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/fios/canais de comunicação,
- mais confiável, por enviar um *bit* de dado por vez, simplificando 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. Em vez 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 e UART

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 protocolo RS-232 para comunicações serias assíncronas Esse protocolo usa um *bit* de **start** para indicar o início de uma nova transmissão de dados, e um *bit* de **stop** para sinalizar o término da transmissão. Os quadros de dados (*data frame* ou caractere) são transmitidos em **pacotes** entre esses dois *bits* responsáveis pelo sincronismo. Opcionalmente, s *bits* de **mark** são inseridos entre os pacotes para indicar que a linha está ociosa, ou seja, nenhum dado específico está sendo transmitido. Além disso, para realiar uma verificação simples de erros durante a transmissão, pode ser incluído no **pacote** um *bit* de paridade [1] (Figura 1).

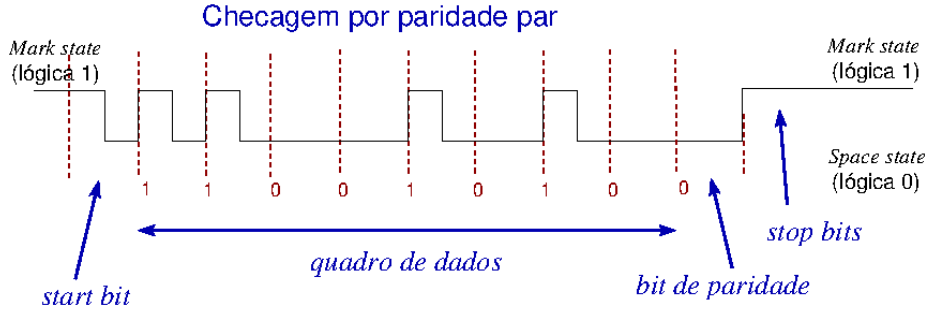

Figura 1: Transmissão por pacote numa comunicação serial assíncrona.

O **protocolo-padrão RS-232** também especifica 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. Ela 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. Embora possua um formato semelhante, seu protocolo não é o padrão RS-232. Isso ocorre porque os níveis de tensão dos sinais transmitidos pela UART são digitais, variando entre 0V e 5V.

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

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. Trata-se de uma técnica simples para detectar erros em transmissões seriais com baixa taxa de erro. 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* [25].

Módulos UARTx

O microcontrolador KL25 é equipado com três módulos dedicados à comunicação serial assíncrona, o que permite conexões simultâneas com até 3 dispositivos: UART0, UART1 e UART2. Além disso, o UART0 pode operar em modo de baixo consumo para uma eficiência energética otimizada.

Cada módulo UARTx atua como uma ponte entre a interface paralela do microcontrolador e uma interface serial assíncrona dos periféricos, utilizando níveis de tensão digitais (Capítulo.39/página 721, Capítulo 40/página 747 em [2], Capítulo 8/ pág. 77 em [3]). O 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]). Por meio dos registradores de deslocamento, os *bytes* a serem transmitidos são serializados e processados, enquanto os *bits* recebidos são amostrados e reagrupados em *bytes* automaticamente, seguindo uma frequência pré-estabelecida.

A habilitação dos sinais de relógio para os módulos UARTx é controlada pelos *bits* do registrador `SIM_SCGC4_UARTx` no módulo `SIM` (Seção 12.2.8/página 204 em [2]). Os sinais de relógio para os registradores de deslocamento de UART1 e UART2 são os mesmos do barramento, enquanto para o UART0, o sinal de relógio, chamado de `UART0` *`clock`* é diferente do sinal de relógio do barramento.

A fonte do `UART0` *`clock`* pode ser selecionada através do registrador de configuração `SIM_SOPT2`. Existem três opções disponíveis, gerenciadas 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 em `MCG`, `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 depende desses sinais de relógio, da taxa de superamostragem e do divisor *prescaler* configurado em `UARTx_BDH` (Seção 39.2.1/ página 725 em [2]) e `UARTx_BDL` (Seção 39.2.2/ página 726 em [2]). Este divisor é composto por 13 *bits*, sendo os 5 *bits* mais significativos armazenados em `UARTx_BDH` e os 8 *bits* menos significativos em `UARTx_BDL`. E a taxa de superamostragem de cada *bit* é definida em `UARTx_C4_OSR` (Seção 39.2.11/ página 736 em [2]).

A técnica de **superamostragem** envolve a coleta de múltiplas amostras do sinal de entrada durante cada período de *bit*, combinando-as para tomar uma decisão precisa sobre o valor do *bit*. Essa técnica é empregada no canal RX para melhorar a precisão na detecção de transições de sinais de entrada, especialmente em ambientes com níveis elevados de ruído.

Nos módulos `UART1` e `UART2`, a taxa de superamostragem é fixada em 16x. Já no `UART0`, é possível selecionar a taxa de superamostragem de 4x a 32x através do registrador de controle/configuração `UARTx_C4`. No caso do `UART0`, se a taxa de amostragem escolhida estiver entre 4x e 7x, é necessário configurar o *bit* `UART0_C5_BOTHEDGE` como ‘1’ (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 meio do campo `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 ajustada de forma independente através dos *bits* `UARTx_S2_RXINV` (Seção 39.2.6/página 731 em [2]) e `UARTx_C3_TXINV` (Seção 39.2.7/página 733 em [2]), respectivamente.

Além disso, o módulo permite incluir o *bit* de paridade aos *bits* de dados, usando os 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 caso do 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 e configuráveis através dos seus registradores, para uma ampla 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]).

Para que a transmissão e recepção ocorram corretamente, os respectivos pinos de saída e de entrada devem ser configurados no modo de multiplexação UARTx, usando os *bits* `PORTx_PCRn_MUX` dos registradores do módulo `PORT`. Os 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 os dados armazenados no registrador `UARTx_D` (Seção 39.2.8/página 734 em [2]) para o registrador de deslocamento. Após a conclusão da transferência, o *bit* de estado `UARTx_S1_TRDE` é setado em '1', indicando a disponibilidade do registrador `UARTx_D` para uma nova transmissão.

No processo de agregação dos dados antes do envio, o registrador de deslocamento adiciona: (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` estiver configurado em '1', e (3) o *bit* de paridade configurado no registrador de configuração `UARTx_C1` (Seção 39.2.3/página 726 em [2]). Além disso, no caso específico do UART0, que suporta envios de caracteres de 10 *bits*, o décimo *bit* `UART0_C4_M10` é agregado ao caractere quando configurado em `UART0_C3` (Seção 39.2.7/página 733 em [2]). O nível lógico dos *bits* transmitidos é selecionável pelo *bit* de configuração `UARTx_C3_TXINV`. Após a conclusão do envio dos *stop bits* e a trnsição do transmissor para o 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]). Para uma compreensão mais detalhada do funcionamento, uma descrição funcional mais completa pode ser encontrada nas Seções 39.3.2/página 739 e 40.3.2/página 762 em [2].

Quando o circuito RX é habilitado, ele realiza uma superamostragem do sinal recebido na frequência configurada (a taxa de superamostragem x *baud rate*) para identificar a borda de descida (do *start bit*) que marca o início do caractere a ser recebido. Assim que essa borda é detectada, o processo de amostragem dos *bits* subsequentes do pacote recebido é iniciado. Após a recepção do *stop bit* e a transferência do 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** na detecção dos *bits*, **ressincronizações** são automaticamente realizadas em cada borda de descida detectada. No entanto, em UART0, existe a possibilidade de desativar essa ressincronização através do *bit* de configuração `UART5_C5_RESYNCDIS` (Seção 39.2.12/página 737 em [2]).

Para **reduzir o consumo de energia**, o RX inclui um **despertador do receptor** (*receiver wakeup*) que o coloca em estado de espera (com o *bit* `UARTx_C2_RWU` em '1') quando identifica que uma mensagem não é destinada a ele. Ele é automaticamente acordado no final da mensagem ou no início da próxima mensagem para retomar a busca por bordas de descida (Seção 39.3.3.2/página 742 em [2]).

Para **garantir a integridade dos dados** recebidos, o canal RX conta com circuitos detectores de erros. Esses erros são sinalizados através das seguintes *flags* de estado: (1) **erro de transbordamento/sobrecarga** (*overrun error*), indicado pela *flag* `UARTx_S1_OR` quando um novo pacote é recebido enquanto o registrador `UARTx_D` ainda está ocupado; (2) **erro de ruído** (*noise error*), indicado pela *flag* `UARTx_S1_NF` quando não há concordância entre os níveis lógicos das

amostras de um mesmo *bit*; (3) **erro de quadro** (*framing error*), indicado pela *flag* `UARTx_S1_FE` quando é detectado o nível lógico '0' onde o *stop bit* é esperado; e (4) **erros de paridade** (*parity error*), indicados pela *flag* `UARTx_S1_PF` quando não há concordância entre o *bit* de paridade dos *bits* amostrados e o *bit* de paridade esperado. 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 bidirecionalmente com dispositivos externos, é preciso designar um pino TxD para o circuito TX (transmissão) e um pino RxD para o circuito RX (receptor), conforme ilustrado nas Figuras 39-1/página 723 e 39-2/página 724 em [2]. Portanto, é necessário alocar os pinos para ambos os circuitos caso estes forem utilizados. Os pinos que podem assumir as funções de `UARTx_TX` e de `UARTx_RX` são especificados pelo fabricante e podem ser consultados na tabela da Seção 10.3.1/página 162 em [2]. Por exemplo, a partir dessa tabela, podemos alocar o par `PTE0` e `PTE1` para as funções de TxD e RxD *Pin* do módulo UART1.

Processamento de Interrupções em UARTx

Os *bits* de habilitação de interrupção correspondentes aos diferentes estados da transmissão, `UARTx_S1_TRDE` e `UARTx_S1_TC`, da recepção, `UARTx_S1_RDRF`, `UARTx_S1_IDLE`, `UARTx_S2_RXEDGIF`, e `UARTx_S2_LBKDIF`, e dos 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 *bit* 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.

Ao consultar o arquivo `Project_Settings/Startup_Code/kinetis_sysinit.c` gerado pelo IDE *CodeWarrior*, oencontramos as rotinas de serviço declarados para as IRQs 12, 13 e 14, que são, respectivamente, `UART0_IRQHandler`, `UART1_IRQHandler` e `UART2_IRQHandler`.

Como cada módulo UARTx tem apenas uma IRQ atribuída, é necessário identificar dentro da rotina de serviço as fontes específicas das interrupções. A técnica mais comum para isso é a varredura dos *bits* de estado por *software*. Essa abordagem envolve testar sequencialmente os *bits* de estado de todos os eventos com interrupções habilitadas e tratar aqueles cujos *bits* de estado estão setados em '1'. Após o tratamento, esses *bits* devem ser resetados em '0' para remover as suas solicitações de interrupção.

Todas as *flags* relacionadas às condições de erro só são resetadas para '0' mediante um acesso de escrita de '1' (*write 1 to clear*). Por outro lado, as *flags* relacionadas à transmissão e 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" aguardando um caractere para resetar a *flag* de disponibilidade do *buffer* de TX (transmissão).

Configuração de Taxa de Transmissão

Taxa de Transmissão (*Baud rate*) indica o número de mudanças de estado que um sinal em um canal de comunicação pode fazer em um segundo. Alguns valores usuais de *baud rate* são: 115200, 9600 e 4800. Para configurar o sinal de relógio do circuito TX/RX do módulo UARTx para operar em uma taxa de transmissão específica, precisamos definir os 13 *bits* de `SBR` (*Serial Baud Rate*). Esses *bits* estão divididos nos campos `UARTx_BDL_SBR` (8 *bits* menos significativos) e `UARTx_BDH_SBR` (5 *bits* mais significativos).

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`:

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

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

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

$$SBR_{calculado} = \frac{UART\,0\,clock\,frequency}{UART\,0\,baud\,rate * (UART\,0\_C\,4\_OSR + 1)}$$
$$SBR_{calculado} = \frac{Bus\,Clock\,Frequency}{ALL\,Other\,UART\,baud\,rate * (16)} \qquad (1)$$

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

$$SBR_{truncado} = (uint\,16\_t)\frac{UART\,0\,clock\,frequency}{UART\,0\,baud\,rate * (UART\,0\_C\,4\_OSR + 1)}$$
$$SBR_{truncado} = (uint\,16\_t)\frac{Bus\,Clock\,Frequency}{ALL\,Other\,UART\,baud\,rate * (16)} \qquad (2)$$

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 predefinida em 16x para os módulos UART1 e UART2, enquanto para o módulo UART0, é ajustável entre 4x a 32x (Seção 39.2.11/página 736 em [2]). Na inicialização de KL25Z, o valor padrão carregado em `UART0_C4_OSR` é 0xF (+1 = 16). Para modificar esse valor, basta substituí-lo, desde que duas condições importantes sejam atendidas: (1) os canais RX e TX devem estar desabilitados, e (2) um acesso de escrita de uma taxa inválida em `UART0_C4_OSR` resultará na escrita do valor 0xF. Essa segunda condição invalida a prática comum de zerar o campo com uma máscara AND e, em seguida, escrever o novo valor com uma máscara OR. Uma alternativa é aplicar uma lógica inversa, 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;
```

Isso garante a correta atualização do valor sem incorrer em problemas relacionados à escrita de taxas inválidas.

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

A conversão de um número inteiro para representações em diferentes bases segue um processo simples: divide-se o número pelo valor da base desejada e registra-se o resto. Esse procedimento é repetido com o quociente resultante até que o quociente seja zero. Os restos registrados são então lidos em ordem reversa para formar o número na nova base. No roteiro 6 [22], apresentamos uma implementação desse processo para a base 10.

Este algoritmo de conversão envolve várias divisões. Porém, como as operações de divisão são computacionalmente mais complexas do que operações aritméticas básicas, como adição, subtração e multiplicação, e também lidam com o caso de divisão por zero, que pode gerar exceções, é importante considerar o impacto na eficiência e robustez do código ao realizar essas conversões.

Quando lidamos com uma base que é uma potência de 2, como no caso binário (2^2), cada dígito pode ser representado por um número fixo de *bits*, que é o logaritmo base 2 da base. Portanto, ao invés de realizar divisões sucessivas, podemos aplicar deslocamentos de *bits* para isolar os dígitos. Por exemplo, para a base 2 (binária), conforme visto no roteiro 1 [17], podemos aplicar uma máscara com o *bit* '1' deslocado m posições (1 << m) sobre o valor para extrair o seu m-ésimo *bit*. Ao percorrer todos os *bits* significativos e concatenar os *bits* isolados, obtemos uma *string* de dígitos binários. Cabe lembrar que no roteiro 3 [27], tivemos a oportunidade de observar como o compilador C integrado ao IDE CodeWarrior otimiza os códigos ao traduzir uma operação aritmética `valor * 32` por uma operação de deslocamento de *bits*.

Para bases maiores, como octal (2^3) ou hexadecimal (2^4), utilizamos máscaras de 3 *bits* (0b111 << m) e 4 *bits* (0b1111 << m), respectivamente, para isolar os dígitos. Uma implementação em C para converter inteiros de 32 *bits* em *strings* hexadecimais envolve deslocar a máscara 0xf (0b1111) em incrementos de 4 *bits*, extrair os 4 *bits* nas posições m, m+1, m+2 e m+3, convertê-los em seus equivalentes ASCII e concatená-los para formar a *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.

Aritmética de Pontos Flutuantes

As representações em ponto flutuante, conforme o padrão IEEE754 [28], são amplamente utilizadas para representar **aproximações dos números reais**. Essas representações permitem descrever a parte fracionária dos valores de diferentes ordens de grandeza com uma precisão considerável, usando uma quantidade fixa de *bytes* (4 para precisão simples e 8 para precisão dupla). A Figura 2 ilustra o padrão IEEE754 da representação de pontos flutuantes.

Ao contrário dos espaçamentos equidistantes dos valores do tipo inteiro, a parte inteira dos valores do tipo `float` é espaçada de forma não uniforme ao longo da reta real, de forma que quanto menores forem os valores, menor será o espaçamento entre eles. Porém, a quantidade de pontos entre dois valores subsequentes, representando a parte fracionária entre eles, permanece a mesma. Assim, a resolução da parte fracionária varia conforme o valor da parte inteira. Quanto menor o valor da parte inteira, maior é a resolução da parte fracionária.

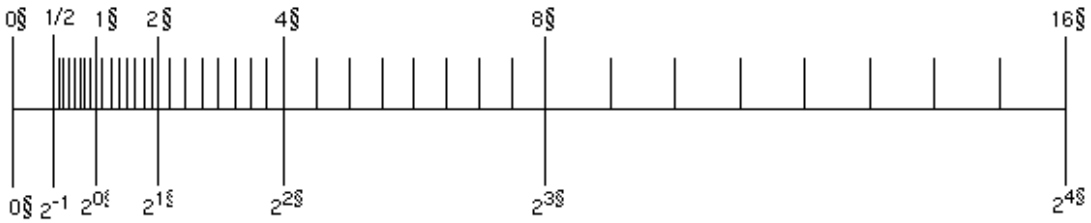

Figura 2: Densidade dos pontos representáveis pelo padrão IEEE754 na reta real (Fonte: [33]).

Em termos de *hardware*, o processamento da aritmética de pontos flutuantes difere da aritmética de inteiros [29]. Essas distinções são consideradas na linguagem de programação C. Dois tipos de dados

nativos, **float** e **double**, são reservados em C para declarar variáveis com valores fracionários, utilizando o ponto '.' para separar as casas inteiras das casas decimais.

O compilador C distingue as operações de inteiros e de pontos flutuantes com base nos tipos de operandos envolvidos. Quando ambos os operandos são inteiros, a aritmética de inteiros é aplicada e o resultado é truncado para um valor inteiro. Por exemplo, o resultado da divisão de duas constantes (1/2) é 0 em ponto fixo, e 0.5 em ponto flutuante. Para usar a aritmética de pontos flutuantes, pelo menos um dos operandos envolvidos na operação deve ser um ponto flutuante. Por exemplo, representar 1 pela convenção de ponto flutuante 1. (1.0) ou fazer uma **conversão explítica** ((float)1). Automaticamente, outros operandos são implicitamente "promovidos" para pontos flutuantes pelo compilador e a aritmética de pontos flutuantes é aplicada [30].

Em KL25Z, há apenas circuitos dedicados para processamento de inteiros, enquanto as operações em pontos flutuantes são emuladas. Para muitos projetos de sistemas embarcados, a aritmética de inteiros é suficiente. Há técnicas para converter os pontos flutuantes para inteiros, como descrito em [31]. Porisso, antes de decidir pela aritmética em ponto flutuante, deve-se ponderar cuidadosamente o compromisso entre o desempenho, a precisão e os desafios adicionais associados à codificação, incluindo a propagação de erros de arredondamento e truncamento implícito [34]. Em certos contextos, o uso de aritmética em ponto flutuante pode gerar resultados inesperados na interação com os usuários, devido a erros de truncamento acumulado na conversão de uma sequência de dígitos antes e depois do ponto para um valor em ponto flutuante. Por exemplo, mostrar 4.99 em vez do esperado 5.00.

Conversão de pontos flutuantes para *strings*

Neste roteiro estamos interessados em converter valores em ponto flutuante para *strings* e renderizá-las no Terminal. Um algoritmo eficaz para essa conversão, que simplifica o problema à conversão de dois valores inteiros, seguiria os seguintes passos:

1. Verifique se o número é negativo. Se for, registre um sinal negativo e altere o valor para positivo.
2. Converta a parte inteira do número para *string* usando a função `itoa`.
3. Multiplique o valor fracionário por uma potência de 10 igual à quantidade de casas decimais desejadas.
4. **Arredonde** o resultado do produto para um valor inteiro.
5. Converta o resultado arredondado para *string* usando a função `itoa` ou o equivalente.
6. Adicione um ponto decimal ao final da *string* da parte inteira.
7. Concatene as duas strings da parte inteira e fracionária para formar a *string* final.
8. Se o número original era negativo, adicione um sinal negativo à *string* final.

Em [32] é apresentada uma implementação do algoritmo em C. Dois casos não foram considerados no algoritmo: (1) o número ser negativo e (2) arredondamento das casas decimais em vez de truncamento.

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 (*Open Standard Device Adapter*). O OpenSDA proporciona uma interface serial USB (*Universal Serial Bus*) com o computador (Seção 5.2/página 7 em [4]). Dentro do OpenSDA estão integrados o *P&E Debug Application,* que possibilita a depuração dos 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, ao usar um

emulador de terminal no computdor-hospedeiro, é possível enviar caracteres para processamento no KL25Z e receber os caracteres gerados pelo KL25Z.

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

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".

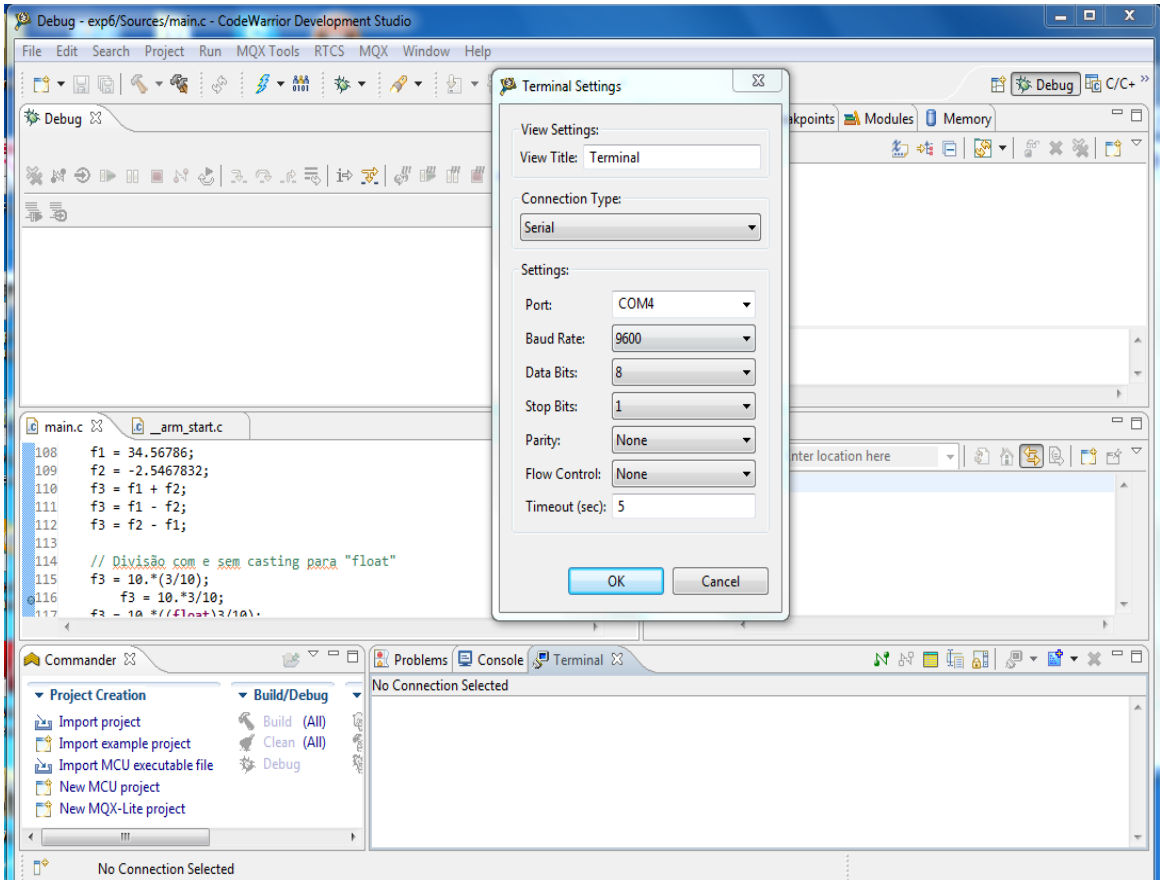

Figura 3: *Pop-up* menu de configuração dos parâmetros seriais do lado do computador-hospedeiro.

Tipo de Dado Struct e Typedef em C

Estrutura (Struct) em C são tipos de dados compostos que permitem agrupar variáveis de diferentes tipos dentro de um único bloco de memória, permitindo acesso a essas variáveis por meio de nomes significativos. Ao utilizar `structs`, podemos acessar os dados de maneira semântica, ou seja, com base em seus significados. Isso torna a compreensão e manutenção do código mais fácil, pois os dados são acessados pelo seu significado, não apenas pelos seus endereços de memória.

A declaração de uma estrutura é feita com a palavra reservada `struct`, seguida pelo nome da `struct` e pelos campos (variáveis-membro) delimitados por chaves. Esses campos podem ser de qualquer tipo de dado válido em C, inclusive outras structs. Essa flexibilidade e organização

oferecidas pelas structs são fundamentais para estruturar dados de forma clara e eficiente em programas C:

`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 manuais relacionados a KL25Z, 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 (aliases) para structs com o comando `typedef`, como

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

Essa linha define um novo nome mais compacto, `PORT_MemMapPtr`, para o ponteiro do tipo de dado `struct PORT_MemMap`, com o qualificador `volatile`. Esse `typedef` é útil para simplificar o código e tornar mais claro o uso de ponteiros para estruturas. No IDE CodeWarrior, esse sinônimo pode ser utilizado para mapear os endereços dos registradores de `PORTx`, facilitando a manipulação desses registradores no código:

```
((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, é possível definir em C um novo tipo de dado para simplificar o gerencimento desses registradores. Por exemplo, podemos criar uma estrutura denominada `struct UART0Config_tag`, que agrupa todos os registradores num único bloco contíguo de memória 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 consideravelmente mais lenta do que a do processador. Para garantir uma sincronização eficiente entre esses dois processos distintos sem comprometer a capacidade de processamento do processador, é comum usar os ***buffers* circulares** [9] nos projetos de sistemas embarcados para armazenar os dados seriais de entrada e de saída.

Esses *buffers*, essencialmente, são estruturas de **fila** com uma peculiaridade: suas extremidades estão conectadas, formando um ciclo. Ao contrário das filas tradicionais, onde a remoção de um elemento requer o deslocamento de todos os elementos restantes, em um *buffer* circular, os elementos permanecem no mesmo lugar após a remoção. Isso é possível graças a dois ponteiros, chamados de *head* (cabeça) e *tail* (cauda), que se movem ciclicamente. Quando um elemento é adicionado, o ponteiro *head* avança; ao remover um elemento do final da fila, o ponteiro *tail* avança.

Na Figura 4, quando removemos o elemento '0' de uma fila clássica, todos os elementos à direita, '1' a '14', são deslocados uma posição para a esquerda, e o ponteiro também é deslocado. No entanto, em um *buffer* circular, a remoção do elemento '0' resulta apenas no deslocamento do ponteiro *tail*, sem afetar o conteúdo da memória. Isso ocorre porque os ponteiros são incrementados ciclicamente no *buffer* circular, mantendo os endereços de memória sempre dentro do espaço reservado. Em contraste, em uma fila tradicional, o ponteiro pode ultrapassar os limites do espaço reservado, resultando em vazamento de memória.

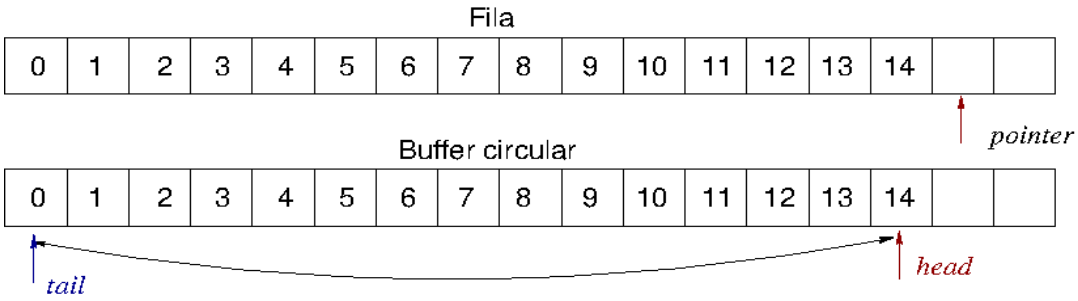

Figura 4: Fila e *buffer* circular.

Em C, é comum definir um novo tipo de dado `struct`, como `BufferCirc_type`, para agrupar um arranjo 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] exemplifica o uso de *buffers* circulares tanto na recepção de caracteres do teclado, onde o usuário e UART0 atuam como produtor e KL25Z como consumidor, quanto na transmissão dos caracteres para o Terminal (produtor = KL25Z e consumidor = UART0+Terminal), embora o problema de compatibilidade da velocidade do produtor KL25Z com a do consumidor UART0 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 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*) 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*, *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 exibida num Terminal.

Extração de Tokens em *Strings*

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 em KL25Z4, é necessário processar os caracteres digitados pelo usuário em linhas distintas.

Cada pacote transmitido por um módulo UARTx contém um quadro de dados de 5 a 10 *bits*. geralmente representado em C pelo tipo de dado `char` quando a quantidade de *bits* é até 8. Em

linguagem C, esses quadros de dados são frequentemente representados pelo tipo de dado `char` quando a quantidade de *bits* é de até 8. Quando uma sequência de pacotes de dados é transmitida, correspondendo a uma linha de caracteres no terminal, ela é armazenada como uma *string*, que é basicamente um vetor de elementos do tipo `char`. Para indicar o término dessa sequência e torná-la uma *string* válida em C, os caracteres de controle "\n\r", necessários para exibição em linhas no Terminal, devem ser substituídos pelo terminador '\0'. Assim, podemos utilizar uma série de funções disponíveis na biblioteca-padrão de C, como `strcmp` e `strlen`.

Antes de processar um comando contido em uma linha recebida, é necessário extrair os **tokens** ou "unidades de informação", como o próprio comando e seus argumentos, que estão presentes na linha. Geralmente, esses tokens são separados por delimitadores como ',' (vírgulas), '.' (pontos), ';' (ponto e vírgula) ou ' ' (espaços em branco). A função `strtok (char *str, const char *lista_delimitadores)` da biblioteca padrão C é um excelente candidato para extrair tokens de uma linha de caracteres digitada no Terminal. Ela 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`). Quando chamada pela primeira vez, a função percorre a linha de caracteres e substitui todos os delimitadores encontrados pelo terminador '\0', indicando assim o fim de cada token. Em seguida, retorna o endereço da primeira *sub-string* encontrada. Nas chamadas subsequentes, quando o primeiro argumento é configurado como `NULL`, a função continua a busca pela próxima *sub-string*, retornando o seu endereço inicial. Esse processo se repete até que a função retorne um endereço `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. A Figura 5 ilustra esse procedimento de forma esquemática.

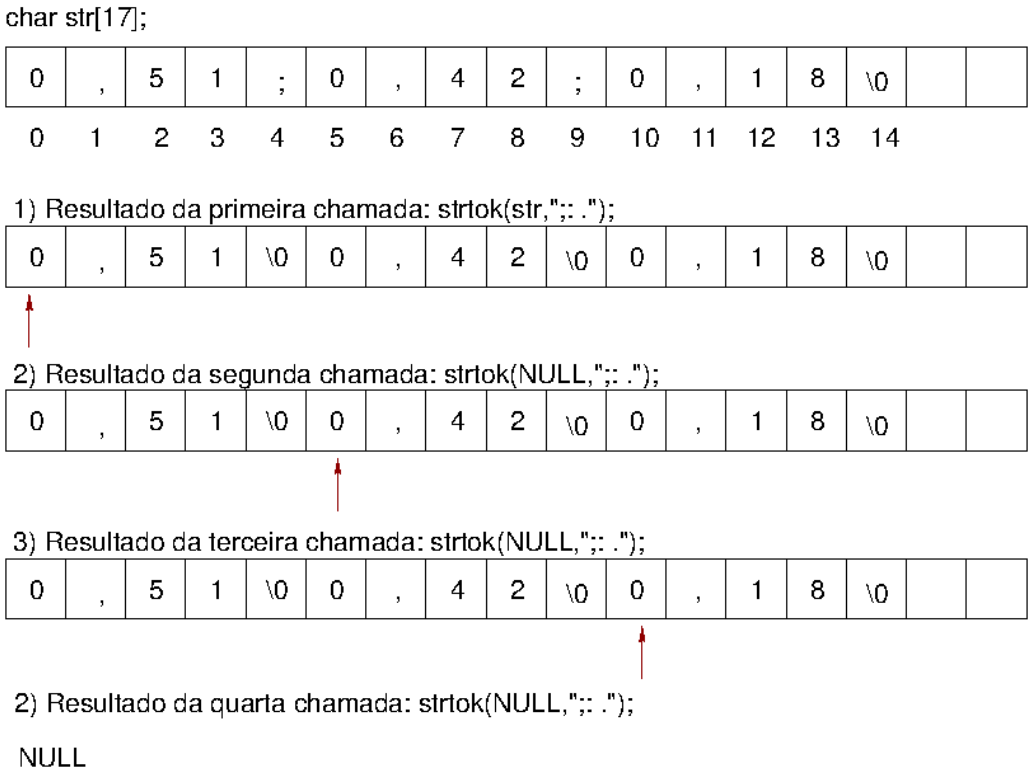

Figura 5: Procedimento implementado em `strtok`

É importante ressaltar que a função `strtok` modifica o conteúdo da *string* original (`str`) durante sua execução. Portanto, é recomendável aplicar essa função a uma cópia da string original se houver a necessidade de preservar o conteúdo original de `str`. Isso evita alterações indesejadas nos dados originais, garantindo a integridade das informações contidas na string original.

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 *string* `source` no endereço `destination` do tipo `char`. 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 copia_str[100];
strcpy (copia_str, str);
i=0;
sub_str[i] = strtok(copia_str, ";:. ");
while (sub_str[i] != NULL) {
     sub_str[++i] = strtok(NULL, ";:. ");
}
```

- `char * strcat ( char * destination, const char * source)` [15]: anexa uma *string* `source` ao final da *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);
```

Diretiva #if em C

Como discutido no roteiro 1 [17], uma **diretiva** para o compilador C é uma instrução que fornece informações adicionais sobre como um programa deve ser compilado. Elas são geralmente precedidas pelo caractere '#'. Até agora, utilizamos diretivas como `#include`, que normalmente inclui os protótipos de funções necessários para um programa, e `#define`, que define constantes ou macros. Além disso, o compilador C suporta **diretivas condicionais**, que permitem a inclusão condicional de blocos de instruções. A sintaxe para a diretiva condicional de inclusão de um bloco de instruções em uma 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 **rotina** que contém o bloco de instruções parametrizado.

Prametrizar um bloco de memória em C significa definir uma mesma estrutura (`struct`) para diferentes regiões de memória, permitindo que os nomes de seus membros sejam reutilizados para acessar diferentes endereços de memória 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], respectivamente, e são acessíveis via a variável `uart[]` do tipo `struct UART_MemMap` definido em `Project_Headers/MKL25Z4.h`.

Usando macros definidas nesse arquivo, como

```
#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 os acessos aos regitradores e seus processamentos, declarando um vetor de ponteiros

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

Essa declaração equivale à criação de um vetor contendo os endereços-base de estruturas para UART1 e UART2:

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

A função `UART_configure` de `rot7_aula` [7] exemplifica a parametrização de um bloco de instruções para configurar os registradores de UART1 e UART2. Mesmo esses registradores estando localizados em endereços distintos, podemos acessá-los com os mesmos identificadores, alterando apenas o valor do elemento do vetor `uart`, seja `uart[0]` ou `uart[1]`. Por exemplo, os identificadores `uart[0]->BDL` e `uart[1]->BDL` representam abstrações dos endereços físicos 0x4004B001 e 0x4006C001 na memória, onde os registradores `UART1_BDL` e `UART2_BDL` estão mapeados. Essa técnica simplifica o código e torna-o mais legível, facilitando a manutenção e reutilização do mesmo bloco de instruções para configuração de diferentes periféricos.

Proteção de Regiões Críticas pelas Restrições nos Estados do Sistema

Embora a recepção e transmissão de caracteres ocorram em canais distintos, ambos podem compartilhar o Terminal para exibir os caracteres em processo. Nesse caso, surge uma **concorrência pelo uso** do recurso do Terminal, o que requer um gerenciamento cuidadoso para evitar acessos conflitantes. No roteiro 6 [22], vimos uma estratégia de proteção envolvendo a desativação das interrupções. Outra abordagem seria a utilização de uma máquina de estados, por meio da implementação de regras de transição e ações restritas a cada estado.

Por exemplo, no projeto rot7_aula [7], definimos 4 estados distintos: `ESPERA`, `EXTRAI`, `MOSTRA` e `LIBERA_BUFFER`, conforme mostra a Figura 6. No estado `EXTRAI`, uma nova entrada em `UART0_IRQHandler` não é permitida, uma vez que uma entrada está sendo processada. No estado `MOSTRA`, embora o sistema esteja atualizando o Terminal, ele permanece pronto para interromper a exibição caso o usuário digite algum caractere. Devido à natureza da comunicação *full-duplex*, essa transição é permitida. No entanto, para evitar acessos concorrentes ao Terminal, a transição do estado `MOSTRA` para o estado `ESPERA` é condicionada à passagem pelo estado `LIBERA_BUFFER`, onde o

buffer circular de saída é esvaziado e os caracteres digitados pelo usuário são descartados. Essa abordagem ajuda a garantir um acesso ordenado e seguro ao recurso compartilhado do Terminal, evitando conflitos e garantindo uma experiência de usuário mais estável.

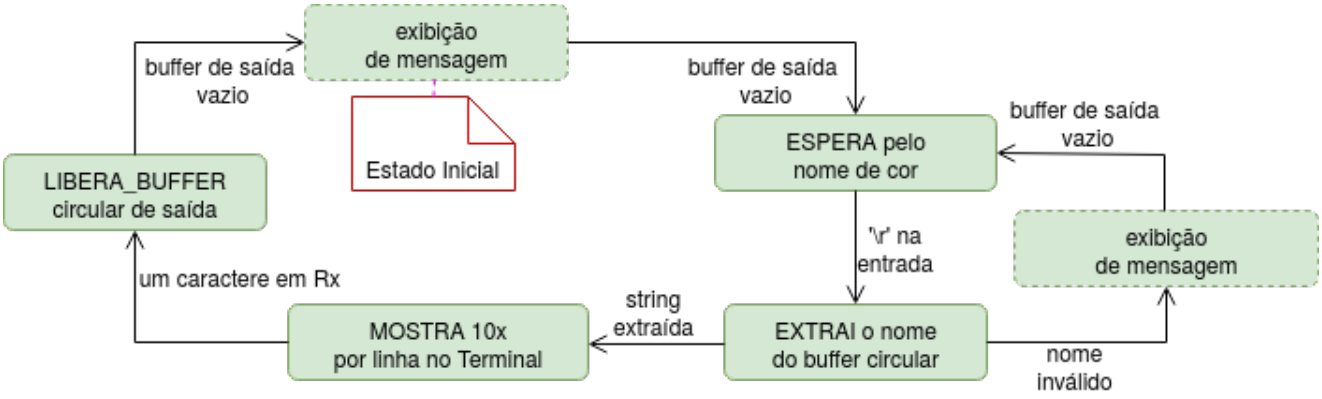

Figura 6: Diagrama de máquina de estados do projeto `rot7_aula` (editado em [19]).

EXPERIMENTO

Neste experimento, iremos desenvolver o projeto `calculadora`. Ela aceita como entrada uma expressão no formato <op1> <op> <op2>, onde <op1> e <op2> são os **operandos do tipo inteiro sem sinal** e <op> é um dos **operadores lógico-aritméticos: + (soma), - (subtração), * (multiplicação), / (divisão)** em **pontos flutuantes**, e **& (E bit a bit), | (OU bit a bit), # (divisão) e % (resto)** em **inteiros**. Os tokens devem ser separados por um espaço em branco.

No início, a MENSAGEM "Entre <op1> <op> <op2> (<op>: +,-,*,/,%,#,&,|)\n\r" é exibida no Terminal. Depois de inserir a EXPRESSAO no formato especificado na linha seguinta à da mensagem, são extraídos e validados os TOKENS usando `strtok`. Caso um erro léxico ou sintático for identificado, gera-se uma mensagem de ERRO; senão é feito o COMPUTO da expressão. Uma mensagem contendo o RESULTADO, que pode ser um número inteiro ou um número de ponto flutuante com duas casas na parte fracionária, é montada e enviada para o Terminal. Após concluir o envio da mensagem, é exibida novamente a MENSAGEM de instrução e aguardada uma nova expressão, e repet-se o ciclo. Figura 7 apresenta um diagrama de máquina de estados da `calculadora`.

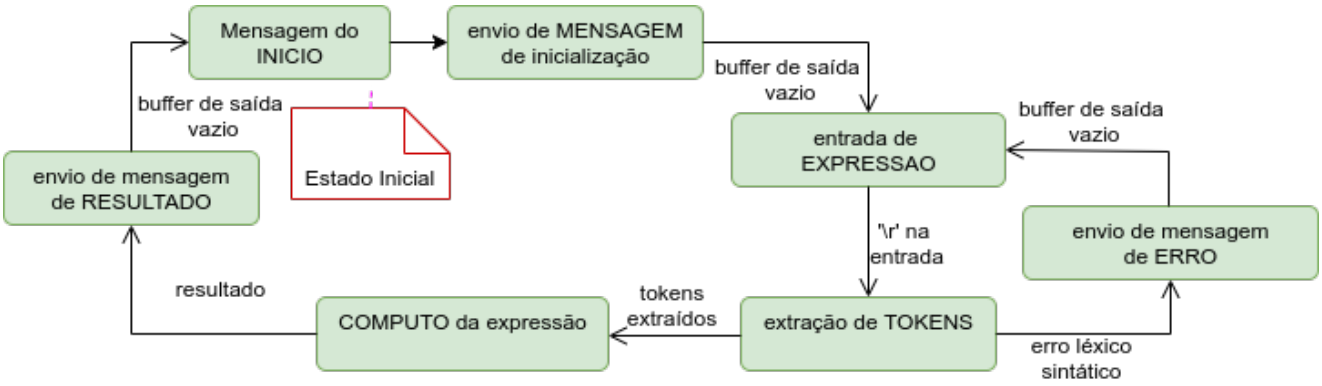

Figura 7: Diagrama de máquina de estados do projeto `calculadora` (editado em [19]).

Se algum erro for detectado durante o cálculo, uma mensagem, contendo o tipo de erro e “Tente novamente <op1> <op> <op2>”, é construída e aguarda-se uma nova tentativa. A Figura 8 ilustra uma

possível interface da `calculadora` no Terminal. Os textos da mensagem e do resultado são apenas sugestões.

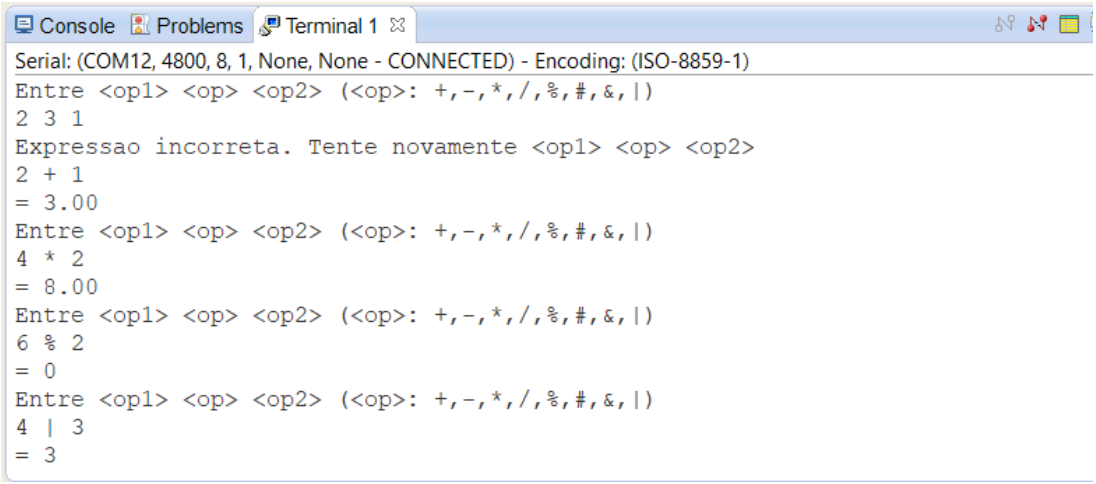

Figura 8: Interface com usuário do projeto `calculadora`.

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.

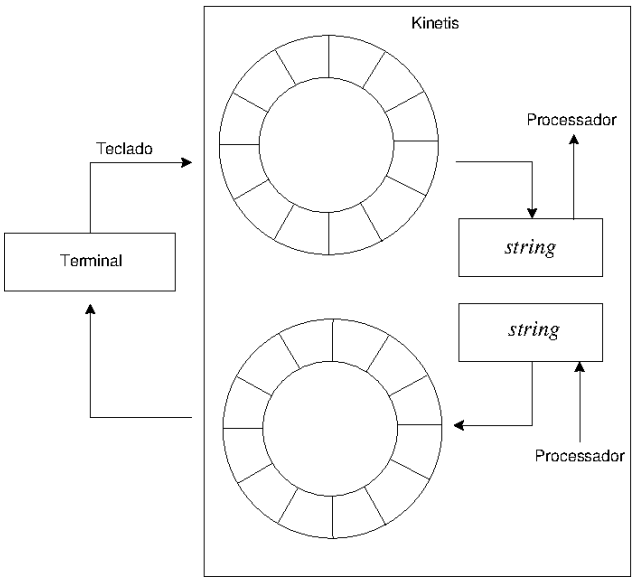

Figura 9: Um *buffer* circular para recepção e outro para transmissão.

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.

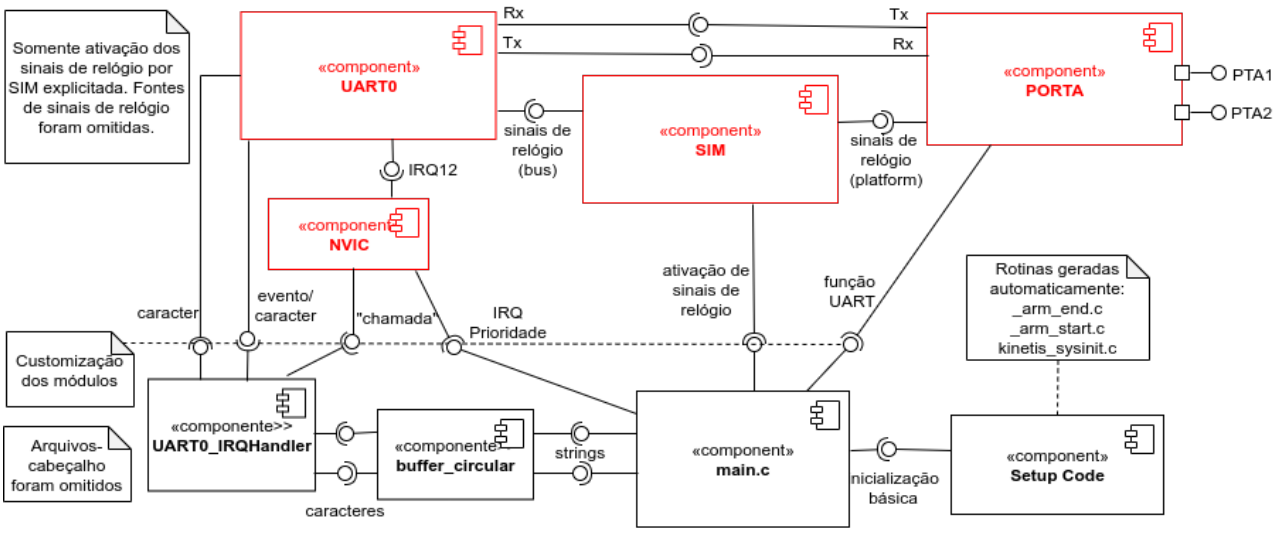

Figura 10: Diagrama de componentes do projeto `calculadora` (editado em [19]).

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 superamostragem 10x. Os *buffers* circulares (1 de entrada e 1 de saída) tem 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 **De conceitos para práticas:** O projeto `rot7_aula` [7] demonstra configurações do módulo UART0 para receber, através do Terminal, uma das cores em maiúsculas: "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 a cor digitada repetida 10 vezes, até que uma nova cor seja inserida. É incluída no projeto uma implementação do *buffer* circular (`Sources/buffer_circular.c` e `Project Headers/buffer_circular.h`).

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). Para isso, é necessário configurar o canal do analisador para interpretar o 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 realizar a amostragem conforme o protocolo *Async Serial*, disponível no menu "*Analyzers*" à direita. Ao selecionar "*Async Serial*", um segundo menu é exibido para configurar os parâmetros de comunicação do sinal a ser amostrado, conforme mostrado na Figura 12. Para o projeto `rot7_aula`, os parâmetros são definidos como *baud rate* de 4800, com 8 *bits* de dados, 1 *stop bit*, sem *bit* de paridade e sem fluxo de controle.

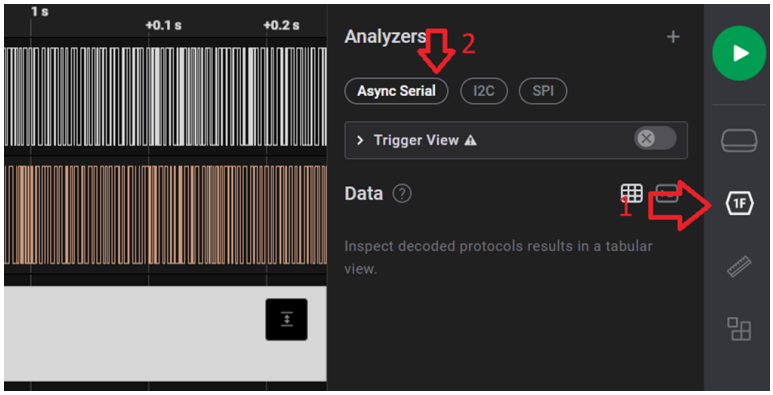

Figura 11: Configuração de um formato específico para as amostras num canal do analisador.

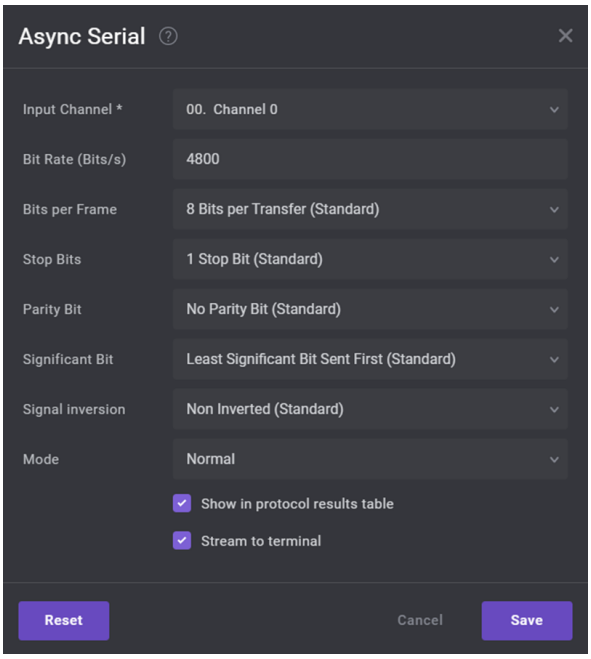

Figura 12: Parâmetros de comunicação serial para um canal do analisador lógico.

1.a **Tipo de Dado Struct**: A variável `config0` do tipo `UART0Config_type` e a variável `config2` do tipo `UARTConfig_type` são declaradas com inicialização para facilitar o gerenciamento dos registradores de UART0 e UART2, respectivamente. Compare a forma de comunicação dos dois módulos com base na inicialização dessas variáveis.

1.b **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.

1.b.1 Identifique os seguintes parâmetros de comunicação num caractere transmitido: os *bits* de dados, *start bit, stop bits,* a terminação dos *bits* (LSB ou MSB) e o nível lógico dos *bits* (nível alto corresponde a '1' ou a '0').

1.b.2 A forma de onda capturada está de acordo com as configurações dos módulos?

1.b.3 Altere a inicialização das variáveis `config0` e `config2` para que a configuração de comunicação serial possua 2 *stop bits* e sem *bit* de paridade. Refaça e execute novamente o executável. Qual é o impacto observado nos sinais capturados após essa modificação? Houve aumento ou redução na taxa de transferência?

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

1.c.1 Meça a largura do pulso de um *bit* e verifique se está de acordo com a taxa de transferência (*baud rate*=4800) ou com os valores configurados nos registradores `UART0_BDH_SBR` e `UART0_BDH_SBR` para o módulo UART2/UART0.

1.c.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. Refaça e execute novamente o executável. Verifique se a forma de onda de um quadro de dados está condizente com as configurações.

1.d **Processamento de Interrupções em UARTx**: O modo de operação dos canais RX e TX é por interrupção. Ao longo do código de `rot7_aula`, as interrupções dos canias RX e TX são habilitadas e desabilitadas. Isso é importante para evitar que o processador seja constantemente interrompido por eventos de comunicação quando não é necessário. 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.

1.e **Proteção de Regiões Críticas:** No projeto `rot7_aula`, as transições de estados e as entradas dos estados são controladas por regras específicas para proteger as regiões críticas do sistema. Identifique no projeto as regras de transição que devem ser satisfeitas nas transições `ESPERA` → `EXTRAI`, `EXTRAI` → `MOSTRA`, `MOSTRA` → `ESPERA`, `MOSTRA` → `LIBERA_BUFFER` e `LIBERA_BUFFER` → `ESPERA`? E as restrições impostas em relação à entrada, por interrupção, dos caracteres digitados pelo usuário nos estados `LIBERA_BUFFER`, `EXTRAI`, `MOSTRA` e `ESPERA`?

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

1.f.1 Qual é a quantidade máxima de caracteres que esses *buffers* podem armazenar?

1.f.2 Qual é a função de cada *buffer* no projeto?

1.f.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`).

1.f.4 Para uma melhor compreensão das funções relacionadas ao *buffer circular*, documente as rotinas no arquivo-cabeçalho `buffer_circular.h` e `UART.h`, seguindo o a sintaxe de Doxygen [18]. Esses arquivos podem ser reusados no projeto `calculadora`.

1.g **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? Por quê UART0 é tratado em separado?

2 **<u>Aprender com os Exemplos dos Manuais</u>**: Os fabricantes costumam fornecer manuais de referência rápida e prática com o código-fonte dos exemplos de projetos envolvendo diferentes módulos de um microcontrolador, visando auxiliar os desenvolvedores a entenderem os recursos disponíveis e iniciarem rapidamente o desenvolvimento de seus próprios projetos.
No entanto, é comum que os exemplos de projetos fornecidos pelos fabricantes sejam destinados a ambientes de desenvolvimento diferentes daqueles escolhidos pelos desenvolvedores. Isso pode exigir algumas adaptações nos códigos para que os desenvolvedores possam aproveitar ao máximo os exemplos fornecidos. Vamos aprender algumas práticas comuns para lidar com essa situação usando exemplos de projetos em [3].

Na seção 8.4/página 84 em [3], são apresentados dois exemplos de configuração de UART0. O Exemplo 1 demonstra uma aplicação em transmissões por *polling*/interrupção, usando o MCGFLLCLK em 48MHz como fonte de sinal de relógio para controle de *baud rate* (Seção 8.4.1/página 84 em [3]). Já o Exemplo 2 ilustra o modo de operação do UART0 em estados de espera com um consumo de potência muito baixo, tendo o MCGIRCLK em 4MHz como fonte de sinal de relógio para controle de *baud rate* (Seção 8.4.2/página 87 em [3]).

É importante observar que o foco desses exemplos está na configuração do UART0, e presume-se que o leitor já possua conhecimento prévio 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 adaptadas para o IDE CodeWarrior. 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*.

2.a **Diferenças de ambiente:** Os projetos `rot7_example1` e `rot7_example2` incluem configurações adicionais àquelas fornecidas pelo fabricante na inicialização do microcontrolador para que satisfaçam os modos de operação especificados.

2.a.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`.

2.a.2 Quais blocos de instruções foram acrescentados em `rot7_example*`? Se removê-los, os programas atenderiam as especificações dadas em [3]?

2.b **Dependências do Projeto**: Compare as rotinas utilizadas pelo fabricante e as rotinas chamadas no IDE CodeWarrior. Houve a necessidade de resolver alguma dependência das funções nos projetos?

2.c **Incompatibilidade de código:** Houve a necessidade de fazer as alterações nos códigos-fonte para que eles sejam compiláveis no IDE CodeWarrior?

2.d **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`).

2.e **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: sem *bit* de paridade. Somente o *baud rate* e a quantidade de *stop bits* mudam: 115200, 1 *stop bit* (`rot7_example1`) e 9600, 2 *stop bits* (`rot7_example2`).

2.f **Processamento de Interrupções em UARTx**: Qual é o modo de operação do canal Tx? Por *polling* ou por interrupção? É adequada a escolha do modo de operação pelo fabricante? Justifique.

2.g **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]. São, de fato, necessários esses cálculos para reproduzir os dois exemplos de projetos fornecidos pelo fabricane?

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

2.h.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`.

2.h.2 Em `rot7_example2` há alguma instrução que define `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].

3 **Praticar as práticas:** Desenvolva o projeto `calculadora` conforme a especificação dada. Procure reusar os códigos dos projetos `rot7_aula` e `rot7_example*`. Recomenda-se os segintes passos de desenvolvimento:

3.a **Especificação funcional:** Detalhe a descrição do comportamento desejado do projeto `calculadora` a partir do diagrama de máquina de estados mostrado na Figura 7, indicando explicitamente as condições que acionam cada transição. Descreva em um documento separado o que acontece dentro de cada estado, detalhando as condições que levam a um estado específico, as atividades ou comportamentos realizados enquanto o sistema está nesse estado e as transições que podem ocorrer a partir desse estado.

3.b **Especificação implementacional:** Identifique os eventos de interrupção que podem ocorrer durante a operação da `calculadora`. Detalhe em diagramas de atividades, ou em uma representação equivalente, as ações dentro de cada estado, incluindo a habilitação e desabilitação desses eventos. Descreva ainda a sequência de interações entre os fluxos de controle durante o tratamento de cada evento com o uso de diagramas de sequência, ou uma representação equivalente. Identifique blocos de instruções comuns nos estados e os parametrize em **funções** para ajudar a simplificar o código e facilitar a manutenção.
Duas funções que podem ser úteis na implementação já se encontram declaradas em `util.h` e/ou implementadas em `util.c`:
`intToStr(uint32_t x, char str[], uint8_t d)`
`void ftoa(float n, char* res, uint8_t afterpoint)`
Outras três funções que podem ajudar:
`uint8_t StrToOp (char *string, char *op, uint32_t *op1, uint32_t *op2)`, que extrai da *string* recebida os dois operandos e o operador usando `strtok`. Caso encontre um erro na conversão, a função retorna um código de erro especificado.
`OpInteiro (char op, uint32_t op1, uint32_t op2, uint32_t *res)`, que computa o resultado inteiro dos dois operandos.
`OpFlutuante (char op, uint32_t op1, uint32_t op2, float *res)`, que computa o resultado em ponto flutuante dos dois operandos.

3.c **Implementação:**Escreva o código com base na especificação implementacional. Procure reusar as funções implementadas. Documente a interface de todas as **novas funções** implementadas seguindo sintaxe Doxygen [18].

3.d **Testes:** Realize testes para garantir que o sistema atenda aos requisitos especificados. Isso pode incluir testes de unidade e testes de integração. Registre os testes conduzidos e os resultados.

3.e **Depuração:** Identifique e corrija quaisquer problemas encontrados durante os testes. Habilite *Print Size* para uma simples análise do tamanho de memória ocupado.

RELATÓRIO

O relatório deve ser devidamente identificado, contendo a identificação do instituto e da disciplina, o experimento realizado, o nome e RA dos alunos do grupo. O prazo para execução deste experimento é de duas semanas, dividido em duas partes. Na primeira semana, é necessário responder as questões do item 2 e realizar a especificação funcional e implementacional do projeto `calculadora`. **Isso inclui computar os valores a serem configurados nos registradores do KL25Z, descrever detalhadamente o comportamento desejado do projeto calculadora, identificar os eventos de interrupção que podem ocorrer durante sua operação, detalhar as ações dentro de cada estado e descrever as sequências de interações entre os fluxos de controle.** Suba dois arquivos no sistema *Moodle*: um contendo as respostas do item 2 e outro, as especificações do projeto `calculadora`. Na segunda semana, deve-se fazer **uma descrição sucinta dos testes conduzidos, incluindo os**

resultados obtidos e quaisquer correções realizadas ao longo do desenvolvimento do projeto. Além disso, é necessário exportar o **projeto `calculadora` devidamente documentado** em um arquivo comprimido no IDE CodeWarrior e subir ambos os 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/EA871/1s2024/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/EA871/1s2024/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/EA871/1s2024/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
[27] Roteiro 3
http://www.dca.fee.unicamp.br/cursos/EA871/1s2024/roteiros/roteiro3.pdf
[28] IEEE754 Converter
http://www.h-schmidt.net/FloatConverter/IEEE754.html
[29] Fixed-point vs. Floating-Point Digital Signal Processing
https://www.analog.com/en/technical-articles/fixed-point-vs-floating-point-dsp.html
[30] Type conversion in C
https://www.geeksforgeeks.org/type-conversion-c/
[31] Simple Fixed-Point Conversion in C
https://embeddedartistry.com/blog/2018/07/12/simple-fixed-point-conversion-in-c/
[32] Acervo Lima. Converta um número de ponto flutuante em string em C
https://acervolima.com/converta-um-numero-de-ponto-flutuante-em-string-em-c/
[33] IEEE Arithmetic
https://docs.oracle.com/cd/E19957-01/806-3568/ncg_math.html
[34] Floating-point data in embedded software
https://www.embedded.com/floating-point-data-in-embedded-software/

Revisado em Fevereiro de 2024
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