

EXPERIMENTO 7 – Interface Serial Assíncrona

FEEC | EA871

Thiago Maximo Pavão - 247381
Vinícius Esperança Mantovani - 247395

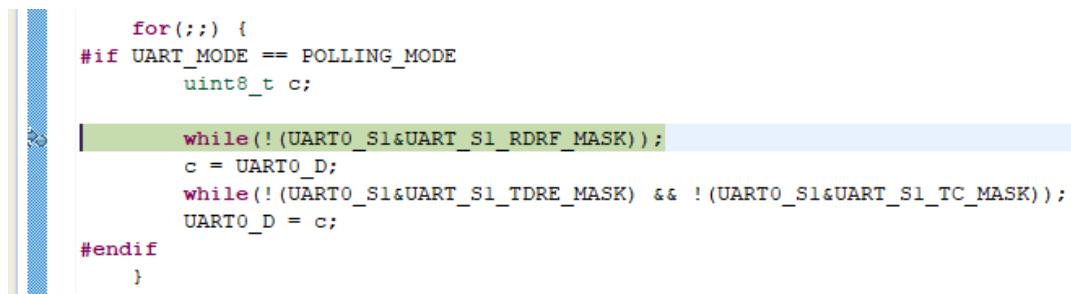
Primeira parte

1.

Inicialmente, o modo de operação configurado é o de polling. O modo configurado é dado por `UART_MODE`, definido em `ISR.h` na linha `#define UART_MODE (modo)`, ele pode ser configurado em `INTERRUPT_MODE` ou `POLLING_MODE`. Estes últimos, são definidos como os números inteiros 1 e 2, nas linhas `#define POLLING_MODE 1` e `#define INTERRUPT_MODE 2`. Assim, por meio de condicionais que consideram o `UART_MODE`, o compilador distingue seu comportamento dependendo do modo selecionado.

```
#define UART_MODE POLLING_MODE
//#define UART_MODE INTERRUPT_MODE
```

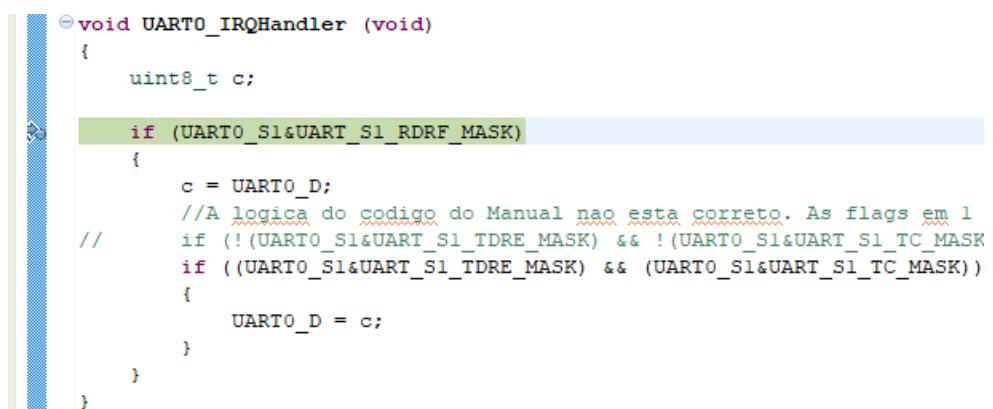
É possível verificar que o modo inicialmente configurado é o de polling colocando um *breakpoint* dentro do `for` na `main` que realiza o polling. Executando o programa em modo debug, observa-se que ele para a execução lá dentro



```
for(;;) {
#if UART_MODE == POLLING_MODE
    uint8_t c;

    while(!(UART0_S1&UART_S1_RDRF_MASK));
    c = UART0_D;
    while(!(UART0_S1&UART_S1_TDRE_MASK) && !(UART0_S1&UART_S1_TC_MASK));
    UART0_D = c;
#endif
}
```

Ademais, após a alteração do modo (usando o modo `INTERRUPT`) e a ativação e configuração do terminal, enviamos um caractere para verificar a entrada na rotina de tratamento de interrupção, conforme a imagem a seguir

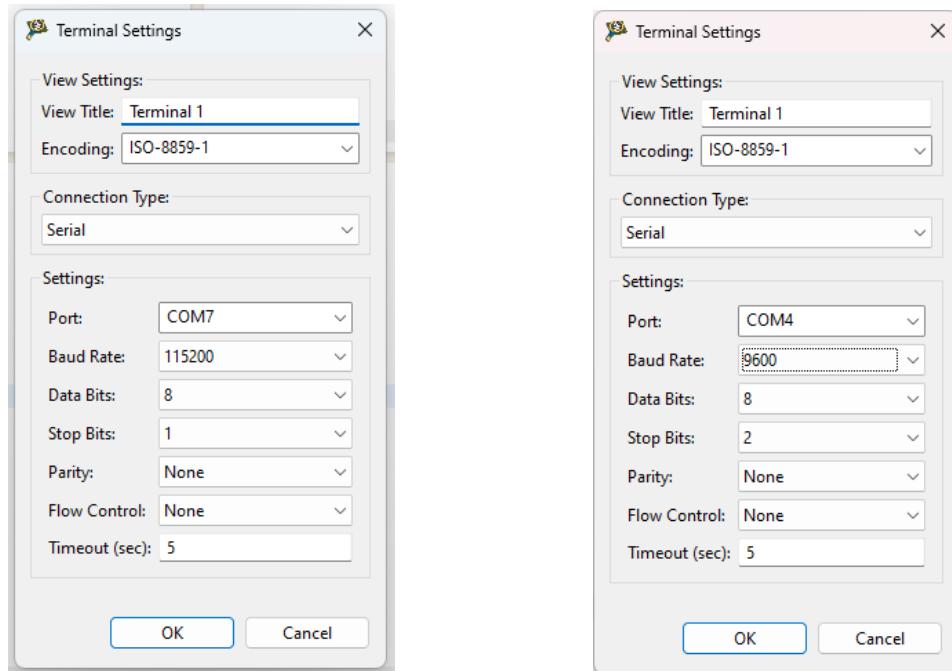


```
void UART0_IRQHandler (void)
{
    uint8_t c;

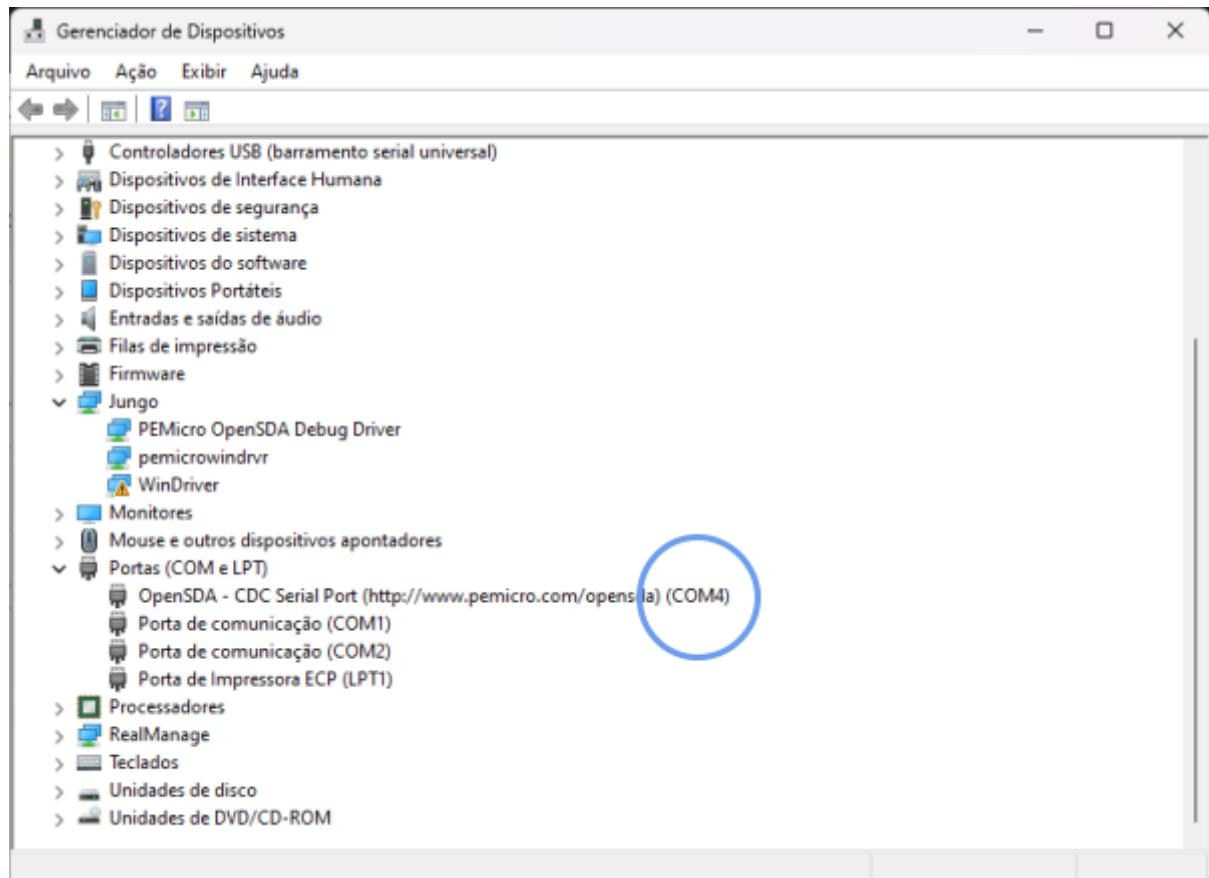
    if (UART0_S1&UART_S1_RDRF_MASK)
    {
        c = UART0_D;
        //A logica do codigo do Manual nao esta correto. As flags em 1
        //if (!(UART0_S1&UART_S1_TDRE_MASK) && !(UART0_S1&UART_S1_TC_MASK)
        if ((UART0_S1&UART_S1_TDRE_MASK) && (UART0_S1&UART_S1_TC_MASK))
        {
            UART0_D = c;
        }
    }
}
```

2.

O terminal já tinha sido configurado para a questão anterior, para o exemplo 1. Configurando também para o outro exemplo temos



Para configurar a porta, acessamos o Gerenciador de Dispositivos



3.1

Algumas diferenças foram encontradas:

1°)

No programa fornecido para o CodeWarrior:

```
PORTA_PCR1 |= PORT_PCR_MUX(0x2);      // UART0_RX
PORTA_PCR2 |= PORT_PCR_MUX(0x2);      // UART0_TX
```

Enquanto a especificação do manual era

```
PORTA_PCR2 = PORT_PCR_ISF_MASK|PORT_PCR_MUX(0x3);
PORTA_PCR1 = PORT_PCR_ISF_MASK|PORT_PCR_MUX(0x3);
```

Aqui é possível observar duas diferenças, a flag de interrupção não precisa ser limpa, provavelmente porque na inicialização já espera-se que não haja pendência. A outra diferença é que no manual é utilizado o operador de atribuição e não o operador “|”, se essas operações são equivalentes certamente o ambiente de desenvolvimento do manual configura todos os registradores em zero inicialmente, retirando o problema de sobreescriver um bit 1 no registrador ao utilizar a atribuição. Esta mesma diferença se repete em outros locais então não será mais comentada.

2°)

Outra diferença foi na habilitação da IRQ12, interrupção causada pelo módulo UART0. No código do manual parecem existir funções para ativar (*enable_irq*) e configurar a prioridade (*set_irq_priority*) já feitas. No caso do código fornecido para o CodeWarrior a função foi criada, ativando a IRQ e estabelecendo a prioridade passada como parâmetro, a função é *UART0_habilitaNVICIRQ12*.

3°)

Ainda sobre interrupções, o endereço da rotina de tratamento da interrupção do UART0 é definida no arquivo *kinetis_sysinit.c*, pelo próprio CodeWarrior, com o nome *UART0_IRQHandler*. No manual, ao que parece esta configuração do vetor de interrupção é feita utilizando macros:

```
extern void uart0_isr(void);
#undef VECT0R_028
#define VECT0R_028 uart0_isr
```

4°)

Outra diferença é na configuração do valor de SBR configurado nos registradores *UART0_BDH* e *UART0_BDL*, no código do manual, o valor foi calculado e os valores a serem guardados em cada registrador (5 bits em um e 8 bits no outro) foram escritos de forma

hardcoded no programa. No programa fornecido para o CodeWarrior, foi escrita a função *UART0_SBR*, que calcula *SBR* em tempo de execução e escreve o valor correto em cada registrador.

No código do manual:

```
UART0_BDH = 0x00;  
UART0_BDL = 0x1A;
```

No programa para o CodeWarrior:

```
uint16_t sbr = UART0_SBR (clock, baud_rate);  
UART0_BDH &= ~UART0_BDH_SBR(0x1F);  
UART0_BDL &= ~UART0_BDL_SBR(0xFF);  
UART0_BDH |= UART0_BDH_SBR(sbr >> 8);  
UART0_BDL |= UART0_BDL_SBR(sbr & 0x00FF);
```

5°)

A forma de configuração do C1 também foi diferente, o efeito é o mesmo, pois *UART0_C1* tem apenas as 8 flags colocadas no programa, portanto a operação apenas seta o registrador inteiro em zero, assim como especificado pelo manual. No entanto, inserir explicitamente o que está sendo feito desta forma ajuda na compreensão do que está sendo configurado.

No manual, temos a versão simplificada:

```
UART0_C1 = 0x00;
```

No programa fornecido é possível ver em detalhe o que está sendo feito:

```
UART0_C1 &= ~(UART0_C1_LOOPS_MASK |  
              UART0_C1_DOZEEN_MASK |  
              UART0_C1_RSRC_MASK |  
              UART0_C1_M_MASK |  
              UART0_C1_WAKE_MASK |  
              UART0_C1_ILT_MASK |  
              UART0_C1_PE_MASK |  
              UART0_C1_PT_MASK  
);
```

6°)

Na configuração do registrador C2, nota-se que a flag RIE não foi configurada em *UART0_config_especifica*, mas ela é configurada em *UART0_habilitaInterruptRxTerminal*, que é chamada posteriormente, habilitando a interrupção após o fim da configuração.

7°)

Na configuração de C3 algo parecido com o caso acima aconteceu. Desta vez, o efeito teoricamente não é o mesmo, pois o registrador também tem 8 bits. Da forma que foi implementado, os campos ORIE, NEIE, FEIE e PEIE não são zerados. Estes campos habilitam interrupções em certas situações de erros, assume-se então que não há a necessidade de ativamente desabilitar estas interrupções quando desenvolvendo no ambiente do CodeWarrior. Talvez no ambiente do manual estes valores também já sejam configurados em zero por padrão, então ao invés de configurar apenas os outros 4 campos em 0 o registrador inteiro é escrito por comodidade. De qualquer forma, mostrar explicitamente quais campos estão sendo configurados ajuda a compreender o que o código está fazendo.

No código do manual:

```
UART0_C3 = 0x00;
```

No programa fornecido para o CodeWarrior:

```
UART0_C3 &= ~(UART0_C3_R8T9_MASK |  
    UART0_C3_R9T8_MASK |  
    UART0_C3_TXDIR_MASK |  
    UART0_C3_TXINV_MASK  
)
```

8°)

A configuração de S1 e S2 também é diferente, a diferença é do mesmo tipo que as anteriores, no código do manual o valor final desejado para o registrador é configurado diretamente, no código fornecido para o CodeWarrior foram utilizadas macros para facilitar na compreensão do que está sendo configurado.

No manual:

```
UART0_S1 |= 0x1F;  
UART0_S2 |= 0xC0;
```

No programa fornecido:

```
UART0_S1 |= (UART0_S1_IDLE_MASK  
    | UART0_S1_OR_MASK  
    | UART0_S1_NF_MASK  
    | UART0_S1_FE_MASK  
    | UART0_S1_PF_MASK);  
  
UART0_S2 |= (UART0_S2_LBKDIF_MASK  
    | UART0_S2_RXEDGIF_MASK);
```

9°)

Após a conclusão da configuração, o manual especifica a execução da instrução em *assembly* de mudança de estado do processador “CPSIE i”. Ela pode ser inserida em código C com a função *asm*. Esta instrução não é necessária no código no ambiente CodeWarrior.

3.2

O bloco que foi adicionado foi o de configuração do módulo MCG, no manual ele não foi mencionado, mas é a primeira rotina executada no programa fornecido pelo CodeWarrior e sem ela o programa não funcionaria pois o Clock não teria sido configurado. A função chamada no exemplo 1 é *MCG_initFLL48MH* e no exemplo 2 é *MCG_initIR4MHFLL20MH*.

4.

Tanto no código do exemplo 1 como no do exemplo 2, a operação do canal TX é feita por meio de polling, uma vez que as interrupções deste canal não foram habilitadas. Isso porque, apesar de ser definida no arquivo *UART.c*, a função de habilitação de interrupções de Tx não é chamada em linha nenhuma de código. Além disso, no exemplo 2, percebe-se a existência do “*while*” na *main*, que executa até que *UART0_S1_TC* seja igual a 1, indicando que a transmissão acabou. Por fim, outro “*while*” que demonstra o funcionamento por polling é o que se encontra na função *UART0_PutStr*, também em *main.c* e chamada na função *main*, na qual *UART0_S1_TDRE* igual a 0 é condição para continuação da transmissão, até que o buffer fique vazio. Já no exemplo 1, percebemos a existência de polling por meio do “*while*” na *main*, que possui como condição de continuação que o buffer de transmissão não esteja vazio e que a transmissão não tenha sido concluída, ou seja: *UART0_S1_TDRE* e *UART0_S1_TC* sejam 0.

5.

Esse cálculo é feito na função *UART0_SBR*, de forma que o valor usado, então, na determinação de *BDH* e *BDL* é o *sbr*, que é retorno dessa função e é dado por:

[1] *sbr* = (*uint16_t*) ((*clock**1.0)/(*baud_rate* * (*osr*+1)))

osr é lido do campo OSR do registrador *UART0_C4*, e é igual a 15, logo *osr* + 1 = 16. *baud_rate* é o valor desejado de taxa de transmissão, que no exemplo 2 é 9600. O *clock* é configurado em 4Mhz, pela função *MCG_initIR4MHFLL20MH*.

Usando a fórmula [1]: *sbr* = 26,041666667, no entanto, truncando, obtemos *sbr* = 26. Por fim, ao passar pelo if na função, o valor de *sbr* passa a ser de 27.

Finalmente, os valores de *BDH* e *BDL* são setados no bloco:

```
UART0_BDH &= ~UART0_BDH_SBR(0x1F);
UART0_BDL &= ~UART0_BDL_SBR(0xFF);
UART0_BDH |= UART0_BDH_SBR(sbr >> 8);
UART0_BDL |= UART0_BDL_SBR(sbr & 0x00FF);
```

Analizando os valores configurados temos

UART0_BDH = 0x00;
UART0_BDL = 0x1B;

Universal Asynchronous Receiver/Transmitter (UART0)	
UART0_BDH	0x00
UART0_BDL	0x1b
UART0_C1	0x00

Que é diferente do especificado pelo manual de

UART0_BDH = 0x00;
UART0_BDL = 0x1A;

O divisor configurado pelo manual faz com que a frequência especificada seja maior do que a taxa de transmissão especificada, o que é um erro por parte do manual.

6.

- 1) O valor é configurado na função *UART0_config_especifica*, ela recebe como parâmetro o valor da superamostragem, *osr*. Primeiro os cinco bits do campo OSR do registrador UART0_C4 são colocados em 1, pela linha
`UART0_C4 |= (0b11111 << UART0_C4_OSR_SHIFT);`

então, é feita uma máscara com todos os bits setados em 1, menos os bits zero do número *osr*, esta máscara é utilizada em uma operação “&=” com *UART0_C4*, terminando de configurar os bits de superamostragem.

```
UART0_C4 &= ((osr << UART0_C4_OSR_SHIFT) | ~UART0_C4_OSR_MASK);
```

Isto é necessário porque a escrita de um valor inválido no campo faz com que o ele seja resetado automaticamente em 0xF, portanto a lógica comum de zerar o campo e depois escrever o valor não funcionaria.

- 2) Em *Rot7_example2*, não há instrução nenhuma que sete *UART0_C4_OSR*, no entanto, isso não é necessário, uma vez que se deseja que essa taxa tenha valor igual a 16 e, por padrão, a taxa é setada em 16, sendo preciso alterá-la apenas no caso de o valor desejado ser algo diferente, entre 4 e 32.

Segunda parte

1.

A opção mais flexível é a implementada no *rot7_aula*, uma vez que, ao usar a struct, torna-se mais fácil configurar um novo modo de operação. A facilidade vem do fato de que podemos

referenciar um campo de um registrador de forma semântica, alterando os parâmetros desejados na configuração em um mesmo local, ao invés de termos de fazer referência pelo nome e campo de cada registrador da configuração. Com a struct, basta que tenhamos, também, uma função que distribua os valores salvos nela para nos respectivos campos dos registradores.

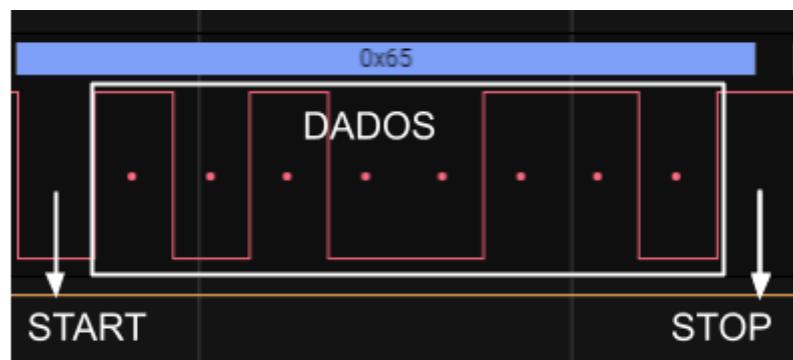
No *rot7_aula* existem duas funções que fazem isso, uma para o UART0 (*UART0_configure*) e outra para UART1 e UART2 (*UART_configure*), dada a diferença na configuração entre eles.

2.

- 1) Foi digitado “VERDE” + ENTER no terminal, o resultado foi a palavra “Verde” sendo impressa repetidamente na tela. No analisador, a seguinte saída foi vista no programa *Logic*



Focando em um dos bytes enviados, temos



A ordem dos bits enviados foi inversa, ou seja, os bits estão ordenados do menos para o mais significativo (LSB). Sendo assim, ao analisar da direita para a esquerda, temos o número 0b01100101, que, em hexadecimal, é igual a 0x65, conforme é apresentado na imagem acima. Por fim, buscando na tabela ASCII, concluímos, inclusive, que esse byte corresponde à letra “e”. Ademais, observando a imagem e o conjunto de bits no programa Logic, notamos que esses dados são precedidos pelo start bit e terminam com um stop bit.

- 2) A configuração original não conta com bit de paridade, então a única alteração feita nas variáveis *config0* e *config1* foi a da quantidade de stop bits, a alteração também foi feita no Logic 2.

```

UART0Config_type config0 = {
    //      .bdh_sbns=0,           ///< Um stop bit
    .bdh_sbns=1,           ///< Dois stop bits
    .sbr=0b100,           ///< MCGFLLCLK/(baud_rate*osr)=4
    .cl_loops=0.           ///< Operacao normal: full-duplex
}

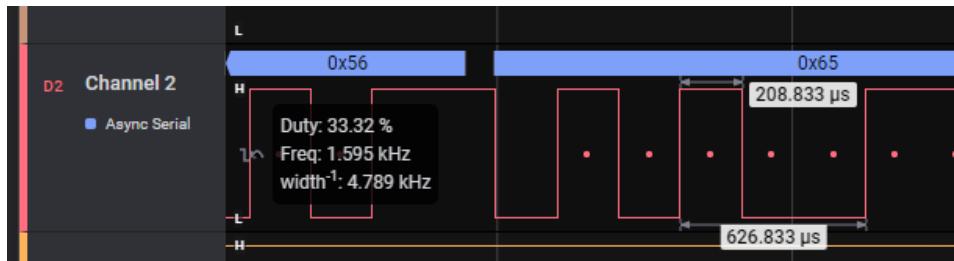
```

Analizando o mesmo caractere, vemos que o número de bits de parada aumentou para dois, nota-se que isso também aumenta o tempo de transmissão de cada caractere, já que um bit a mais é transmitido.

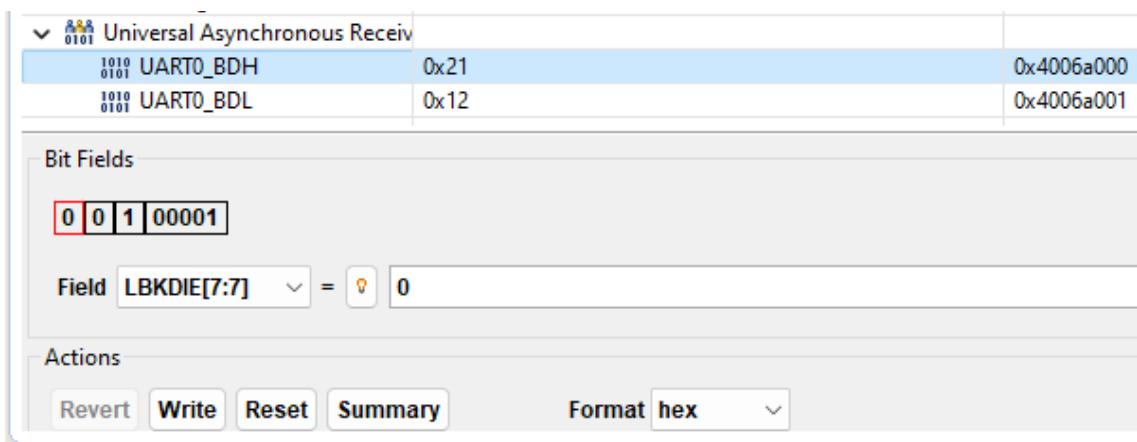


3.

- 1) Medindo a largura de um dos bits do caractere é de aproximadamente 209 us



calculando o valor baseado no baud rate, ou seja, pela razão 1/4800, temos o resultado 208,3333 us, bastante condizente com o amostrado na imagem, embora aproximadamente 0,5us menor.



Temos que $UART0_BDH_SBR = 0b00001$, $UART0_BDL_SBR = 0b00010010$, então $SBR = 0b100010010 = 274$, com este valor de SBR a taxa de transmissão fica igual à aproximadamente 4784, ao invés de 4800. Isso acontece pois SBR é truncado e arredondado

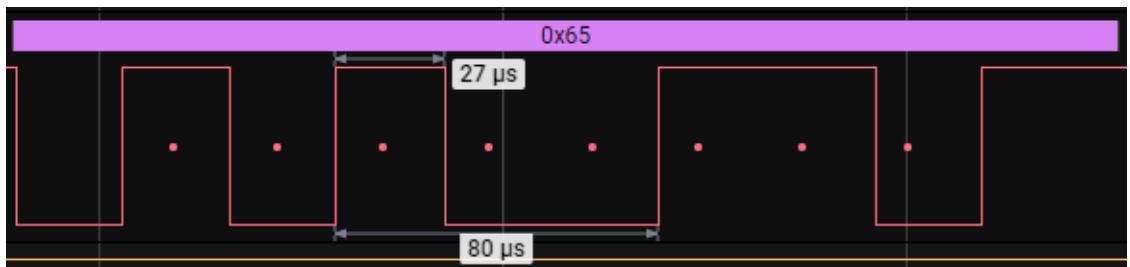
para o inteiro mais próximo. Para esta taxa temos uma largura de pulso de aproximadamente 209 us.

Por fim, comparando os três modos (medida em Logic, cálculo por baud rate e cálculo pelos valores nos registradores), percebemos que a medida pelo Logic é mais próxima do cálculo feito por meio dos valores contidos nos registradores. Enquanto que destoa um pouco daquela calculada por meio do valor de baud rate. Isso porque, o valor de baud rate de 4800 não é perfeitamente preciso, sendo aproximadamente 4784, pois o cálculo feito por código, apesar de ser baseado nesse valor, não resulta exatamente nele, o que ocasiona uma diferença de aproximadamente 0,7us no resultado.

2)

Alterando no código e no programa do analisador, temos

```
// Baud rates: 300, 1200, 2400
//#define BAUD_RATE 4800
#define BAUD_RATE 38400
```



A largura medida é de aproximadamente 27 us, que difere levemente do valor esperado para um baud rate de 38400, de 26 us, essa diferença se deve novamente à diferença no SBR que é configurado, devido à aproximação feita. Nota-se também que o ponto esperado para a medida, mostrado no programa Logic está se deslocando do centro do pulso, isso ocorre devido à diferença na frequência esperada e a que foi realmente configurada e poderia se tornar um problema se fossem transmitidos mais bits por vez.

4.

Quanto ao canal RX, a interrupção é habilitada na *main*, por meio da chamada da função *UART0_habilitaInterrupRxTerminal()* e não é desabilitada em ponto nenhum do código, assim qualquer dado pode ser recebido à qualquer momento do terminal.

Já no canal TX, temos uma lógica mais complexa, as strings são enviadas por interrupção, existem algumas funções que fazem com que elas sejam enviadas pelo pino TX: *ISR_EnviaString10x*, *ISR_EnviaString* e *ISR_Realignamento*. Essas funções seguem um mesmo princípio de funcionamento: Elas escrevem os caracteres da string a ser enviada, em ordem, em um *buffer* circular e habilitam as interrupções. No caso da função *EnviaString* temos

```

void ISR_EnviaString (char *string) {
    uint8_t i;

    while (BC_push( &buffer0, string[0]) == -1);
    UART0_C2 |= UART0_C2_TIE_MASK;
}

```

Isso faz com que uma interrupção seja gerada e a rotina de tratamento da interrupção do UART0 seja executada, como é possível ver abaixo, com a máscara TDRE é possível identificar que a interrupção causada foi devido ao canal TX, e o tratamento é retirar um caractere do buffer e enviá-lo pelo pino, isto é feito escrevendo o caractere no registrador UART0_D. Neste momento, o módulo começa o envio do dado e quando finaliza gera uma interrupção que ativa novamente a rotina e envia o próximo caractere, isto é feito até que não haja mais itens no buffer, momento em que as interrupções são desabilitadas.

```

void UART0_IRQHandler()
{
    char item;

    if (UART0_S1 & UART0_S1_RDRF_MASK) {
        // ...
    }
    else if (UART0_S1 & UART0_S1_TDRE_MASK) {
        /*!
         * Interrupção solicitada pelo canal Tx
         */
        if (BC_isEmpty(&buffer0))
            UART0_C2 &= ~UART0_C2_TIE_MASK;      // < desabilita Tx
        else {
            BC_pop (&buffer0, &item);
            UART0_D = item;
        }
    }
}

```

5.

A transição ESPERA→EXTRAI ocorre quando o botão enter do teclado é pressionado. Nesse momento, é adicionado um terminador ('\0') no buffer circular e, ocorre uma espera durante o tempo em que a flag TDRE tem valor igual a 0, ou seja, o buffer de transmissão está cheio. Quando o buffer é liberado, adiciona-se uma quebra de linha '\n' no buffer de transmissão para que o cursor do usuário no terminal seja jogado para a linha de baixo automaticamente. Por fim, o estado é setado como EXTRAI. O que se percebe no if do print a seguir

```

if (item == '\r') {
    // inserir o terminador e avisar o fim de uma string
    BC_push (&bufferE, '\0');
    while (!(UART0_S1 & UART_S1_TDRE_MASK));
    UART0_D = '\n';
    flag = EXTRAI;
}

```

A transição EXTRAI→MOSTRA ocorre após o fim da extração da string digitada pelo usuário antes de pressionar a tecla ENTER. Ela está armazenada no buffer circular e é retirada de lá e salva na string passada como parâmetro para a função *ISR_extraiString*. Esta função é chamada no *loop* infinito localizado na *main*, que verifica constantemente o estado e chama a função quando ele é igual a EXTRAI. Após o fim da extração o estado é alterado para MOSTRA.

```
for(;;) {
    if (ISR_LeEstado() == EXTRAI) {
        ISR_extraiString (string);
        ISR_escreveEstado (MOSTRA);
    }
}
```

A transição MOSTRA→ESPERA ocorre se a palavra digitada pelo usuário não estiver dentre as opções previstas, VERDE, VINHO, VIOLETA, VERMELHO ou VIRIDIANO. No *loop* da *main* é possível ver que no estado MOSTRA, o código utiliza a função de comparação de string *strcmp* para identificar qual palavra foi digitada e enviá-la dez vezes na mesma linha. Se a palavra não for reconhecida, a mensagem "Digite novamente" é enviada e o estado é reiniciado em ESPERA.

```
if (ISR_LeEstado() == MOSTRA) {
    if (strcmp (string, "VERDE") == 0) {
        ISR_EnviaString10x ("Verde");
        ISR_Realignamento();
    } else if (strcmp (string, "VINHO") == 0) {
        ISR_EnviaString10x ("Vinho");
        ISR_Realignamento();
    } else if (strcmp (string, "VIOLETA") == 0) {
        ISR_EnviaString10x ("Violeta");
        ISR_Realignamento();
    } else if (strcmp (string, "VERMELHO") == 0) {
        ISR_EnviaString10x ("Vermelho");
        ISR_Realignamento();
    } else if (strcmp (string, "VIRIDIANO") == 0) {
        ISR_EnviaString10x ("Viridiano");
        ISR_Realignamento();
    }
    else {
        ISR_EnviaString ("Digite novamente\n\r");
        ISR_escreveEstado (ESPERA); //reseta estado
    }
}
```

A transição MOSTRA→LIBERA_BUFFER ocorre quando o usuário envia um caractere enquanto as linhas com o nome da cor repetido dez vezes são impressas, esse envio indica que é necessário parar de enviar estas palavras e mostrar para o usuário as opções de cores novamente. A troca de estado é feita na rotina de tratamento de interrupção *UART0_IRQHandler*, dentro do *if* de solicitações do canal de recepção, RX. Também é possível ver que neste caso o caractere é apenas descartado, não sendo nem ecoado para o usuário, pois a função retorna.

```
if (flag == MOSTRA) {
    // Parar o preenchimento de buffers circulares
    flag = LIBERA_BUFFER;
    return;
}
```

Finalmente, a transição LIBERA_BUFFER→ESPERA ocorre na main por meio de um if, no qual o programa espera com espera ocupada, o esvaziamento do buffer de saída (BC2), garantindo que ele já esteja liberado de um possível envio prévio que foi solicitado e, em seguida envia a string pedindo para o usuário digitar uma cor no terminal. Até que, na última instrução dentro da condicional, o estado é alterado para ESPERA. Conforme se percebe a seguir

```

} else if (ISR_LerEstado() == LIBERA_BUFFER) {
    //Aguardar o envio completo dos caracteres
    while (!ISR_BufferSaidaVazio());
    ISR_EnviaString ("Digite [VERDE|VINHO|VIOLETA|VERMELHO|VIRIDIANO]:\n\r");
    ISR_escreveEstado (ESPERA);      //reseta estado
}

```

6.

- 1) Esses buffers têm uma quantidade máxima de 128 caracteres, o tamanho é definido na função BC_init(), chamada em ISR_inicializaBC() que, por sua vez, é chamada na *main*. Sendo assim, notando ainda que o parâmetro “tamanho” de BC_init() recebe o valor da macro TAM_MAX na chamada feita em ISR_inicializaBC(), concluímos que o elemento “tamanho” do buffer é setado como 128, conforme se segue com os prints do processo após a chamada na *main*

```

// Setup
ISR_inicializaBC();



---


void ISR_inicializaBC () {
    /*
     * Inicializa um buffer circular de entrada
     */
    BC_init (&bufferE, TAM_MAX);

    /*
     * Inicializa dois buffers circulares de saída
     */
    BC_init (&buffer0, TAM_MAX);
    BC_init (&buffer2, TAM_MAX);
}



---


void BC_init(BufferCirc_type *buffer, unsigned int tamanho)
{
    buffer->dados = malloc(tamanho * sizeof(char));
    buffer->tamanho = tamanho;
    buffer->escrita = 0;
    buffer->leitura = 0;
}

Macro: #define TAM_MAX 128

```

- 2) O buffer circular *bufferE* armazena os caracteres digitados pelo usuário até que ele envie um caractere ‘\r’, ao pressionar a tecla ENTER. Isto fica evidente dentro do caso de tratamento de interrupção causado pelo canal RX, que chama a função *BC_push* com este buffer:

```

if (item == '\r') {
    //inserir o terminador e avisar o fim de
    BC_push (&bufferE, '\0');
    while (!(UART0_S1 & UART_S1_TDRE_MASK));
    UART0_D = '\n';
    flag = EXTRAI;
} else {
    BC_push (&bufferE, item);
}

```

O *buffer0* armazena dados a serem enviados pelo pino TX, seguindo a lógica de interrupções já explicada. A string é armazenada no buffer e cada interrupção causada pelo TX faz com que um caractere seja enviado, até que todos sejam. O mesmo vale para o *buffer2*, porém nele ficam guardados os caracteres a serem enviados por UART2, que serão lidos com o analisador lógico, efetivamente espelhando o sinal sendo enviado pelo pino do UART0.

Vale destacar que apesar dos mesmos conteúdos serem enviados para os dois módulos (0 e 2), ainda são armazenados dois buffers que são lidos e esvaziados conforme as respectivas interrupções causadas pelos módulos, que funcionam de forma independente.

No caso de interrupção causada por TX, para o UART0, vemos o uso de *BC_POP* com o *buffer0*.

```

else if (UART0_S1 & UART0_S1_TDRE_MASK) {
    /*
     * Interrupcao solicitada pelo canal Tx
     */
    if (BC_isEmpty(&buffer0))
        UART0_C2 &= ~UART0_C2_TIE_MASK;
    else {
        BC_pop (&buffer0, &item);
        UART0_D = item;
    }
}

```

E na rotina de tratamento de interrupção do *UART2*, também para interrupção causada por TX, o uso de *BC_POP* com o *buffer2*.

```

/*
 * Interrupcao solicitada pelo canal Tx
 */
if (BC_isEmpty(&buffer2))
    UART2_C2 &= ~UART_C2_TIE_MASK; ///<
else {
    BC_pop (&buffer2, &item);
    UART2_D = item;
}

```

- 3) Inicialmente, vale entender os campos “escrita” e “leitura” dos buffers. Isso porque, o campo “escrita” armazena o índice do elemento do vetor no qual será escrito o caractere seguinte. Enquanto que o campo “leitura” armazena o índice do próximo elemento do vetor a ser lido.

As funções BC_push e BC_pop, calculam, respectivamente, a posição do próximo caractere a ser escrito/lido (não o que será escrito/lido nesta chamada, mas na próxima) e armazena na variável *next*, este valor é calculado somando-se um no campo *escrita/leitura*, em seguida o valor é comparado com o tamanho do *buffer*, se for igual a posição é inválida, pois a última posição do vetor é tamanho-1. Neste caso, a próxima posição é a primeira do vetor e é isso que faz com que a lógica fique circular.

```
next = buffer->escrita+1;
if (next == buffer->tamanho) next = 0;

next = buffer->leitura + 1;
if (next == buffer->tamanho) next = 0;
```

O *buffer* é declarado como cheio se a próxima posição (*next*) é igual ao campo *leitura*. Neste caso, o caractere não é armazenado no *buffer* e a função retorna -1, de forma a indicar que houve um erro para que o bloco de código que fez a chamada.

7.

A função de configuração dos módulos UART1 e UART2, *UART_configure*, acessa a posição *x* do vetor *UART*, declarado no topo do arquivo *UART.c*, o mesmo arquivo em que a função é implementada. *x* é um valor passado como parâmetro para a função, indicando qual ponteiro utilizar.

```
UART_MemMapPtr UART[] = UART_BASE_PTRS ;
```

O vetor é igualado com o uso de uma macro, que por sua vez utiliza duas outras macros, os ponteiros para *structs* *UART_MemMapPtr*, com os endereços base dos registradores de cada módulo. Estas macros são definidas no arquivo *MKL25Z4.h*, o mesmo arquivo que fornece as outras diversas macros que são utilizadas

```
/** Peripheral UART1 base pointer */
#define UART1_BASE_PTR          ((UART_MemMapPtr) 0x4006B000u)
/** Peripheral UART2 base pointer */
#define UART2_BASE_PTR          ((UART_MemMapPtr) 0x4006C000u)
/** Array initializer of UART peripheral base pointers */
#define UART_BASE_PTRS           { UART1_BASE_PTR, UART2_BASE_PTR }
```

A característica em comum entre os registradores dos dois módulos é que os campos de seus registradores têm as mesmas funções. No entanto, não seria possível utilizar essa mesma generalização para UART0, já que, alguns campos de alguns de seus registradores têm funções destoantes daquelas assumidas pelos mesmos campos em UART1 e UART2.