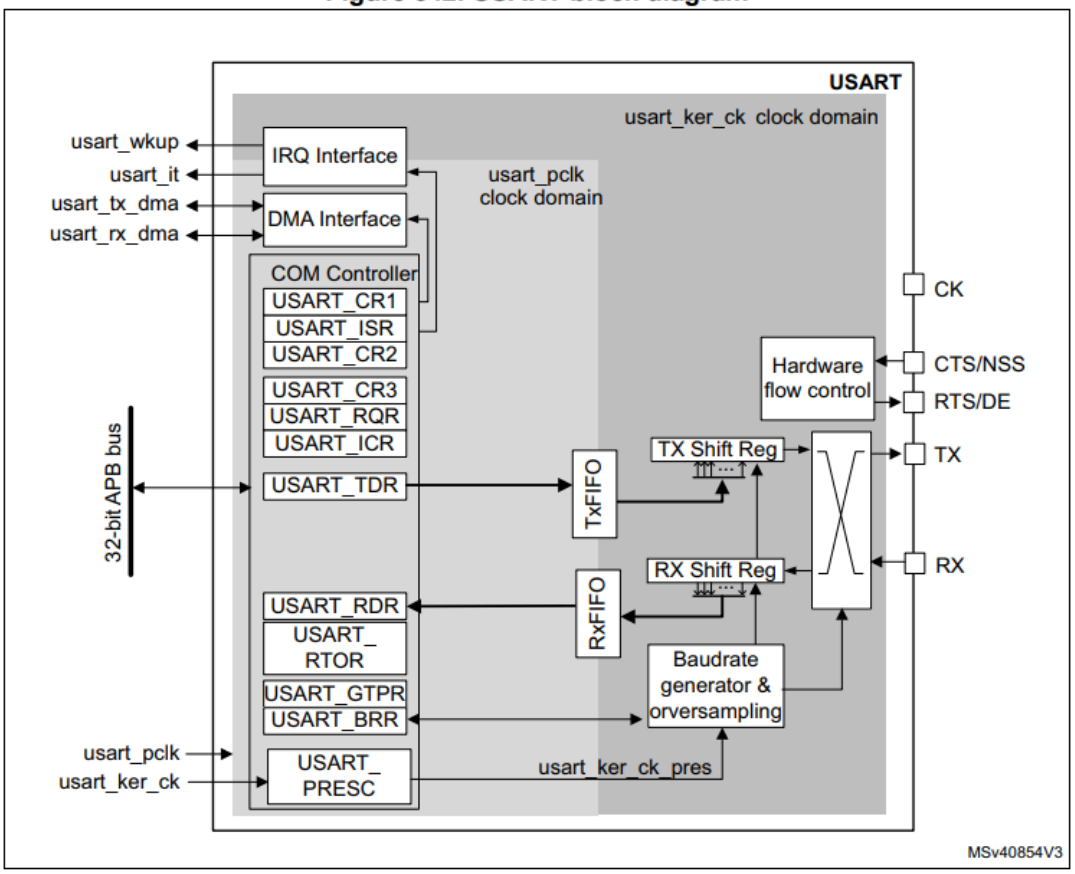
Quanto à **detecção e controle de erros**, o USART está equipado com mecanismos para verificar paridade, detectar sinais de *break* e identificar erros de enquadramento (em inglês, *framing error*), assegurando a integridade dos dados durante a transmissão. O periférico também suporta controle de fluxo 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, o 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 diagrama de blocos do USART, mostrado na figura, ilustra detalhadamente os componentes e o fluxo de dados envolvidos na transmissão e recepção de dados seriais assíncronos. 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.

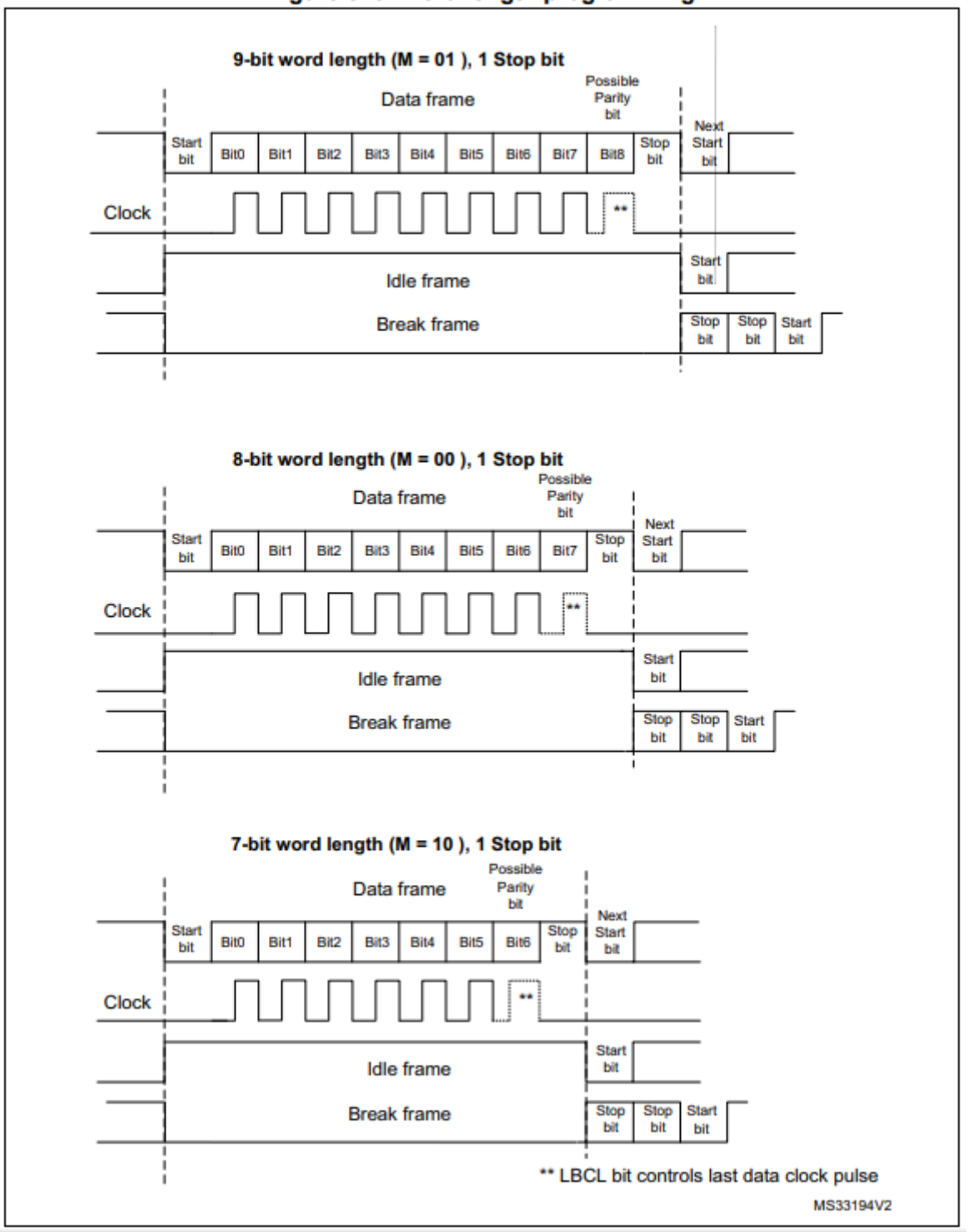
Figure 542. USART block diagram



O controlador de comunicação (COM) lida aspectos importantes da transmissão, como o **tamanho da palavra**, a **paridade** e o **número de *bits* de parada**, 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 *bits* de parada 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). Neste modo, 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 parada, 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 parada. 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 a sinalização 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.

Por outro lado, 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 o 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*.

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.

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*). 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 pelos *bits* `USART_CR3_RTSE` e `USART_CR3_CTSE` no registrador `USART_CR3`, respectivamente.

A **habilitação dos canais** de recepção (Rx) e transmissão (Tx) num 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 do USART, assegurando a precisão na comunicação.

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 ao 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 do USART/UART poder gerar múltiplos eventos de interrupção, há apenas uma única linha de interrupção (IRQ) associada a cada módulo, como mostra a figura

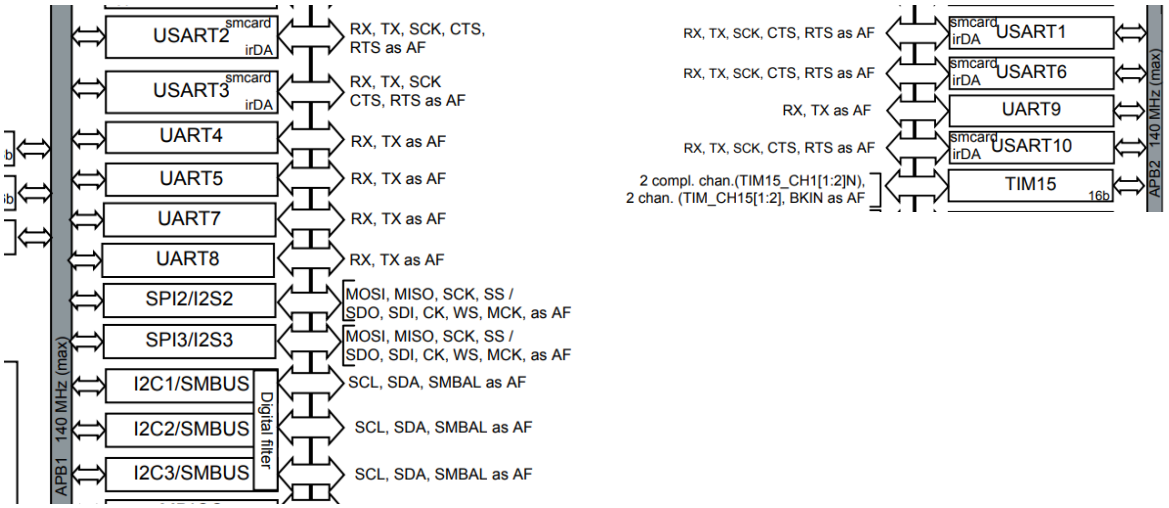
a seguir. Isso significa que quando ocorre uma interrupção, o processador é notificado através dessa **única IRQ**, e o controlador de interrupções precisa identificar qual evento específico causou a interrupção. Portanto, dentro da rotina de serviço de interrupção (ISR), é necessário verificar as bandeiras (em inglês, *flags*) ou registradores do módulo UART/USAT 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.

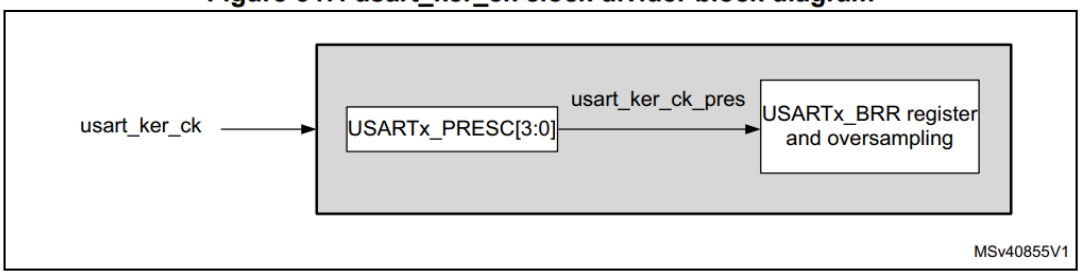
O **bloco de interface DMA** (do inglês *Direct Memory Access*) permite a comunicação contínua usando DMA para transmissão e recepção de dados. Os *bits* `USART_CR3_DMAT` e `USART_CR3_DMAR` no registrador `USART_CR3` habilitam o modo DMA para transmissão e recepção, respectivamente.

O diagrama 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 o acesso aos registradores do USART, como mostra o recorte da figura no *Datasheet*. O `usart_ker_ck` é a fonte de *clock* para o próprio USART, controlando seu funcionamento interno. 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



A taxa de transmissão/recepção num 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 do USART, seja *oversampling* por 8 ou 16. No caso do *oversampling* por 16, por exemplo, para uma taxa de transmissão desejada de 9600bps 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 mais próximo e maior, 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:

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

.

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.

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.

Uma das características importantes no UART é o modo FIFO (do inglês, *First-In, First-Out*), que utiliza *buffers* para o envio e recebimento de dados, tornando a comunicação mais eficiente. O 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 o UART e continue com outras tarefas enquanto o UART processa a transmissão. Por sua vez, o RXFIFO armazena os dados recebidos, possibilitando que o processador leia um bloco de dados do 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 do 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.