EA871 – LAB. DE PROGRAMAÇÃO BÁSICA DE SISTEMAS DIGITAIS

EXPERIMENTO 7 – Interface Serial Assíncrona

Profa. Wu Shin-Ting

OBJETIVO: Apresentação de uma interface serial assíncrona.

ASSUNTOS: Interface serial assíncrona UART, comunicação serial do MKL25Z128 com PC via porta COM, programação do MKL25Z128 para processamento de sinais de uma comunicação UART.

O que você deve ser capaz ao final deste experimento?

Entender o princípio de comunicação via UART.

Saber configurar um módulo UARTx de KL25Z para uma comunicação serial assíncrona.

Programar MKL25Z128 para processamento de sinais de uma comunicação UART.

Saber alocar pinos para um módulo UARTx.

Saber calcular o divisor de frequência para uma dada taxa de transmissão.

Saber configurar o divisor de frequência e a taxa de superamostragem em KL25Z.

Saber configurar um Terminal serial para acessar um sinal de uma porta serial.

Saber utilizar *buffer* (circular) numa transferência serial.

Saber fazer conversão entre *strings* de algarismos e valores numéricos que elas representam.

Saber usar `struct` e parametrização de blocos de memória.

Saber aplicar as funções da biblioteca padrão C para manipular *strings*.

Saber implementar os exemplos de aplicação dos módulos do microcontrolador apresentados em [3].

Saber aplicar máquina de estados na proteção de regiões críticas.

INTRODUÇÃO

Um dos padrões mais usados para comunicações entre sistemas ou partes de um sistema é o padrão serial. Na **interface serial**, os *bits* são enviados em sequência, um de cada vez, ao invés de grupos de 8 ou mais *bits* simultaneamente (paralelamente), demandando mais tempo para envio de uma informação de tamanho maior que um *bit* [1]. Mesmo assim, ela é comumente aplicada na comunicação entre microcontroladores, comunicação com dispositivos-periférico e comunicação via redes, pois ela é

- mais eficiente em termos do uso de recursos, por utilizar menos pinos,
- mais confiável, por enviar um *bit* de dado por vez que simplifica o circuito detector de erros, e
- mais flexível, por ser facilmente adaptável para diferentes taxas de transmissão e protocolos de comunicação.

Existem duas variantes deste padrão: *síncrona* e *assíncrona*. Basicamente, a diferença está na presença ou ausência de um sinal de relógio que sincroniza a transmissão de *bits*. Neste experimento veremos um padrão assíncrono, que exige que os dois sistemas tenham seus relógios individuais ajustados, bem como o estabelecimento prévio da velocidade de transmissão e um formato específico para o sinal digital de comunicação. No lugar de um circuito de processamento de sinais digitais de propósito geral

(GPIO), usaremos um módulo denominado ***Universal Asynchronous Receiver/Transmitter*** (**UART**) dedicado a um formato de comunicação serial. A transmissão de dados é *full duplex*, ou seja, ambos os lados podem transmitir e receber simultaneamente, através das linhas TX e RX, respectivamente. A linha TX de um lado deve ser conectada à linha RX do outro, e vice-versa.

Protocolos de Comunicação

Um **protocolo de comunicação** é um conjunto de regras e procedimentos que permitem a comunicação entre dispositivos em uma rede ou entre sistemas. Ele define como os dados serão transmitidos, formatados, verificados e processados. Existem diversos tipos de protocolos de comunicação serial. O protocolo tipicamente usado em comunicações seriais assíncronas é RS-232. Esse protocolo usa um *bit* de **start** para indicar o início de uma nova transmissão de dados e um *bit* de **stop** para indicar o final da transmissão. Os quadros de dados (*data frame* ou caractere) são transmitidos por **pacotes** entre esses dois *bits* responsáveis pelo sincronismo. Os *bits* de **mark** (opcional) são inseridos entre os pacotes para indicar que um dado específico está sendo transmitido. Para uma verificação simples dos possíveis erros na transmissão, pode ser incluído no **pacote** um *bit* de paridade [1] (Figura 1).

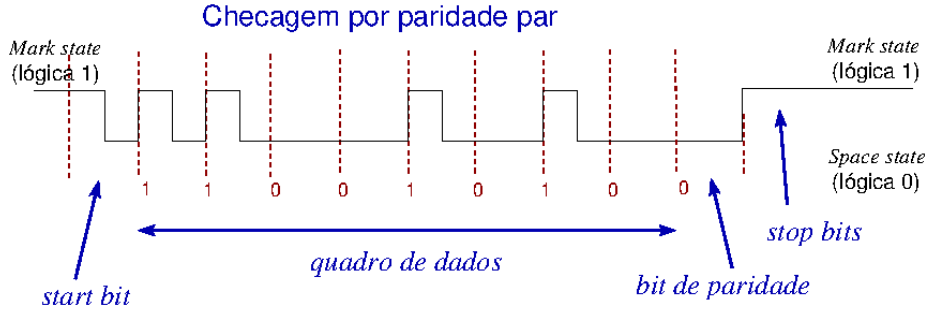

Figura 1: Transmissão por pacote numa comunicação serial assíncrona.

O **protocolo-padrão RS-232** estabelece ainda o nível de tensão dos sinais. O nível lógico 1 (*mark state*) está associado a uma tensão entre -3V a -15V enquanto o nível lógico 0 (*space state*) a uma tensão entre 3V a 15V. Este padrão é adotado nas portas seriais, também chamadas de portas de comunicação COM, dos PCs.

UART (*Universal Asynchronous Receiver/Transmitter*) é um circuito de interface serial padronizada em muitos microcontroladores para comunicações com outros dispositivos através de uma interface serial. Ele inclui dois registradores (um para transmissão e outro para recepção), além de circuitos para geração e detecção de sinais de *start*, *stop* e *mark bits*, e para controle de fluxo. Os níveis de tensão dos seus sinais são digitais, entre 0V a 5V.

Paridade de uma Palavra (Dado de n *Bits*)

A **paridade** de uma palavra é a paridade da quantidade de *bits* '1' na palavra. Ela é **par** se essa quantidade é par; do contrário, a paridade é considerada **ímpar**. É uma forma simples para detectar erros em transmissões seriais com baixa taxa de erro. Consiste em adicionar um *bit* (de paridade) para indicar se o número de *bits* '1' na palavra é par ('0') ou ímpar ('1'), antes de transmitir a palavra. Depois, no receptor, é contada novamente a quantidade de *bits* '1' e comparada com o *bit* de paridade. Se eles são diferentes, é caracterizado com um erro na transmissão.

Uma forma de determinar a paridade de uma palavra é usando o **algoritmo XOR**. 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, a paridade será 0. Se o número de *bits* '1' na palavra for ímpar, a paridade será 1. Este algoritmo é eficiente pois não requer contagem de *bits* 1 na palavra e pode ser implementado usando operações *bit* a *bit* simples. O procedimento pode ser acelerado, se dividirmos os *bits* de um inteiro recursivamentemente em duas metades e tomar seu XOR até restar apenas 1 *bit* [25].

Módulos UARTx

O microcontrolador KL25 possui três módulos dedicados à comunicação serial assíncrona permitindo conexões simultâneas com até 3 dispositivos, UART0, UART1 e UART2, podendo UART0 operar no modo de baixo consumo. Cada módulo UARTx atua como uma "ponte" entre a interface paralela do microcontrolador e uma interface serial assíncrona dos periféricos, em níveis de tensão digitais (Capítulo.39/página 721, Capítulo 40/página 747 em [2], Capítulo 8/ pág. 77 em [3]). UARTx contém um circuito transmissor (canal TX, Figura 39-1/página 723 em [2]) e um receptor (canal RX, Figura 39-2/ página 724 em [2]). Através dos registradores de deslocamento, os *bytes* a serem transmitidos são serializados e processados, e os *bits* recebidos são amostrados e reagrupados em *bytes* automaticamente, a uma frequência pré-estabelecida.

Os sinais de relógio de UARTx são habilitados pelos *bits* do registrador `SIM_SCGC4_UARTx` do módulo `SIM` (Seção 12.2.8/página 204 em [2]). Pela Tabela 5-2/página 121 em [2], os sinais de relógio dos registradores de deslocamento de UART1 e UART2 são os mesmos do barramento (*bus clock*), enquanto o de UART0, denominado `UART0 clock`, é distinto do barramento. A fonte de `UART0 clock` é selecionável, via o registrador de configuração `SIM_SOPT2` (campos `SIM_SOPT2_PLLFLLSEL` e `SIM_SOPT2_UART0SRC`), dentre os 3 sinais gerenciados pelo módulo `MCG` (Figura 5-1/página 116 em [2]): sinais internos de referência `MCGIRCLK`, sinais externos de referência `OSCERCLK` e sinais gerados por um laço de sincronismo `MCGFLLCLK/MCGPLLCLK` (Seção 5.7.7/ página 125 em [2]). A frequência dos sinais de relógio que pulsam os registradores de deslocamento nos canais TX e RX dependem desses sinais de relógio, do divisor *prescaler* configurado em 13 *bits*, 5 *bits* mais significativos em `UARTx_BDH` (Seção 39.2.1/ página 725 em [2]) e 8 *bits* menos significtivos em `UARTx_BDL` (Seção 39.2.2/ página 726 em [2]), e da taxa de superamostragem de cada *bit* setada em `UARTx_C4_OSR` (Seção 39.2.11/ página 736 em [2]).

A **superamostragem** consiste em coletar várias amostras do sinal de entrada durante cada período de *bit* e combiná-las para tomar uma decisão sobre o valor do *bit*. Ela é aplicada no canal RX para melhorar a precisão na detecção de bordas de sinais de entrada, especialmente em condições de ruído elevado. A taxa de superamostragem é fixa em 16x nos módulos `UART1` e `UART2` e selecionável em `UART0`, entre 4x a 32x, via o registrador de controle/configuração `UARTx_C4`. Como UART0 trabalha com uma taxa de superamostragem maior que 7x, é necessário setar em '1' o *bit* `UART0_C5_BOTHEDGE` se for escolhida uma taxa de amostragem entre 4x e 7x (Seção 39.2.12/ página 737 em [2]).

Em relação aos *bits* contidos em cada pacote de dados, UARTx permite configurar a quantidade de *stop bits* através do campo `UARTx_BDH_SBNS` (Seção 39.2.1/página 725 em [2]), e a quantidade de *bits* de dados por `UARTx_C1_M` (Seção 39.2.3/página 726 em [2]). A interpretação lógica dos sinais de *bits* nos canais receptor e transmissor pode ser configurada de forma independente, respectivamente pelos *bits* `UARTx_S2_RXINV` (Seção 39.2.6/página 731 em [2]) e `UARTx_C3_TXINV` (Seção 39.2.7/página 733 em [2]). O módulo permite também incluir o *bit* de paridade aos *bits* de dados, através dos campos de configuração `UARTx_C1_PE` e `UARTx_C1_PT` (Seção 39.2.3/página 726 em [2]). Esses *bits* são contabilizados como *bits* de dados. O *bit* que segue o *start bit* é o *bit* mais significativo (MSB) em UART1 e UART2. No módulo UART0, a terminação dos *bits* é configurável pelo *bit* `UART0_S2_MSBF` (*bit* menos significativo (0) ou mais significativo (1)) do registrador de estado e configuração `UART0_S2` (Seção 39.2.6/página 731, em [2]).

Os circuitos transmissor (TX) e receptor (RX) são bem versáteis, configuráveis através dos seus registradores, para uma grande gama de modos de operação. Eles são habilitados individualmente pelos *bits* `UARTx_C2_RE` e `UARTx_C2_TE` (Seção 39.2.4/página 728 em [2]). Os respectivos pinos de saída e de entrada devem ser configurados no modo de multiplexação UARTx, através dos *bits* `PORTx_PCRn_MUX` dos registradores do módulo `PORT`, para transferência dos *bits*. Pinos que podem assumir as funções `UARTx_RX` e `UARTx_TX` são listados na tabela da Seção 10.3.1/página 161 em [2].

Quando o circuito TX é habilitado, ele transfere o que estiver no registrador de dados UARTx_D (Seção 39.2.8/página 734 em [2]) para o registrador de deslocamento. Assim que o conteúdo de UARTx_D conclui uma transferência, o *bit* de estado UARTx_S1_TRDE é setado em '1' indicando a disponibilidade de UARTx_D. No registrador de deslocamento são agregados antes do envio (1) os *stop bits* configurados em UARTx_BDH (Seção 39.2.1/página 725 em [2]), (2) o nono *bit* UARTx_C3_T8 se o *bit* UARTx_C1_M está setado em '1', e (3) o *bit* de paridade configurado no registrador de configuração UARTx_C1 (Seção 39.2.3/página 726 em [2]), UART0 suporta ainda envios de caracteres de 10 *bits*. Quando estiver configurado neste estado em UART0_C3 (Seção 39.2.7/página 733 em [2]), o décimo *bit* UART0_C4_M10 é agregado ao caractere (Seção 39.2.11/página 736 em [2]). A lógica dos *bits* transmitidos é selecionável pelo *bit* de configuração UARTx_C3_TXINV. Quando termina o envio dos *stop bits* e o transmissor entra no modo ocioso (*idle*, UARTx_S1_IDLE em '1'), o *bit* de estado UARTx_S1_TC é setado em '1' (Seção 39.2.5/página 729 em [2]). Uma descrição funcional mais completa se encontra nas Seções 39.3.2/página 739 e 40.3.2/página 762 em [2].

Quando o circuito RX é habilitado, ele superamostra o sinal recebido na frequência (de taxa de superamostragem x *baud rate*) configurada em busca de uma borda de descida (do *start bit*), a fim de sincronizar com o caractere a ser recebido. Assim que detectar uma borda de descida inicia-se a amostragem dos outros *bits* subsequentes do pacote recebido. Depois de receber o *stop bit* e tranferir o caractere recebido do registrador de deslocamento para o registrador UARTx_D, o *bit* de estado UARTx_S1_RDRF é setado em '1'. Para **aumentar a confiabilidade** nos *bits* detectados, **resincronizações** são automaticamente conduzidas em cada borda de descida detectada. Em UART0 há a opção de desabilitar a resincronização através do *bit* de configuração UART5_C5_RESYNCDIS (Seção 39.2.12/página 737 em [2]). Para **reduzir o consumo energético**, é integrado no RX um **despertador do receptor** (*receiver wakeup*) que o chaveia para o estado de espera (*bit* UARTx_C2_RWU em '1') quando identifica que uma mensagem não é destinada a ele e o acorda automaticamente no fim da mensagem ou no início da próxima mensagem para que ele volte a buscar bordas de descida (Seção 39.3.3.2/página 742 em [2]). Para **assegurar a integridade** dos dados recebidos, o canal RX contém circuitos **detectores de erros** que possam ocorrer numa transmissão e sinalizam estes erros através das seguintes *flags* de estado: (1) de **erro de transbordamento/sobrecarga** (*overrun error*), UARTx_S1_OR, setado quando se recebe um novo pacote com UARTx_D ainda ocupado, (2) de **erro de ruído** (noise *error*), UARTx_S1_NF, setada quando não há concordância entre os níveis lógicos das amostras de um mesmo *bit*, (3) de **erro de quadro** (*framing error*), UARTx_S1_FE, setado quando é detectado o nível lógico '0' onde o *stop bit* é esperado, e (4) de **erros de paridade** (*parity error*), UARTx_S1_PF, quando não há concordância entre o *bit* de paridade dos *bits* amostrados e o *bit* de paridade amostrado. Uma descrição funcional mais completa se encontra nas Seções 39.3.3/página 740 e 40.3.3/página 764 em [2].

Alocação de Pinos para UARTx

Para se comunicar com o mundo externo, é necessário alocar um pino TxD Pin para o circuito TX e um pino RxD para o circuito RX como mostram a Figura 39-1/página 723 e Figura 39-2/página 724 em [2]. É, portanto, necessário alocar os pinos para os dois circuitos caso eles sejam utilizados. Os pinos que podem assumir a função de UARTx_TX e de UARTx_RX são fornecidos pelo fabricante e podem ser consultados na tabela da Seção 10.3.1/página 162 em [2]. Por esta tabela, podemos alocar por exemplo o par PTE0 e PTE1 para assumir a função de TxD e RxD Pin do módulo UART1.

Processamento de Interrupções em UARTx

Os *bits* de habilitação de interrupção correspondentes aos estados de transmissão, UARTx_S1_TRDE e UARTx_S1_TC, aos estados de recepção, UARTx_S1_RDRF, UARTx_S1_IDLE, UARTx_S2_RXEDGIF, e UARTx_S2_LBKDIF, e aos estados de erros, UARTx_S1_OR, UARTx_S1_NF, UARTx_S1_FE e UARTx_S1_PF, estão nos registradores UARTx_C2 (Seção 39.2.4/página 728 em [2]) e UARTx_C3 (Seção 39.2.7/página 733 em [2]). Quando ocorre um

evento de interrupção (*bit* de estado e o de habilitação de interrupção correspondente estiverem em 1), é gerada automaticamente uma requisição de interrupção com o número de vetor igual a 28 (IRQ=12) para UART0, 29 (IRQ=13) para UART1, ou 30 (IRQ=14) para UART2 (Tabela 3-7, página 52, em [2]). Se o controlador NVIC estiver configurado para atender a linha de interrupção solicitante, o fluxo de controle é desviado para a rotina de serviço.

Consultando o arquivo `Project_Settings/Startup_Code/kinetis_sysinit.c` gerado pelo IDE *CodeWarrior*, os nomes das rotinas de serviço declarados para as IRQs 12, 13 e 14 são, respectivamente, `UART0_IRQHandler`, `UART1_IRQHandler` e `UART2_IRQHandler`. Tendo apenas uma IRQ atribuída a cada módulo, é necessário identificar dentro da rotina de serviço as fontes das interrupções. A técnica de varredura dos *bits* de estados, por *software*, é a mais aplicada Ela consiste em testar sequencialmente os *bits* de estado de todos os eventos com interrupções habilitadas e tratar aqueles que tiverem com os *bits* de estado setados em '1'. Após o tratamento, esses *bits* devem ser resetados em '0' para remover as suas solicitações de interrupção.

Todas as *flags* relativas às condições de erros só são resetadas em '0' com um acesso de escrita de '1' (*write 1 to clear*), enquanto as relacionadas com a transmissão e a recepção são automaticamente resetadas em '0' na ocorrência de eventos específicos (Seção 39.2.5/página 729 em [2]). Os *bits* `UARTx_S1_TRDE` e `UARTx_S1_TC` são resetados em '0' quando ocorre um acesso de escrita no registrador `UARTx_D`. O *bit* `UARTx_S1_RDRF` é resetado em '0' quando ocorre um acesso de leitura do registrador `UARTx_D`. Cabe aqui uma ressalva em relação à habilitação da interrupção do canal TX. Essa só deve ser feita quando há produção de caracteres para transmissão; caso contrário, o sistema pode ficar "preso" no aguardo de um caractere para resetar a *flag* de disponibilidade do *buffer* de TX.

Configuração de Taxa de Transmissão

Taxa de Transmissão (*Baud rate*) é o número de vezes que um sinal em um canal de comunicação pode mudar de estado em um segundo. Alguns valores usuais de *baud rate* são: 115200, 9600 e 4800. Para que o sinal de relógio do circuito TX/RX do módulo UARTx opere numa taxa de transmissão específica, devemos setar os 13 *bits* de `SBR`, representados nos campos `UARTx_BDL_SBR` (8 *bits* menos significativos) e `UARTx_BDH_SBR` (5 *bits* mais significativos), de tal forma que seja satisfeita a relação mostrada na Figura 2 (Seção 8.3.2/página 78 em [3]), onde *Source Clock Frequency* corresponde à frequência do sinal `UART0 clock`:

$$\text{UART0 baud rate} = \frac{\text{Source Clock Frequency}}{\text{SBR}*(\text{UART0\_C4[OSR]}+1)}$$

$$\text{All Other UART baud rate} = \frac{\text{Bus Clock Frequency}}{\text{SBR}*16}$$

Figura 2: Relação entre as taxas de transmissão e os parâmetros configuráveis em UARTx.

O divisor *prescaler* SBR, em ponto flutuante, pode ser calculado com as expressões

$$SBR_{calculado} = \frac{UART0\,clock\,frequency}{UART0\,baud\,rate * (UART0\_C4\_OSR + 1)} \qquad (1)$$

$$SBR_{calculado} = \frac{Bus\,Clock\,Frequency}{ALL\,Other\,UART\,baud\,rate * (16)}$$

A frequência setada não deve ser maior do que a taxa de transmissão especificada. Quando $SBR_{calculado}$ é um valor fracionário, deve-se arredondar para o próximo inteiro

$$SBR_{truncado} = (uint\ 16\_t)\frac{UART\ 0\ clock\ frequency}{UART\ 0\ baud\ rate * (UART\ 0\_C4\_OSR + 1)}$$
$$SBR_{truncado} = (uint\ 16\_t)\frac{Bus\ Clock\ Frequency}{ALL\ Other\ UART\ baud\ rate * (16)} \quad (2)$$

ou seja, SBR a ser setado nos registradores `UARTx_BDH_SBR|UARTx_BDL_SBR` é

```
SBR = SBR_truncado;

if (SBR_calculado > SBR_truncado) SBR = SBR++;
```

Dos 16 bits alocados a SBR, somente os 13 *bits* menos significativos são de interesse

```
SBR = SBR & 0x1FFF;
```

Desses 13 *bits*, os 5 *bits* mais significativos devem ser setados em `UART0_BDH_SBR`

```
UART0_BDH |= UART0_BDH_SBR(SBR >> 8);
```

e os outros 8 *bits* menos significativos em `UART0_BDH_SBR`

```
UART0_BDL |= UART0_BDL_SBR(SBR);
```

Verifique este procedimento com os valores dados nos dois exemplos das Seções 8.4.1/página 84 e 8.4.2/página 87 em [3].

Configuração de Taxa de Superamostragem

A taxa de superamosgragem de um sinal no canal RX é fixa em 16x para os módulos UART1 e UART2 e é configurável sntre 4x a 32x para o módulo UART0 (Seção 39.2.11/página 736 em [2]). O valor carregado em `UART0_C4_OSR` na inicialização de KL25Z é 0xF (+1 = 16). Se quisermos alterá-lo, basta sobreescerver o novo valor observado as duas condições necessárias: (1) os canais RX e TX devem estar rdesabilitados, e (2) um acesso de escrita de uma taxa inválida em `UART0_C4_OSR` faz com que 0xF seja armazenado. Essa segunda condição invalida a prática de zerar o campo com uma máscara AND e escrever o novo valor com uma máscara OR. Uma alternativa é aplicar uma lógica invertida, setando o campo em 0b11111 com uma máscara OR:

```
UART0_C4 |= (0b11111 << UART0_C4_OSR_SHIFT);
```

e construir uma máscara AND com o novo valor, `novo_valor`:

```
mascara = (novo_valor << UART0_C4_OSR_SHIFT) | ~UART0_C4_OSR_MASK
```

para setá-lo usando operador lógico AND:

```
UART0_C4 &= mascara;
```

Conversão de Inteiros para *Strings* de Dígitos Binários e Hexadecimais

A conversão de um inteiro para representações em diferentes bases é feita dividindo o número inteiro pelo valor da base desejada e anotando o resto. O processo é repetido com o quociente da divisão até que o quociente fique zero. Os restos anotados são então lidos ao contrário para formar o número na nova base. No roteiro 6 [22] apresentamos uma implementação de conversão para base 10. Esse algoritmo envolve divisões. Por envolver várias operações de subtração e comparação para encontrar o quociente e o resto, uma divisão é computacionalmente mais complexa que outras operações aritméticas básicas, como adição, subtração e multiplicação, além de lidar com o caso de divisão por zero que gera exceções.

Quando a base é uma potência de 2, os possíveis 2^n valores de cada dígito são representáveis por `n` *bits*. Portanto, no lugar de divisões sucessivas, podemos aplicar deslocamentos sucessivos de `n` *bits* para isolar os dígitos. Por exemplo, para a base 2 (binária), vimos no reoteiro 1 [17] que se aplica uma máscara com o *bit* '1' deslocado de m posições (1<<m) sobre o valor para isolar o seu m-ésimo *bit*. Ao percorrer m, de 0 até o *bit* mais significativo, e concatenar todos os *bits* isolados, teremos uma *string* de dígitos binários. Em [26] encontra-se uma implementação em C dessa técnica baseada em deslocamentos. Para bases de potências maiores, como octal (2^3) ou hexadecimal (2^4), usamos máscaras de 3 *bits* (0b111<<m) e 4 *bits* (0b1111<<m), respectivamente, para isolar os dígitos. Uma implementação em C de conversão de inteiros em 32 *bits* para *strings* hexadecimais é deslocar a máscara 0xf=0b1111 de m *bits* em incrementos de 4 *bits,* extrair os 4 *bits nas* posições/ordens m, m+1, m+2 e m+3, convertê-los para ASCII e concatená-los para formar uma *string* de dígitos hexadecimais em ASCII.

Para a conversão de um algarismo `v` em hexadecimal para um código ASCII, que pode ser um dígito decimal ('0'-'9' ou 0x30-0x39), uma letra minúscula ('a'-'f' ou 0x61-0x66) ou uma letra maiúscula ('A'-'F' ou 0x41-0x46), precisamos diferenciar os dígitos, cujos valores numéricos estão entre 0 a 9, das letras, cujos valores estão entre 10 a 15. Para a primeira faixa de valores, basta `v+0x30`, como vimos no roteiro 6 [22]. E para o segundo intervalo de valores, podemos aplicar `v-10+'a'` (minúsculas) ou `v-10+'A'` para obter códigos ASCII correspondentes.

Conversão de *Strings* de Dígitos Binários e Hexadecimais para Inteiros

Um número é representado por uma sequência de algarismos posicionais, cujos pesos no valor do número dependem da sua posição/ordem `n` , contando a partir de 1 da direita para esquerda, na sequência e da base `b` em que o número é representado. Para converter uma sequência de algarismos posicionais no valor numérico N do número, iniciamos com `N=0`. Varremos os algarismos da direita para esquerda. Para cada algarismo `A`, extraimos o seu valor numérico `v` e acumular este valor ponderado pelo seu peso posicional em `N`. Se os algarismos estiverem codificados em ASCII, a extração do seu valor numérico pode ser feita pelo comando condicional `if-else` em C:

```
if (A >= 0x30 && A <= 0x39)  v = A - '0';
else if (A >= 0x41 && A <= 0x46)  v=A-'A';
else if (A >= 0x61 && A <= 0x66)  v=A-'a';
```

Em C você pode ainda usar a função `strtol` da biblioteca `stdlib.h` para converter *strings* de algarismos de qualquer base para valores inteiros.

Comunicação entre UART0 e Terminal (Periférico)

No *kit* de desenvolvimento FRDM-KL25Z, o módulo UART0 é conectado através dos pinos PTA1 e PTA2 ao microcontrolador do OpenSDA (OpenSDA MCU), uma interface serial USB (*Universal Serial Bus*) (Seção 5.2/página 7 em [4]). No OpenSDA estão residentes o *P&E Debug Application,* com o qual podemos depurar os códigos executados no MCU, e a interface CDC (USB *Communications Device Class*) que faz a "ponte" entre as linhas TX e RX do processador-alvo e a interface USB. Portanto, se tivermos um emulador de terminal no nosso computador-hospedeiro podemos digitar caracteres no terminal e enviá-los para serem processados no MCU e receber os caracteres gerados no MCU.

Em [5] encontram-se as dicas para instalar o *plugin* RxTx ("Terminal") no IDE *CodeWarrior*. A comunicação do "Terminal" [6] com o microcontrolador se dá através do módulo UART0. Para abrí-lo no ambiente IDE, basta seguir o caminho **Windows > Show View > Other … > Terminal**. (Seção 2.8.1/página 43 em [8]). Aparecerá uma aba na janela do canto inferior direito e uma janela "*Terminal Settings*", através da qual é possível configurar os parâmetros de comunicação serial no módulo UARTx, como ilustra a Figura 3.

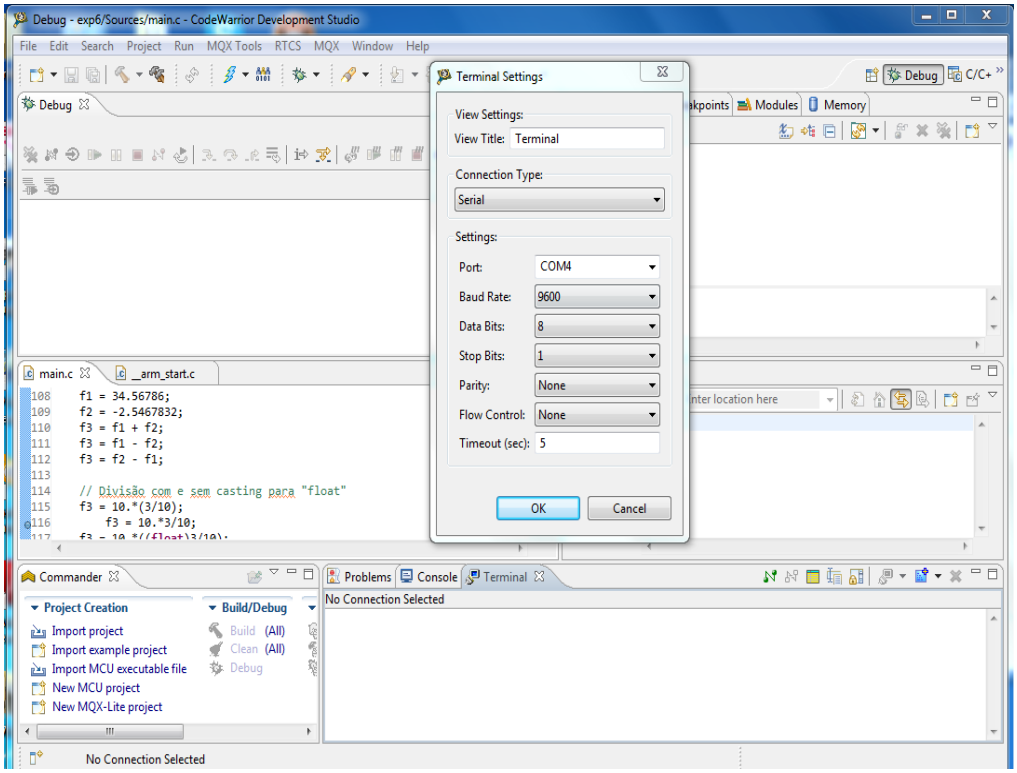

Figura 3: *Pop-up* menu de configuração dos parâmetros seriais do lado do computador-hospedeiro.

Para se comunicar, por exemplo, com o projeto `rot7_aula` [7] instalado no microcontrolador, a configuração do "Terminal" deve ser: porta COM com a qual o microcontrolador está conectado, *baud rate* 9600, caracter de 8 *bits*, sem *bit* de paridade, 1 *stop bit* e sem fluxo de controle. Para identificar a porta em que o microcontrolador está conectado, acesse o Gerenciador de Dispositivos, expanda o item "Portas (COM e LPT)" e veja a COM em que está conectado o dispositivo "OpenSDA- CDC Serial Port".

Tipo de Dado Struct e Typedef em C

Estrutura (Struct) em C é um tipo de dado composto que permite declarar variáveis com diferentes tipos de dados dentro de um mesmo bloco de memória em uma única estrutura, e essas variáveis são acessadas através de nomes significativos. A utilização de structs em C permite acessar o conteúdo de memória de forma semântica, ou seja, com significado. Isso facilita a compreensão e manutenção do código, pois os dados são acessados com base em seu significado semântico, e não apenas com base nos seus endereços de memória. A declaração dela é através da palavra reservada `struct`, seguida pelo nome da struct e pelas variáveis-membro (campos) delimitadas pelas chaves. Essas variáveis-membro podem ser de qualquer tipo de dado válido em C, incluindo outras structs.

`struct` <nome> {

<tipo_de_dado> membro1;

<tipo_de_dado> membro2;

⋮

}

Ao usá-la para instanciar uma variável, os seus membros podem ser acessados via o operador '`.`'. E ao usá-la para instanciar um ponteiro, os seus campos podem ser acessados com o operador "→". No arquivo `Project_Headers/MKL25Z4.h` gerado pelo IDE *CodeWarrior* são definidas várias structs que mapeiam os endereços físicos dos registradores em nomes utilizados pelos fabricante nos seus manuais, como

```
struct PORT_MemMap {
  uint32_t PCR[32]; /**< Pin Control Register n, array offset: 0x0, array
step: 0x4 */
  uint32_t GPCLR; /**< Global Pin Control Low Register, offset: 0x80 */
  uint32_t GPCHR; /**< Global Pin Control High Register, offset: 0x84 */
  uint8_t RESERVED_0[24];
  uint32_t ISFR;  /**< Interrupt Status Flag Register, offset: 0xA0 */
}
```

Para aumentar a legibilidade do código, podemos criar sinônimos (alias) para structs com o comando `typedef`, como

```
typedef struct PORT_MemMap volatile *PORT_MemMapPtr;
```

que define para o ponteiro do tipo de dado `struct PORT_MemMap` com qualificador `volatile` o novo nome mais compacto `PORT_MemMapPtr`, o qual o IDE *CodeWarrior* utilizou para mapear os endereços dos registradores de PORTx:

```
((PORT_MemMapPtr)0x40049000u)

((PORT_MemMapPtr)0x4004A000u)

((PORT_MemMapPtr)0x4004B000u)

((PORT_MemMapPtr)0x4004C000u)

((PORT_MemMapPtr)0x4004D000u)
```

Em vista da quantidade de parâmetros (campos) distribuídos em 12 registradores para configurar o modo de operação do módulo UART0, podemos definir em C um novo tipo de dado, por exemplo `struct UART0Config_tag,` que agrupa todos os registradores num único registro e processá-los como uma única instância:

```
typedef struct UART0Configuration_tag {
      uint8_t bdh_sbns;   ///< selecionar quantidade de stop bits (0 = 1; 1 = 2)
      uint16_t sbr;               ///< divisor prescaler de baud rate
      uint8_t c1_loops;   ///< operação em loop (normal = 0)
      uint8_t c1_dozeen;  ///< habilitar espera (doze)
      uint8_t c1_rsrc;    ///< habilitar a saída do TX em operação de loop
      uint8_t c1_m;       ///< 8-bit (0) ou 9-bit de dados
      uint8_t c1_wake;    ///< forma de wakeup do RX (0-idle line; 1-address-mark)
      uint8_t c1_ilt;     ///< selecionar a forma de detecção de "idle line"
      uint8_t c1_pe;      ///< habilitar paridade
      uint8_t c1_pt;      ///< tipo de paridade (0-par; 1-ímpar)
      uint8_t c2_rwu;     ///< setar o RX no estado de standby aguardando pelo wakeup
      uint8_t c2_sbk;     ///< habilitar o enfileiramento de caracteres break
      uint8_t s2_msbf;    ///< setar endianess para MSB (0 = LSB;1 = MSB)
      uint8_t s2_rxinv;   ///< habilitar a inversão da polaridade dos bits do RX
      uint8_t s2_rwuid;   ///< habilitar o set do IDLE bit durante standby do RX
      uint8_t s2_brk13;   ///< selecionar o comp. de caracter break (0=10 bits; 1=13 bits)
      uint8_t s2_lbkde;   ///< habilitar detecção de caractere break longo
      uint8_t c3_r8t9;    ///< bit 8 (RX)/bit 9 (TX)
    uint8_t c3_r9t8;      ///< bit 9 (RX)/bit 8 (TX)
    uint8_t c3_txdir;     ///< para RSRC=1, configurar o sentido de TX (0=entrada; 1=saída)
    uint8_t c3_txinv;     ///< habilitar a inversão da polaridade dos bits de TX
    uint8_t c4_maen1;     ///< habilitar controle de "match address" 1
    uint8_t c4_maen2;     ///< habilitar controle de "match address" 2
```

```
        uint8_t c4_m10;          ///< selecionar o modo de bits (0=8/9 bits;1=10 bits)
        uint8_t c4_osr;          ///< taxa de super-amostragem (default=16(0b01111)
        uint8_t c5_tdmae;   ///< habilitar transmissão por DMA
        uint8_t c5_rdmae;   ///< habilitar recepção por DMA
        uint8_t c5_bothedge;     ///< amostrar os dados em ambas as bordas do clock de baud rate
        uint8_t c5_resyncdis;    ///< desabilitar o resincronismo nas transições de 1 para 0
    } UART0Config_type;
```

No projeto `rot7_aula` [7] é aplicada essa estrutura para declarar a variável `config0`, cujo endereço pode ser acessado através do operador endereço-de '`&`'. Os seus membros são acessados pelo operador '.', inclusive nas suas inicializações no momento da declaração de uma variável.

***Buffers* Circulares**

A velocidade de processamento dos módulos UARTx é muito menor do que a do processador. Para compatibilizar os dois passos distintos de processamento sem sacrificar a capacidade de processamento do processador, é bastante comum usar nos projetos de sistemas embarcados os ***buffers* circulares** [9] para armazenar os dados seriais de entrada e de saída. Um *buffer* circular é, de fato, uma estrutura de dados fila com as pontas conectadas (o elemento seguinte ao último é o primeiro da fila). O diferencial dessa estrutura em relação às clássicas filas está na forma como é feita a reorganização dos dados em cada remoção e na forma como o espaço de memória é ocupado. Na fila, todos os elementos devem ser deslocados de uma casa quando se remove o primeiro elemento. Porém, no *buffer* circular, os seus elementos não são deslocados quando se retira um elemento da fila. A manipulação dos dados é feita por dois ponteiros, denominados *head* (cabeça da fila) e *tail* (final da fila). Quando se adiciona um elemento, o *head* é incrementado ciclicamente, e quando se retira um elemento do final da fila, o *tail* é incrementado ciclicamente. Na Figura 4, sob a perspectiva de uma fila clássica, se removermos o elemento '0' da fila, os elementos '1' a '14' serão deslocados de uma casa para esquerda e *pointer* tem o seu endereço deslocado de uma casa. Já no *buffer* circular, a remoção do elemento '0' faz com que somente *tail* tenha o endereço deslocado de uma casa. O conteúdo da memória não é alterado. Como os ponteiros são incrementados ciclicamente no *buffer* circular, os endereços de memória acessados estão sempre dentro do espaço pré-reservado, enquanto na fila, o *pointer* pode vazar para fora do espaço previamente reservado.

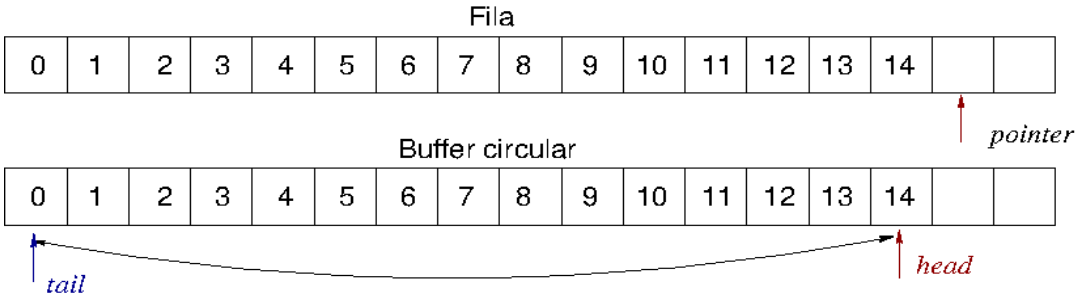

Figura 4: Fila e *buffer* circular.

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

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

O projeto `rot7_aula` [7] demonstra o uso de *buffers* circulares tanto na recepção de caracteres teclados (produtor = usuário+UART e consumidor = KL25Z) quanto na transmissão dos caracteres para o Terminal (produtor = KL25Z e consumidor = UART+Terminal), embora o problema de compatibilidade da velocidade do produtor com a do consumidor seja mais crítico na transmissão.

Caracteres de Controle

Um **caractere de controle** é um caractere não renderizável, pertencente a um conjunto de códigos que representam símbolos de escrita, com uma função específica reconhecida universalmente. Todos os códigos abaixo de 32 (0x20) da tabela ASCII são caracteres de controle. Ao serem inseridos numa *string*, eles podem alterar a disposição doe 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*) sobreescreve o último caractere renderizado, 0x0A (*line feed*) marca o fim de uma linha, e 0x0D (*carriage return*) retorna o carro (da máquina de escrever) para a primeira coluna. Para representá-los 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*, *line feed* e *carriage return* são, respectivamente, "\a", "\b", "\n" e "\r" [10], No projeto `rot7_exemple2` [20] é demonstrado o uso de caracteres de controle "\n\r" no controle da forma como uma mensagem é mostrada num Terminal.

Extração de Tokens em *Strings*

Terminais usam uma interface de linha de comando (*command-line interface*, CLI), porque ela oferece uma forma poderosa e flexível de interagir com o sistema operacional, é independente de plataforma e requer menos recursos do sistema. Uma linha de comando é uma linha contendo um comando e os seus argumentos separados por separadores. Ela é executada quando se pressiona "enter". Para implementarmos uma interface de linha de comando em KL25Z4, precisamos processar os caracteres digitados pelo usuário por linhas.

Cada pacote transmitido por um módulo UARTx contém um quadro de dados de 5 a 9 *bits*, que é tipicamente representado em C pelo tipo de dado `char`. Naturalmente, uma sequência de pacotes de dados correspondentes a uma linha de caracteres no terminal é armazenada como uma *string* (vetor de elementos do tipo `char`) depois de adicionar ao final da sequência o terminador '\0'. Pode-se, então, aplicar uma série de funções disponíveis na biblioteca-padrão de C, como `strcmp`, `strlen` [13], para manipulá-las. Antes de processar o comando contido numa linha, precisamos extrair os tokens ou "unidades de informação", como o comando e os seus argumentos, contidos na linha. Esses são comumente separados por separadores como ',', '.', ';', ou espaço branco.

A função `strtok (char *str, const char *lista_delimitadores)` da biblioteca padrão C é um excelente candidato para extrair os tokens, espaçados pelos separadores que constam em `lista_delimitadores`, de uma linha de caracteres `str` digitada no Terminal [14]. A função varre `str` quando é chamada pela primeira vez, **substitui** todos os delimitadores em `lista_delimitadores` pelo terminador '\0' e retorna o endereço da primeira *sub-string*. Nas chamadas subsequentes de `strtok` com o primeiro argumento setado em NULL, a função retorna o endereço inicial da seguinte *sub-string*, e assim sucessivamente até que `strtok` retorne um endereço NULL ou esgota todos os caracteres da *string* original. Como resultado final, temos os endereços das sub-strings que compõem `str`. Figura 5 ilustra o procedimento.

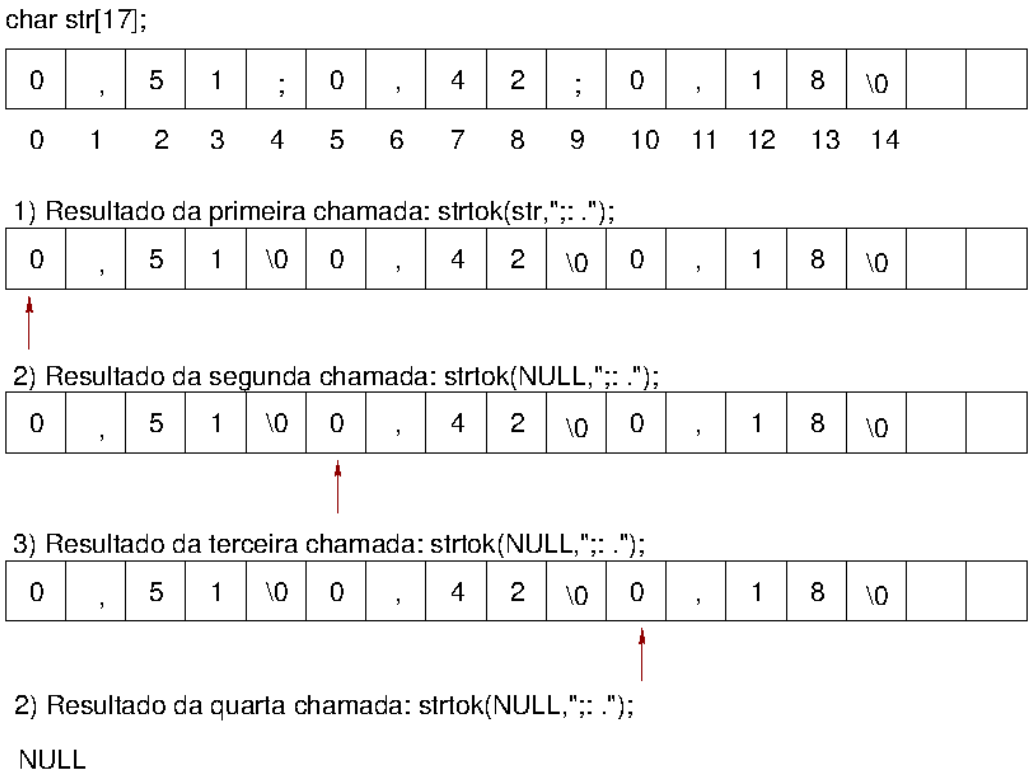

Figura 5: Procedimento implementado em `strtok`

Processamento de *Strings*

Duas outras funções da biblioteca padrão C que podem ser úteis na implementação do projeto deste roteiro são:

- `char * strcpy ( char * destination, const char * source)` [12]: fazer uma cópia da *sub-string* `source` em *sub-string* `destination`. O espaço de memória alocado a `destination` deve ser maior ou igual ao espaço de `source`. Por exemplo, a fim de preservar uma versão original da *string,* podemos fazer uma cópia de uma *string* antes de processá-la com `strtok` como ilustram as seguintes chamadas

```
char str_tmp[100];
strcpy (str_tmp, string);
strtok (str_tmp, "; -");
```

- `char * strcat ( char * destination, const char * source)` [15]: anexa uma *sub-string* `source` ao final da *sub-string* `destination`. É necessário alocar à variável `destination` um espaço de memória suficiente para a quantidade total de caracteres em `destination` e `source`. Por exemplo, para construir uma frase "Ciclo de trabalho: 0.45", podemos usar as seguintes instruções

```
char string[100]="Ciclo de trabalho: ";
char sub_string[5]="0.45";
strcat (string, sub_string);
```

Vale destacar que o conteúdo de uma *string* `str` é alterado na execução do comando `strtok`. Recomenda-se aplicar esse comando sobre uma cópia de `str`. Traduzindo em código C essa ideia de cópia e a ideia apresentada na Figura 5, segue-se uma implementação de extração dos endereços dos tokens de `str` num vetor `sub_str`

```
strcpy (copia_str, str);
```

```
i=0;
sub_str[i] = strtok(copia_str, ";:. ");
while (sub_str[i] != NULL) {
   sub_str[++i] = strtok(NULL, ";:. ");
}
```

Diretiva #if em C

Como vimos no roteiro 1 [17], uma **diretiva** para compilador C é uma instrução que fornece informações adicionais para o compilador sobre como um programa deve ser compilado. Elas são geralmente precedidas por '#'. Já usamos as diretivas `#include`, que inclui tipicamente os protótipos das funções usadas em um programa, e `#define`, que define uma constante ou uma macro. O compilador C suporta **diretivas condicionais**. A sintaxe da diretiva condicional de inclusão de um bloco de instruções na condição verdadeira é

```
#if <expr>
//bloco de código a ser compilado se <expr> == Verdadeira
#endif
```

onde `<expr>` é uma expressão constante que pode ser verdadeira ou falsa.

Quando se quer incluir um bloco de instruções alternativo caso `<expr>` seja falsa, usa-se a seguinte diretiva

```
#if <expr>
//bloco de código a ser compilado se <expr> == Verdadeira
#else
//bloco de código a ser compilado se <expr> == Falsa
#endif
```

A diretiva `#if` é usada no exemplo 1 de configuração de UARTx na Seção 8.4.1/página 84 em [3].

Parametrização de Blocos de Memória

Parametrizar um bloco de instruções significa definir um conjunto de variáveis de entrada de um determinado trecho de código, de forma que um mesmo bloco de instruções possa ser reutilizado para diferentes conjuntos de valores. Tipicamente, esses valores são passados como argumentos para a função ou método que contém o bloco de instruções parametrizado.

Prametrizar um bloco de memória em C significa definir uma mesma struct para diferentes regiões de memória, de forma que os nomes dos seus membros possam ser reutilizados para acessar diferentes endereços e serem processados por um mesmo bloco de instruções. Por exemplo, os registradores de UART1 e UART2 estão mapeados nos blocos de endereços [0x4006B000,0x4006B008] e [0x4006C000,0x4006C008] e acessíveis via a `struct UART_MemMap` definida em `Project_Headers/MKL25Z4.h`. Através das macros definidas nesse arquivo

```
#define    UART1_BASE_PTR      ((UART_MemMapPtr)0x4006B000u)
#define    UART2_BASE_PTR      ((UART_MemMapPtr)0x4006C000u)
#define    UART_BASE_PTRS      { UART1_BASE_PTR, UART2_BASE_PTR }
```

podemos parametrizar acessos aos registradores e seus processamentos, declarando um vetor de ponteiros

```
UART_MemMapPtr uart[] = UART_BASE_PTRS;
```

que equivale à declaração de um vetor dos endereços-base de duas structs:

```
UART_MemMapPtr      uart[]      =      {((UART_MemMapPtr)0x4006B000u),
((UART_MemMapPtr)0x4006C000u)};
```

A função `UART_configure` de `rot7_aula` [7] parametriza um bloco de instruções que configuram os registradores de UART1 e UART2. Mesmo de endereços distintos, só precisamos alterar o valor do elemento do vetor `uart`, `uart[0]` ou `uart[1]`, para acessá-los com os mesmos nomes. Por exemplo, os identificadores `uart[0]->BDL` e `uart[1]->BDL` são abstrações dos endereços físicos 0x4004B001 e 0x4006C001 da memória onde os registradores `UART1_BDL` e `UART2_BDL` são mapeados.

Proteção de Regiões Críticas pelas Restrições nos Estados do Sistema

Embora a recepção e a transmissão de caracteres aconteçam em canais distintos, elas podem compartilhar o Terminal para mostrar os caracteres que estão processando. Neste caso, dizemos que há uma **concorrência ao uso** do (recurso) Terminal que precisamos gerenciar para evitar acessos conflitantes. Vimos no roteiro 6 [22] uma estratégia de proteção é desabilitando as interrupções. Uma outra estratégia seria usar uma máquina de estados através da implementação de regras de transição de estado e regras que restrinjam as ações permitidas nos estados. Por exemplo, no projeto `rot7_aula` [7] distinguimos 4 estados, `ESPERA`, `EXTRAI`, `MOSTRA` e `LIBERA_BUFFER` (Figura 6). No estado `EXTRAI` não é permitida uma nova entrada em `UART0_IRQHandler`, já que há uma entrada sendo processada. No estado `MOSTRA`, embora o sistema esteja ocupando o Terminal, ele está pronto para interromper o que está mostrando quando o usuário digita algum caractere – é permitida a entrada de um novo caractere. Mas, para evitar acessos concorrentes do Terminal, a transição do estado `MOSTRA` para o estado `ESPERA` é condicionada à passagem pelo estado `LIBERA_BUFFER` em que o *buffer circular* de saída é esvaziado e os caracteres digitados pelo usuário não são ecoados.

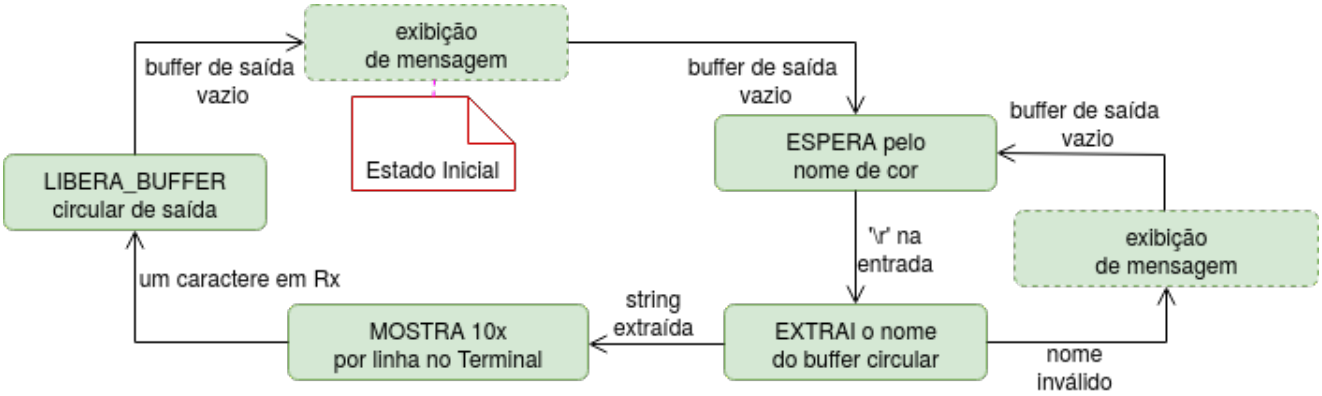

Figura 6: Diagrama de máquina de estados do projeto `rot7_aula` (editado em [19]).

EXPERIMENTO

Neste experimento vamos desenvolver o projeto `paridade`, É aceito como entrada uma linha de comando `<op> <valor>`, em que `<valor>` é um número de 32 bits e `<op>` o tipo de paridade. O valor é precedido de H/h(exadecimal), B/b(inário) ou sem nada (decimal) e os dois tipos de paridade são identificados com P/p(ar) e I/i(mpar). Os tokens `<valor>` e `<op>` devem ser separados por um espaço branco. Assuma que letras podem ser maiúsculas ou minúsculas em hexadecimal. A figura 7 mostra um diagrama de máquina de estados do projeto, mostrando as transições entre os seis estados do sistema.

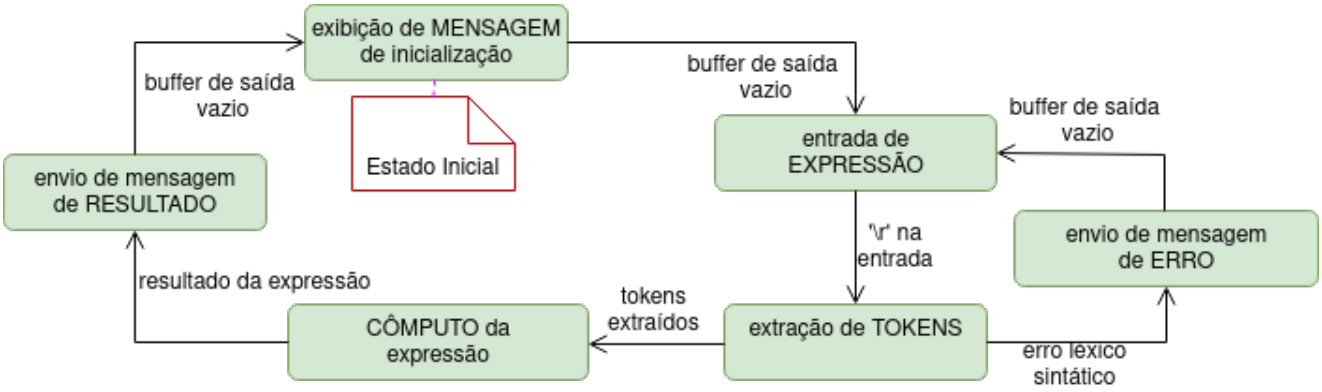

Figura 7: Diagrama de máquina de estados do projeto `paridade` (editado em [19]).

No estado `MENSAGEM` é enviado ao Terminal a *string*: `"Entre <P/p/I/i> <tipo><valor> (tipo: b/B/h/H)\n\r"`. Ao inserir uma linha de expressão, finalizada com a tecla *Return*, são extraídos e validados os 2 TOKENS. Se não ocorrer nenhum `ERRO` (isto é, os valores numéricos e o operador válidos), é feito o CÔMPUTO da paridade de `<valor>` e gerado o resultado em *bit* de paridade ímpar (1 se a quantidade de *bits* é ímpar; 0 se é par). Monta-se uma mensagem contendo os RESULTADOs: `"<valor em representação binária> tem ~~a paridade <tipo de paridade> em <bit de paridade>~~uma quantidade impar de 1. O digito de paridade <tipo de paridade>: <bit de paridade>."` e enviada para o Terminal. Assim que concluir o envio da mensagem, volta-se para o estado `MENSAGEM` aguardando por um novo comando, e assim sucessivamente. Caso algum erro for detectado na extração de `TOKENS`, é construída uma das seguintes mensagens de aviso de erro, `"Valor inteiro invalido.\n\r"`, `"Quantidade de tokens incorreta"`, `"Tipo de paridade incorreto"`, e aguarda por uma nova `EXPRESSAO`. Os textos da MENSAGEM e do RESULTADO são sugestões. A figura 8 ilustra uma interface do aplicativo.

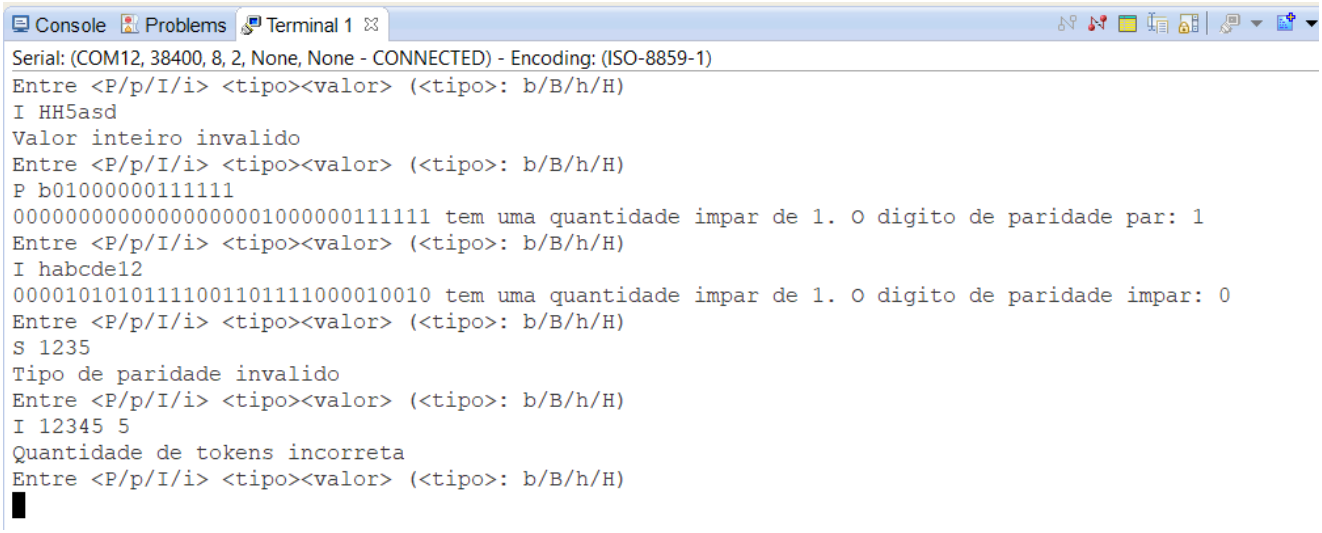

Figura 8: Interface do projeto `paridade`.

Dois *buffers* circulares, `bufferE` e `bufferS` com 80 elementos (caracteres), são aplicados no projeto, um para recepção de caracteres e outro, para transmissão como ilustra a Figura 9. Todos os caracteres recebidos pelo canal RX são armazenados no *buffer* circular de entrada e, antes do processamento, são extraídos e reformatados em *strings* substituindo o caractere de controle '\r' pelo terminador '\0'. Os resultados são armazenados no *buffer* circular de saída e enviados, **por interrupção**, ao Terminal.

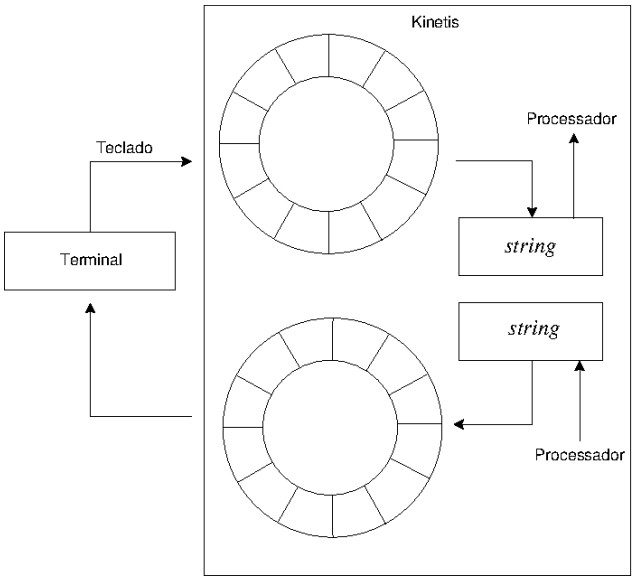

Figura 9: Um *buffer* circular para recepção e outro para transmissão.

A figura 10 mostra, através de um diagrama de componentes, como os componentes de *hardware* (em vermelho) e *software* (em preto) interagem para atender a configuração necessária ao projeto.

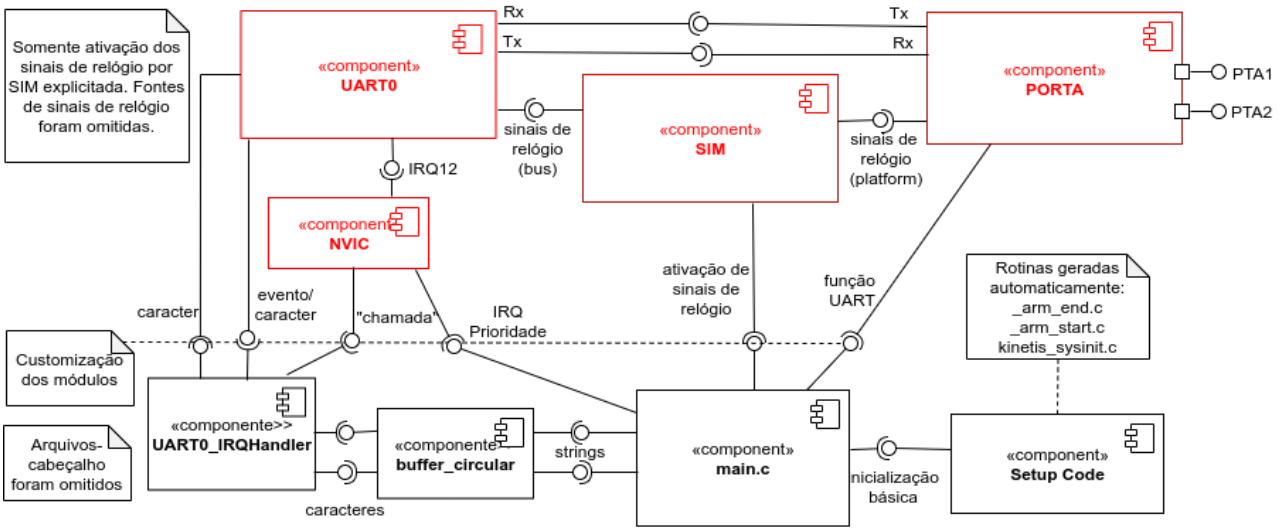

Figura 10: Diagrama de componentes do projeto `paridade` (editado em [19]).

A fonte de sinais de relógio é a padrão, MCGFLLCLK em 20971520Hz. O modo de operação do módulo UART0 especificado é *baud rate 38400*, caracter de 8 *bits*, 2 *stop bits*, sem *bit* de paridade e taxa de amostragem 12x. Os *buffers* circulares (1 de entrada e 1 de saída) tem ~~80~~ 100 *bytes*.

Segue-se um roteiro para o desenvolvimento do projeto. Usa-se na implementação do projeto as macros do arquivo-cabeçalho `derivative.h`.

1 **Aprender com os Exemplos dos Manuais**: Na seção 8.4/página 84 em [3] são apresentados dois exemplos de configuração de UART0. O Exemplo 1 é uma aplicação em transmissões por *polling*/interrupção usando como MCGFLLCLK em 48MHz como fonte de sinal de relógio para controle de taxa de transmissão (Seção 8.4.1/página 84 em [3]). E o Exemplo 2 demonstra o modo de operação do UART0 em estados de espera com um consumo de potência muito baixo tendo como MCGIRCLK em 4MHz a fonte de sinal de relógio para controle de taxa de transmissão (Seção 8.4.2/página 87 em [3]). Como o foco é na configuração do UART0, assume-se que o leitor já tenha conhecimento da configuração do módulo MCG (*Multipurpose Clock Generator)* para implementar

os exemplos a partir dos blocos de instruções sugeridos. As instruções em *assembly* utilizadas nos exemplos são detalhadas nas Seções A6.7.76/página 198 (`WFI`) e B1.4.3/página 214 (`CPSIE`) em [21].

Os projetos `rot7_example1` [16] e `rot7_example2` [20] são as implementações completas em C dos dois exemplos. As instruções de configuração recomendadas são divididas em 3 partes: configuração da fonte dos sinais de relógio (`MCG_` e `SIM_`), habilitação do sinal de relógio e multiplexação dos pinos (`UART0_config_basica`), e configuração do módulo propriamente dito (`UART0_config_especifica`). Execute os dois projetos no modo *Debug*.

1.1 **Diretiva #if em C**: O projeto `rot7_example1` suporta dois modos de operação de recepção, *polling* e interrupção. Qual é o modo de operação configurado para a versão disponível? Qual modificação deve ser feita em `ISR.h` para chavear para o outro modo de operação antes de regerar um novo executável? Certifique as suas respostas colocando dois pontos de parada, um dentro do laço for da função `main` (`main.c`) e outro dentro da rotina de serviço `UART0_IRQHandler` (`ISR.c`).

1.2 **Protocolos de Comunicação:** Ajuste os parâmetros de comunicação serial do **Terminal** e conecte-o com a porta Serial OpenSDA antes de executar os programas carregados no microcontrolador: ~~*1 stop bit,*~~ sem *bit* de paridade. Somente a taxa de transmissão e a quantidade de *stop bits* mudam: 115200, 1 *stop bit* (`rot7_example1`) e 9600, 2 *stop bits* (`rot7_example2`).

1.3 **Módulos UARTx:** Os dois projetos mostram as configurações adicionais às do fabricante na inicialização do microcontrolador para termos os modos de operação especificados.

1.3.1 Associe aos blocos de instruções recomendados em [3] os blocos de instruções implementados em `rot7_example*` responsáveis por tais configurações adicionais. Quais são as diferenças encontradas? Tente justificar as diferenças encontradas. Se tiver dificuldades para seguir o fluxo do código, execute, passo a passo, as funções `MCG_`, `SIM_`, `UART0_config_basica` e `UART0_config_especifica`.

1.3.2 Quais blocos de instruções foram acrescentados em `rot7_example*`? Se removê-los, os programas atenderiam as especificações dadas em [3]?

1.4 **Processamento de Interrupções em UARTx**: Qual é o modo de operação do canal Tx? Por *polling* ou por interrupção? Justifique.

1.5 **Configuração de Taxa de Transmissão**: Nos exemplos os valores setados em `UART0_BDH` e `UART0_BDL` são pré-calculados e codificados. Em `rot7_example*` eles são calculados em função da frequência da fonte de sinal de relógio, da taxa de transmissão e da taxa de superamostragem. Identifique o bloco de instruções que faz esse cálculo e compare os valores computados em `rot7_example*` com os valores calculados em [3].

1.6 **Configuração de Taxa de Superamostragem**: Nos dois exemplos é especificado "*oversampling ratio 16*".

1.6.1 Compare o bloco de instruções que seta essa taxa em `rot7_example1` e as instruções recomendadas em [3]. Descreva sucintamente o procedimento implementado em `rot7_example1`.

1.6.2 Em `rot7_example2` há alguma instrução que seta `UART0_C4_OSR`? É de fato necessárias essas instruções quando a taxa especificada é 16? Responda com base na descrição de `UART0_C4` na Seção 39.2.11/página 736 em [2].

2 O projeto `rot7_aula` [7] demonstra configurações do módulo UART0 para receber pelo Terminal uma das cores em maiúscula "VERDE", "VINHO", "VIOLETA", "VERMELHO" e "VIRIDIANO". Além de ecoar o que foi digitado, o módulo UART0 envia para o Terminal uma sequência de linhas

contendo 10 vezes a cor digitada, até que uma nova cor seja entrada. As configurações do módulo UART0 foram espelhadas no módulo UART2 para possibilitar a visualização das formas de onda geradas em um canal do analisador lógico (pino 2 do *header* H5). Deve-se configurar o canal do analisador para interpretar o seu sinal segundo o padrão "*Async Serial*", conforme os parâmetros de comunicação serial do módulo UART2. A figura 11 ilustra a configuração do canal 0 do analisador para fazer amostragem segundo o protocolo *Async Serial*, no menu "*Analyzers*" à direita. Selecionando "*Async Serial*" aparece um outro menu através do qual se especifica os parâmetros de comunicação do sinal a ser amostrado.

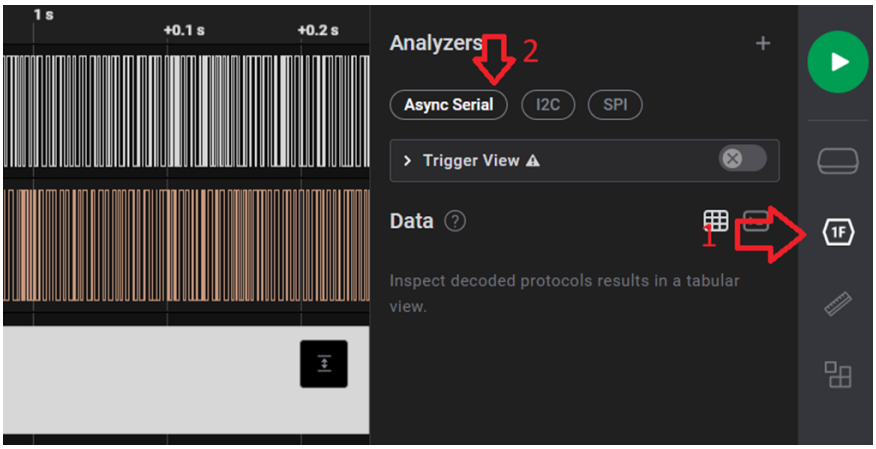

Figura 11: Configuração de um formato específico para as amostras num canal do analisador.

A figura 12 ilustra a especificação do sinal a ser amostrado. Ele possui *baud rate* de 4800, com 8 *bits* de dados, 1 *stop bit* e sem *bit* de paridade.

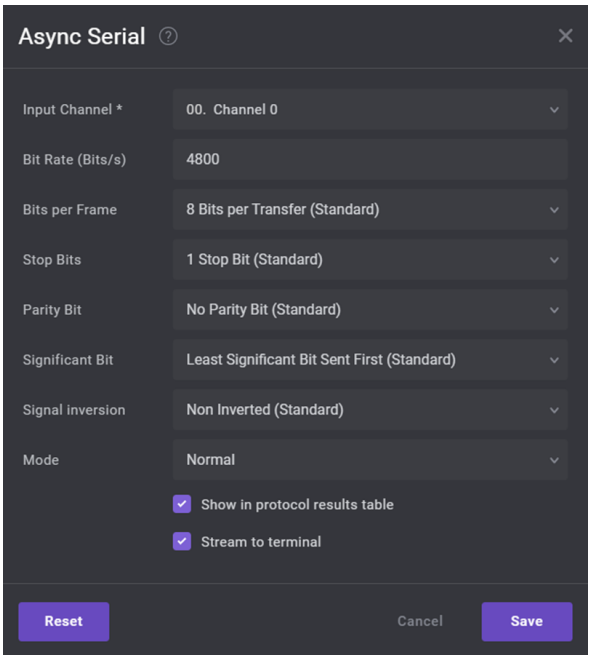

Figura 12: Parâmetros de comunicação serial para um canal do analisador lógico.

O projeto `rot7_aula` [7] inclui uma implementação do *buffer* circular (`Sources/buffer_circular.c` e `Project Headers/buffer_circular.h`).

2.1 **Tipo de Dado Struct**: Compare a configuração de um modo de operação do módulo UART0 nos projetos `rot7_example*` com a configuração implementada em `rot7_aula` usando o

tipo de dado `struct UART0Config_type` com inicializações. Qual das duas opções é mais flexível e simples na reconfiguração de um novo modo de operação?

2.2 **Protocolos de Comunicação**: Ajuste os parâmetros de comunicação serial do **Terminal** e do **canal 2 do analisador lógico** em *baud rate* 4800, 1 *stop bit*, sem *bit* de paridade. Execute `rot7_aula` e capture os sinais com o analisador lógico por um intervalo de 2s.

2.2.1 Identifique os *bits* de dados, *start bit, stop bits,* a terminação dos *bits* (LSB ou MSB do caractere após o *start bit*?) e a lógica dos *bits* (Nível alto corresponde a '1').

2.2.2 Altere os valores das variáveis `config0` e `config2` para que a configuração de comunicação serial possua 2 *stop bits* e sem *bit* de paridade. Qual é o impacto de "2 *stop bits*" no sinal capturado?

2.3 **Configuração de Taxa de Transmissão**: Em `rot7_aula` a frequência de MCGFLLCLK é a frequência-padrão, 20971520Hz.

2.3.1 Meça a largura de pulso de um *bit* e certifique se está condizente com a taxa de transferência (*baud rate*), ou com os valores setados nos registradores `UART0_BDH_SBR` e `UART0_BDH_SBR`, configurada para o módulo UART2/UART0?

2.3.2 Altere o código do projeto para que a configuração de comunicação serial tenha um *baud rate* 38400, 2 *stop bits* e sem *bit* de paridade. Certifique se a forma de onda de um quadro de dados está condizente com as configurações.

2.4 **Processamento de Interrupções em UARTx**: O modo de operação dos canais RX e TX é por interrupção. Indique no código os pontos em que as interrupções dos canais RX e TX são habilitadas e desabilitadas. Justifique a estratégia aplicada nas habilitações e desabilitações.

2.5 **Proteção de Regiões Críticas:** Quais regras de transição devem ser satisfeitas nas transições `ESPERA` → `EXTRAI`, `EXTRAI` → `MOSTRA`, ~~`EXTRAI`~~`MOSTRA` → `ESPERA`, `MOSTRA` → `LIBERA_BUFFER` e `LIBERA_BUFFER` → `ESPERA`? Quais restrições são impostas em relação à entrada, por interrupção, dos caracteres digitados pelo usuário nos estados `LIBERA_BUFFER`, `EXTRAI`, `MOSTRA` e `ESPERA`?

2.6 ***Buffers* Circulares**: Foram alocados 3 *buffers* circulares, `bufferE`, `buffer0` e `buffer2`.

2.6.1 Qual é a quantidade máxima de caracteres que esses *buffers* podem armazenar?

2.6.2 Qual é a função de cada *buffer* no projeto?

2.6.3 Qual é a técnica aplicada para assegurar processamentos cíclicos de um vetor de elementos no projeto? E qual estratégia adotada quando um *buffer* estiver cheio para receber um novo elemento? Dica: Veja em `BC_push` e `BC_pop` (`buffer_circular.c`).

2.6.4 Complete a documentação das funções no arquivo-cabeçalho `buffer_circular.h` e `UART.h`, seguind o a sintaxe de Doxygen [18]. (Essa parte pode ser entregue junto com o projeto)

2.7 **Parametrização de Blocos de Memória**: Na função `UART_configure` (`UART.c`) são usados os ponteiros `UART[x]` para processarem os registradores distintos de dois módulos UARTx, onde x = 1 ou 2, por um mesmo bloco de instruções. Onde são declarados esses ponteiros? Qual é a característica comum dos registradores dos dois módulos para que a parametrização seja possível? Dica: Note que UART0 é tratado em separado.

3 Desenvolva o projeto `paridade`. Recomenda-se os seguintes passos que procuram reusar os códigos dos projetos `rot7_aula` e `rot7_example1`

3.1 Crie um novo projeto paridade (Seção 2.1/página 4 em [23]).

3.2 Adicione os arquivos `SIM.*` (do projeto `rot7_example1`), `UART.*` (do projeto `rot7_example1`), `buffer_circular.*`, `util.*` (`rot7_aula`) para reuso (Seção 2.2.3/página 14 em [23]). Crie novos arquivos `ISR.*` (Seção 2.2.2/página 14 em [23]).

3.3 Defina os estados e as regras de transições válidas para cada par de estados mostrados no diagrama de máquina de estados da figura 7. Especifique as ações permitidas em cada estado. Adicione o tipo de dado `enum estado_tag` em `ISR.h`, redefinido como `tipo_estado`, que nomeia os valores constantes associados aos diferentes estados com os nomes intuitivos dos estados, `MENSAGEM`, `EXPRESSAO`, `TOKENS`, `COMPUTO`, `RESULTADO` e `ERRO`.

3.4 Inicialize o sistema usando como fonte de sinais de relógio MCGFLLCLK (20971520Hz). Faça ajustes nas funções `UART0_config_basica` e `UART0_config_especifica` para que o modo de operação do módulo UART0 seja *baud rate 38400*, caracter de 8 *bits*, 2 *stop bits*, sem *bit* de paridade e taxa de amostragem 12x. Faça **testes de unidade** da conexão com o Terminal, ecoando por *polling* a tecla digitada como em `rot7_example1`.

3.5 Entrada e Saída dos Dados. Declare dois *buffers* circulares, 1 de entrada `bufferE` e outra de saída `bufferS`, em `ISR.c`. Ajuste as funções `ISR_inicializaBC` e `ISR_EnviaString` (`rot7_aula`) para processamento desses dois *buffers* e adicione-as em `ISR.*`. Copie ainda a função `ISR_extraiString` (`rot7_aula`) para `ISR.*`. Use `ISR_inicializaBC` para alocar um espaço de memória de 80 *bytes* para cada *buffer*. Faça testes de unidade de entrada e saída de *strings* com a seguinte rotina de serviço (`ISR.c`) que insere, **por interrupção**, os caracteres digitados em `bufferE` e extrai, **por interrupção**, os caracteres retidos em `bufferS` até esvaziá-lo.

```
void UART0_IRQHandler()
{
   char item;
   if (UART0_S1 & UART0_S1_RDRF_MASK) {

      item= UART0_D;
      UART0_D = item;
      if (item == '\r') {
         BC_push (&bufferE, '\0');
         while (!(UART0_S1 & UART_S1_TDRE_MASK));
         UART0_D = '\n';
      } else {
         BC_push (&bufferE, item);
      }
   } else if (UART0_S1 & UART0_S1_TDRE_MASK) {
      if (BC_isEmpty(&bufferS))
         UART0_C2 &= ~UART0_C2_TIE_MASK;
      else {
         BC_pop (&bufferS, &item);
         UART0_D = item;
      }
   }
}
```

3.5.1 Para testes de entrada (RX), use uma função `main` (`min.c`) devidamente inicializada, incluindo a habilitação da interrupção do RX, e com o laço `for` vazio. Coloque um ponto de parada na linha "`UART0_D = '\n';`". Digite uma linha concluída com a tecla "Return" e certifique se os caracteres digitados foram transferidos para o `bufferE`.

3.5.2 Para testes de saída (TX), use uma função `main` (`min.c`) devidamente inicializada, sem a habilitação da interrupção do RX, e com o laço `for` vazio. Insira a chamada de `ISR_EnviaString ("EA8871 - LE30: teste\n\r")` antes do laço `for` em `main`. Execute o programa e ceritifque se a mensagem é mostrada no Terminal.

3.6 Processamento da Expressão digitada. A tarefa efetiva do sistema é

3.6.1 **Extração de Tokens em *Strings***: extrair 2 *tokens* a partir da string extraída do *buffer circular*. Implemente a função `uint8_t ExtraiString2Tokens (char *str,`

`uint8_t *i, char **tokens)` (`main.c`) que retorna no argumento os endereços dos 2 *tokens*, <op> e <valor> . Quando a quantidade de *tokens* é diferente de 2, retorna um código de erro igual a 1. Quando <op> é diferente de 'I', 'i', 'P' ou 'p', retorna o código de erro igual a 3.

3.6.2 **Conversão de *Strings* de Dígitos Binários e Hexadecimais para Inteiros**: verificar o primeiro caractere do segundo *token* para decidir a base e aplicar diferentes algoritmos de conversão. Implemente as funções `uint8_t ConvertBitStringtoUl32 (char *str, uint32_t *valor)`, `uint8_t ConvertDecStringtoUl32 (char *str, uint32_t *valor)` e `uint8_t ConvertHexStringtoUl32 (char *str, uint32_t *valor)` em `util.*` para conversão de `str` na base 2, 10 e 16, respectivamente, em `valor`. Essas funções devem retornr o código de erro 2 se `str` não é um valor válido.

3.6.3 **Paridade de uma Palavra (Dado de n *Bits*)**: computar o *bit* de paridade correspondente ao valor digitado (segundo *token*) conforme o tipo de paridade especificado no primeiro *token*. Reuse a implementação em C, `uint8_t findParity(uint32_t x)`, apresentada em [25]. Insira-a em `util.*`. Note que o procedimento é para paridade ímpar. Se o usuário tiver especificado paridade par, ajustes devem ser feitos no *bit* de paridade retornado pela função `findPariuty`.

3.6.4 **Conversão de Inteiros para *Strings* de Dígitos Binários e Hexadecimais**: converter o valor digitado, `valor`, para uma *string* de algarismos binários `bin`. Implemente em `util.*` a função `char *ConvertUl32toBitString (uint32_t valor, char *bin)`.

3.6.5 **Processamento de *Strings***: enviar o resultado no formato especificado para o Terminal. Para isso monta-se uma string no formato especificado usando as funções padrão da biblioteca C, como `strcpy` e `strcat`.

3.7 Testes de Unidade do porocessamento da expressão. Como a entrada da expressão é por interrupção, coletada dentro de `UART0_IRQHandler` (`ISR.c`), deslocamos o processamento da expressão para a função `main` (`main.c`). Para **testes de unidade** das funções implementadas em 3.6, sugerimos que inclua quaisquer dois estados no projeto só para poder chavear entre a entrada e o processamento. Por exemplo, os estados `EXPRESSAO` e `TOKENS`. Copie para `ISR.*` as funções `ISR_LeEstado` e `ISR_escreveEstado` (`rot7_aula`). Insira em `UART0_IRQHandler`, depois do comando "`item = UART0_D;`" o comando "`if (estado != EXPRESSAO) return;`". Insira ainda `ISR_escreveEstado (TOKENS)` após a linha "`UART0_D = '\n';`". Em `main`, insira no laço `for` o seguinte bloco de instruções:

```
ISR_escreveEstado(EXPRESSAO);
for(;;) {
  switch(ISR_LeEstado ()) {
  case TOKENS:
     //Quando chega neste ponto numa execução, já deve ter
     //um nova linha no buffer circular
     //Inserir aqui as funções que quiser testar e colocar um
     //ponto de parada durante a execução para análise
     ISR_escreveEstado (EXPRESSAO);
     break;
  default:
    break;
  }
}
```

3.8 Implmentação da Máquina de Estados. Insira todos os estados no fluxo principal de `main`. Com base nas regras definidas em 3.3, distribua as funções implementadas entre os estados e adicione os comandos de transições dos estados nos estados pertinentes. Insira em

`UART0_IRQHandler` as restrições em suas operações para proteger regiões críticas. Faça **testes funcionais** do projeto.

3.9 Refinamento. Fazer ajustes nas mensagens de interação com usuários e na documentação das funções implementadas.

3.10 Habilite *Print Size* para uma simples análise do tamanho de memória ocupado. Gere um executável e refaça os **testes funcionais** do projeto para diferentes situações para ver se a resposta está condizente com a especificação.

3.11 Gere uma documentação do projeto com Doxygen [18].

RELATÓRIO

O relatório deve ser devidamente identificado, contendo a identificação do instituto e da disciplina, o experimento realizado, o nome e RA do aluno. O prazo para execução deste experimento é duas semanas. O relatório é dividido em duas partes. Para a primeira semana, responda num arquivo em pdf, as perguntas dos itens 1 e 2 e suba o arquivo no sistema *Moodle*. Para a segunda semana, faça uma descrição sucinta dos testes conduzidos ao longo do desenvolvimento do projeto `paridade`, junto com algumas imagens ilustrativas, num arquivo em pdf. Exporte o projeto `paridade` **devidamente documentado** num arquivo comprimido no IDE CodeWarrior. Suba os dois arquivos no sistema *Moodle*. **Não se esqueça de limpar o projeto (*Clean* …) e apagar as pastas html e latex geradas pelo Doxygen antes**.

REFERÊNCIAS

[1] Jimb0. Serial Communication.
https://learn.sparkfun.com/tutorials/serial-communication

[2] KL25 Sub-Family Reference Manual – Freescale Semiconductors (doc. Number KL25P80M48SF0RM), Setembro 2012.
https://www.dca.fee.unicamp.br/cursos/EA871/references/ARM/KL25P80M48SF0RM.pdf

[3] Kinetis L Peripheral Module Quick Reference – Freescale Semiconductors, Setembro 2012.
https://www.dca.fee.unicamp.br/cursos/EA871/references/ARM/KLQRUG.pdf

[4] FRDM-KL25Z User's Manual
https://www.dca.fee.unicamp.br/cursos/EA871/references/ARM/FRDMKL25Z.pdf

[5] Erich Styger. Using Serial Terminal and COM Support in Eclipse Oxygen and Neon
https://mcuoneclipse.com/2017/10/07/using-serial-terminal-and-com-support-in-eclipse-oxygen-and-neon/

[6] Joel_E_B e Jimb0. Serial Terminal Basics.
https://learn.sparkfun.com/tutorials/terminal-basics

[7] rot7_aula.zip
http://www.dca.fee.unicamp.br/cursos/EA871/1s2023/codes/rot7_aula.zip

[8] Wu, S.T. Ambiente de Desenvolvimento de Software
https://www.dca.fee.unicamp.br/cursos/EA871/references/apostila_C/AmbienteDesenvolvimentoSoftware_V1.pdf

[9] Siddharth. Implementing Circular Buffer in C.
https://embedjournal.com/implementing-circular-buffer-embedded-c/

[10] Sequências de escape
https://learn.microsoft.com/pt-br/cpp/c-language/escape-sequences?view=msvc-170

[11] Bits em Linguagem C – Conceito e Aplicação
https://embarcados.com.br/bits-em-linguagem-c/

[12] strcpy
https://www.cplusplus.com/reference/cstring/strcpy/

[13] The Embedded Experts: string.h
https://studio.segger.com/index.htm?https://studio.segger.com/string_h.htm

[14] strtok
https://www.cplusplus.com/reference/cstring/strtok/

[15] strcat
https://www.cplusplus.com/reference/cstring/strcat/
[16] rot7_example1.zip
http://www.dca.fee.unicamp.br/cursos/EA871/1s2023/codes/rot7_example1.zip

[17] Roteiro 1
http://www.dca.fee.unicamp.br/cursos/EA871/1s2023/roteiros/roteiro1.pdf
[18] Doxygen
https://www.doxygen.nl/manual/docblocks.html
[19] Diagrams.net
https://www.diagrams.net/
[20] rot7_example2.zip
http://www.dca.fee.unicamp.br/cursos/EA871/1s2023/codes/rot7_example2.zip
[21] ARMv6-M Architecture Reference manual
https://www.dca.fee.unicamp.br/cursos/EA871/references/ARM/ARMv6-M.pdf
[22] Roteiro 6
http://www.dca.fee.unicamp.br/cursos/EA871/1s2023/roteiros/roteiro6.pdf
[23] Wu, S.T. Ambiente de Desenvolvimento de Software
https://www.dca.fee.unicamp.br/cursos/EA871/references/apostila_C/AmbienteDesenvolvimentoSoftware_V1.pdf
[24] Stackoverflow. Converting int to binary String in C
https://stackoverflow.com/questions/68069279/converting-int-to-binary-string-in-c
[25] Calcular a paridade de um número usando uma tabela de pesquisa
https://www.techiedelight.com/pt/compute-parity-number-using-lookup-table/
[26] Converting int to binary string in C
https://stackoverflow.com/questions/68069279/converting-int-to-binary-string-in-c

Revisado em Janeiro de 2023
Revisado Março e Outubro de 2022
Revisado em Junho e Julho de 2021
Revisado em Novembro de 2020
Revisado em Fevereiro e Julho de 2017
Agosto de 2016