

EXPERIMENTO 7 – Interface Serial Assíncrona

FEEC | EA871

Thiago Maximo Pavão - 247381

Vinicius Esperança Mantovani - 247395

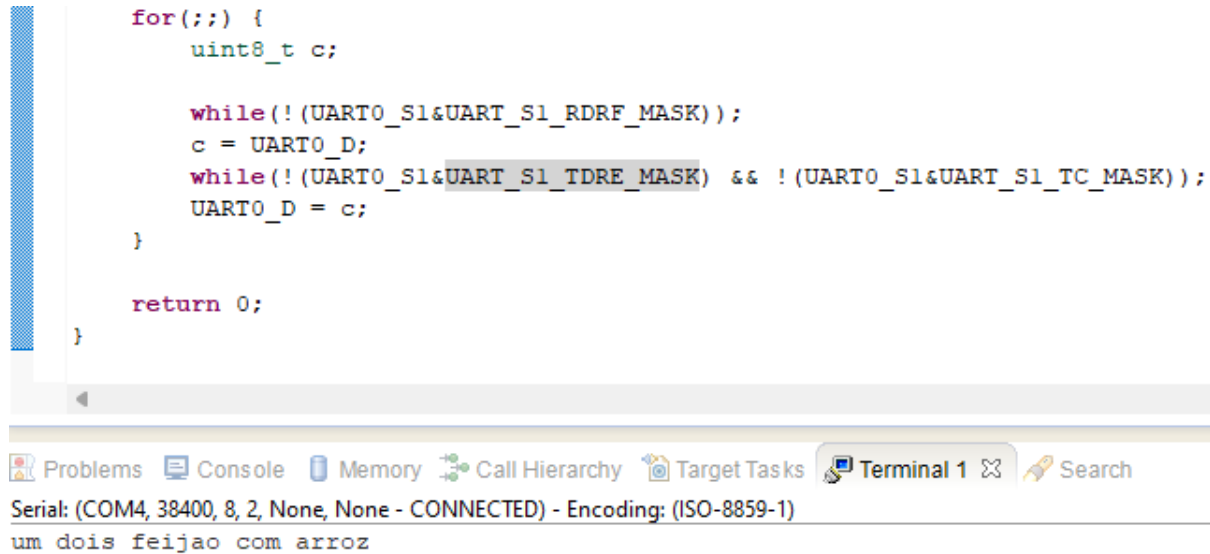
Terceira parte

3.4

```
for(;;) {
    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;
}

return 0;
}
```



Foi feita a inicialização, e alguns caracteres foram digitados no terminal. Foi possível vê-los enquanto eles eram digitados então a inicialização/conexão e o ecoamento dos caracteres está funcionando.

3.5.1

Foi implementada a rotina de tratamento de interrupção do módulo UART0, para testar, foi feita a conexão e alguns caracteres foram enviados com o programa executando em modo debug e com um breakpoint dentro do caso de pressionamento de ENTER. Após digitar alguns caracteres e pressionar ENTER, vimos que a execução para na linha esperada:

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

Analisando a aba de variáveis do debugger, nota-se que o vetor de dados do buffer circular se encontra no endereço de memória `0x1ffff068`.

Variables Breakpoints Registers Memory Modules		
Name	Value	Location
(x) item	'\r'	0x20002fcf
(x) flag	' '	0x00000000
▼ bufferE	0x1ffff008	0x1ffff008
> dados	0x1ffff068	0x1ffff008
(x) tamanho	100	0x1ffff00c
(x) leitura	0	0x1ffff010
(x) escrita	28	0x1ffff014
> bufferS	0x1ffff018	0x1ffff018

Acessando a aba de memória do debugger, no endereço do vetor de dados, vemos

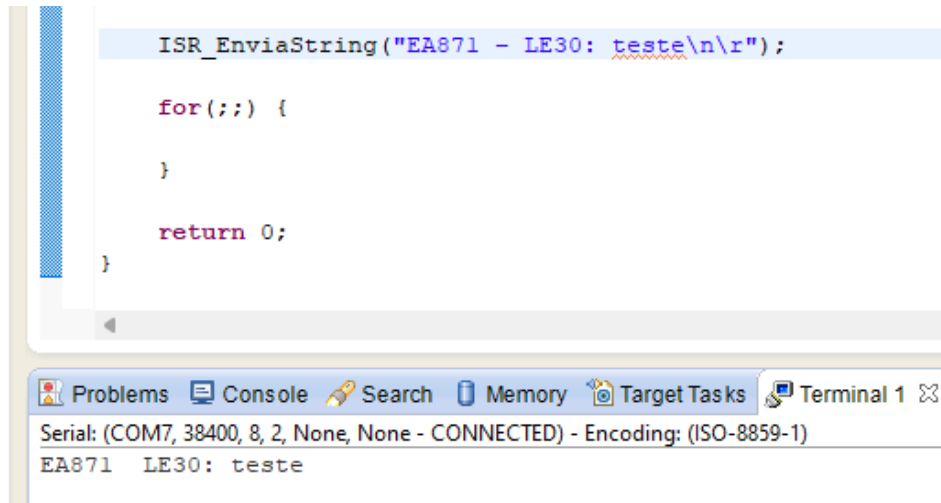
Address	0	1	2	3
1FFFF05C	44	F1	FF	1F
1FFFF060	D0	00	00	00
1FFFF064	D0	F0	FF	1F
1FFFF068	74	72	65	73
1FFFF06C	20	71	75	61
1FFFF070	74	72	6F	20
1FFFF074	66	65	69	6A
1FFFF078	61	6F	20	6E
1FFFF07C	6F	20	70	72
1FFFF080	61	74	6F	00
1FFFF084	B4	F0	FF	1F

Nota-se que o primeiro valor 0 ocorre no endereço `0x1ffff083`. Que indica que este deve ser o terminador da string enviada. Alterando a interpretação dos dados como caracteres, pelo debugger, vemos o que foi digitado e se encontra no buffer circular *“tres quatro feijao no prato”*.

Address	0	1	2	3
1FFFF05C	D	ñ	ÿ	
1FFFF060	Ð	�	�	�
1FFFF064	Ð	ð	ÿ	
1FFFF068	t	r	e	s
1FFFF06C		q	u	a
1FFFF070	t	r	o	
1FFFF074	f	e	i	j
1FFFF078	a	o		n
1FFFF07C	o		p	r
1FFFF080	a	t	o	�
1FFFF084	'	ð	ÿ	

3.5.2

O teste de unidade foi bem sucedido.



```
ISR_EnviaString("EA871 - LE30: teste\n\r");

for(;;) {

}

return 0;
}
```

Problems Console Search Memory Target Tasks Terminal 1

Serial: (COM7, 38400, 8, 2, None, None - CONNECTED) - Encoding: (ISO-8859-1)

EA871 LE30: teste

3.7

Inicialmente, na questão 3.6.1, escrevemos a função `ExtraiString2Tokens` sem muitas dificuldades quanto às condições de erros. No entanto, ao fazer testes de unidade, percebemos um erro recorrente, que fazia com que o programa fosse direcionado para a rotina default de tratamento de interrupções, por conta da exceção gerada. Assim, procurando pelo erro, percebemos que estava relacionado com o modo como declaramos a string passada para a função `strtok`, que havia sido declarada de tal forma que era armazenada como constante na memória, mas, trocando `char *string` por `char string[100]`, chegamos ao resultado esperado e, não caímos mais na rotina de tratamento de exceção.

Em seguida, partindo para o item 3.6.2, de maneira minimamente destoante do indicado, optamos por fazer uma única função que discerne entre bases de acordo com o que entra do terminal e, converte a string em int. Para tanto, não tivemos grandes complicações, sendo consideravelmente tranquilo o desenvolvimento e, obtendo sucesso nos testes de unidade.

Seguindo, no item 3.6.3, criamos a função `paridade()` responsável por determinar se um número tem, em binário, paridade par ou ímpar. Sendo assim, o retorno da função é 0 se sua paridade é par e 1 caso seja ímpar. Como no item 3.6.2, não tivemos problemas nos testes de unidade desta função, até pela lógica não muito complexa envolvida.

Ademais, desenvolvemos a função `ConvertUl32toBitString` no item 3.6.4, que está encarregada de converter um número inteiro em uma string na base binária. Esta função também não apresentou problemas na execução dos testes de unidade. Ainda assim, a fizemos de um modo, primeiramente, que optamos por alterar, para simplificar o código e diminuir o tempo computacional (embora não seja muito significativo). Isso de modo que, ao invés de coletarmos a string ao contrário após o cálculo dos valores dos bits e continuarmos com o preenchimento com 0's nos bits não usados, resolvemos receber os bits do fim para o começo da string e continuarmos o mesmo loop até o índice 0 da string, de modo que, a partir do último bit de valor 1, a string é preenchida com 0's.

Por fim, executamos a tarefa exigida no item 3.6.5, tratamos a string para enviarmos da maneira como se deseja ao terminal. Para tanto, além de convertermos o número inteiro para uma string, tivemos de concatenar strings para formar toda a sentença a ser enviada para o terminal,

fazendo, para tanto, inclusive, um vetor contenedor dos valores “par” e “ímpar”, nesta ordem, o que nos facilitou no preenchimento dos campos das strings impressas relacionados à paridade e ao número de 1 no número.

A função de paridade utilizada foi a mesma implementação fornecida da web. O resultado fornecido por ela foi utilizado para descobrir qual o dígito de paridade deveria ser gerado. O roteiro pedia que a própria função paridade já fizesse isso, mas pensamos que fosse melhor fazer desta outra forma.