

8

UART Universal Asynchronous Receiver and Transmitter

Versão 1.0 21/08/2020

Dúvida: devemos usar UCBR, UCBRF e UCBRS ou colocar o x no final UCBRx, UCBRFx e UCBRSx? Acho que esse "x", apesar de indicar que os valores são específicos para cada USCI_Ax, complicam. Ao programar, esse "x" não é usado. O texto está misturando as duas.

A necessidade de comunicação é uma constante nos sistemas microprocessados. Normalmente, precisamos nos comunicar com os sistemas que projetamos para obter valores das diversas variáveis de controle e para enviar parâmetros de configuração. A comunicação também é muito útil enquanto estamos desenvolvendo programas, depurando-o e buscando por erros. Um meio de comunicação muito empregado é a comunicação serial assíncrona. Ela é interessante pela simplicidade do protocolo e do cabo de conexão. Na forma mais simples, bastam três fios: um para transmissão, outro para recepção e o último ligado à terra para fornecer a referência de tensão.

Devemos comparar esse cabo com o cabo usado numa comunicação paralela de 8 bits, por exemplo, que deve fazer uso de pelo menos dez condutores. Especialmente quando se trabalha com grandes distâncias, o cabo serial oferece uma boa economia. O preço pago por essa simplicidade é a velocidade, pois, podemos dizer que a comunicação serial é mais lenta que a paralela. Isso é intuitivo: é mais rápido enviar 8 bits de uma só vez, por 8 caminhos independentes, que enviar 8 bits sequencialmente através de um único caminho. Entretanto, com a evolução da tecnologia, estão surgindo canais seriais extremamente rápidos, que conseguem conciliar a simplicidade do cabo com a velocidade, citamos USB, SATA e PCI-Express (fazem uso paralelo de canais seriais).

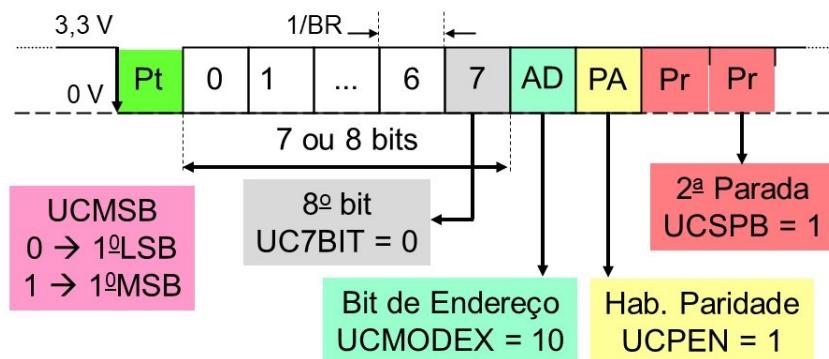
Neste estudo do MSP430 veremos os protocolos seriais mais simples: UART, I²C e SPI. Entretanto, é costume usar o termo Porta Serial para fazer referência à comunicação

serial assíncrona (UART). Isto acontece porque ela foi a primeira porta serial de largo uso. A sigla **UART** significa em inglês “*Universal Asynchronous serial Receiver and Transmitter*”, a tradução seria “Transmissor e Receptor Serial Assíncrono Universal”. Algumas vezes encontramos a sigla **USART** que inclui o recurso síncrono e significa em inglês “*Universal Synchronous and Asynchronous serial Receiver and Transmpter*”.

8.0. Quero Usar a Porta Serial e não Pretendo Ler Todo este Capítulo

Para o leitor que não tem experiência em comunicação serial assíncrona (ou é humilde), recomendamos o avanço para o tópico 8.1. Para os demais, abordarmos neste tópico o mínimo necessário para configurar e empregar as duas UARTs disponíveis (F5529).

A figura abaixo resume as possibilidades de operação de uma porta serial assíncrona do MSP430. Ela trabalha com campos de dados de 7 ou 8 bits, podendo enviar primeiro o bit mais significativo ou primeiro o bit menos significativo, com ou sem paridade (par ou ímpar), oferece recurso para sinalizar um caractere como dado ou endereço e ainda disponibiliza a escolha de 1 ou 2 bits de parada.



Somente as unidades USCI_A0 e USCI_A1 (MSP430 F5529) podem operar no modo UART. A USCI_A0 tem pinos dedicados exclusivamente a ela enquanto a USCI_A1 precisa lançar mão de dois pinos da porta P4, através de mapeamento. Tudo está resumido na tabela abaixo (a LaunchPad disponibiliza apenas P4.0, 1, 2 e 3).

	USCI_A0	USCI_A1
TXD	P3.3	Mapear em P4.0, P4.1, P4.2 ou P4.3

RXD	P3.4	Mapear em P4.0, P4.1, P4.2 ou P4.3
-----	------	---------------------------------------

Para disponibilizar os pinos para a USCI_A0 é necessária a habilitação da função alternativa em P3SEL.

```
P3.SEL |= BIT4 | BIT3;
```

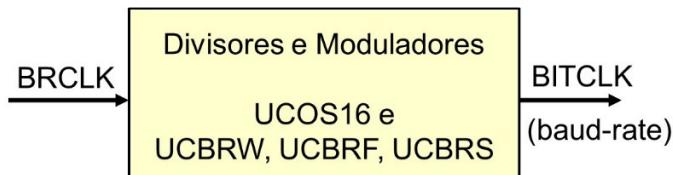
Para disponibilizar os pinos para a USCI_A1 é necessário selecionar 2 pinos da porta P4 e fazer o mapeamento com a sequência abaixo.

```
P4SEL |= BITi | BITj; //Disponibilizar P4.i e P4.j
PMAPKEYID = 0X02D52; //Liberar mapeamento de P4
P4MAPi = PM_UCA1TXD; //P4.i = TXD
P4MAPj = PM_UCA1RXD; //P4.j = RXD
```

Para configurar uma unidade USCI_Ax, é necessário obedecer à sequência abaixo.

1. Ativar a condição de *reset*, ou seja, fazer UCSWRST = 1.
2. Inicializar os registradores, incluindo UCAxCTL1.
3. Configurar as portas de I/O usadas.
4. Finalizar a condição de *reset*, ou seja, fazer UCSWRST = 0.
5. Se for o caso, habilitar as interrupções (UCRXIE e UCTXIE).

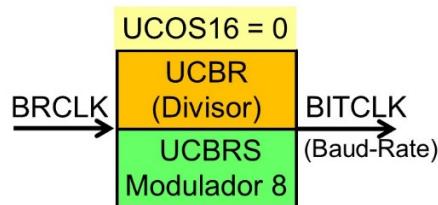
A programação da velocidade de transmissão (*baud-rate*) envolve a questão de gerar o relógio BITCLK a partir do relógio BRCLK, como mostrado na figura.



Existe dois modos para a geração do *baud-rate*:

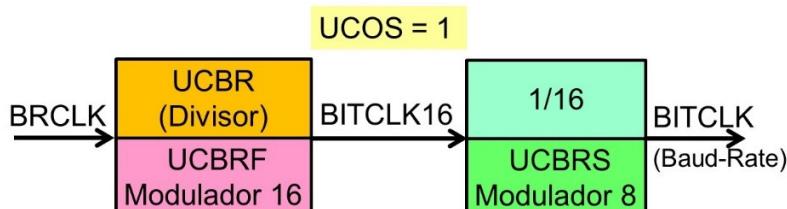
- (UCOS16 = 0) Modo Baixa Frequência e
- (UCOS16 = 1) Modo Super Amostragem.

O Modo Baixa Frequência é usado quando relação entre BRCLK está próximo de BITCLK (maior que 3). Ao usar frequências baixas, diminui-se o consumo de energia. Entretanto, quando a relação entre esses dois sinais for bem próxima de 3, a incerteza na detecção do bit de partida aumenta a chance de erros. Apresentamos seu diagrama de blocos e o algoritmo de cálculo. É recomendada a consulta à Tabela 36-4 (pag 951) do Manual do Usuário do MSP430.



1. Calcular $n = BRCLK / BITCLK$; // É preciso $n \geq 3$
2. $N = INT(n)$; // $N =$ parte inteira de n
3. $UCBR = N$; // Programar divisor, $UCBR = N$
4. $m8 = (n - N) \times 8$; // Calcular valor para o Modulador 8
5. $M8 = round(m8)$; // Arredondar
6. $UCBRS = M8$; // Programar Modulador 8, $UCBRS = M8$

O Modo Super Amostragem, é usado quando a relação entre BRCLK e BITCLK é maior que 16. Se a relação entre eles for muito grande, os instantes de amostragem da linha de recepção ficarão muito próximos, o que aumenta a vulnerabilidade ao ruído. O uso de uma frequência mais elevada, implica num maior consumo. Apresentamos seu diagrama de blocos e o algoritmo de cálculo. Recomendamos a consulta da Tabela 36-5 (pag 953) do Manual do Usuário do MSP430.



1. Calcular $BITCLK16 = 16 \times BITCLK$; // Calcular sinal intermediário
2. Calcular $n = BRCLK / BITCLK16$; //
3. $N = INT(n)$; // $N =$ parte inteira de n
4. $UCBR = N$; // Programar divisor, $UCBR = N$
5. $m16 = (n - N) \times 16$; // Calcular valor para o Modulador 16
6. $M16 = round(m16)$; // Arredondar (preferir para baixo)
7. $UCBRF = M16$; // Programar Modulador 16
8. $BITCLK16 = BRCLK / (N + M / 16)$; // Recalcular BITCLK16
9. $n = BITCLK16/BITCLK$; // Calcular novo n
10. $m8 = (n - 16) \times 8$; // Calcular valor para o Modulador 8
11. $M8 = round(m8)$; // Arredondar
12. $UCBRS = M8$ // Programar Modulador 8

Após configurada a porta serial, sua operação é muito simples e faz uso de dois *buffers*:

- **UCAxTXBUF** é o *buffer* de transmissão. O caractere aqui escrito é copiado para o registrador de deslocamento e transmitido pela linha TXD. A *flag* UCTXIFG vai para 1 sempre que este registrador ficar vazio e é zerada automaticamente por ocasião da escrita. Ela deve ser encarada como sendo a autorização para escrever neste *buffer*.
- **UCAxRXBUF** é o *buffer* de recepção. Ele recebe o caractere que recém chegou para linha RXD. Este caractere deve ser lido antes da chegada do próximo. A *flag* UCRXIFG vai a 1 para indicar a existência de um dado neste *buffer* e é zerada quando este dado é lido.

As *flags* UCTXIFG e UCRXIFG estão no registrador UCAxIFG e podem provocar interrupções caso elas estejam habilitadas no registrador UCAxIE. A seguir apresentamos um gabarito para a configuração dos registradores envolvidos na comunicação serial assíncrona.

Registros de 8 bits

	7	6	5	4	3	2	1	0
UCAxCTL0	UCPEN	UCPAR	UCMSB	UC7BIT	UCSPB	UCMODEx		UCSYNC
	0	0	0	0	0	0		0
UCAxCTL1	UCSSEL		UCRXEIE	UCBRKIE	UCDORM	UCTXADDR	UCTXBRK	UCWRST
	0		0	0	0	0	0	1
UCAxMCTL	UCBRFx				UCBRSx			UCOS16
	0				0			0
UCAxSTAT	UCLISTEN	UCFE	UCOE	UCPE	UCBRK	UCRXERR	UCADDR/ UCIDLE	UCBUSY
	0	0	0	0	0	0	0	0
UCAxABCTL	-	-	UCDELIMx		UCSTOE	UCBTOE	-	UCABDEN
	0	0	0		0	0	0	0
UCAxIE	-	-	-	-	-	-	UCTXIE	UCRXIE
	0	0	0	0	0	0	0	0
UCAxIFG	-	-	-	-	-	-	UCTXIFG	UCRXIFG
	0	0	0	0	0	0	1	0
UCAxRXBUF								
UCAxTXBUF								

Registros de 16 bits

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
--	----	----	----	----	----	----	---	---	---	---	---	---	---	---	---	---

UCAxBRW	Divisor para gerar o <i>Baud-rate</i> 0
UCAxIV	Vetores de interrupção 0

O leitor é fortemente recomendado a estudar o restante deste capítulo. Existem muitos outros recursos que não estiveram presentes nesta rápida abordagem.

8.1. Fundamentos sobre Comunicação Serial

Um conceito básico na comunicação serial é a distinção entre comunicação síncrona e comunicação assíncrona. Na comunicação serial síncrona, como o nome indica, existe um sinal que marca o instante em que cada bit é disponibilizado no canal serial (é o caso do SPI e do I²C). Esse sinal recebe o nome de relógio (*clock*). Podemos pensar num cabo bem simples que faz uso de três fios, um para o canal de dados serial, outro para o relógio e o terceiro, denominado terra, para a referência das tensões. Pode-se imaginar a que as linhas sejam bidirecionais, sendo o transmissor responsável por gerar os dados seriais e o relógio.

A Figura 8.1 apresenta um exemplo de transmissão serial síncrona, onde são transmitidos os bytes 0x97 (1001 0111 binário) e 0xB6 (1011 0110 binário). Note que os bits são enviados na ordem inversa da que se usa para escrevê-los, ou seja, transmite-se primeiro o bit menos significativo e por último o bit mais significativo. A validade de cada bit, no caso deste exemplo, foi marcada pelo flanco positivo do relógio. Podem-se imaginar diversas outras formas de validação do bit de dado, por exemplo, ele pode ser validado no flanco negativo, ou quando o relógio permanecer em nível alto, ou ainda, enquanto o relógio permanecer em nível baixo. O mais comum é empregar um dos flancos do relógio.

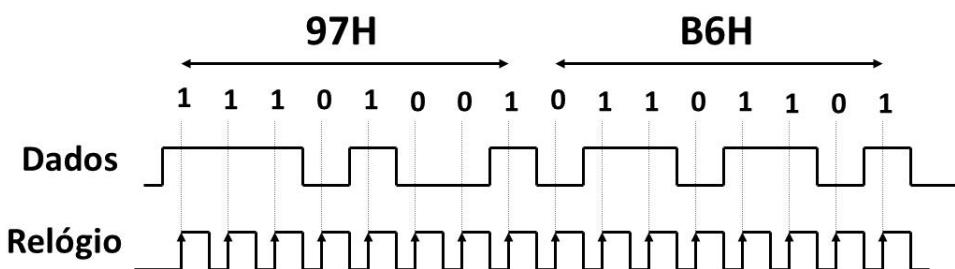


Figura 8.1. Exemplo de uma transmissão serial síncrona.

Já na comunicação serial assíncrona, não existe o relógio para validar os bits de dados. Na sua forma mais simples são necessários apenas três condutores: um para transmissão, outro para recepção e um terceiro para ser a referência de terra. Como não existe relógio, é preciso que, antes de se iniciar a comunicação, se saiba quantos bits serão transmitidos por segundo, pois isto define a janela de tempo na qual o transmissor envia (e o receptor recebe) cada bit. Por exemplo, uma velocidade de 9.600 bauds significa que são transmitidos 9.600 bits por segundo, assim cada bit é apresentado durante uma janela de tempo de $1/9.600$ segundo ($104,17 \mu s$). O nome usado é *baud-rate*, que abreviaremos por BR, já que a unidade de medida é o **baud** (quantidade de mudanças na linha de transmissão). É costume usar as expressões “taxa de bits” ou “velocidade de transmissão”. São comuns as taxas de 9.600 bauds, 19.200 bauds, 28.800 bauds etc. Para que o receptor possa reconhecer o início de uma transmissão usa-se um bit especial denominado “bit de partida” e o final é indicado com um ou outro bit especial denominado “bit de parada”. A Figura 8.2 apresenta o exemplo de uma transmissão serial assíncrona.

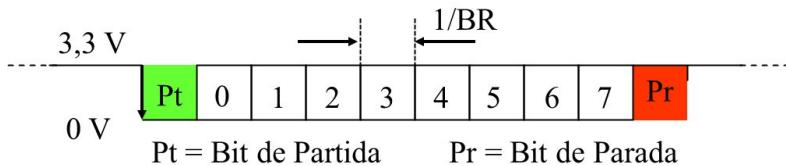


Figura 8.2. Exemplo de uma transmissão serial assíncrona.

É importante olharmos com um pouco de atenção para entendermos o que significam os bits de partida e parada. Note que, como mostrado na Figura 8.2, a comunicação inativa é indicada pela linha em nível alto (3,3 V no caso do MSP430). Quando o transmissor vai iniciar o envio, ele coloca a linha em nível baixo, por um intervalo de tempo igual a $1/BR$. Este é o bit de partida! Logo em seguida, o transmissor coloca a linha em nível alto ou nível baixo, de acordo com os valores dos bits a serem transmitidos, em intervalos de tempo iguais a $1/BR$. Note que o bit menos significativo, o bit 0, é transmitido primeiro e, em consequência, o bit mais significativo, o bit 7, é transmitido por último.

Após transmitir o último bit, o transmissor deixa a linha em nível alto por um intervalo de tempo igual a $1/BR$. Este é o bit de parada! Note que o bit de parada é igual à linha inativa. Essa é a finalidade do bit de parada: garantir que a linha fique algum tempo no estado inativo, para assim assegurar que o próximo bit de partida seja percebido. Vemos então que os bits de partida e parada, em verdade, nada têm de especial. A figura 8.3 apresenta o exemplo da transmissão assíncrona dos bytes 97H e B6H em sequência.

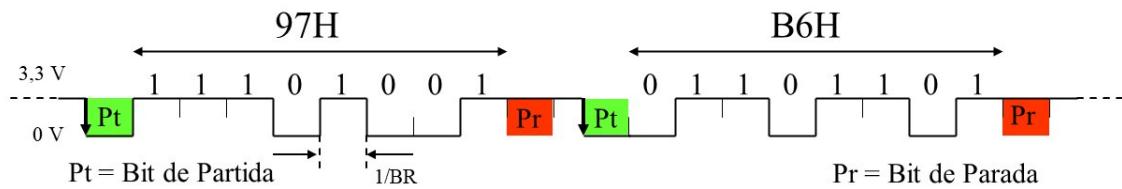


Figura 8.3. Exemplo da transmissão serial assíncrona dos bytes 0x97 e 0xB6.

Quando se trabalha com comunicação serial é importante caracterizar se o canal serial empregado permite transmissão e recepção ao mesmo tempo, ou se permite a transmissão ou a recepção em instantes diferentes. No exemplo apresentado na Figura 8.1 a comunicação precisa ser unidirecional, ou seja, não é possível fazer transmissão e recepção ao mesmo tempo, ficando o transmissor responsável por gerar o relógio. Já no caso da comunicação assíncrona, dado ao fato de se ter duas linhas de dados, é possível transmitir e receber ao mesmo tempo. A Figura 8.4 ilustra esses dois casos.

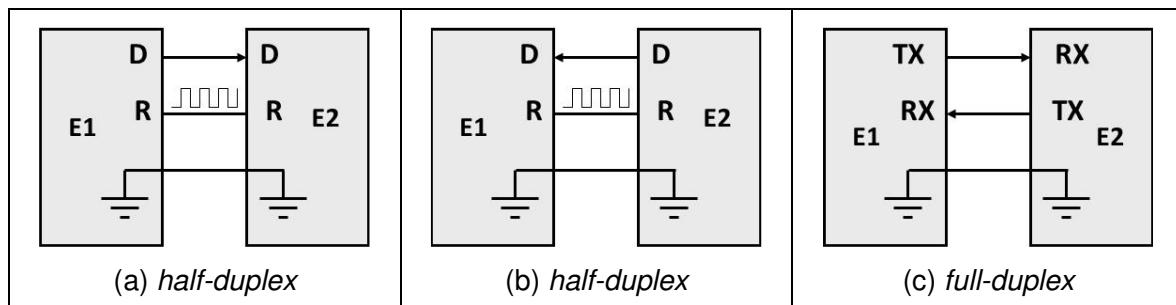


Figura 8.4. Exemplos de comunicação unidirecional (half-duplex) e bidirecional (full-duplex), onde: D = dado, R = relógio, TX = transmissor e RX = receptor.

Note que na Figura 8.4.a, o equipamento E1 transmite para o equipamento E2 através da linha de dados D, ao mesmo tempo em que envia o relógio pela linha R. A Figura 8.4.b apresenta a transmissão no sentido inverso. Já na Figura 8.4.c, ambos equipamentos transmitem e recebem ao mesmo tempo. Quando o canal de comunicação, apesar de ser bidirecional, só pode ser usado em um sentido por vez recebe o nome de *half-duplex*. Já quando o canal permite simultaneidade na transmissão e recepção, recebe o nome *full-duplex*. Para tornar completo o conceito, falta abordar o termo *simplex* que é usado para indicar situações onde existe apenas um transmissor e um receptor, ou seja, o canal de comunicação é usado num único sentido apenas.

8.1.1. Imprecisão no Baud-rate

Como vimos na Figura 8.2, o *baud-rate* define a janela de tempo durante a qual cada bit está disponível. O bit de Partida permite que o receptor sincronize sua recepção. Assim, vemos que há uma sincronização implícita entre o transmissor e o receptor, que é refeita a cada bit de Partida. Essa sincronização é mantida durante toda a duração do trem de bits.

Tanto o transmissor como o receptor trabalham com o *baud-rate* a partir de seus relógios internos. Por isso, a comunicação entre dois MSPs não deve enfrentar dificuldades no ajuste do *baud-rate*, já que o *hardware* dos relógios é semelhante. Os problemas surgem quando são envolvidos dispositivos diferentes. Por exemplo, o caso da conexão UART de um MSP com um Arduino ou com um Raspberry Pi. Vamos então estudar os problemas que surgem devido a imprecisão no *baud-rate*.

Iniciamos com a ótica do receptor para a leitura de um bit. Supondo que conhecemos a janela de tempo deste bit, perguntamos em que instante ele deve ser lido (amostrado): no início da janela de tempo ou no final, ou ainda, na metade? É óbvio que os momentos próximos ao início e fim da janela de tempo devem ser “perigosos”, provavelmente sujeitos a transientes. Assim, é de certa forma óbvio que, a leitura do bit deve acontecer na metade da janela de tempo.

Será que apenas uma leitura é suficiente? Bem, se o bit está estável durante boa parte da janela, por que não melhorar a confiabilidade e fazer várias leituras antes de decidir se o bit recebido é 1 ou 0? O mais comum é realizar 3 leituras e decidir pela maioria. Isso garante uma boa imunidade a ruídos.

A Figura 8.5 apresenta dois exemplos de leituras (amostragens) dentro da janela de tempo. Na porção da esquerda, a janela de tempo é dividida em 4 partes que indicam o momento de cada leitura. Neste caso, temos as 3 leituras “bem” espalhadas dentro da janela de tempo, mas ainda com uma guarda de 25% para as extremidades. É uma solução bastante robusta. Na porção da direita desta figura, a janela foi dividida em 14 intervalos, sendo que as 3 leituras foram feitas na marca central e nas duas vizinhas. Esta solução se afasta do perigo das extremidades, mas tem os instantes de leitura bem próximos, o que indica uma maior vulnerabilidade ao ruído. É importante ver que, à medida em que se aumenta a quantidade de intervalos, maior será essa vulnerabilidade ao ruído.

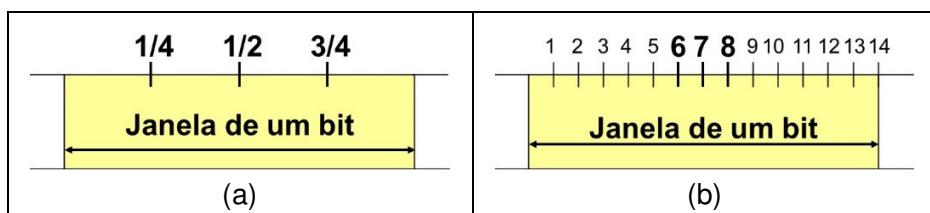


Figura 8.5. Duas soluções para as 3 leituras usadas para estimar o valor do bit recebido.

Vamos agora mostrar o efeito danoso de uma imprecisão no *baud-rate*. Tomemos como exemplo um caso que foi muito comum no passado. Era o problema de comunicação serial entre um PC e o microcontrolador 8051 acionado por um cristal de 12 MHz. Digamos que seja selecionada a taxa de 9.600 bauds (9.600 bps). O PC tinha um oscilador especial e era capaz de gerar 9.600 Hz com grande precisão. Entretanto, o *baud-rate* do 8051 era ditado pela equação abaixo, onde N é a variável inteira que o programador tem liberdade para controlar. O número 384 que aparece no denominador era uma imposição do *hardware* do 8051.

$$N = \frac{CLK}{384 \times BR} = \frac{12 \times 10^6}{384 \times 9600} = 3,25$$

A conta resulta em $N = 3,25$. Porém, é preciso programar um valor inteiro, por isso arredondamos para $N = 3$. Estamos assim cometendo um erro. A taxa gerada pelo 8051 será de 10.416,7 bauds, de acordo com a equação abaixo. Será que sob estas condições os dois dispositivos têm condições de se comunicar? Este caso está ilustrado na figura abaixo.

$$BR = \frac{CLK}{384 \times N} = \frac{12 \times 10^6}{384 \times 3} = 10.416,7 \text{ Bauds}$$

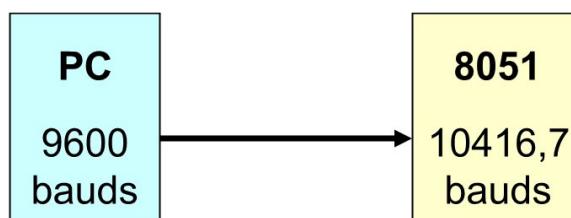


Figura 8.6 Ilustração do caso em que dois dispositivos operam com taxas de bits próximas. Será que a comunicação acontece sem erros?

Podemos pensar que o erro na taxa de bits é pequeno. Ele é de 8,5% acima do planejado ($10.416 / 9.600$). Seria esse erro grande o suficiente para inviabilizar a comunicação? O problema está na sincronização implícita feita pelo bit de partida. Essa sincronização perdura por todo o período de recepção do dado. Isto faz com que o erro na janela de tempo de cada bit se acumule, resultando num valor final que pode ser grande.

Vamos fazer algumas contas:

- 9.600 bauds → janela de um bit é de 104,17 µs e
- 10.416,7 bauds → janela de um bit é de 96 µs

Vemos que a diferença entre as janelas de tempo de cada bit é de $8,17 \mu s$ (erro perto de 9%). Sim, essa diferença é pequena para um bit, mas se acumula e ao final de 10 bits teremos uma diferença acumulada igual a $81,7 \mu s$, o que obviamente é muito grande, quase equivalente à janela de um bit. A conclusão é que sob essas condições os dois dispositivos não conseguem se comunicar. A solução adotada na época era a de trocar o cristal do 8051 para um com a frequência igual a 11,0592 MHz ($9600 \times 3 \times 384$).

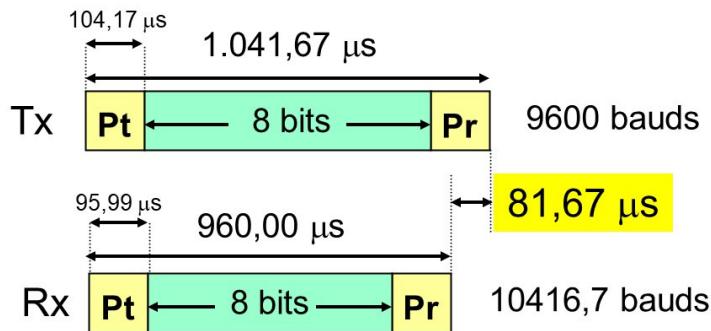


Figura 8.7. Ilustração de dois dispositivos UART tentando de comunicar com uma pequena diferença no baud-rate.

Vem agora pergunta: até quanto podemos tolerar de erro no *baud-rate*? Vamos considerar o frame típico que tem 10 bits (bit partida, 8 bits de dados e bit de parada). A resposta mais segura é a de que o erro acumulado após esses 10 bits deve ser menor que 25% da janela de tempo de um bit, vide Figura 8.5.a. Vamos então considerar BR como sendo a taxa desejada e BRx a taxa que conseguimos gerar. Chegamos então à inequação abaixo.

$$10 \left| \frac{1}{BR} - \frac{1}{BRx} \right| < 0,25 \frac{1}{BR}$$

Trabalhando um pouco esta inequação chegamos à solução final abaixo.

$$0,975 < \frac{BR}{BRx} < 1,025$$

Assim, para o caso de 9.600 bauds, os limites são 9.360 e 9.840 bauds. Os mais otimistas, aqueles que se consideram com sorte, podem estender essa diferença acumulada em 10 bits menor que 50%. O leitor é convidado a refazer os cálculos para esta condição.

8.2. A Unidades Seriais (USCI) do MSP430

Como existe uma certa semelhança entre as formas seriais de comunicação (UART, SPI, I²C etc), o projeto do MSP430 traz unidades seriais que podem ser configuradas para os diversos protocolos. Essas unidades são denominadas de Interface de Comunicação Serial Universal e usam a sigla USCI (do inglês, *Universal Serial Communication Interface*).

As USCI não precisam ser idênticas. Diferentes USCI oferecem diferentes serviços e são identificadas por uma letra, por exemplo, USCI_A, USCI_B etc. Quando o MSP possui mais de uma instância de cada tipo, elas recebem números para permitir a identificação. No caso do MSP430 F5529 temos: USCI_A0, USCI_A1, USCI_B0 e USCI_B1. A Tabela 8.1 resume os recursos oferecidos por essas unidades, onde se pode ver que apenas a USCI_A opera no modo UART.

Tabela 8.1. Recursos oferecidos pelas diferentes USCI

	USCI_A	USCI_B
UART	X	
IrDA	X	
SPI	X	X
I ² C		X

8.3. Recursos da USCI_Ax no Modo UART

Quando a USCI_Ax opera no modo UART, ela conecta dois dispositivos seriais usando os pinos UCAxRXD (recepção) e UCAxTXD (transmissão). Os recursos, quando neste modo, estão listados a seguir.

- Campo de dados de 7 ou 8 bits;
- Paridade par, ímpar ou sem paridade;
- Registradores de deslocamento independentes;
- *Buffers* de transmissão e recepção independentes;
- Seleção para transmitir primeiro o bit mais significativo ou o menos significativo;
- Protocolos para comunicação multiprocessador;
- Recurso para despertar a CPU com a detecção do bit de partida;
- *Baud-rate* programável, com recursos para a parte fracionária;
- *Flags de status* para indicar erros e
- Interrupções separadas para transmissão e recepção.

A Tabela 8.2 apresenta os pinos dedicados às duas instâncias da USCI_Ax. Para USCI_A0 estão destinados os pinos P3.3 e P3.4. Já para a USCI_B, não existe uma destinação fixa. O programador deve mapear dois pinos da porta P4 para usá-los com a USCI_A1. Lembrar que na barra de terminais da LaunchPad temos apenas os bits P4.0, 1, 2 e 3. Por ocasião da inicialização da LaunchPad esses pinos são mapeados para uso com a UCSI_B1. O programador pode alterar isso. É recomendado, então, que o leitor volte ao tópico 3.5 do Capítulo 3, que trata de GPIO para relembrar detalhes do mapeamento da porta P4.

Tabela 8.2. Distribuição dos pinos entre as unidades USCI_Ax

	USCI_A0	USCI_A1
UCAxTXD	P3.3	Mapear P4.3, P4.2, P4.1 ou P4.0
UCAxRXD	P3.4	Mapear P4.3, P4.2, P4.1 ou P4.0

Tabela 8.3. Constantes para permitir o mapeamento de bits da porta P4 para a USCI_A1

Valor	Mnemônico	Função
11	PM_UCA1RXD	RXD para a UART USCI_A (a direção é controlada pela USCI)
12	PM_UCA1TXD	TXD para a UART USCI_A (a direção é controlada pela USCI)

8.4. Diagrama de Blocos de uma USCI_Ax no Modo UART

A seguir vamos estudar a USCI_Ax quando opera no Modo UART. Iniciamos com o diagrama de blocos que, para facilitar a explicação, foi quebrado nas três partes apresentadas abaixo (Figuras 8.8, 8.9 e 8.10). Como sempre fizemos, nestes diagramas de blocos, as caixas verdes com cantos arredondados são configurações controladas pelo programador e as caixas amarelas indicam bits de sinalização.

8.4.1. Geração do Relógio para Transmissão e Recepção

Na Figura 8.8 temos a porção responsável pela geração dos relógios para a transmissão e para a recepção. Usando um multiplexador, o programador pode selecionar como base (BRCLK) um dos três relógios: ACLK, SMCLK ou UCAxCLK. A entrada UCAxCLK permite

que o usuário use um relógio externo para gerar seu *baud-rate*. Temos UCA0CLK em P2.7 e UCA1CLK pode ser mapeado através de um dos pinos da porta P4.

Uma vez selecionado o BRCLK, o programador tem à sua disposição um divisor e um modulador para gerar os relógios que serão usados na recepção (RxCLK) e na transmissão (TxCLK). Esse conjunto é bastante sofisticado e possibilita grande precisão na geração do *baud-rate*. Mais adiante estudaremos como isso é feito. **O controle UCABEN é fonte de dúvida. Ele parece ser um controle interno, não acessível ao programador. O manual apresenta este controle em suas figuras, entretanto o texto não faz qualquer referência explicativa.**

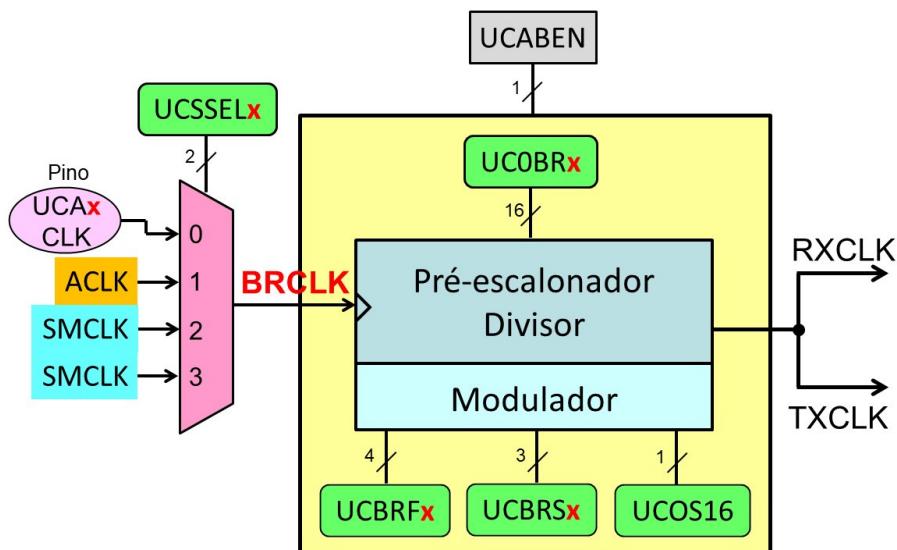


Figura 8.8. Diagrama de blocos da porção responsável pela geração dos relógios para transmissão e recepção ($x = 0$ ou 1).

8.4.2. Módulo Receptor

A Figura 8.9 apresenta o diagrama de blocos da porção responsável pela recepção. Iniciamos a explicação pelo canto inferior direito, onde se pode ver o pino UCAxRXD que é a entrada por onde chegam os bits serials. Note que o multiplexador logo a seguir é controlador pelo bit UCLISTEN. Se UCLISTEN = 0, então a recepção é feita pelo pino UCAxRXD. Por outro lado, se UCLISTEN = 1, o que é passado adiante é o sinal de transmissão serial (UCATXRXD), ou seja, se recebe o que foi transmitido (é um recurso para se “escutar” a transmissão, ou *loop back*). Isto é interessante para durante a fase de depuração dos programas.

O bit UCIREN permite passar o sinal recebido através do decodificador IrDA (infra vermelho) ou entregá-lo diretamente para o registrador de deslocamento. O IrDA não será

abordado neste estudo, ou seja, sempre usaremos UCIREN = 0. À medida que os bits vão chegando (na verdade, seguindo a janela de tempo ditada pelo *baud-rate*), de acordo com a configuração, eles vão sendo deslocados para dentro do Registrador de Deslocamento. Quando todos os bits são recebidos, o dado é transferido para o *buffer* de recepção (UCAxRXBUF) e já se pode iniciar uma nova recepção. O programador deve ler o dado do *buffer* de recepção antes que o próximo dado seja recebido. Do contrário, o novo dado irá sobrescrever o anterior, o que caracteriza o erro de Atropelamento (*Overrun*). A *flag* UCRXIFG, marcada com um contorno mais grosso, é muito importante. Ela vai a 1 quando o dado recebido é transferido para o UCAxRXBUF. Esta *flag* é zerada por ocasião da leitura do *buffer* de recepção.

A Máquina de Estados de RX, de acordo com a configuração, é responsável por acompanhar os estados da recepção e acionar *flags* para sinalizar os erros e as informações de *status*.

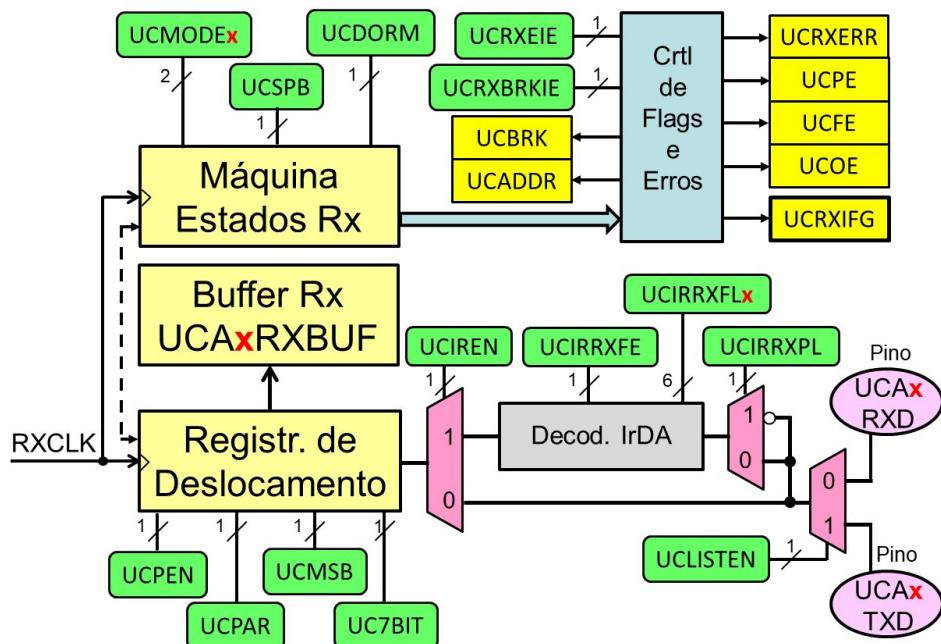


Figura 8.9. Diagrama de blocos da parte responsável pela recepção ($x = 0$ ou 1).

8.4.3. Módulo Transmissor

A Figura 8.10 apresenta o diagrama de blocos da porção responsável pela transmissão serial. O usuário escreve no *buffer* UCAxTXBUF o dado a ser transmitido. Esse dado é copiado para o registrador de deslocamento que inicia sua transmissão. Toda vez que o *buffer* UCAxTXBUF fica vazio, a *flag* UCTXIFG vai para 1. Ela é muito importante e está marcada com um contorno mais grosso. Essa *flag* é automaticamente apagada quando o

usuário escreve no UCA_xTXBUF. A Máquina de Estados, de acordo com a configuração é responsável por controlar o Registrador de Deslocamento de forma a que os bits sejam enviados corretamente. Como já foi dito no tópico anterior, neste estudo o bit UCIREN vai estar sempre em 0, ou seja, não é usado o Codificador IrDA (infra vermelho).

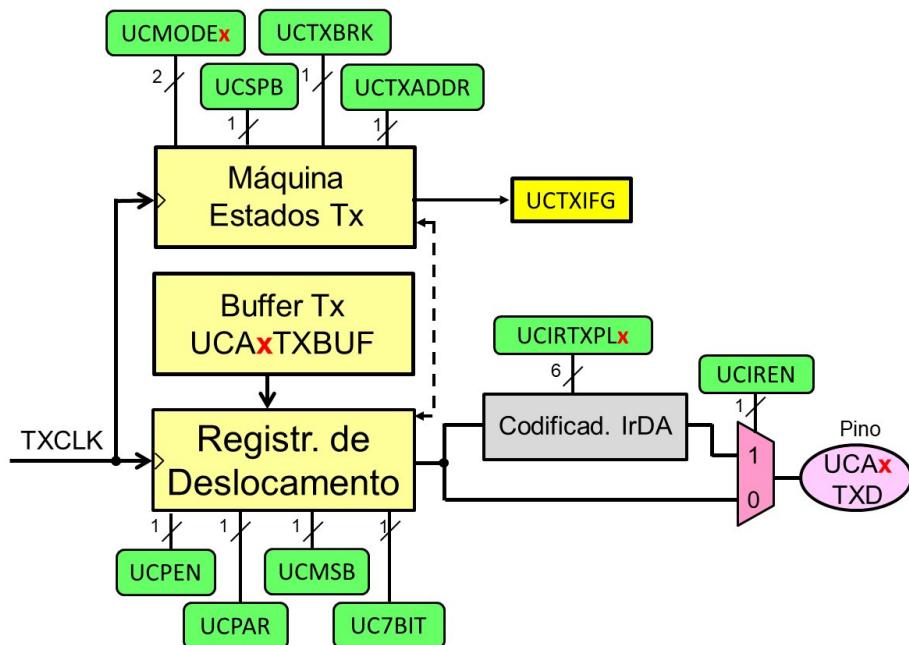


Figura 8.10. Diagrama de blocos da parte responsável pela transmissão ($x = 0$ ou 1).

8.5. Configuração das USCI_Ax do MSP430

A inicialização de uma UART envolve as seguintes definições:

- *Baud-rate*;
- Quantidade de bits de dados;
- Uso de paridade, caso positivo, par ou ímpar;
- Quantidade de bits de parada e
- Algum outro recurso especial.

No caso do MSP430, esses mesmos passos devem ser cumpridos para inicializar a USCI_Ax no modo UART. A USCI_Ax entra na condição de *reset* quando o programa ativa o bit UCSWRST. Esse bit é também ativado pelo PUC (*Power-up Clear*) quando o MSP é energizado. Então, após o MSP ser ligado, a USCI_Ax está em *reset*.

É importante citar que a USCI_Ax só pode ser configurada com UCSWRST ativado e os seguintes passos devem ser usados para a configuração:

1. Ativar a condição de *reset*, ou seja, fazer UCSWRST = 1;
2. Inicializar os registradores, incluindo UCAXCTL1;
3. Configurar as portas de I/O usadas;
4. Finalizar a condição de *reset*, ou seja, fazer UCSWRST = 0 e
5. Se for o caso, habilitar as interrupções (UCRXIE e UCTXIE).

8.5.1. Os Moduladores de 8 e 16 bits

Como vimos no tópico 8.1.1, a programação da taxa de transmissão (*baud-rate*) é muito importante e, se não for bem feita, pode inviabilizar toda a comunicação ou resultar em comunicação com dados equivocados. O MSP430 possui recursos para gerar *baud-rate* com uma boa precisão. Isso é conseguido com o uso de divisores (*prescalers*) e de moduladores.

O típico caso para gerar um *baud-rate* envolve a divisão (por um número inteiro) de um relógio com frequência elevada para se chegar à frequência desejada. É comum que o resultado dessa divisão seja arredondado para o inteiro mais próximo, e assim, a parte fracionária fica desprezada. Com o modulador, poderemos ajustar também essa parte fracionária. Vamos a um caso típico com o MSP. Precisamos trabalhar a 9.600 bauds e temos disponível o ACLK (32.786 Hz). Fazemos então a conta mostrada abaixo para calcular o valor do divisor N.

$$N = \frac{32.786}{9.600} = 3,41$$

Como somos forçados a usar um valor inteiro e para cometer um menor erro, arredondamos para baixo ($N = 3$), e acabamos por usar o *baud-rate* de 10.922,7 bauds. Pelo que já estudamos, nessas condições nossa comunicação serial não vai funcionar. O modulador permite ajustar (um pouco) esta parte fracionária de 0,41 e com isso conseguimos chegar bem mais perto da taxa desejada.

São dois moduladores, um de 8 bits e outro de 16 bits. O modulador, simplesmente adiciona um período extra em determinadas porções do sinal de saída. Esse período extra tem como referência o sinal original. Para facilitar a explicação e deixarmos o conceito bem claro, vamos inventar um Modulador de 4 bits. A tabela abaixo apresenta o padrão para esse hipotético Modulador de 4 bits. Nessa tabela, as posições marcadas com 1 indicam que o modulador aumenta da duração do sinal de saída em um período do relógio original, que é o BRCLK.

*Tabela 8.4. Padrão de modulação do Modulador de 4 bits, aqui inventado.
(Este modulador não existe no MSP430, sua finalidade é puramente didática)*

MD	Bit 0	Bit 1	Bit 2	Bit 3	Fração
----	-------	-------	-------	-------	--------

0	0	0	0	0	0	0
1	0	1	0	0	1/4	0,25
2	0	1	0	1	2/4	0,5
3	0	1	1	1	3/4	0,75

A Figura 8.11 ilustra as 4 opções desse modulador. Na parte superior da figura temos o BRCLK, que é o relógio a partir do qual será feita a divisão por 3, como calculamos acima. O sinal indicado por f_0 na figura, corresponde ao sinal BRCLK dividido por 3, exatamente. A parte em azul abrange 4 períodos de f_0 , já que nosso modulador é de 4 bits. Note que com $MD = 0$, nenhuma alteração é feita e 12 períodos de BRCLK correspondem a 4 períodos de f_0 . A relação é de 3,0 (12 / 4).

Já o sinal f_1 é criado com o modulador em 1 ($MD = 1$). De acordo com a tabela, no intervalo correspondente ao bit 1 é inserido um período extra de BRCLK. Este período extra está desenhado em vermelho. Note que a faixa azul ficou um pouco maior. Agora, a relação entre os dois sinais é de 3,25 (13 / 4).

O sinal f_2 é criado com o modulador em 2 ($MD = 2$). De acordo com a tabela, os bits 1 e 3 têm um período extra de BRCLK. Estes períodos extras estão em vermelho. Note que a faixa azul aumentou ainda mais. Agora, a relação entre os dois sinais é de 3,50 (14 / 4).

Para terminar, o sinal f_3 é criado com o modulador em 3 ($MD = 3$). De acordo com a tabela, os bits 1, 2 e 3 têm um período extra de BRCLK. Estes períodos extras estão em vermelho. Note que a faixa azul é bem maior. Agora, a relação entre os dois sinais é de 3,75 (14 / 4).

A conclusão é que podemos selecionar o divisor para 3,00; 3,25; 3,50 ou 3,75. No caso de nosso problema, selecionamos $MD = 2$ e o divisor passa a ser de 3,50 que está muito mais próximo do desejado 3,41. Temos então:

$$BR = \frac{32.768}{3,50} = 9.362,3 \text{ Hz}$$

Será que essa frequência está próxima o suficiente para a UART funcionar corretamente? Usando o que foi exposto em 8.1.1, o leitor é convidado a verificar e concluir que deve funcionar, apesar de estar próximo do que especificamos como limite de segurança.

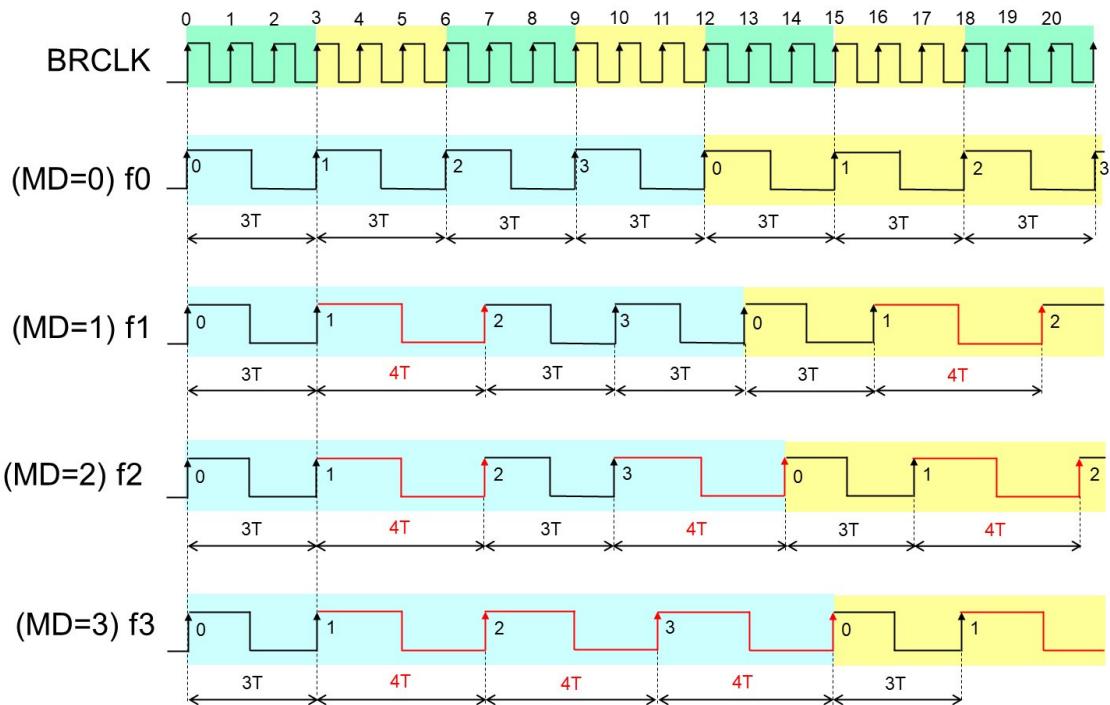


Figura 8.11. Diagrama de tempo expondo as 4 possibilidades de escolha para o hipotético Modulador de 4 bits.

Voltando ao MSP, temos dois moduladores: um de 8 e outro de 16 bits. A tabela do modulador de 8 bits está apresentada abaixo. A configuração desse modulador é feita com o campo de 3 bits denominado de UCBRSx. A tabela ainda apresenta, na última coluna, a fração que é possível aproximar. Uma preocupação nesses modulares é a distribuição desses períodos extras ao longo dos 8 períodos. Note a distribuição dos "1s" da tabela.

Tabela 8.4. Padrão para o Modulador de 8 bits

UCBRSx	Bit 0 Start	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Fração
0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	1/8
2	0	1	0	0	0	1	0	0	2/8
3	0	1	0	1	0	1	0	0	3/8
4	0	1	0	1	0	1	0	1	4/8
5	0	1	1	1	0	1	0	1	5/8
6	0	1	1	1	0	1	1	1	6/8

7	0	1	1	1	1	1	1	1	7/8
---	---	---	---	---	---	---	---	---	-----

Para indicar uma forma simples de se determinar qual o melhor valor para UCBRSx, voltemos ao problema original: gerar 9.600 Hz a partir de 32.768 Hz. Já vimos que o resultado da divisão foi de 3,41. Então para escolher a melhor opção para aproximar esse 0,41, multiplicamos ele por 8 ($8 \times 0,41 = 3,28$). Arredondamos para 3 e usamos UCBRSx = 3. O programador ainda deve verificar as duas possibilidades (3 e 4) para checar se era realmente o melhor resultado (há incertezas na detecção do bit de partida). De acordo com as contas apresentadas abaixo, a primeira solução é a melhor e fica dentro da margem indicada no Tópico 8.1.1.

$$BR = \frac{32.768}{3 + 3/8} = 9.709,0 \text{ Hz} \rightarrow \text{erro} = 109,0 \text{ Hz}$$

$$BR = \frac{32.768}{3 + 4/8} = 9.362,3 \text{ Hz} \rightarrow \text{erro} = 237,7 \text{ Hz}$$

A tabela do modulador de 16 bits está apresentada abaixo. Seu uso é semelhante à tabela do modulador de 8 bits.

Tabela 8.5. Padrão para o Modulador de 16 bits

UCBRFx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Fração
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1/16	
2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2/16	
3	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	3/16	
4	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	4/16
5	0	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	5/16
6	0	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1	6/16
7	0	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	7/16
8	0	1	1	1	1	0	0	0	0	0	0	0	1	1	1	1	8/16
9	0	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1	9/16
10	0	1	1	1	1	1	0	0	0	0	0	1	1	1	1	1	10/16
11	0	1	1	1	1	1	1	0	0	0	0	1	1	1	1	1	11/16
12	0	1	1	1	1	1	1	0	0	0	1	1	1	1	1	1	12/16
13	0	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1	13/16
14	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	14/16
15	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	15/16

8.5.2. Geração de *Baud-rate*

O MSP430 possui recursos para gerar *baud-rate* nos valores mais comuns (EIA RS-232) 600, 1200, 2400, 4800, 9600, 19200, 38400 bauds a partir de seus relógios internos. Como já dissemos, isso é conseguido com o uso de divisores (*prescalers*) e de moduladores.

O usuário deve escolher entre dois modos de geração de *baud-rate*, selecionados pelo bit UCOS16. Mais adiante veremos em qual registrador está este bit.

- UCOS16 = 0: Modo Baixa Frequência e
- UCOS16 = 1: Modo Super Amostragem (*oversampling*).

(UCOS16 = 0) Geração de *Baud-rate* Frequência Baixa

O Modo Baixa Frequência (UCOS16 = 0) é usado quando o relógio base (BRCLK) e o *baud-rate* desejado (BITCLK) estão próximos. A relação entre os dois deve ser de, pelo menos, 3. Se a relação chegar a 32, é melhor mudar para o Modo Super Amostragem.

$$32 > \frac{BRCLK}{BITCLK} \geq 3$$

Observação: com a relação BRCLK/BITCLK maior que 16, já podemos usar o Modo Super Amostragem, porém o resultado não é bom. Assim, preferimos indicar o limite como sendo 32.

É importante lembrar que quanto menor forem as frequências envolvidas numa comunicação serial, menor será o consumo de energia. A Figura 8.12 apresenta um diagrama de bloco. A parte inteira da divisão apresentada acima é programada em (UCBRx) e se faz o ajuste da parte fracionária no Modulador de 8 bits (UCBRSx), como descrito no algoritmo a seguir.

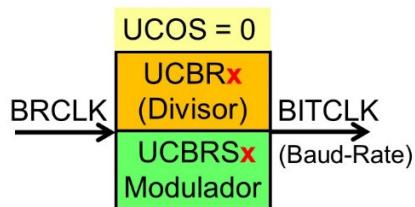


Figura 8.12. Esquema para geração de *baud-rate* no Modo de Baixa Frequência.

Algoritmo para configurar *baud-rate* no Modo Baixa Frequência (UCOS16 = 0)

7. Calcular $n = BRCLK/BITCLK$; // É preciso $n \geq 3$
8. $N = INT(n)$; // N = parte inteira de n
9. $UCBRx = N$; // Programar divisor, $UCBRx = N$
10. $m8 = (n - N) \times 8$; // Calcular valor para o Modulador 8
11. $M8 = round(m8)$; // Arredondar
12. $UCBRSx = M8$; // Programar Modulador 8, $UCBRSx = M8$

No Manual do Usuário do MSP430 traz a Tabela 36-4 (pag 951) onde já estão calculados os valores a serem programados nos diversos registradores e os piores casos de erros de transmissão e recepção, considerando a janela de um bit. Quando fazemos o cálculo, o manual ainda recomenda que se verifique se o incremento ou decremento de USCRSx não gera um menor valor erro máximo, quando se analisam todos os bits. Este detalhe não será levado em consideração no presente texto.

Observação: Veja que na Tabela 36-4 o pior caso é quando se opera em 9.600 bauds usando o ACLK (32.768 Hz). Os autores puderam constatar na prática que esta configuração gera erros. Por isso, não recomendamos seu uso.

Exemplo 8.1: Indique a configuração ($UCBRx$ e $UCBRSx$) para operar em 4.800 bps empregando o ACLK. Qual o valor exato da frequência usada? Tomando como referência a janela de tempo de um bit, qual o erro percentual acumulado quando se considera um campo de dados de 8 bits, incluindo os bits de Partida e Parada?

Solução:

- $n = \frac{32.768}{4.800} = 6,826 \rightarrow N = 6$ (maior que 3).
- $M8 = round(0,826 \times 8) = round(6,64) = 7 \rightarrow M8 = 7$.
- Programamos $UCBR = 6$ e $UCBRS = 7$.
- $BR = \frac{32.768}{6+\frac{7}{8}} \rightarrow BR = 4.766,25 \text{ Hz}$.

Cálculo do período de um bit:

- $T_{4.800} = \frac{1}{4.800} = 208,333 \mu\text{s}$.
- $T_{4.766} = \frac{1}{4.766,25} = 209,809 \mu\text{s}$.

O período de um bit ficou em 209,809 μs , quando deveria ser de 208,333 μs . Isto resulta num erro de 1,476 μs por bit. Em 10 bits temos um erro acumulado de 14,76 μs . Corresponde a 7% ($100\% \times 14,76/208,333$) da janela de tempo de um bit. Vemos então que o ajuste está muito bom.

Exemplo 8.2: Comente sobre a configuração (UCBRx e UCBRSx) para operar em 1.200 bps empregando o ACLK.

Solução:

- $n = \frac{32.768}{1.200} = 27,306 \rightarrow N = 27$ já é um valor alto. Mais adiante será explicado o porquê. A melhor solução seria passar para o modo alta frequência. Entretanto, podemos continuar.
- $M8 = round(0,306 \times 8) = round(2,45) = 2 \rightarrow M8 = 2$.
- Programamos UCBRx = 27 e UCBRSx = 2.
- $BR = \frac{32.768}{27 + \frac{2}{8}} = 6,826 \rightarrow BR = 1.202,5 \text{ Hz}$.

Dúvida: Como definir um limite para o valor do divisor UCBRx? Será que 27 já não é grande e traz vulnerabilidade ao ruído? O manual apresenta valor de até 2.083 (20 MHz para gerar 9.600)

Exemplo 8.3: Comente sobre a configuração (UCBRx e UCBRSx) para operar em 19.200 bps empregando o ACLK.

Solução:

- $n = \frac{32.768}{19.200} = 1,706 \rightarrow N < 3$. Não funciona! Selecionar SMCLK para relógio base (BRCLK).

(UCOS16 = 1) Geração de *Baud-rate* em Super Amostragem

O Modo Super Amostragem (UCOS16 = 1) é usado quando a relação entre o relógio base (BRCLK) e o *baud-rate* desejado (BITCLK) está acima de 16.

$$\frac{BRCLK}{BITCLK} \geq 16$$

Observação: com a relação BRCLK / BITCLK maior que 16, já podemos usar o Modo Super Amostragem, porém o resultado próximo de 16, às vezes, não é bom. Por exemplo, trabalhar com 38.400 bauds usando SMCLK (1.048.576 Hz) e UCOS16 = 1. A prática não mostrou um bom resultado. Verificar. Essa configuração não aparece na Tabela 36.5 do MSP User's Guide

Como mostrado na Figura 8.13, este modo usa um divisor (UCBRx) e um modulador de 16 bits (UCBRFx) para gerar um sinal intermediário, denominado de BITCLK16, que é

aproximadamente 16 vezes o BITCLK desejado. Depois, um segundo estágio com um divisor fixo em 16 e um modulador de 8 bits (UCBRSx), permite um segundo ajuste. Se o registrador UCBRx é configurado em 0 ou 1, então BITCLK16 = BRCLK e o modulador de 16 não realiza qualquer ação.

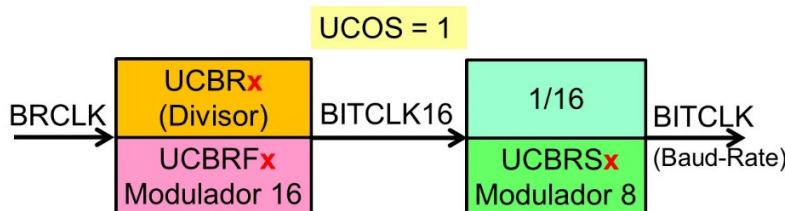


Figura 8.13. Esquema para geração de baud-rate no Modo de Super Amostragem.

Algoritmo para configurar *baud-rate* no Modo Super Amostragem (UCOS16 = 1)

13. Calcular $\text{BITCLK16} = 16 \times \text{BITCLK}$; //Calcular sinal intermediário
14. Calcular $n = \text{BRCLK}/\text{BITCLK16}$; //
15. $N = \text{INT}(n)$; // N = parte inteira de n
16. $\text{UCBRx} = N$; // Programar divisor, $\text{UCBRx} = N$
17. $m16 = (n - N) \times 16$; // Calcular valor para o Modulador 16
18. $M16 = \text{round}(m16)$; // Arredondar (ou menor inteiro mais próximo)
19. $\text{UCBRFx} = M16$; // Programar Modulador 16
20. $\text{BITCLK16} = \text{BRCLK}/(N+M/16)$; // Recalcular BITCLK16
21. $n = \text{BITCLK16}/\text{BITCLK}$; // Calcular novo n
22. $m8 = (n-16) \times 8$; // Calcular valor para o Modulador 8
23. $M8 = \text{round}(m8)$; // Arredondar
24. $\text{UCBRSx} = M8$ // Programar Modulador 8

No Manual do Usuário do MSP430 está disponível a Tabela 36-5 (pag 953) que lista os valores a serem programados nos diversos registradores e os piores casos de erros de transmissão e recepção, considerando a janela de um bit. O manual do MSP430 recomenda que se verifique se o incremento ou decremento de USCRFx não gera um menor valor erro máximo, considerando UCBRSx excursionando de 0 até 7. Isto não será feito no presente texto.

Exemplo 8.4: Indique a configuração (UCBRx, UCBRFx e UCBRSx) para operar em 9.600 bps empregando o SMCLK. Qual o valor exato da frequência gerada? Tomando como referência a janela de tempo de um bit, qual o erro percentual acumulado quando se considera um campo de dados de 8 bits, incluindo os bits de partida e parada?

Solução:

- $\text{BITCLK16} = 16 \times \text{BITCLK} = 16 \times 9.600 = 153.600 \text{ Hz}$.

- $n = \frac{1.048.576}{153.600} = 6,827 \rightarrow N = 6.$
- $M16 = round(0,827 \times 16) = round(13,2) = 13 \rightarrow M16 = 13.$
- Programamos UCBRx = 6 e UCBRFx = 13.
- $BITCLK16 = \frac{1.048.576}{6+\frac{13}{16}} = 153.919,4 \text{ Hz}.$
- $n = \frac{153.919,4}{9.600} = 16,033.$
- $M8 = round(0,033 \times 8) = round(0,264) = 0 \rightarrow M8 = 0.$
- Programamos UCBRSx = 0.
- $BR = \frac{153.919,4}{16+\frac{9}{8}} = 9.619,96 \rightarrow BR = 9.620 \text{ Hz}.$

Cálculo do período de um bit:

- $T_{9.600} = \frac{1}{9.600} = 104,17 \mu\text{s}.$
- $T_{9.619} = \frac{1}{9.619,96} = 103,95 \mu\text{s}.$

O período de um bit ficou em 103,95 μs , quando deveria ser de 104,17 μs , o que resulta num erro de 0,217 μs por bit. Em 10 bits temos um erro acumulado de 2,17 μs . Corresponde a 2 % ($100\% \times 2,17/104,17$) da janela de tempo de um bit. O ajuste está muito bom!

Observação: O Manual do Usuário do MSP430, no item 36.3.10.2 (pag. 949) especifica o arredondamento no cálculo de M16 (UCBRFx), entretanto, acreditamos que deve ser usada apenas a parte inteira, sem qualquer aproximação. Isto porque, quando o arredondamento é para o inteiro acima, a relação entre BITCLK16 / BITCLK fica abaixo de 16, o que dificulta o uso do Modulador de 8 bits (UCBRSx).

Considerações sobre a Geração de *Baud-rate*

O leitor irá constatar que, algumas vezes, seu cálculo de UCBRx, UCBRFx e UCBRSx não corresponde aos valores apresentados nas tabelas 36-4 e 36-5 do manual Guia do Usuário do MSP430. Isso acontece porque as tabelas apresentam o caso que resulta no menor erro considerando todas as opções das janelas de bit.

Para explicar melhor, voltemos ao Exemplo 8.1, onde foi pedido para gerar 9.600 bps no Modo de Baixa Frequência ($UCOS16 = 0$) e tomando como referência o ACLK. Mostramos abaixo um gráfico que apresenta todas possibilidades do Modulador 8 para este exemplo. Como se pode ver, a configuração $UCBRSx = 3$ é a opção que gera a melhor solução, pois é a que está mais próxima de 9.600 Hz. Neste caso, como os demais pontos estão bastante afastados da frequência desejada, pode-se dispensar a verificação do máximo erro considerando todos os bits.

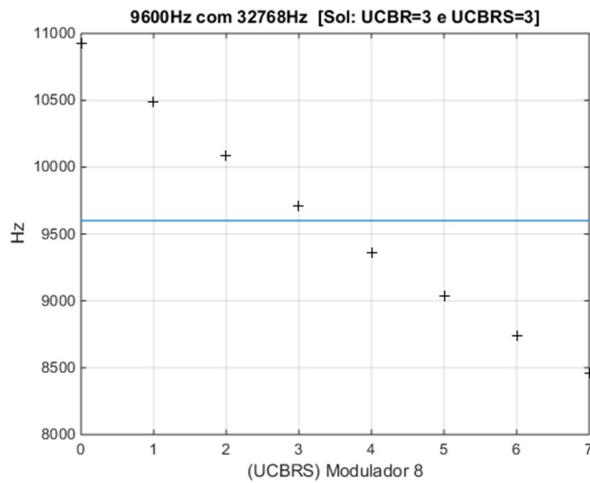


Figura 8.14. Todas as opções do Modulador de 8 bits para o caso de se usar ACLK para gerar 9.600 Hz com $UCOS16 = 0$.

Abaixo está a listagem de um *script Matlab* para gerar o gráfico com as opções do Modulador de 8 bits, como mostrado na figura acima. Ele permite definir o BITCLK desejado e o BRCLK disponível. A linha horizontal em azul sinaliza o BITCLK desejado e os sinais “+” indicam a frequência gerada a partir das 8 opções do modulador.

Script Matlab para calcular $UCBRx$ e $UCBRSx$ e ainda mostrar todas as opções para o Modulador de 8 bits ($UCBRSx = 0, 1, \dots, 7$)

```
% BR - UCOS16=0
% Gerar BITCLK Hz partir de BRCLK Hz

aclk = 2^15;
smclk = 2^20;

% Definir bitclk e brclk
bitclk = 9600; %<<===== alterar esta linha de acordo com o caso
brclk=aclk;      %<<===== alterar esta linha de acordo com o caso
```

```
%brclk=2e7;

% Calcular N e M8
n=brclk/bitclk;
N=fix(n);
m8=n-N;
M8=round(8*m8);

% Desenhar gráfico
MM8=[0:7];
bps=brclk./(N+(MM8./8));
plot(MM8,bps,'+k');
grid;
msg=sprintf('%dHz      com      %dHz      [Sol:      UCBR=%d      e
UCBRS=%d]',bitclk,brclk,N,M8);
title(msg);
xlabel('(UCBRS) Modulador 8');
ylabel('Hz');
line([0 7],[bitclk bitclk]);
msg=sprintf('%d Hz',bitclk);
text(0.5,br,msg);
```

A Figura 8.15 apresenta um caso mais extremo, onde se quer BITCLK = 9.000 Hz e se dispõe de BRCLK na frequência 20×10^6 Hz. O leitor pode constatar que todas as opções do Modulador de 8 bits geram soluções que podem ser usadas. A mais próxima de 9.600 Hz é conseguida com UCBRS_x = 3, será então que deveríamos escolher esta? Entretanto, se o leitor for conferir na Tabela 36-4 do manual, a solução indicada é UCBRS_x = 2, porque esta é a que resulta num menor erro máximo. A figura mostra que para os dois valores vizinhos à linha de 9.600 Hz (M8 = 2 e M8 = 3) o erro na frequência está abaixo de 0,5 Hz. Em tais casos, se faz a escolha pela opção que gera o menor erro máximo. Como já foi afirmado, aqui neste estudo não faremos a análise deste erro.

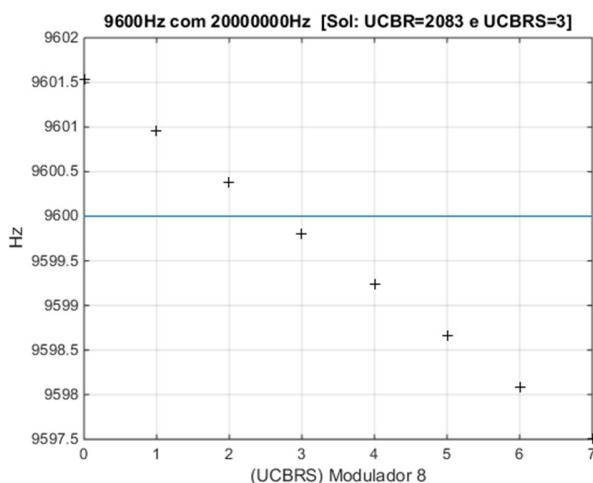


Figura 8.15. Todas as opções do Modulador de 8 bits para o caso de se usar 20×10^6 Hz para gerar 9.600 Hz.

Para o caso de super amostragem ($UCOS16 = 1$) há mais flexibilidade. A Figura 8.16 apresenta todas as possíveis opções para o Exemplo 8.4, que pedia para gerar 9.600 Hz a partir do SMCLK. Para cada configuração do Modulador 16 ($UCBRFx$) temos 8 possibilidades para o Modulador 8 ($UCBRSx$). Nesta figura, cada linha inclinada representa o percurso por essas 8 opções. É possível de ver que existem 6 possibilidades e que o algoritmo usado nos levou para a última opção ($UCBRF = 13$ e $UCBRS = 0$). Não o faremos aqui, porém este seria um bom caso para analisar os piores casos de erros nos bits de transmissão e recepção.

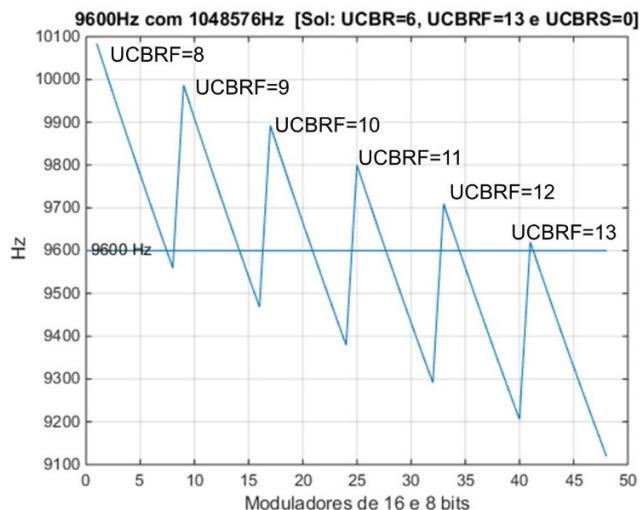


Figura 8.16. Opções do Modulador de 16 bits ($UCBRF$) e todas as opções do Modulador de 8 bits para o caso de se usar SMCLK para gerar 9.600 Hz.

A Figura 8.17 apresenta a mesma Figura 8.16, mas agora as opções válidas estão indicadas com “+”. Nesta figura se pode ver que as opções $UCBRFx = 9$ com $UCBRSx = 5$ e $UCBRFx = 10$ com $UCBRSx = 4$ são as que mais se aproximam da frequência desejada. Entretanto, elas não estão indicadas na Tabela 36-5 do manual do MSP porque devem provocar maiores erros totais nos bits de transmissão e recepção.

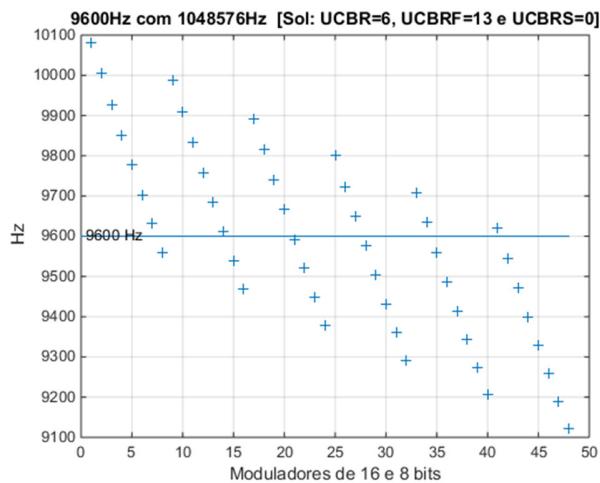


Figura 8.17. Repetição da Figura 8.16, mas agora sinalizando com “+” as opções válidas.

Pode-se ver as 6 opções (8, 9, ..., 13) do Modulador de 16 bits (UCBRF) e todas as opções do Modulador de 8 bits para o caso de se usar SMCLK para gerar 9.600 Hz.

Abaixo está a listagem de um *script Matlab* para gerar o gráfico com as opções dos Moduladores de 16 e de 8 bits, como mostrado na figura acima. Ele permite definir o BITCLK desejado e o BRCLK disponível.

Script Matlab para mostrar os resultados das opções de UCBRx, UCBRFx e UCBRSx (é preciso um ajuste na faixa de UCBRFx a ser analisada) quando no Modo Super Amostragem

```
% BRb - UCOS16=1
% Gerar BITCLK Hz partir de BRCLK Hz

aclk = 2^15;
smclk = 2^20;

% Definir bitclk e brclk
bitclk = 9600; %<===== alterar esta linha de acordo com o caso
brclk=smclk; %<===== alterar esta linha de acordo com o caso
%brclk=2e7;

% Calcular bitCLK16
bitclk16=16*bitclk;

% Calcular M16
n=brclk/bitclk16;
N=fix(n);
m16=n-N;
M16=round(16*m16); %operador round ou floor ?
```

```
% Calcular M8
n=bitclk16/bitclk;
m8=n-16;
M8=round(8*m8);

% Desenhar gráfico
bps=zeros(1,1);
x=1;
for MM16=8:13      %<== Faixa a analisar
    for MM8=0:7
        bps(x)=brclk/(N+(MM16/16));
        bps(x)=bps(x)/(16+(MM8/8));
        x=x+1;
    end
end
%eixo=8+[0:1/8:5+7/8];
plot(bps);
%plot(bps,'+');
grid;
msg=sprintf('%dHz     com     %dHz           [Sol:      UCBR=%d,      UCBRF=%d      e
UCBRS=%d]',bitclk,brclk,N,MM16,M8);
title(msg);
xlabel('Moduladores de 16 e 8 bits');
ylabel('Hz');
line([0 48],[bitclk bitclk]);
msg=sprintf('%d Hz',bitclk);
text(0.5,bitclk,msg);
```

Considerações sobre os Instantes da Amostragem do Canal Serial

Café, vamos discutir este tópico. Ver figuras 36-10 e 36-11 do User Guide e Figura 10.21 do Davies

O próximo tópico vai deixar um pouco mais claro o que deve ser levado em conta quando se tem várias opções para programação do *baud-rate*. O primeiro ponto a se preocupar são os instantes em que acontecem as 3 amostragens para decidir por votação o valor do bit recebido. O relógio usado para marcar estes instantes de amostragem é o BRCLK no caso de UCOS16 = 0 (baixa frequência) ou o BITCLK16 no caso de UCOS16 = 1 (super amostragem). **De onde retirei isso?**

A atuação do UCOS16, dos divisores e moduladores, resulta em uma janela de bit (um período do BITCLK). O que interessa no momento é saber se essa janela contém um número par ou número ímpar de períodos do relógio base (BRCLK ou BITCLK16).

Vamos então analisar primeiro o caso, quando a janela de bit é formada por uma quantidade par de períodos. A Figura 8.18 apresenta um exemplo considerando a janela composta por 4 períodos do relógio de referência. Note que a amostragem acontece bem no centro dessa janela. Uma amostra bem no centro, outra meio período antes e a outra meio período depois.

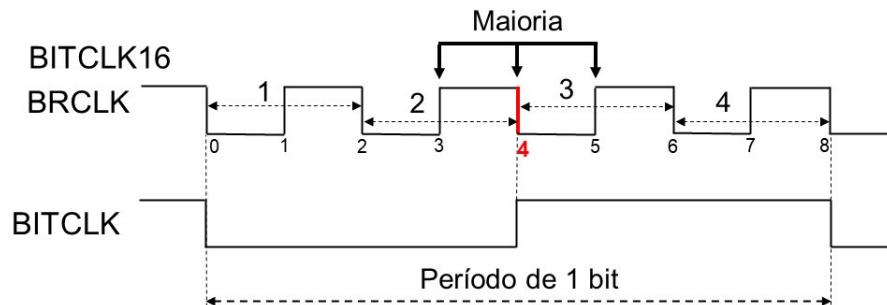


Figura 8.18. Instante de amostragem considerando a janela de um bit igual a 4 períodos do relógio de referência (BRCLK ou BITCLK16). A linha vermelha indica a metade da janela.

Quando a janela do bit é formada por um número ímpar de períodos do relógio de referência (BRCLK ou bitCLK16), os instantes de amostragem são atrasados (meio período) em relação ao centro desta janela. Isso acontece porque a metade exata da janela resulta em um número fracionário de períodos. A Figura 8.19 apresenta um exemplo considerando 5 períodos do relógio de referência.

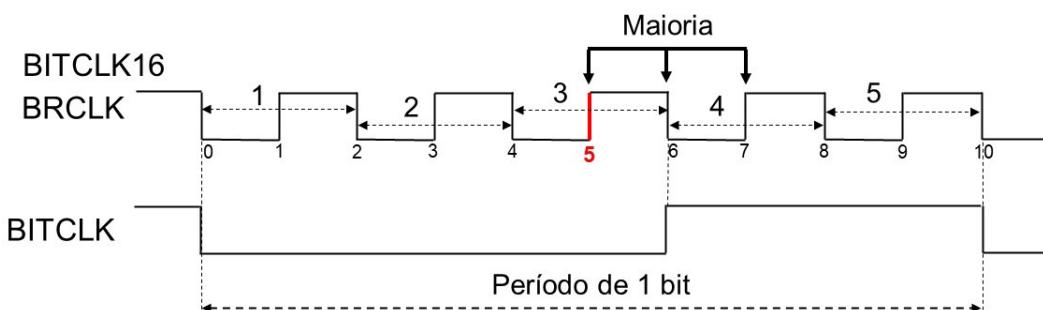


Figura 8.19. Instante de amostragem considerando a janela de um bit igual a 5 períodos da referência (BRCLK ou bitCLK16). A linha vermelha indica a metade da janela.

Acho que errei, se são 5 períodos, então a amostragem acontecer um período mais cedo.

A recepção ainda depende do instante em que o receptor percebe o bit de Partida. Isto pode resultar no erro de até um período do relógio base, pois a busca pelo bit de Partida acontece uma única vez por período (flanco de descida) deste relógio. Quando a relação entre o relógio base e o bitCLK é alto, este erro não importa muito. Entretanto, quando a

relação é baixa, como o caso mostrado na Figura 8.20, o instante de amostragem pode se aproximar das “extremidades” da janela de bit. Isso ainda pode ficar mais delicado se lembarmos que o Modulador de 8 bits pode inserir períodos extras. (**Acho que isso não ficou bem explicado. Que tal refazer a figura inserindo o que o transmissor envia e a óptica do receptor com o erro de um período? É complicar muito?**). Veja na Tabela 36-4 do Gui do Usuário do MSP430 que os maiores erros surgem quando UCBRx é pequeno.

Será que vale a pena entrar nos detalhes de cálculos dos erros das Tabelas 36-4 e 36-5?

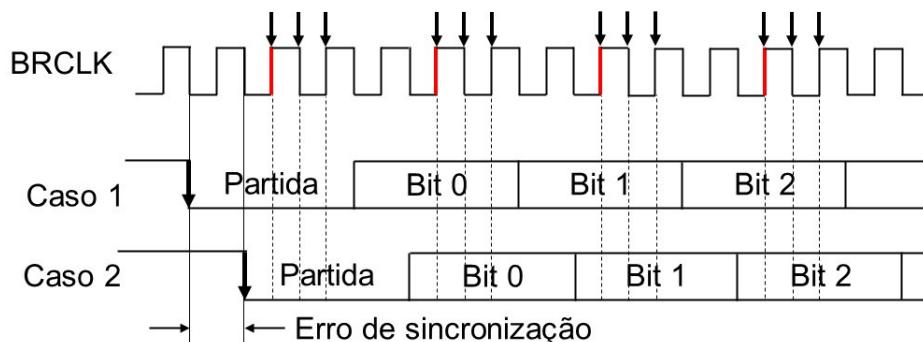


Figura 8.20. Possíveis erros de sincronização para o caso em que a relação entre BRCLK e bitCLK é igual a 3 (UCOS16 = 0). A linha vermelha indica a metade de cada janela.

Depois de tudo isso, vamos a algumas orientações. Quando se usa o Modo Baixa Frequência (UCOS16 = 0), um valor baixo do divisor (UCBRx) aumenta vulnerabilidade aos erros de sincronização pois os instantes de amostragem se aproximam de uma das extremidades da janela de bit. Nunca use UCBRx menor que 3. Por outro lado, quando o valor do divisor (UCBRSx) é muito alto, os instantes de amostragem ficam muito próximos, o que reduz a imunidade ao ruído. Se a comunicação serial está sujeita a um ambiente muito ruidoso, acreditamos que o divisor dentro da faixa de 10 a 20 deve trazer bons resultados.

Já o Modo de Super Amostragem (UCOS16 = 1) não sofre com os problemas acima, pois a relação entre bitCLK16 e bitCLK sempre será próxima de 16. O Modulador de 8 bits, se empregado, pode introduzir apenas uma pequena variação. A deficiência deste Modo de Super Amostragem está no emprego de frequências mais elevadas, o que aumenta o consumo de energia.

8.5.2. Formato do Trem de bits

As possibilidades de formato de dados oferecidas pela USCI estão resumidas na Figura 8.21. O campo de dados pode ser composto por 7 ou 8 bits (UCBIT7). O usuário pode selecionar entre um ou dois bits de parada (UCSPB). Como o bit de parada é idêntico à

linha inativa, o uso de dois bits de parada aumenta a separação entre dois envios sequenciais, além de propiciar uma folga maior para o receptor. Temos ainda o bit de paridade e o bit indicador de endereço.

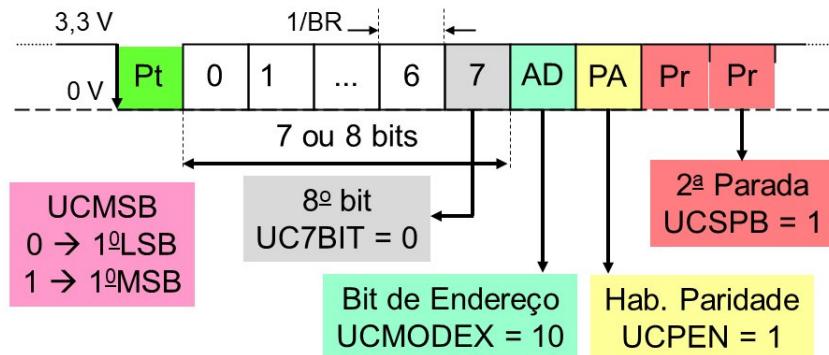


Figura 8.21. Opções do trem de bits gerados pelas USCI quando no modo UART.

O programador pode habilitar ou não uso de um bit de paridade (UCPEN). Se a paridade estiver habilitada, é possível escolher entre a paridade par e a paridade ímpar. A finalidade da paridade é permitir, na recepção, uma forma simples de detecção de erro. Aqui se faz necessária uma pequena explicação. O bit de paridade é um bit extra adicionado após o campo de dados. Na paridade par, este bit extra é ajustado de tal forma que a quantidade total de bits iguais a “1” seja par. O funcionamento é semelhante no caso da paridade ímpar.

Vamos ao exemplo da tabela abaixo em que se transmite em 8 bits de dados o código ASCII das letras A (0x41), B (0x42) e C (0x43), sendo que o bit de paridade (par ou ímpar) está sinalizado em vermelho. Na recepção, será possível detectar a alteração de um bit, porque a paridade vai estar errada. Note que este recurso apenas indica que aconteceu um erro. Ele é uma forma fraca de proteção já que o caso de dois erros no mesmo campo de dados, resulta num resultado válido.

Tabela 8.6 – Ilustração do uso do bit de paridade com Paridade Par e com Paridade Ímpar

Letra	ASCII	Paridade Par	Paridade Ímpar
A	0100 0001	0100 0001 0	0100 0001 1
B	0100 0010	0100 0001 0	0100 0001 1
C	0100 0011	0100 0011 1	0100 0001 0

8.5.3. Comunicação Multiprocessador

O protocolo serial assíncrono foi criado para o caso de comunicação entre dois equipamentos, onde as linhas de transmissão e recepção são cruzadas, como mostrado na figura abaixo.

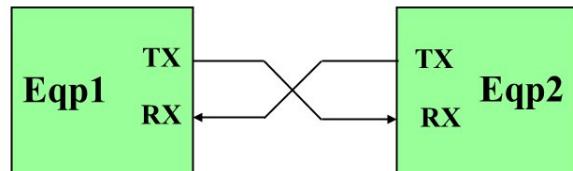


Figura 8.22. Comunicação serial full-duplex entre dois equipamentos.

Porém, as aplicações contemporâneas, muitas vezes, demandam pela comunicação entre 3 ou mais equipamentos, ou no nosso caso, processadores. O que é apresentado a seguir pretende solucionar este problema, oferecendo uma forma de endereçamento para os processadores envolvidos. É importante comentar que é oferecido apenas um recurso simples de endereçamento. Toda a parte lógica da comunicação deve ser construída pelo programador. Não existe qualquer tipo de recurso para tratar as colisões ou garantir alguma forma de arbitragem.

Com isso buscam-se por duas vantagens. A primeira é a de poder selecionar um processador dentre vários e a segunda é o fato de todos os processadores ficarem no modo de baixo consumo, sendo acordados apenas quando recebem um endereço. Neste caso, o programa de cada um deve verificar se ele foi endereçado.

É preciso então imaginar o caso de vários processadores conectados, cada um com um endereço único, definido previamente. O programador deve indicar um processador para ser o mestre e os demais serão escravos. O mestre controla toda a forma de comunicação, como mostrado na Figura 8.23. Nesta figura, somente o escravo endereçado tem autorização para conectar seu pino de transmissão. Todos os demais permanecem com sua saída de transmissão em alta-impedância. **Outra opção é dispensar o mestre e permitir que um processador qualquer enderece outro processador e inicie a comunicação (será que dá para fazer isso? Pensar!).**

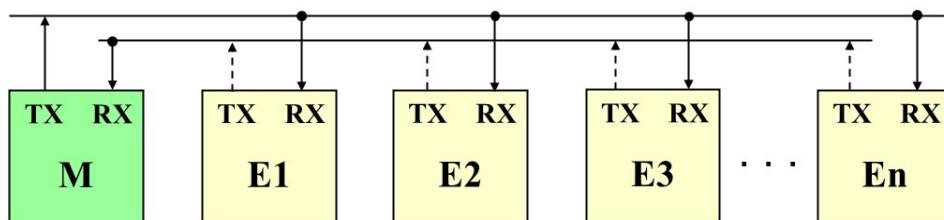


Figura 8.23. Sugestão de uma arquitetura de comunicação multiprocessador com um mestre e diversos escravos, cada um com um endereço.

O campo UCMODEX do registrador UCAxCTL0, que será apresentado mais adiante, permite 4 opções ao usuário, como indicado na tabela abaixo. A primeira (UART) é para a operação serial corriqueira. As duas seguintes são destinadas a comunicação multiprocessador. A última é para a detecção da velocidade de transmissão (*baud-rate*).

Tabela 8.7 – Modos de operação da USCI

UCMODEX	Descrição	Multiprocessador
0	UART	Não
1	Linha Ociosa	Sim
2	Bit de Endereço	Sim
3	UART (detecção de <i>baud-rate</i>)	Não

Formato de Comunicação Multiprocessador Usando Linha Ociosa

Este formato é habilitado com UCMODEX = 1. Vamos iniciar indicando o que, neste caso, se entende por “linha ociosa” (*idle-line*). Em uma comunicação serial típica, usamos um trem de 10 bits: 1 bit de partida, 8 bits de dados e 1 bit de parada. Para este modo de operação, a linha é considerada ociosa quando permanece inativa por um intervalo de tempo igual ou superior ao tempo consumido para se transmitir 10 bits. Vale lembrar que linha está ociosa quando em nível alto.

Este formato de comunicação multiprocessador se resume a uma simples convenção: o primeiro caractere recebido após um período de linha ociosa deve ser interpretado como endereço. De acordo com a sugestão feita na Figura 8.23, o mestre deixa a linha entrar no estado de ociosa e em seguida envia um endereço. O escravo endereçado conecta seu pino de transmissão e os dois podem conversar. A conversação termina quando o mestre deixa sua linha de transmissão entrar no estado de ociosa, novamente.

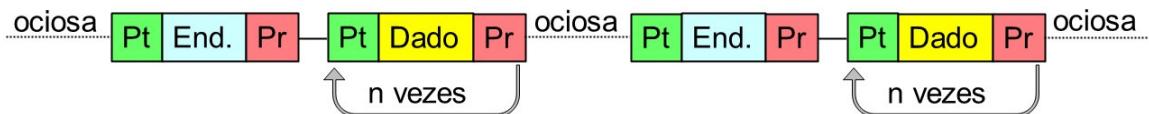


Figura 8.24. Endereçamento usando o formato Linha Ociosa.

O bit de controle UCDORM é usado para permitir economia de energia quando se usa este modo. Se UCDORM é igual a 1, somente os endereços são transferidos para o *buffer* de recepção, ativando TXIFG, o que permite gerar interrupção e retirar a CPU do modo de baixo consumo de energia. Ao receber um endereço, o processador deve conferir se é o seu endereço. Caso positivo, ele deve zerar UCDORM e continuar a receber os dados.

A USCI permite a geração precisa de um período de ocioso. Se o programador ativar o bit de controle UCTXADDR, então próximo dado a ser transmitido é precedido por um período ocioso equivalente ao tempo da transmissão de 11 bits. Então, este dado será interpretado como endereço. O bit UCTXADDR é zerado automaticamente após a transmissão do endereço.

Formato de Comunicação Multiprocessador Usando bit de Endereço

Este formato é habilitado com UCMODEX = 2. Agora, cada trem de bits, tem um bit extra para indicar se o caractere transmitido é um endereço (AD = 1) ou um dado (AD = 0). O arranjo de bits está indicado na Figura 8.21 e a Figura 8.25, logo abaixo, deixa clara a ideia. A transmissão com AD = 1, faz a seleção de um escravo e depois as demais transmissões, com AD = 0, trafegam dados com o escravo previamente endereçado.

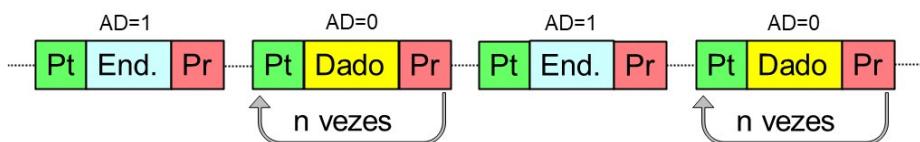


Figura 8.25. Endereçamento usando o bit de endereço (AD).

Para enviar um endereço, o transmissor primeiro ativa o bit UCTXADDR e depois escreve no *buffer* de transmissão o byte que será enviado como endereço. O bit UCTXADDR é automaticamente zerado quando a transmissão é iniciada. Então, para depois enviar os dados, nada precisa ser feito. Neste modo de operação (UCMODEX = 1), o bit UCTXADDR é sempre copiado para o trem de bits a ser transmitido.

O receptor, para economizar energia, pode ficar no estado dormente (UCDORM = 1). Neste estado, o dado serial só é transferido para o *buffer* de recepção se o bit AD for igual a 1. Neste caso a flag UCRXIFG é ativada e pode provocar interrupção, retirando a CPU do modo de baixo consumo. Então, o programa pode conferir se é o seu endereço. Caso positivo, zera o bit UCDORM e permanece acordado para a comunicação. Caso negativo, nada precisa ser feito e a CPU volta para o modo de baixo consumo ao finalizar a interrupção.

8.5.4. Emprego da Sinalização *Break*

A sinalização de *break* (quebra) é usada para que o transmissor desperte a atenção do receptor, usualmente, para mudar a forma de comunicação ou para fazer uma sinalização de erro grave. No caso do transmissor, o programador, em seu código, é responsável por selecionar as situações ativarão a sinalização de *break*. Já no receptor, o código deve indicar o tipo de ação a ser tomada. Em suma, é uma forma adicional de sinalização entre o transmissor e o receptor.

Uma condição de *break* acontece quando a linha de comunicação fica em nível baixo por um longo período, usualmente, superior ao tempo gasto para a transmissão de um caractere. O leitor pode se perguntar se isso não ocorre quando se transmite um byte igual a zero? Não, porque ao final temos o bit de parada, que obrigatoriamente é em nível alto. Então, a condição de *break* pode ser explicada, independentemente do modo selecionado, como sendo uma transmissão com todos os bits iguais a zero, incluindo dados, paridade e até as janelas destinadas ao bit de parada. Numa comunicação normal, isso nunca acontece.

Para transmitir um *break*, o programa deve ativar o bit UCTXBRK e escrever 0 no *buffer* de transmissão. Assim que inicia a transmissão, o bit UCTXBRK é automaticamente zerado. A sinalização de *break* pode ser usada nos modos 0, 1 ou 2 (UCMODEX = 0, 1, ou 2).

No receptor, o *break* é sinalizado pela *flag* UCBRK indo para 1. Se a interrupção estiver habilitada (UCBKIE = 1), então ela acontece. A *flag* (UCRXIFG) que sinaliza a recepção também é ativada e o *buffer* de recepção pode ser lido, sendo que ele vai retornar 0.

8.5.4. Detecção Automática de *Baud-rate*

Na detecção automática de *baud-rate*, o transmissor envia um padrão de bits que o receptor usa para estimar a duração da janela de bits e assim calcular o *baud-rate* e atualizar os registradores necessários (UCAxBRCTL e UCAxMCTL). O padrão é composto por uma sinalização de *break*, seguida pelo envio do byte 0x55, como mostrado na figura abaixo.

Quando neste modo, após a sinalização de *break* o receptor se prepara para medir o intervalo de tempo caracterizado por 5 flancos de descida. Com isso, ele mede a janela de tempo usada para se transmitir 8 bits. Conhecendo o tamanho dessa janela, fica fácil estimar o *baud-rate*.

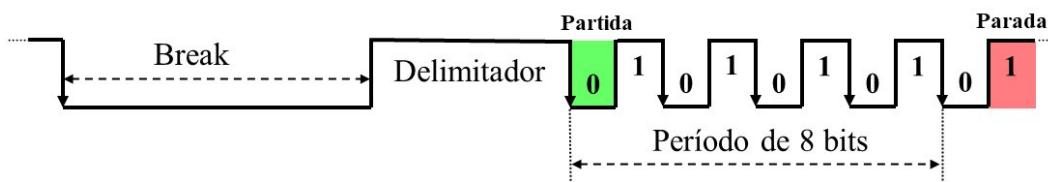


Figura 8.26. Sequência para realizar a detecção de baud-rate.

Para enviar a sequência de sincronismo, o transmissor deve obedecer às etapas:

1. Selecionar Modo 3 (UCMODEx = 3) e configurar UCAXBRCTL e UCAXMCTL de acordo com o *baud-rate* desejado.
2. Escrever o byte 0x55 no TXBUF (autorizado por TXIFG = 1), não usar paridade e configurar para um bit de parada.
3. Deste ponto em diante, o transmissor já pode começar a enviar seus dados.

O receptor deve estar com bit UCABDEN em 1, do contrário, o padrão é recebido, mas a medição não é realizada. O resultado dessa medição é transferido para os registradores de controle de *baud-rate* (UCAXBRCTL e UCAXMCTL). Os limites inferiores para este modo de detecção de *baud-rate* são:

- Modo Baixa Frequência (UCOS16 = 0) → 30 bauds e
- Modo Super Amostragem (UCOS16 = 1) → 488 bauds.

8.5.5. Detecção e Sinalização de Erros

Uma das piores possibilidades de erro para o receptor é a detecção de um falso bit de partida, que pode ser provocado pelo ruído. Para evitar isso, as entradas UCAXRXD contam com um filtro que elimina pulsos em nível baixo com duração inferior a 150 ns, aproximadamente. Quando um pulso em nível baixo excede este período, são feitas 3 amostragens (essa amostragem já foi estudada) para, numa votação, decidir se é realmente um pulso de partida.

Com relação ao trem de bits recebido, a USCI_A tem condições de detectar e sinalizar os erros listados na tabela abaixo. Logo a seguir é feita uma breve descrição de cada um deles. A flag UCRXERR é ativada quando acontece um dos erros listados.

Tabela 8.8. Indicação de Erros (UCRXERR = 1)

Erro	Flag	Descrição
Quadro de bits	UCFE	Bit de parada em nível em baixo.
Paridade	UCPE	Paridade errada.
Atropelamento	UCOE	Dado atual sobrescreve o anterior que não foi lido.

<i>Break</i>	UCBRK	Detectada condição de <i>Break</i> (não chega a ser um erro).
--------------	-------	---

O erro Quadro de bits pode acontecer quando se recebe um dado com o *baud-rate* errado ou com um tamanho diferente, assim, no instante em que deveria estar o bit de parada, está um bit de dado. Note que a detecção deste erro é fraca. Por exemplo, no caso de o transmissor operar numa taxa acima da que erroneamente foi programada no receptor, este erro nunca vai acontecer.

O erro de Paridade é simples e já foi explicado. A paridade sempre inclui todos os bits do campo habilitado. Por exemplo, no caso de uso do bit de endereço, ele também é incluído nesta verificação de paridade.

O erro de Atropelamento é quando chega um novo dado sendo que o anterior não foi lido. Este novo dado sobrescreve o anterior, que é então perdido. Portanto, esta indicação sinaliza ao programa que um dado foi perdido.

A Condição de *Break* já foi estudada e serve para fazer uma “sinalização forte” ao receptor. Então, em geral, não é uma condição de erro.

Quando um desses sinalizadores de erro (UCFE, UCPE, UCOE, UCBRK ou UCRXERR) vai para nível alto, ele assim permanece até que o usuário leia o *buffer* de recepção (UCAxRXBUF) ou apague “manualmente” a *flag*. Para garantir a interpretação correta da sinalização de erros, é recomendada a seguinte sequência:

1. Esperar a sinalização de que chegou um dado (UCAxRXIFG = 1);
2. Ler o registrador UCAxSTAT para verificar os possíveis erros, incluindo UCOE;
3. Ler o *buffer* de recepção (UCAxRXBUF), isto apaga toda a sinalização de erro;
4. Verificar novamente o bit UCOE, pois um novo dado pode ter sido recebido entre a leitura dos registradores UCAxSTAT e UCAxRXBUF (entre os passos 2 e 3) e vai sobre escrever o anterior.

Com o uso do bit UCRXEIE, o programador indica à USCI_Ax se deseja receber os caracteres com erros. Ao colocar este bit em 1, o dado errado é recebido e a *flag* UCRXIFG é ativada, podendo provocar interrupção. Com UCRXEIE em zero, o dado errado é rejeitado e a *flag* UCRXIFG não é ativada.

8.5.5. Interrupções da USCI_Ax no Modo UART

Somente as duas *flags* abaixo podem provocar a interrupção da USCI_Ax quando no Modo UART. Para que a interrupção ocorra, é necessário habilitá-la, o que é feito com os bits UCTXIE e UCRXIE, respectivamente, para transmissão e recepção.

- UCTXIFG → vai a 1 para indicar que o *buffer* de transmissão (UCAxTXBUF) está pronto para receber um novo dado. Esta *flag* é automaticamente apagada quando se escreve no *buffer* de transmissão.
- UCRXIFG → vai a 1 para indicar que o *buffer* de recepção (UCAxRXBUF) tem um novo dado para ser lido. Esta *flag* é automaticamente apagada quando se lê o *buffer* de recepção.

Quando o MSP é energizado ou quando o bit de *reset* (UCSWRST) é ativado, as *flags* vão para o seguinte estado: UCTXIFG é ativado e UCRXIFG é zerado. Para a recepção, ainda é preciso levar em conta as seguintes particularidades na geração de interrupções.

- Se UCAxRXEIE = 0, então caracteres errados não ativam UCRXIFG;
- Se UCDORM = 1, no modo multiprocessador, apenas os caracteres de endereço ativam UCRXIFG, se no modo normal, nenhum caractere ativa UCRXIFG.
- Se UCBRKIE = 1, a condição de *break* ativa UCRXIFG e UCBRK.

Da Tabela de Vetores de Interrupção, somente uma posição está dedicado à essas duas possibilidades de interrupção (UCTXIFG e UCRXIFG), por isso, é preciso consultar o registrador UCAXIV. A leitura desse registrador retorna um número correspondente à interrupção pendente de maior prioridade e apaga a *flag* que a provocou. Neste registrador, a recepção tem a maior prioridade.

8.6. Registradores da USCI_Ax no Modo UART

Nesta seção são descritos os registradores da USCI_Ax quando opera no modo UART. Alguns podem ser acessados como um registrador de 16 bits ou em duas porções de 8 bits, como mostrado na Tabela 8.9. Outros possuem acesso exclusivo em 8 ou 16 bits. Para o programador em C, isso não faz grande diferença, já que o compilador cuida dos detalhes. Na descrição particular de cada um, para facilitar a ocupação do espaço disponível em cada página deste texto, demos preferência ao acesso usando as porções de 8 bits. O registrador dedicado ao IRDA (UCAxIRCTL) não será estudado.

Tabela 8.9. Registradores da USCI_Ax quando opera no modo UART.

16 bits	8 bits	Acesso	Reset	Ordem
UCAxCTLW0	UCAxCTL1	R/W	0x1	LSB
	UCAxCTL0	R/W	0	MSB
UCAxBRW	UCAxBR0	R/W	0	LSB
	UCAxBR1	R/W	0	MSB
-	UCAxMCTL	R/W	0	-

-	UCAxSTAT	R/W	0	-
-	UCAxRXBUF	R/W	0	-
-	UCAxTXBUF	R/W	0	-
-	UCAxABCTL	R/W	0	-
UCAxICTL	UCAxIE	R/W	0	LSB
	UCAxIFG	R/W	0	MSB
UCAxIV	-	R/W	0	-
UCAxIRCTL	UCAxIRTCTL	R/W	0	LSB
	UCAxIRRCTL	R/W	0	MSB

8.6.1. UCAxCTL0 – Registrador 0 de Controle (USCI_Ax Control Register 0)

Este registrador controla o acesso ao modo UART, o tamanho do campo de dados, a ordem de transmissão dos bits e a habilitação da paridade, como mostrado abaixo.

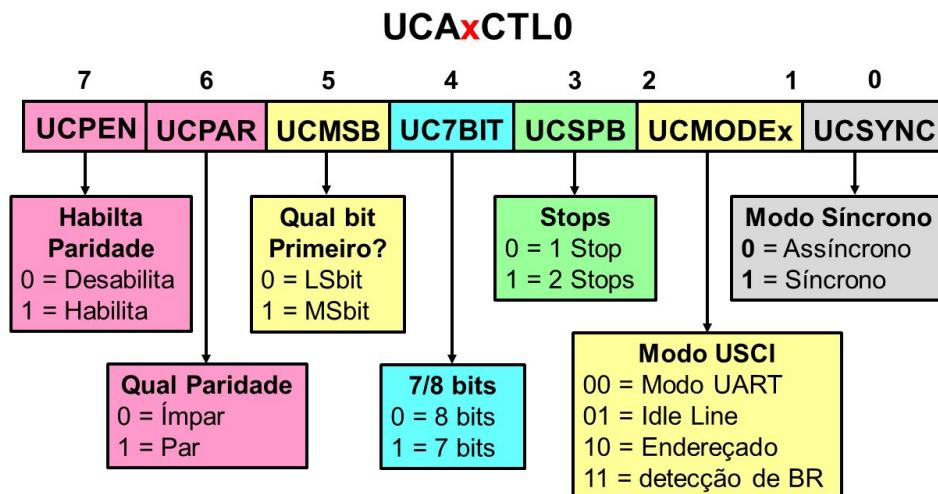


Figura 8.27. Descrição dos bits do registrador UCAxCTL0.

(R/W) bit 7: UCPEN – Habilitar a paridade

(Parity enable)

O usuário habilita ou desabilita ao uso da paridade. Se a paridade é habilitada, ela é esperada tanto na transmissão quanto na recepção. No modo de comunicação multiprocessador usando o bit de endereço, este bit é também incluído no cálculo da paridade.

UCPEN = 0 → Paridade desabilitada.
 UCPEN = 1 → Paridade habilitada.

**(R/W) bit 6: UCPAR – Seleção do tipo de paridade
*(Parity select)***

Este bit só tem sentido com UCPEN = 1, ou seja, com a paridade habilitada.
 UCPAR = 0 → Selecionada paridade ímpar.
 UCPAR = 1 → Selecionada paridade par.

**(R/W) bit 5: UCMSB – Seleção do MSB primeiro
*(MSB first select)***

O usuário seleciona a ordem dos bits, se primeiro deve ser enviado o bit mais significativo (MSB) ou primeiro o bit menos significativo (LSB).
 UCMSB = 0 → Envia primeiro o bit menos significativo (mais comum).
 UCMSB = 1 → Envia primeiro o bit mais significativo.

**(R/W) bit 4: UC7BIT – Seleção do tamanho do campo de dados
*(Character length)***

O usuário seleciona se quer trabalhar com campo de dados de 7 ou 8 bits.
 UC7BIT = 0 → Dados com 8 bits.
 UC7BIT = 1 → Dados com 7 bits.

**(R/W) bit 3: UCSPB – Seleção da quantidade de bits de parada
*(Stop bit select)***

O programador seleciona se quer usar 1 ou 2 bits de parada.
 UCSPB = 0 → Um bit de parada.
 UCSPB = 1 → Dois bits de parada.

**(R/W) bits 2,1: UCMODEx – Modo da USCI
*(USCI mode)***

Faz a seleção entre os modos assíncronos quando UCSYNC = 0.
 UCMODEx = 0 → Modo UART.
 UCMODEx = 1 → Modo Multiprocessador com endereçamento por linha ociosa.
 UCMODEx = 2 → Modo Multiprocessador com bit de endereço.
 UCMODEx = 3 → Modo UART com detecção automática de *baud-rate*.

**(R) bit 0: UCSYNC – Habilitar modo síncrono
*(Synchronous mode enable)***

Este bit deve estar em zero para permitir a operação com a UART.
 UCSYNC = 0 → Modo assíncrono (UART).
 UCSYNC = 1 → Modo síncrono.

8.6.2. UCAxCTL1 – Registrador 1 de Controle

(USCI_Ax Control Register 1)

Este registrador contém o bit para ativar o *reset* da USCI_A além permitir a seleção de algumas outras opções de operação e indicação do relógio (BRCLK) a ser usado para gerar o *baud-rate* (BITCLK), como mostrado abaixo.

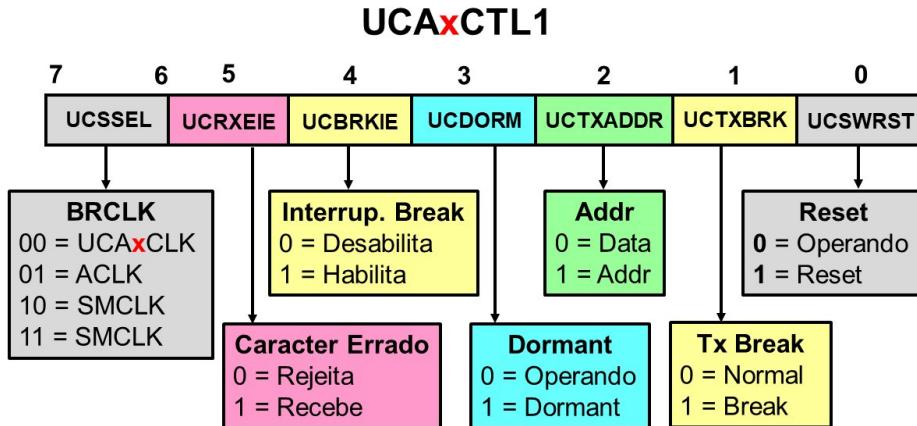


Figura 8.28. Descrição dos bits do registrador UCAxCTL1.

(R/W) bits 7,6: UCSSEL – Seleção do relógio para a USCI (USCI clock source select)

Com esses dois bits o programador seleciona o relógio (BRCLK) que será usado para gerar o *baud-rate* (BITCLK).

UCSSEL = 0 → UCAxCLK, relógio por um pino da CPU. Não disponível na F5529.

UCSSEL = 1 → ACLK

UCSSEL = 2 → SMCLK

UCSSEL = 3 → SMCLK

(R) bit 5: UCRXEIE – Habilita recepção de dado errado (Receive erroneous-character interrupt enable)

Com este bit o programador indica o comportamento da USCI para quando recebe um dado com erro. Se for orientada para receber o dado com erro, a *flag* UCRXIFG é ativada e pode provocar interrupção.

UCRXIE = 0 → o dado errado é rejeitado e UCRXIFG não é ativado.

UCRXIE = 1 → o dado errado é recebido e UCRXIFG é ativado.

(R/W) bit 4: UCBRKIE – Habilita interrupção para a condição de Break (Receive break character interrupt enable)

O programador seleciona se a condição de *break* ativa a *flag* UCRXIFG, podendo então provocar interrupção.

UCBRKIE = 0 → Condição de *break* não ativa UCRXIFG e não provoca interrupção.

$UCBRKIE = 1 \rightarrow$ Condição de *break* ativa UCRXIFG e pode provocar interrupção.

(R/W) bit 3: UCDORM – Estado dormente

(*Dormant*)

Coloca a USCI no estado dormente (dormindo).

$UCDORM = 0 \rightarrow$ Não dormente. Todos dados são recebidos e ativam UCRXIFG.

$UCDORM = 1 \rightarrow$ Dormente. No modo de comunicação multiprocessador, apenas os endereços ativam UCRXIFG. No modo de detecção de *baud-rate*, apenas a condição de *break* seguida pelo sincronismo ativa a UCRXIFG.

(R/W) bit 2: UCTXADDR – Transmitir endereço (Modo Multiprocessador)

(*Transmit address*)

Quando no Modo de Comunicação Multiprocessador usando bit de endereço, ao ativar esta opção o programador indica que a próxima transmissão deve ter o bit de endereço em ativado.

$UCTXADDR = 0 \rightarrow$ A próxima transmissão será de um dado (bit de endereço = 0).

$UCTXADDR = 1 \rightarrow$ A próxima transmissão será de um endereço (bit de endereço = 1).

(R/W) bit 1: UCTXBRK – Transmitir uma condição de *Break*

(*Transmit Break*)

Transmitir uma condição de *break* com a próxima escrita no *buffer* de transmissão (UCAxTXBUF). Se estiver no modo de detecção automática de *baud-rate*, o próximo dado a ser escrito deve ser 0x55. Nas demais condições, deve ser escrito zero.

$UCTXBRK = 0 \rightarrow$ A próxima transmissão não é um *break*.

$UCTXBRK = 1 \rightarrow$ A próxima transmissão é um *break* ou um *break* seguido pelo sincronismo.

(R/W) bit 0: UCSWRST – Reset habilitado por software

(*Software reset enable*)

O programador faz o *reset* da USCI. Vale lembrar que a configuração da USCI deve acontecer com este bit ativado.

$UCSWRST = 0 \rightarrow$ *Reset* desabilitado, USCI liberada para operar.

$UCSWRST = 1 \rightarrow$ *Reset* habilitado.

8.6.3. UCAxBRW – Registrador de Controle do *Baud-rate* (*USCI_Ax Baud-rate Control Word*)

Este registrador permite programar o divisor usado na programação do *baud-rate*. É recomendado consultar o tópico 8.5.2 para ver os detalhes de sua programação. Ele é formado por dois registradores de 8 bits: UCAxBR1 e UCAxBR0, como mostrado a seguir.

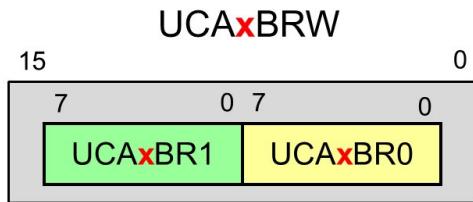


Figura 8.29. Descrição do registrador UCAXBRW.

8.6.4. UCAXMCTL – Controle dos Moduladores (USCI_Ax Modulation Control)

Este registrador disponibiliza o controle dos moduladores de 8 e 16 bits, além de permitir a seleção dos modos de baixa frequência ou super amostragem, como mostrado a seguir.

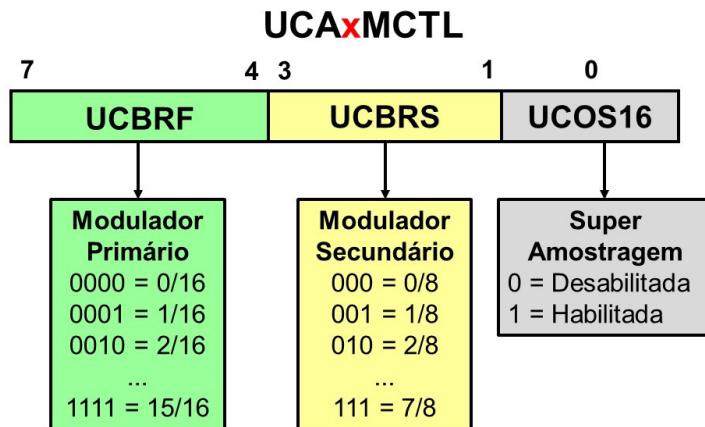


Figura 8.30. Descrição dos bits do registrador de controle dos moduladores.

(R/W) bits 7,6,5,4: UCBRF – Seleção para o Primeiro Modulador (16 bits) (First modulation stage select)

Com esses quatro bits se faz a seleção do padrão usado na modulação do BITCLK16 usado no Modo Super Amostragem. Vide Tabela 8.5. Eles bits são ignorados quando UCOS16 = 0.

(R/W) bits 3,2,1: UCBRS – Seleção para o Primeiro Modulador (16 bits) (First modulation stage select)

Com esses três bits se faz a seleção do padrão usado modulação do BITCLK. Vide Tabela 8.4.

(R/W) bit 0: UCOS16 – Habilitação do Modo Super Amostragem (Oversampling mode enable)

O modo de Super Amostragem ($UCOS16 = 1$) deve ser selecionado quando a relação entre o relógio tomado como base ($BRCLK$) e o *baud-rate* desejado está acima de 16. Do contrário usar o modo de Baixa Frequência ($UCOS16 = 0$).

$UCOS16 = 0 \rightarrow$ Desabilitado.

$UCOS16 = 1 \rightarrow$ Habilitado.

8.6.5. UCAxSTAT – Estado do USCI_Ax (*USCI_Ax Status Register*)

Neste registrador estão informações sobre o estado da USCI_Ax. Ele deve ser consultado para verificar a paridade do dado recebido, erro no trem de bits, condição de Break e muitas outras informações úteis para a operação, como mostrado a seguir.

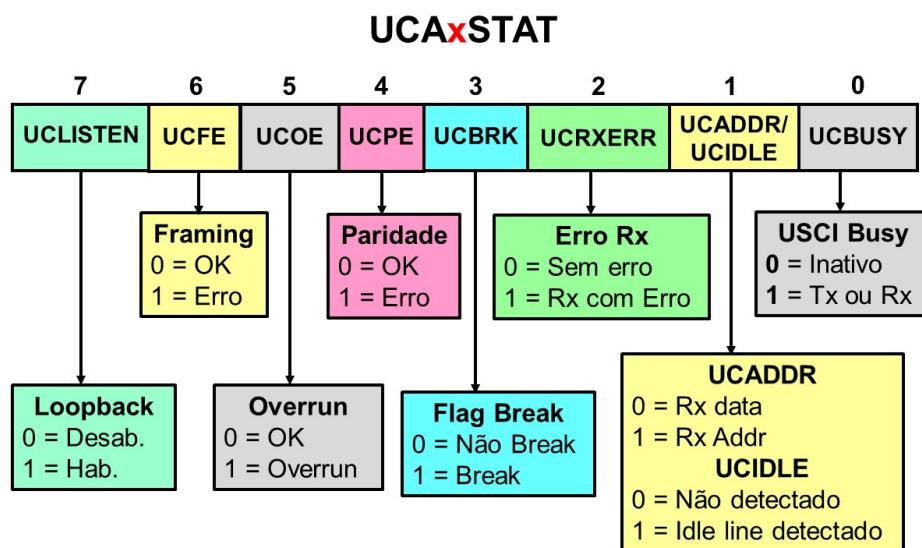


Figura 8.31. Descrição dos bits do Registrador de Estado.

(R/W) bit 7: UCLISTEN – Habilitação do Modo Ouvinte (*Listen enable*)

Na verdade, este não é um bit de estado. Quando ativado, é fechado um curto entre a saída e a entrada serial. Em outras palavras, a USCI_Ax recebe de volta tudo o que transmite (*loopback* em inglês). É interessante para a fase de testes de um projeto. Este bit só pode ser alterado com o *reset* ativado ($UCSWRST = 1$).

$UCLISTEN = 0 \rightarrow$ Desabilitado.

$UCLISTEN = 1 \rightarrow$ Habilitado.

(R/W) bit 6: UCFE – Erro no Trem de Bits (*Framing Error flag*)

Esta *flag* indica que há algo errado no trem de bits. Ela é ativada quando o bit de parada é lido em nível baixo. Ela é apagada por ocasião da leitura de UCAXRXBUF (*buffer* de recepção).

UCFE = 0 → Sem erros.

UCFE = 1 → O bit de parada do caractere recebido estava em nível baixo.

(R/W) bit 5: UCOE – Erro de Atropelamento

(*Overrun Error flag*)

Esta *flag* é ativada quando um novo caractere é escrito no UCAXRXBUF (*buffer* de recepção) antes da leitura do anterior. Isto significa que este dado anterior foi perdido.

Esta *flag* é apagada automaticamente por ocasião da leitura do *buffer*. O manual indica que ela não deve ser apagada por *software*, sob pena e de não funcionar corretamente.

UCOE = 0 → Sem erros.

UCOE = 1 → Aconteceu um atropelamento e o dado anterior foi perdido.

(R/W) bit 4: UCPE – Erro de Paridade

(*Parity Error flag*)

Quando se faz uso de paridade (UCPEN = 1), esta *flag* indica o resultado da verificação da paridade do caractere recebido. Ela é apagada automaticamente por ocasião da leitura do *buffer* de recepção (UCAXRXBUF). Se a opção paridade estiver desabilitada (UCPEN = 0), esta *flag* é sempre lida como 0.

UCPE = 0 → Sem erros.

UCPE = 1 → Caractere recebido com erro de paridade.

(R/W) bit 3: UCBRK – Detecção da Condição de Break

(*Break detect flag*)

Esta *flag* indica que foi detectado uma condição de Break. Ela é apagada automaticamente por ocasião da leitura do *buffer* de recepção (UCAXRXBUF).

UCPE = 0 → Nenhum break detectado.

UCPE = 1 → Foi detectado a condição de break.

(R/W) bit 2: UCRXERR – Indicação de Erro de Recepção

(*Receive error flag*)

Esta *flag* sumariza as condições de erro. Ela vai para 1 quando uma das seguintes *flags* é ativada: UCFE, UCPE, UCOE. Ela é apagada automaticamente por ocasião da leitura do *buffer* de recepção (UCAXRXBUF).

UCRXERR = 0 → Recepção sem erros.

UCRXERR = 1 → Foi detectado erro na recepção.

(R/W) bit 1: UCADDR/UCIDLE – Recepção de Endereço ou Linha Ociosa

(*Address received / Idle line flag*)

Com UCMODEx = 1, que é o Modo de Comunicação Multiprocessador usando linha ociosa, esta *flag* indica a detecção de que a linha está ociosa. Ela é apagada automaticamente por ocasião da leitura do *buffer* de recepção (UCAXRXBUF).

UCIDLE = 0 → Não foi detectada linha ociosa.
 UCIDLE = 1 → Foi detectada linha ociosa.

Com UCMODE_x = 2, que é o Modo de Comunicação Multiprocessador usando o bit de endereço, esta *flag* indica se foi recebido um endereço ou um dado. Ela é apagada automaticamente por ocasião da leitura do *buffer* de recepção (UCAxRXBUF).

UCADDR = 0 → Foi recebido um caractere de dado.

UCADDR = 1 → Foi recebido um caractere de endereço.

(R) bit 0: UCBUSY – USCI_Ax Ocupada

(USCI Busy)

Esta *flag* indica se está em curso uma operação de transmissão ou recepção.

UCBUSY = 0 → USCI_Ax ociosa.

UCBUSY = 1 → USCI_Ax está transmitindo ou recebendo.

8.6.6. UCAxRXBUF – Registrador de Recepção **(USCI_Ax Receive-data Buffer)**

Este é o registrador de 8 bits, só de leitura, que contém o último caractere recebido pela USCI_Ax. É o *buffer* de recepção. A leitura deste registrador apaga a *flag* UCRXIFG, além das *flags* indicadoras de erro e ainda a *flag* UCIDLE/UCADDR. No caso de se operar em 7 bits, o bit mais à esquerda sempre estará em zero.



Figura 8.32. Descrição do buffer de recepção, denominado UCAxRXBUF.

8.6.7. UCAxTXBUF – Registrado de Transmissão **(USCI_Ax Transmit-data Buffer)**

Este é o registrador de 8 bits, só de escrita, onde se escreve o dado a ser transmitido. Neste registrador, o dado fica aguardando o momento de ser transferido para o registrador de deslocamento. A escrita neste registrador apaga a *flag* UCTXIFG. No caso de se operar em 7 bits, o bit mais à esquerda sempre estará em zero.

UCAxTXBUF

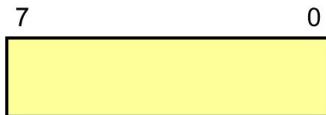


Figura 8.33. Descrição do buffer de transmissão, denominado UCAxTXBUF.

8.6.8. UCAxIRTCTL – Registrador de Controle de Transmissão do IrDA (*IrDA Transmit Control Register*)

Este destina-se ao IrDA e, como já foi dito, não será estudado.

8.6.9. UCAxIRRCTL – Registrador de Controle de Recepção do IrDA (*IrDA Receive Control Register*)

Este destina-se ao IrDA e, como já foi dito, não será estudado.

8.6.10. UCAxABCTL – Controle da Detecção Automática de *Baud-rate* (*USCI_Ax Auto Baud-rate Control Register*)

Este registrador destina-se ao controle do modo de detecção automática de *baud-rate* e está descrito na figura a seguir.

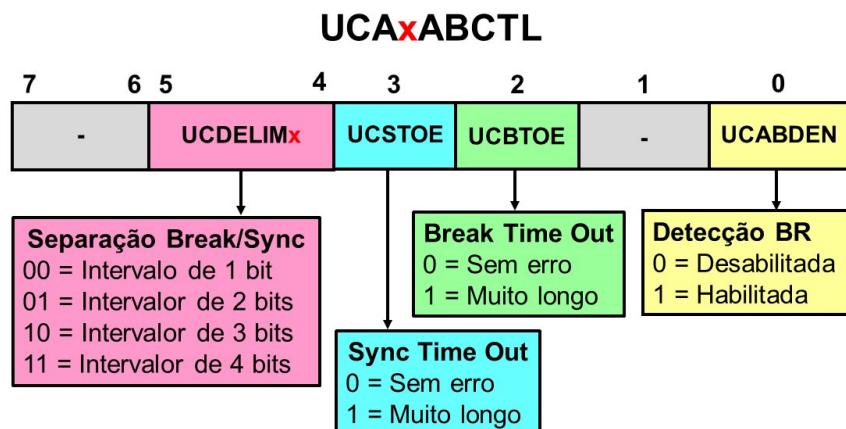


Figura 8.34. Descrição dos bits do Registrador de Controle da detecção automática de baud-rate.

(R) bits 7,6: Reservados

Bits reservados, sua leitura sempre resulta em 0.

**(R/W) bits 5,4: UCDELIMx – Separador entre Break e o Campo de Sincronismo
(Break and Sync delimiter length)**

Este campo permite controlar o intervalo entre a condição de *break* e o início do campo de sincronismo.

UCDELIMx = 0 → Separação correspondente ao período de 1 bit.

UCDELIMx = 1 → Separação correspondente ao período de 2 bits.

UCDELIMx = 2 → Separação correspondente ao período de 3 bits.

UCDELIMx = 3 → Separação correspondente ao período de 4 bits.

**(R/W) bit 3: UCSTOE – Erro na Duração do Campo de Sincronismo
(Sync field time out error)**

Indica se o campo de sincronismo ultrapassou o limite especificado (ver tópico 8.5.4).

UCSTOE = 0 → Sem erros.

UCSTOE = 1 → Duração do campo de sincronismo ultrapassou o limite mensurável.

**(R/W) bit 2: UCBTOE – Erro na Duração da Condição de Break
(Break time out error)**

Indica se a condição de *break* ultrapassou o limite especificado (ver tópico 8.5.4).

UCBTOE = 0 → Sem erros.

UCBTOE = 1 → Duração da condição de *break* ultrapassou o intervalo correspondente ao período de 22 bits.

(R) bits 1: Reservado

Bit reservado, sua leitura sempre resulta em 0.

**(R/W) bit 0: UCABEN – Habilitação da Detecção Automática de *Baud-rate*
(Automatic baud-rate detect enable)**

Habilita o recurso de detecção automática de *baud-rate* (ver tópico 8.5.4).

UCABEN = 0 → Detecção desabilitada. A duração da condição de *break* e do campo de sincronismo não são medidos.

UCABEN = 1 → Detecção habilitada. A duração da condição de *break* e do campo de sincronismo são medidos e os registradores que controlam o *baud-rate* são alterados.

**8.6.11. UCAxIE – Registrador para Habilitação das Interrupções
(USCI_Ax Interrupt Enable Register)**

A USCI_Ax, quando no modo UART, tem apenas duas interrupções: a de transmissão e a de recepção. Este registrador permite controlar suas habilitações, como mostrado a seguir.

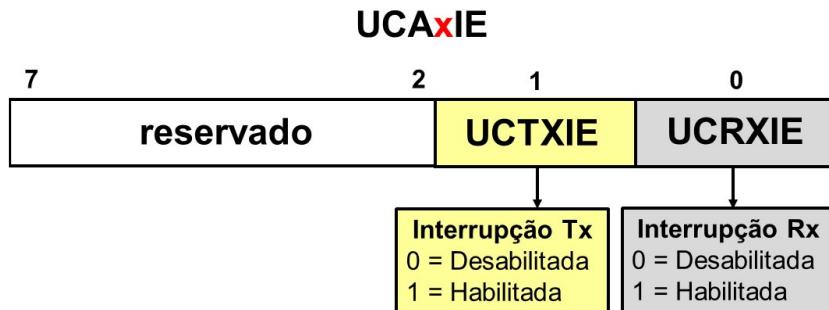


Figura 8.35. Descrição dos bits do Registrador de habilitação das interrupções.

(R) bits 7,6,...,2: Reservados

Bits reservados, sua leitura sempre resulta em 0.

(R/W) bit 1: UCTXIE – Habilitação da Interrupção por Transmissão.

(Transmit interrupt enable)

Habilita a interrupção quando termina a transmissão de um caractere (TXIFG = 1).

UCTXIE = 0 → Interrupção desabilitada.

UCTXIE = 1 → Interrupção habilitada.

(R/W) bit 0: UCRXIE – Habilitação da Interrupção por Recepção.

(Receive interrupt enable)

Habilita a interrupção quando termina a recepção de um caractere (RXIFG = 1).

UCRXIE = 0 → Interrupção desabilitada.

UCRXIE = 1 → Interrupção habilitada.

8.6.12. UCAxIFG – Registrador de *Flags* de Interrupção

(USCI_Ax Interrupt Flag Register)

Este registrador hospeda as duas *flags* que indicam a finalização da transmissão ou da recepção e podem provocar interrupções, como mostrado a seguir.

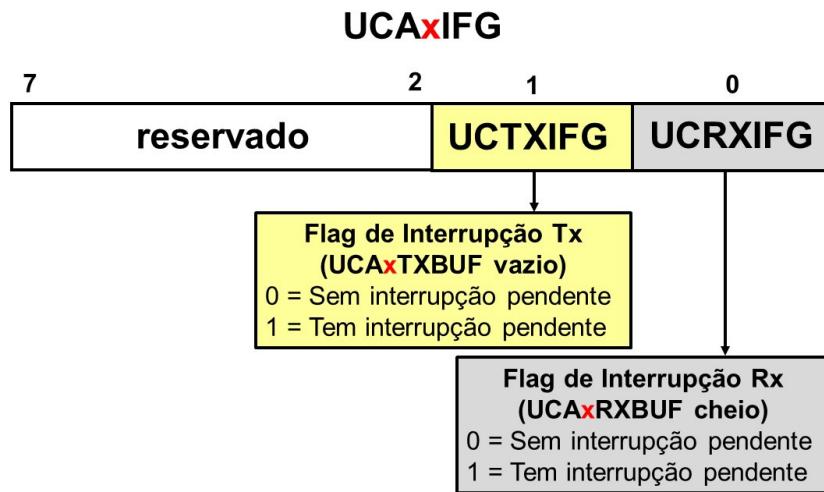


Figura 8.36. Descrição dos bits do Registrador de habilitação das interrupções.

(R) bits 7,6,...,2: Reservados

Bits reservados, sua leitura sempre resulta em 0.

(R/W) bit 1: UCTXIFG – Flag de Interrupção por Transmissão.

(Transmit interrupt flag)

Esta *flag* é ativada sempre que o *buffer* de transmissão (UCA_XTXBUF) está vazio.

UCTXIFG = 0 → Nenhuma interrupção pendente.

UCTXIFG = 1 → Interrupção pendente.

(R/W) bit 0: UCRXIFG – Flag de Interrupção por Recepção.

(Receive interrupt flag)

Esta *flag* é ativada sempre que o *buffer* de recepção (UCA_XRXBUF) é preenchido com o caractere recebido.

UCRXIFG = 0 → Nenhuma interrupção pendente.

UCRXIFG = 1 → Interrupção pendente.

8.6.13. UCA_XIV – Registrador de Vetor de Interrupção

(USCI_Ax Interrupt Vector Register)

Este é um registrador de 16 bits e indica o evento mais prioritário dentre os 2 possíveis que podem provocar interrupção da USCI_Ax. A cada leitura, este registrador retorna um número correspondente à interrupção de maior prioridade dentre as duas pendentes. Em seguida apaga a *flag* desta interrupção. Na próxima leitura é retornado um número correspondente à segunda mais prioritária dentre as pendentes. A Tabela 8.10 resume as opções.



Figura 8.37. Descrição Registrador de Vetores de Interrupção. É um registrador de 16 bits e, separadamente, seus bits não têm significado especial.

Tabela 8.10. Descrição dos valores retornados pela leitura do registrador UCAXIV

Prioridade	Valor	Flag	Significado
	0x0	-	Nenhuma interrupção pendente
Maior	0x2	UCRXIFG	Buffer de recepção (UCAxRXBUF) recebeu um caractere
Menor	0x4	UCTXIFG	Buffer de transmissão (UCAxTXBUF) vazio.

8.7. Exercícios Resolvidos

Para os primeiros exercícios faremos uso da opção de *loop back*, que é o caso onde a USCI_Ax recebe de volta tudo o que transmite. Depois, fecharemos um curto entre os pinos de transmissão (TXD) e recepção (RXD). Porém, para propor exercícios mais sofisticados, evidentemente, precisaremos de uma segunda UART. Felizmente o MSP430 F5529 oferece duas instâncias da USCI_A que podem operar no modo UART. Então, por facilidade, vamos na praticar com a USCI_A0 sendo que a USCI_A1 vai receber um programa padrão de forma a gerar respostas.

ER 8.1. Neste primeiro exercício, vamos ativar a opção *loop back* (UCLISTEN = 1) que, dentro do *chip*, fecha um curto entre a saída e a entrada serial. Assim, a USCI_A recebe de volta tudo o que transmite. Vamos então escrever um programa que envia, repetitivamente, pela porta serial as letras de 'A' até 'Z'. Aqui vamos usar o *polling* para cadenciar a transmissão e a recepção. A recepção deve conferir se as letras estão chegando na sequência correta. O *led* verde (P4.7) acende enquanto tudo estiver correto e o *led* vermelho (P1.0) acende em caso de erro. Configuração serial: 9.600 bauds, 8 bits, sem paridade em com um bit de parada. Use o modo de baixa frequência com a ACLK (32.768 Hz).

Solução:

A solução é muito simples. A configuração dos registradores está mostrada logo abaixo e é bem óbvia. O cálculo necessário é para o divisor gerar 9.600 Hz a partir do ACLK (32.768 Hz), no modo de baixa frequência (UCOS16 = 0).

- $n = \frac{32.768}{9.600} = 3,413 \rightarrow N = 3$ (note que estamos no limite, que é 3).
- $M8 = \text{round}(0,413 \times 8) = \text{round}(3,306) = 3 \rightarrow M8 = 3$.
- Programamos UCBRx = 3 e UCBRSx = 3.
- $BR = \frac{32.768}{3+\frac{3}{8}} \rightarrow BR = 9.709,03 \text{ Hz (aceitável)}$.

Tabela 8.11. Configuração dos Registradores da USCI_A0

	7	6	5	4	3	2	1	0
UCA0CTL0	UCPEN	UCPAR	UCMSB	UC7BIT	UCSPB	UCMODEx		UCSYNC
	0	0	0	0	0	0		0
UCA0CTL1	UCSSEL		UCRXIE	UCBRKIE	UCDORM	UCTXADDR	UCTXBRK	UCWRST
	1		0	0	0	0	0	1/0
UCA0MCTL	UCBRF				UCBRS			UCOS16
	0				3			0
UCA0STAT	UCLISTEN	UCFE	UCOE	UCPE	UCBRK	UCRXERR	UCADDR/ UCIDLE	UCBUSY
	1	0	0	0	0	0	0	0

(UCA0BRW = 3 não está mostrado na tabela)

A solução está apresentada na listagem a seguir. Ela é bem simples. A variável `letra_tx` contém a letra (de A até Z, ciclicamente) a ser transmitida. No laço principal, o programa aguarda a ativação da flag UCTXIFG, que é a autorização para escrever a variável `letra_tx` no buffer de transmissão (UCA0TXBUF). Em seguida o programa aguarda a ativação da flag UCRXIFG, que é a indicação de que recebeu um caractere e que, então, o buffer de recepção (UCA0RXBUF) pode ser lido. A leitura deste buffer é armazenada em `letra_rx`. A comparação indica se o resultado foi o esperado. Caso positivo o led verde acende e caso negativo é o vermelho que acende.

É interessante comentar que a cada ciclo de transmissão e recepção, a variável `letra_tx` é incrementada. O limite é quando ela ultrapassa a letra Z, indicada pela comparação com o valor 'Z' + 1. Quando isto acontece, seu valor é restaurado para A.

O leitor pode ainda notar a presença de 4 funções para acionar os *leds*, que facilitam a preparação do programa.

Listagem da solução do ER 8.1

```
// ER8.1 USCI_A0
//
// Comunicação serial com loop back (UCLISTEN = 1)
// Usar polling
// USCI_A0 Transmite e recebe letras de A até Z
// 9.600 bauds (ACLK), 8 bits, sem paridade, 1 stop

#include <msp430.h>

#define TRUE 1
#define FALSE 0

void USCI_A0_config(void);
void led_VM(void);
void led_vm(void);
void led_VD(void);
void led_vd(void);
void leds_config(void);

int main(void){
    char letra_tx,letra_rx;
    WDTCTL = WDTPW | WDTHOLD;      // stop watchdog timer
    leds_config();
    USCI_A0_config();
    letra_tx='A';
    while(TRUE){                  //Laço principal
        while ( (UCA0IFG&UCTXIFG)==0); //Esperar TXIFG=1
        UCA0TXBUF=letra_tx;
        while ( (UCA0IFG&UCRXIFG)==0); //Esperar RXIFG=1
        letra_rx=UCA0RXBUF;

        if (letra_tx == letra_rx){   //Verificar
            led_VD();  led_vm();      //OK
        }
        else{
            led_vd();  led_VM();      //Erro
        }

        if (++letra_tx == 'Z'+1)  letra_tx='A'; //Próxima letra
    }
    return 0;
}
```

```

// Configurar USCI_A0
void USCI_A0_config(void){
    UCA0CTL1 = UCSWRST;      //RST=1 para USCI_A0
    UCA0CTL0 = 0;            //sem paridade, 8 bits, 1 stop, modo UART
    UCA0STAT = UCLISTEN;    //Loop Back
    UCA0BRW = 3;             // Divisor
    UCA0MCTL = UCBRS_3;     //Modulador = 3 e UCOS=0
    UCA0CTL1 = UCSSEL_1;    //RST=0 e Selecionar ACLK
}

// Controle dos leds
void led_VM(void){ P1OUT |= BIT0; } //led Vermelho aceso
void led_vm(void){ P1OUT &= ~BIT0; } //led Vermelho apagado
void led_VD(void){ P4OUT |= BIT7; } //led Verde aceso
void led_vd(void){ P4OUT &= ~BIT7; } //led Verde apagado

// Configurar Leds
void leds_config(void){
    P1DIR |= BIT0;          P1OUT &= ~BIT0; //Led vermelho
    P4DIR |= BIT7;          P4OUT &= ~BIT7; //Led verde
}

```

ER 8.2. Este exercício é semelhante ao anterior, porém vamos fazer o controle de transmissão e recepção usando interrupção. A conferência das letras será feita pelo programa principal. Este é um problema simples. Porém, se não for tomado cuidado, pode resultar em erros.

Solução:

A configuração dos registradores é mesma do exercício anterior, com a exceção de fazer UCARXIE = 1 para ativar a interrupção por recepção. Note que não ativamos a interrupção por transmissão, alguns leitores poderiam esperar.

As variáveis `letra_tx` e `letra_rx`, que agora são compartilhadas com a interrupção foram declaradas como globais e ainda usando o modificador `volatile` para evitar otimizações do compilador. A rotina de interrupção faz uso da variável estática `letra`. Ela é o “contador” da letra a ser transmitida.

Note que se fez uso apenas da interrupção de recepção. Isso facilita o cadenciamento entre a transmissão e a recepção. A cada caractere recebido, um novo é transmitido. Apenas a primeira transmissão precisa ser feita no corpo do programa principal, para gerar a primeira interrupção de recepção. Depois disso, tudo entra em ritmo. O programa principal fica ocupado apenas conferindo se as variáveis `letra_tx` e `letra_rx` são idênticas. Esta conferência poderia ser feita dentro da rotina de interrupção. Neste caso, o laço principal ficaria vazio.

Como temos a certeza de que apenas a interrupção por recepção foi habilitada, não é necessário checar de forma completa o resultado do registrar UCA0IV. Ele é lido apenas para apagar a *flag* UCARXIFG.

Listagem da solução do ER 8.2.a

```
// ER8.2a USCI_A0 - Correto
//
// Comunicação serial com loop back (UCLISTEN = 1)
// Usar interrupção
// USCI_A0 Transmite e recebe letras de A até Z
// 9.600 bauds (ACLK), 8 bits, sem paridade, 1 stop

#include <msp430.h>

#define TRUE 1
#define FALSE 0

void USCI_A0_config(void);
void led_VM(void);
void led_vm(void);
void led_VD(void);
void led_vd(void);
void leds_config(void);

volatile char letra_tx,letra_rx;

int main(void){
    WDTCTL = WDTPW | WDTHOLD;      // stop watchdog timer
    leds_config();
    USCI_A0_config();
    __enable_interrupt();

    UCA0TXBUF='A';                //Primeira transmissão
    while(TRUE){                   //Laço principal
        if (letra_tx == letra_rx){ //Verificar
            led_VD();  led_vm(); //OK
        }
        else{
            led_vd();  led_VM(); //Erro
        }
    }
    return 0;
}

// Interrupção da USCI_A0
// #pragma vector = 56
#pragma vector = USCI_A0_VECTOR
```

```

__interrupt void usci_a0_int(void) {
    volatile static letra='A';
    UCA0IV;           //Apagar RXIFG
    letra_rx=UCA0RXBUF; //Letra recebida
    letra_tx=letra;   //Letra transmitida
    if (++letra=='Z'+1) //Próxima letra
        letra='A';     //Voltar para letra A
    UCA0TXBUF=letra; //Transmitir
}

// Configurar USCI_A0
void USCI_A0_config(void){
    UCA0CTL1 = UCSWRST; //RST=1 para USCI_A0
    UCA0CTL0 = 0;       //sem paridade, 8 bits, 1 stop, modo UART
    UCA0STAT = UCLISTEN; //Loop Back
    UCA0BRW = 3;        // Divisor
    UCA0MCTL = UCBRS_3; //Modulador = 3 e UCOS=0
    UCA0CTL1 = UCSSEL_1; //RST=0 e Selecionar ACLK
    UCA0IE = UCRXIE;   //Hab. Interrup recepção
}

// Copiar do ER 8.1 as funções abaixo:
void led_VM(void),void led_vm(void),void led_VD(void),
void led_vd(void) e void leds_config(void)

```

O leitor pode estar inquieto com o fato de termos transmissão e recepção e no entanto usarmos apenas a interrupção da recepção. Preparamos então uma nova solução, agora usando essas duas interrupções. Como a listagem é muito semelhante à anterior, trechos idênticos foram removidos. O leitor pode comprovar que ela não funciona e é convidado a explicar o porquê e a corrigir esta solução.

Listagem da solução do ER 8.2.b (Não funcional!)

```

// ER8.2b USCI_A0 - Faz o teste de forma errada!
...
volatile char letra_tx,letra_rx;
volatile char flag;           //Iniciar qdo chega um novo dado

int main(void){
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer
    leds_config();
    USCI_A0_config();
    __enable_interrupt();

    flag=FALSE;
    while(TRUE){               //Laço principal
        while (flag==FALSE);  //Esperar recepção

```

```

    flag=FALSE;
    if (letra_tx == letra_rx){           //Verificar
        led_VD();   led_VM();           //OK
    }
    else{
        led_VD();   led_VM();           //Erro
    }
}
return 0;
}

// Interrupção da USCI_A0
//#pragma vector = 56
#pragma vector = USCI_A0_VECTOR
__interrupt void usci_a0_int(void){
    volatile static letra='A';
    switch(__even_in_range(UCA0IV,0x4)){
        case 0x0: break;
        case 0x2: letra_rx=UCA0RXBUF;           //Recepção
                  flag=TRUE;                   //Flag indica recepção
                  break;
        case 0x4: letra_tx=letra;                //Transmissão
                  UCA0TXBUF=letra++;
                  if (letra=='Z'+1) letra='A'; //Próxima letra
                  break;
    }
}

// Configurar USCI_A0
void USCI_A0_config(void){
...
    UCA0IE = UCTXIE | UCRXIE;    //Habilitar ambas interrupções
}
...

```

ER 8.3. Este exercício é semelhante ao ER 8.1, porém ao invés de habilitar o *loop back* (UCLISTEN = 1), vamos fechar um curto entre os pinos de transmissão (P3.3) e recepção (P3.4), como mostrado na Figura 8.38. Agora será necessário configurar esses pinos para serem usados pela porta serial. Com o programa rodando, podemos ainda remover momentaneamente o cabo para constar a sinalização de erro.

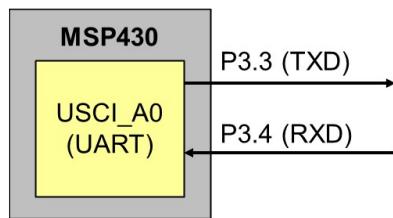


Figura 8.38. Colocação de um curto-circuito entre os pinos P3.3 e P3.4 de forma que a USCI_A0 receba de volta tudo o que transmitir.

Solução:

A solução é muito semelhante à do ER 8.1. Precisamos apenas adicionar uma sub-rotina para configurar os pinos. Um ponto merece destaque. O leitor vai constatar que ao remover o cabo que está ligando P3.3 com P3.4, surpreendentemente o *led* vermelho, que indica erro, não acende e o programa fica preso no laço aguardando a chegada de um caractere (aguarda UCARXIFG). Por que será que isso acontece?

A explicação é simples. Ao remover o cabo, provavelmente vamos provocar um erro na recepção. Da forma como foi feita a configuração da USCI_A0, ela está preparada para recusar caracteres com erro de recepção. Assim, ela recusa o caractere errado que chegou e fica preso esperando um novo caractere que nunca chega. Note que o programa está num laço que transmite e recebe. Se habilitarmos a recepção de caracteres com erros, aí sim o *led* vermelho acende (troque o comentário entre as duas linhas assinaladas em vermelho). Nesse caso, ao recolocarmos o cabo, o programa continua.

Aqui cabe uma segunda pergunta. Ao removermos o cabo, causamos a recepção de um caractere errado e o *led* vermelho acende. Em seguida, mesmo sem o cabo, o programa vai transmitir um novo caractere e vai parar esperando a chegada de um caractere. Mas sem o cabo, esse caractere nunca chega. Então, como que o programa volta a rodar quando recolocamos o cabo? É porque a reconexão do cabo provoca rebotes que são interpretados como bits de partida.

Listagem da solução do ER 8.3

```

// ER8.3 USCI_A0
//
// Curto entre P3.3(TXD) e P3.4(RXD)
// Usar polling
// USCI_A0 Transmite e recebe letras de A até Z
// 9.600 bauds (ACLK), 8 bits, sem paridade, 1 stop

#include <msp430.h>

#define TRUE 1

```

```

#define FALSE 0

void pin_config(void);
void USCI_A0_config(void);
void led_VM(void);
void led_vm(void);
void led_VD(void);
void led_vd(void);
void leds_config(void);

int main(void) {
    char letra_tx,letra_rx;
    WDTCTL = WDTPW | WDTHOLD;      // stop watchdog timer
    leds_config();
    pin_config();
    USCI_A0_config();
    letra_tx='A';
    while(TRUE){                  //Laço principal
        while ( (UCA0IFG&UCTXIFG)==0); //Esperar TXIFG=1
        UCA0TXBUF=letra_tx;
        while ( (UCA0IFG&UCRXIFG)==0); //Esperar RXIFG=1
        letra_rx=UCA0RXBUF;

        if (letra_tx == letra_rx){ led_VD(); led_vm(); } //OK
        else                      { led_vd(); led_VM(); } //Erro

        if (++letra_tx == 'Z'+1)   letra_tx='A'; //Próxima letra
    }
    return 0;
}

// Configurar pinos P3.3 (TXD) e P3.4 (RXD)
void pin_config(void){
    P3SEL |= BIT4 | BIT3;
}

// Configurar USCI_A0
void USCI_A0_config(void){
    UCA0CTL1 = UCSWRST;          //RST=1 para USCI_A0
    UCA0CTL0 = 0;                //sem paridade, 8 bits, 1 stop, modo UART
    UCA0BRW = 3;                 // Divisor
    UCA0MCTL = UCBRS_3;          //Modulador = 3 e UCOS=0
    UCA0CTL1 = UCSSEL_1;          //Recusar caractares errados
    //UCA0CTL1 = UCSSEL_1|UCRXIE; //Receber caractares errados
}

// Copiar do ER 8.1 as funções abaixo:
void led_VM(void),void led_vm(void),void led_VD(void),
void led_vd(void) e void leds_config(void)

```

ER 8.4. Este exercício propõe usar as duas instâncias da USCI_A, conectadas como mostrado na Figura 8.38. A USCI_A1 trabalhará como unidade auxiliar, com a simples tarefa de retornar o complemento a 2 de todo número (8 bits com sinal) que receber. Para que fique transparente, esta unidade USCI_A1 deverá operar por interrupção. Assim, a USCI_A0 soma o número enviado ao recebido, e o resultado deve ser zero. Configuração serial: 19.200 bauds, 8 bits, sem paridade em com um bit de parada. Use o modo de super amostragem (UCOS16 = 1) com SMCLK (1.048.576 Hz).

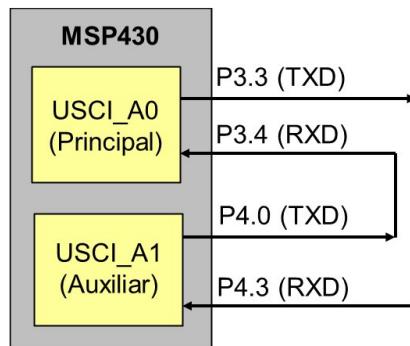


Figura 8.39. Sugestão para conexão entre as duas instâncias da USCI_A com o emprego de dois cabos.

Solução:

Iniciamos com os cálculos para configurarmos o *baud-rate*, que estão apresentados logo abaixo. O único comentário é sobre o arredondamento no cálculo do Modulador de 16 bits (M16). Poderíamos afirmar que o valor 6,56 deveria ser arredondado para 7. Entretanto, se fizermos isso a relação entre o BRCLK16 e o *baud-rate* (BR) fica abaixo de 16.

- $\text{BITCLK16} = 16 \times \text{BITCLK} = 16 \times 19.200 = 307.200 \text{ Hz.}$
- $n = \frac{1.048.576}{307.200} = 3,41 \rightarrow N = 3.$
- $M16 = \text{round}(0,41 \times 16) = \text{round}(6,56) = 6 \rightarrow M16 = 6.$
- Programamos UCBRx = 3 e UCBRFx = 6.
- $\text{BITCLK16} = \frac{1.048.576}{3 + \frac{6}{16}} = 310.689,18 \text{ Hz.}$
- $n = \frac{\text{BITCLK16}}{BR} = \frac{310.689,18}{19.200} = 16,18.$

- $M8 = \text{round}(0,18 \times 8) = \text{round}(1,44) = 1 \rightarrow M8 = 1.$
- Programamos UCBRSx = 1.
- $BR = \frac{310.689,18}{16 + \frac{1}{8}} = 9.619,96 \rightarrow BR = 19.267,54 \text{ Hz}.$

As duas tabelas a seguir apresentam a configuração das duas unidades. Elas são semelhantes. Apenas a unidade USCI_A1 tem habilitada a interrupção por recepção.

Tabela 8.12. Configuração dos Registradores da USCI_A0

	7	6	5	4	3	2	1	0
UCA0CTL0	UCPEN	UCPAR	UCMSB	UC7BIT	UCSPB	UCMODEx		UCSYNC
	0	0	0	0	0	0		0
UCA0CTL1	UCSSEL		UCRXEIE	UCBRKIE	UCDORM	UCTXADDR	UCTXBRK	UCWRST
	2		0	0	0	0	0	1/0
UCA0MCTL	UCBRF				UCBRS			UCOS16
	6				1			1
UCA0STAT	UCLISTEN	UCFE	UCOE	UCPE	UCBRK	UCRXERR	UCADDR/UCIDLE	UCBUSY
	0	0	0	0	0	0	0	0

(UCA0BRW = 3 não está mostrado na tabela)

Tabela 8.13. Configuração dos Registradores da USCI_A1

	7	6	5	4	3	2	1	0
UCA1CTL0	UCPEN	UCPAR	UCMSB	UC7BIT	UCSPB	UCMODEx		UCSYNC
	0	0	0	0	0	0		0
UCA1CTL1	UCSSEL		UCRXEIE	UCBRKIE	UCDORM	UCTXADDR	UCTXBRK	UCWRST
	2		0	0	0	0	0	1/0
UCA1MCTL	UCBRF				UCBRS			UCOS16
	6				1			1
UCA1STAT	UCLISTEN	UCFE	UCOE	UCPE	UCBRK	UCRXERR	UCADDR/UCIDLE	UCBUSY
	0	0	0	0	0	0	0	0
UCA1IE	Reservado						UCTXIE	UCRXIE
	0						0	1

(UCA1BRW = 3 não está mostrado na tabela)

A unidade USCI_A1 não tem pinos exclusivos, entretanto, sua entrada (RXD) e sua saída (TXD) podem ser mapeadas em bits da porta P4. Na LaunchPad temos disponíveis apenas P4.3, 2, 1, 0. Como P4.2 e P4.3 já são configurados para operar com a USCI_B1, preferimos então lançar mão de P4.0 para a transmissão (TXD) e P4.3 para a recepção (RXD). O Capítulo de GPIO explica como mapear a porta P4. O pequeno trecho abaixo realiza esta operação.

```
PMAPKEYID = 0X02D52;      //Liberar mapeamento de P4
P4MAP0 = PM_UCA1TXD;     //P4.0 = TXD
P4MAP3 = PM_UCA1RXD;     //P4.3 = RXD
```

A listagem abaixo apresenta a solução. Ela é semelhante à listagem do ER 8.1. Chamamos a atenção para a configuração da USCI_A1. O mapeamento de P4 foi realizado com a unidade ainda em *reset*. Como ela vai operar por interrupção, sua tarefa é muito simples. Habilitamos apenas a interrupção de recepção. Cada vez que essa interrupção acontece, a rotina de interrupção escreve no UCA1TBUF o complemento de 2 de UCA1RBUF. A leitura de UCA1IV foi colocada apenas para apagar a *flag* de interrupção (RXIFG). Pudemos fazer isso porque temos certeza de que a única interrupção habilitada é a de recepção.

Listagem da solução do ER 8.4

```
// ER8.4 USCI_A0 e USCI_A1
//
// USCI_A0    P3.3 (TXD) ===> P4.3 (RXD) USCI_A1
//                  P3.4 (RXD) <== P4.0 (TXD)
// USCI_A1 retornar complemento a 2 do número que receber
// 115.200 bauds (SMCLK), 8 bits, sem paridade, 1 stop

#include <msp430.h>

#define TRUE 1
#define FALSE 0

void USCI_A0_config(void);
void USCI_A1_config(void);
void led_VM(void);
void led_vm(void);
void led_VD(void);
void led_vd(void);
void leds_config(void);

int main(void) {
    signed char tx=0,rx;
```

```

WDTCTL = WDTPW | WDTHOLD;      // stop watchdog timer
leds_config();
USCI_A0_config();
USCI_A1_config();
__enable_interrupt();
while(TRUE) {                  //Laço principal
    while ( (UCA0IFG&UCTXIFG)==0); //Esperar TXIFG=1
    UCA0TXBUF=tx;                //Transmitir
    while ( (UCA0IFG&UCRXIFG)==0); //Esperar RXIFG=1
    rx=UCA0RXBUF;               //Receber
    if ( (tx+rx) == 0) { led_VD(); led_vm(); } //OK
    else                      { led_vd(); led_VM(); } //Erro
    tx++;
    //if (tx == -128)          tx=0;      //Ver observação no texto
}
return 0;
}

// Interrupção da USCI_A1
//#pragma vector = 46
#pragma vector = USCI_A1_VECTOR
__interrupt void usci_a1_int(void){
    UCA1IV;                     //Apagar RXIFG
    UCA1TXBUF=-UCA1RXBUF;      //Transmitir complemento a 2
}

// Configurar USCI_A0
void USCI_A0_config(void){
    UCA0CTL1 = UCSWRST;        //RST=1 para USCI_A0
    UCA0CTL0 = 0;              //sem paridade, 8 bits, 1 stop, modo UART
    UCA0BRW = 3;               // Divisor
    UCA0MCTL = UCBRF_6 | UCBRS_1 | UCOS16; //Modulador = 3 e UCOS=0
    P3SEL |= BIT4 | BIT3;     //Disponibilizar P3.3 e P3.4
    UCA0CTL1 = UCSSEL_2;       //RST=0 e Selecionar SMCLK
}

// Configurar USCI_A1
void USCI_A1_config(void){
    UCA1CTL1 = UCSWRST;        //RST=1 para USCI_A0
    UCA1CTL0 = 0;              //sem paridade, 8 bits, 1 stop, modo UART
    UCA1BRW = 3;               // Divisor
    UCA1MCTL = UCBRF_6 | UCBRS_1 | UCOS16; //Modulador = 3 e UCOS=0
    P4SEL |= BIT3 | BIT0;     //Disponibilizar P4.3 e P4.0
    PMAPKEYID = 0X02D52;      //Liberar mapeamento de P4
    P4MAP0 = PM_UCA1TXD;      //P4.0 = TXD
    P4MAP3 = PM_UCA1RXD;      //P4.3 = RXD
    UCA1CTL1 = UCSSEL_2;       //RST=0 e Selecionar SMCLK
    UCA1IE = UCRXIE;          //Interrupção por recepção
}

```

```
// Copiar do ER 8.1 as funções abaixo:  
void led_VM(void),void led_vm(void),void led_VD(void),  
void led_vd(void) e void leds_config(void)
```

Vamos aproveitar este programa para indicar algumas armadilhas que pegam programadores desatentos. Ao rodar o programa, o usuário vai perceber que o *led* vermelho apresenta umas piscadas rápidas. Isso é indicação de erro. Será que o leitor consegue explicar o porquê. Use um *break point* para consultar o valor das variáveis *tx* e *rx* quando acontece o erro. Ainda mais, verifique que se a linha marcada em vermelho for usada (remover o indicador de comentário “//”) o erro não acontece.

Outra armadilha mais. As variáveis *tx* e *rx* são do tipo *char* com sinal. Tente remover o modificador *signed* de sua declaração. O que aconteceu?

ER 8.5. O HC-05, muito comum no mercado, é um módulo Bluetooth com entrada serial. Ele tem 6 terminais, dos quais usaremos apenas 4, como mostrado na figura abaixo. Este exercício pede o uso deste módulo para enviar dados para o PC. Isto pode ser muito útil em diversos outros experimentos. Para este exercício, cuja finalidade é apenas comprovar o funcionamento, envie apenas as letras de A até Z, repetidamente. O leitor vai precisar de um monitor serial. Sugestões: Monitor serial do Arduino, Teraterm, [Café](#), [conhece algum outro mais?](#)

Ante de rodar o programa, será necessário parear seu computador com o HC-05. A senha mais comum é “1234”, porém, se esta não funcionar tente “0000”.

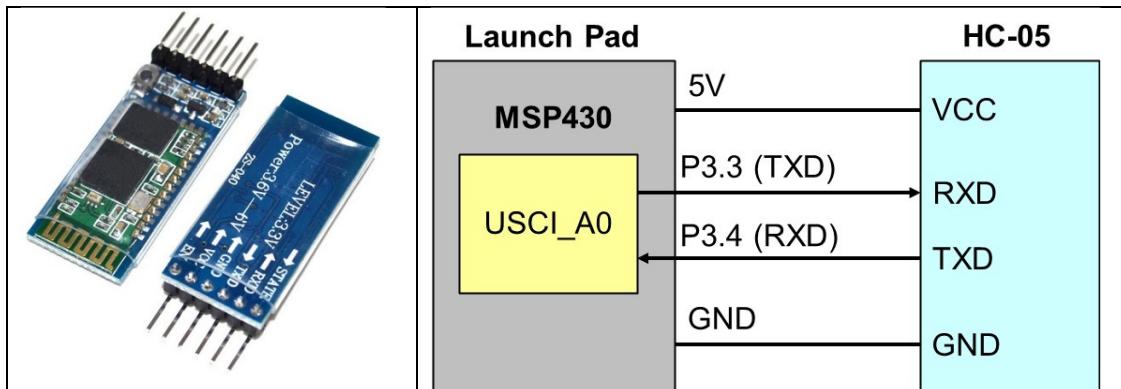


Figura 8.40. Foto do Módulo Bluetooth HC-05 e sua conexão com a LaunchPad.

Solução:

O HC-05 é um dispositivo muito versátil, ele pode operar como mestre ou como escravo numa rede Bluetooth. De fábrica ele vem programado para operar como escravo com a porta serial em 9.600 bauds, 8 bits de dados e sem paridade. A entrada de VCC (5 V) alimenta um regulador interno de 3,3 V. Todo restante da placa funciona com 3,3 V, assim, suas linhas de transmissão (TXD) e de recepção (RXD) podem ser ligadas diretamente à LaunchPad.

Uma vez pareado com o computador, o HC-05 envia por Bluetooth tudo que recebe por sua entrada serial e, por outro lado, ele envia por sua saída serial tudo que recebe por Bluetooth. Em suma, ele funciona como um transceptor serial/Bluetooth totalmente transparente.

A listagem do programa solução está apresentada logo a seguir. Ela é muito simples e dispensa maiores comentários. Note que após o envio da letra “Z”, o programa envia o caractere de nova linha ('\n'). Assim, cada linha do terminal serial apresenta as letras de “A” até “Z”.

Listagem da solução do ER 8.5

```
// ER8.5 USCI_A0 com HC-05
//
// HC-05: VCC = +5 V
// HC-05: GND = GND
// HC-05: RXD = P3.3 (USCI_A0: TXD)
// HC-05: TXD = P3.4 (USCI_A0: RXD)
// USCI_A0 Transmite letras de A até Z
// 9.600 bauds (ACLK), 8 bits, sem paridade, 1 stop

#include <msp430.h>

#define TRUE 1

void USCI_A0_config(void);

int main(void){
    char letra='A';
    WDTCTL = WDTPW | WDTHOLD;      // stop watchdog timer
    USCI_A0_config();
    while(TRUE){                   //Laço principal
        while ( (UCA0IFG&UCTXIFG)==0); //Esperar TXIFG=1
        UCA0TXBUF=letra++;          //Próxima letra
        if (letra == 'Z'+1)   letra='\n'; //Nova linha
        if (letra == '\n'+1)  letra='A'; //Volta ao 'A'
    }
    return 0;
}
```

```
// Configurar USCI_A0
void USCI_A0_config(void){
    UCA0CTL1 = UCSWRST;          //RST=1 para USCI_A0
    UCA0CTL0 = 0;                //sem paridade, 8 bits, 1 stop, modo UART
    UCA0BRW = 3;                 //Divisor
    UCA0MCTL = UCBRS_3;          //Modulador = 3 e UCOS=0
    P3SEL |= BIT4 | BIT3;         //Disponibilizar pinos
    UCA0CTL1 = UCSSEL_1;          //RST=0, ACLK
}
```

Para podermos apreciar o resultado do programa, é necessário o uso de um Terminal Serial. A sugestão é pelo monitor serial do Arduino. Uma outra opção é o uso do TeraTerm. A Figura 8.41 apresenta o resultado com o emprego de cada um deles. Todo dispositivo Bluetooth pareado com um computador é usado sob a forma de uma porta serial. Assim, após o pareamento com o HC-05, é preciso descobrir o número da porta serial que foi designada no seu computador. São usadas as denominações “COM” seguidas por um número. No caso deste exemplo, foi usada a COM8, entretanto o pareamento resultou em COM8 e COM9. **Ainda não pesquisamos o porquê, mas cada vez que um HC-05 é pareado, surgem duas portas seriais, sendo que somente um funciona.**

Um ponto merece comentário. O programa envia apenas o caractere de nova linha ('\n'). Alguns terminais pedem também o caractere de retorno do carro ('\r'). É possível programar como o terminal vai interpretar o caractere de nova linha. No caso do TeraTerm foi necessário acessar a opção “Setup, Terminal, LF”. É importante comentar que no PC, a velocidade (*baud-rate*) do terminal serial pouco importa, pois os dados trafegam pelo Bluetooth e a porta serial é apenas uma simulação conveniente.

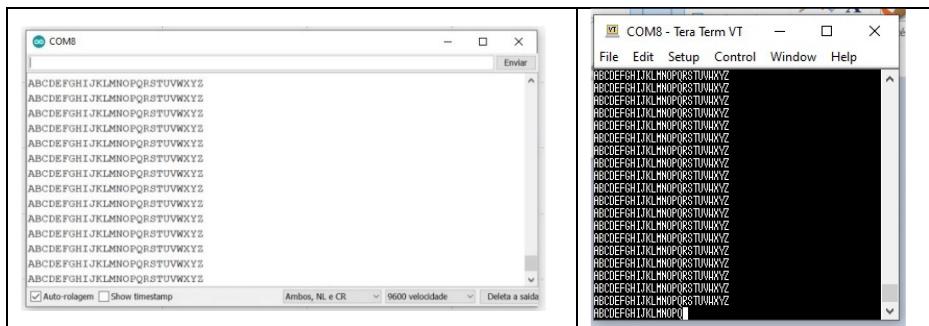


Figura 8.41. Resultado do exercício. No lado esquerdo foi usado o monitor serial do Arduino e na direita o TeraTerm.

ER 8.6. Após executar o código do ER 8.5, temos a certeza de conseguimos conectar a LaunchPad com o HC-05. Vamos agora a uma comunicação um pouco mais sofisticada.

Como desafio, este programa pede uma comunicação nos dois sentidos. O leitor vai escolher 5 frases famosas, numeradas de 1 até 5. Cada vez que o leitor digitar um número em seu terminal, a frase de número correspondente será impressa. O caractere “?” resulta na impressão da quantidade de frases disponíveis.

Solução:

A solução deste exercício é relativamente simples e sua listagem está apresentada logo abaixo. Vamos começar com alguns detalhes da linguagem C. A constante `TOTAL` especifica a quantidade de frases disponíveis. O leitor pode acrescentar mais frases e alterar seu valor. Pergunta: o que acontece se `TOTAL` for maior que 9? A melhor forma de se arrumar as frases (*strings*) é usando um vetor de ponteiros. Veja a declaração da variável `char *frase[]`. O Apêndice C oferece mais detalhes sobre essa solução. Aqui, basta informar que o compilador armazena essas frases em algum lugar na memória de programa (*flash*) e grava no vetor de ponteiros o início de cada uma. Usando os recursos do CCS, o leitor é convidado a procurar por essas frases (tente 0x4400). As frases estão sem os acentos porque, em geral, os monitores seriais não os aceitam.

Duas mensagens são importantes. Uma é a mensagem “Opcao nao valida”. Ela é acessada com o ponteiro da posição zero do vetor `frase`. A segunda mensagem depende do total de frases. Veja no corpo do programa a variável `char msg[]` que é inicializada com a *string* "Opcoes de 1 ate X.\n". Esse letra 'X', durante a execução do programa é alterada em função do `TOTAL` de opções do programa. Para viabilizar esta alteração, a mensagem foi colocada no corpo do programa e, assim, é criada na memória RAM.

Quando se usa um terminal serial, o que é digitado só enviado após o usuário teclar ENTER. Quando isso acontece, o monitor envia pela porta serial o texto que o usuário digitou. De acordo com a configuração do monitor, após o texto o monitor acrescenta o caractere de nova linha ('\\n' = 0xA) e o retorno do carro ('\\r' = 0xD). Sabendo disso, foi criada a função `char bt_ler(void)` que joga fora os caracteres '\\n' e '\\r' que, porventura, tenham sido recebidos e retorna apenas o próximo byte. Essa função atende apenas à necessidade do atual programa. Ela é muito simples e limitada. Por exemplo, tente enviar “123”, o programa deveria imprimir as frases 1, 2 e a 3. Porém isso não acontece. Por quê?

Para imprimir na porta serial, foi criada a função básica `void bt_char(char c)`. Ela espera a autorização (UCTXIFG = 1) para escrever no *buffer* serial (UCA0TXBUF) e envia o caractere que recebeu como argumento. Aproveitando essa função, foi preparada a função `void bt_str(char *vet)`, que é responsável por enviar uma *string* pela porta serial. Essas funções começam com `bt_` para indicar que fazem uso do Bluetooth.

Fica a sugestão para o leitor criar diversas outras funções, como para imprimir números em decimal, hexadecimal etc. Todas elas devem tomar coma base a função `bt_char()`.

Listagem da solução do ER 8.6

```
// ER8.6 USCI_A0 com HC-05 - Brincadeira com Frases
//
// HC-05: VCC = +5 V
// HC-05: GND = GND
// HC-05: RXD = P3.3 (USCI_A0: TXD)
// HC-05: TXD = P3.4 (USCI_A0: RXD)
// USCI_A0 Recebe números e envia frases
// 9.600 bauds (ACLK), 8 bits, sem paridade, 1 stop

#include <msp430.h>

#define TRUE 1
#define FALSE 0
#define TOTAL 5      //Quantidade de frases

char bt_ler(void);
void bt_str(char *vet);
void bt_char(char c);
void USCI_A0_config(void);

char *frase[]={
    "Opcão não valida.\n",                                //0
    "O otimista não sabe o que o espera.\n",              //1, Millor
    "De onde menos se espera, dai eh que não sai nada.\n", //2, B. de Itararé
    "Em terra de saci, todo chute eh uma voadora.\n",     //3, Ditado pop.
    "Eu pretendo viver para sempre, ou morrer tentando.\n", //4, Groucho Marx
    "A mao que joga a pedra, eh a mesma que apedreja.\n"}; //5, Falcão

int main(void){
    char opc;
    char msg[]{"Opcoes de 1 ate h X.\n"};           //Mensagem de limite
    WDTCTL = WDTPW | WDTHOLD;                      // stop watchdog timer
    msg[17]=0x30+TOTAL;                            //Limite de escolhas
    USCI_A0_config();
    while(TRUE){
        bt_str("\n>> Digite um numero ou '?'.");
        opc=bt_ler();
        bt_str(" [");  bt_char(opc);  bt_str("]\n");    //Indicar escolha
        if (opc=='?')  bt_str(msg);                  //Escolha = ?
        else{
            opc = opc-0x30;                      //Foi número?
            if ( opc>0 && opc<(TOTAL+1))  bt_str(frase[opc]);
            else bt_str(frase[0]);
        }
    }
}
```

```

        }
    }
    return 0;
}

// Receber um caractere serial. Elimina '\n' ou '\r'
// Prende esperando chegar algo
char bt_ler(void){
    char c='\n';
    while (c=='\n' || c=='\r'){           //Consumir '\n' e '\r'
        while ( (UCA0IFG&UCRXIFG)==0); //Esperar RXIFG=1
        c=UCA0RXBUF;                  //Ler buffer
    }
    return c;
}

// Enviar uma string pela serial
void bt_str(char *vet){
    unsigned int i=0;
    while (vet[i] != '\0')
        bt_char(vet[i++]);
}

// Enviar um caracter pela serial
void bt_char(char c){
    while ( (UCA0IFG&UCTXIFG)==0); //Esperar TXIFG=1
    UCA0TXBUF=c;
}

void USCI_A0_config(void){...} //Copiar do ER 8.5

```

ER 8.7. O dispositivo Bluetooth HC-05 pode ser configurado. É possível alterar a velocidade de sua porta serial, alterar seu nome, senha e vários outros parâmetros. Isto é feito com os “Comandos AT”. Abaixo estão alguns exemplos. É simples colocar o HC-05 no modo de configuração. Note um pequeno botão, bem próximo aos terminais. Se o HC-05 foi energizado com esse pequeno botão pressionado, ele entra neste modo de configuração, sendo que o *baud-rate* passa a ser de 38.400 bauds. Então, pressione primeiro este botão e só depois faça a conexão do VCC.

Sugestões para obter documento sobre os Comandos AT para o HC-05:

- <https://electrosome.com/wp-content/uploads/2018/01/Bluetooth-AT-Commands.pdf>
- http://www.liotux.ch/arduino/HC-0305_serial_module_AT_command_set_201104_revised.pdf

O presente exercício pede que se coloque o HC-05 no modo de Comandos AT e que se prove alguns comandos, como os listados abaixo. Busque na Internet a lista completa

desses comandos AT. Ela é bem grande. O leitor pode fazer algumas alterações interessantes, como mudar o *baud-rate* para uma frequência mais alta.

Tabela 8.14. Exemplo de alguns comandos AT e respostas do HC-05

Comando	Resposta
AT	OK
AT+UART?	+UART:9600,0,0 OK
AT+UART=115200,0,0 (erro: no doc pdf tem ";" a mais)	OK
AT+PSWD?	+PIN:"1234" OK
AT+VERSION?	VERSION:3.0-20170601 OK
AT+ADDR? (endereço ?Bluetooth)	+ADDR:98D3:31:F5B0E1 OK
AT+NAME?	+NAME:HC-05
AT+NAME=NOVO NOME	OK
AT+NAME?	+NAME:NOVO NOME

Solução:

Configurar o HC-05 com Comandos AT é, algumas vezes, um pouco trabalhoso, pois ele deixa de operar como Bluetooth. O que pretendemos oferecer com este exercício é uma forma simples e confortável de se executar esta tarefa. Entretanto, vamos precisar de dois HC-05, como mostrado na Figura 8.42.

HC-05 COM → faz a comunicação (Bluetooth) com o Terminal Serial que está no PC e
HC-05 AT → unidade a ser configurada com os comandos AT.

Seguindo este esquema, o programador pode, confortavelmente, digitar os comandos AT no seu PC (terminal serial) e nele receber as respostas. A USCI_A0, configurada para 9.600 bauds “conversa” com o PC e a USCI_A1, configurada para 38.400 bauds, recebe os comandos do MSP e envia suas respostas.

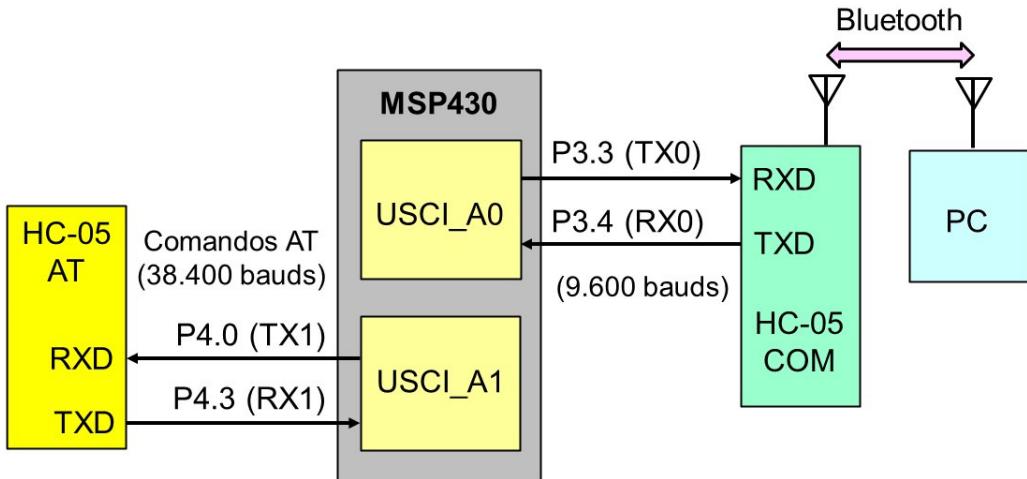


Figura 8.42. Sugestão para enviar Comandos AT para uma HC-05 enquanto se usa uma outra HC-05 para se comunicar, via Bluetooth, com o terminal serial o PC.

Neste exercício, a tarefa do MSP430 é muito simples, um mero atravessador. O MSP deve enviar por TX1 tudo que receber por RX0 e, por outro lado, enviar por TX0 tudo que receber por RX1. Porém, as velocidades são diferentes, o que pode resultar num problema ao receber (RX1) a 38.400 bauds e enviar (TX0) a 9.600 bauds. Fica clara a necessidade de um *buffer* em memória pois, neste caso, os dados chegam mais rápidos do que saem.

Vamos então aproveitar a oportunidade deste exercício e fazer uma estrutura do fluxo de dados. Toda recepção de dados será por interrupção e fazendo uso de uma fila circular. Já as transmissões, como são coordenadas pelo MSP e não precisam deste recurso. A Figura 8.43 apresenta a sugestão para o fluxo de dados. O Exercício Resolvido C.1, Apêndice C, trata da criação e operação de filas circulares. Seria interessante que o leitor o estudasse, em especial o ER C.1.

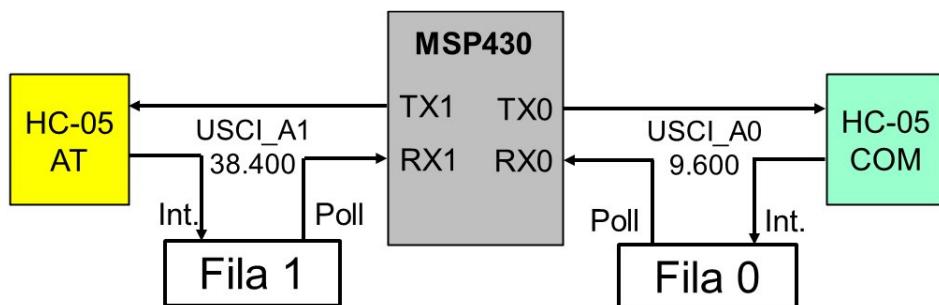


Figura 8.43. Fluxo dos dados na solução deste exercício. Note que toda recepção faz uso de filas circulares.

Como o leitor pode ver na figura acima, são duas filas circulares que são preenchidas por interrupção. Já a retirada dos dados é feita por *polling*. Na listagem abaixo o leitor pode ver as funções para cada uma das filas. Note que cada função que retira dado da fila, desabilita as interrupções antes de operar os ponteiros da fila. Isto evita o perigo de uma interrupção também alterar o mesmo ponteiro. Caso a fila fique cheia, o programa sinaliza erro acendendo o *led* vermelho e fica preso num laço infinito. A Figura 8.44 apresenta o uso do monitor serial do Arduino para exibir as respostas do HC-05 aos comandos AT. Neste caso, os comandos devem ser digitados na linha superior deste monitor e serem seguidos pela tecla <ENTER>.

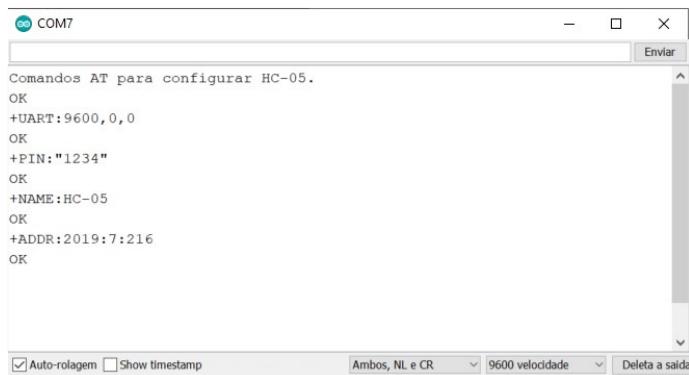


Figura 8.44. Respostas da HC-05 aos comandos AT. No caso foi usado o monitor serial do Arduino. A porta usada, “COM7”, é alterada de acordo com o computador.

Listagem da solução do ER 8.7

```
// ER8.7 Enviar Comandos AT para HC-05
//
// USCI_A0    P3.3 (TXD) ==> HC-05 ==> Terminal
// 9.600      P3.4 (RXD) <== COM <== Serial
//
// USCI_A1    P4.0 (TXD) ==> HC-05
// 38.400     P4.3 (RXD) <== CMDOS AT

#include <msp430.h>

#define TRUE      1
#define FALSE     0
#define FILA0_TAM 128 //Tamanho da fila 0
#define FILA1_TAM 128 //Tamanho da fila 1

void fila0_inic(void);
char fila0_poe(char dado);
char fila0_tira(char *pt);
void fila1_inic(void);
char fila1_poe(char dado);
```

```

char fila1_tira(char *pt);
void USCI_A0_config(void);
void USCI_A1_config(void);
void bt_str(char *vet);
void bt_char(char c);
void led_VM(void);
void led_vm(void);
void led_VD(void);
void led_vd(void);
void leds_config(void);

// Variáveis para as filas
volatile char fila0[FILA0_TAM],fila1[FILA1_TAM]; //Buffers para as filas
volatile unsigned char pin0,pout0; //Ponteiros fila0
volatile unsigned char pin1,pout1; //Ponteiros fila1
volatile unsigned char vet[100],ix;

int main(void){
    char x;
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer
    leds_config();
    USCI_A0_config();
    USCI_A1_config();
    bt_str("Comandos AT para configurar HC-05.\n"); //Boas vindas
    fila0_inic();
    fila1_inic();
    __enable_interrupt();
    while(TRUE){ //Laço principal
        if ((UCA1IFG&UCTXIFG) == UCTXIFG){ //TX1 ocioso?
            if (fila0_tira(&x) == TRUE) //Tirar da Fila 0
                UCA1TXBUF=x; //Enviar
        }

        if ((UCA0IFG&UCTXIFG) == UCTXIFG){ //TX0 ocioso?
            if (fila1_tira(&x) == TRUE) //Tirar da Fila 1
                UCA0TXBUF=x; //Enviar
        }
    }
    return 0;
}

// Interrupção da USCI_A0
//#pragma vector = 56
#pragma vector = USCI_A0_VECTOR
__interrupt void usci_a0_int(void){
    UCA0IV; //Apagar RXIFG
    fila0_poe(UCA0RXBUF); //Por dado na fila
}

// Interrupção da USCI_A1

```

```

//#pragma vector = 46
#pragma vector = USCI_A1_VECTOR
__interrupt void usci_a1_int(void){
    UCA1IV;                      //Apagar RXIFG
    fila1_poe(UCA1RXBUF);        //Por dado na fila
}

////////////////// FILA 0 ///////////////////////////////
// Inicializar fila 0
void fila0_inic(void){
    pin0=1;
    pout0=0;
}

// Colocar dado na fila 0
// Retorna TRUE se conseguiu colocar na fila
// Retorna FALSE se a fila está cheia
char fila0_poe(char dado){
    if (pin0 == pout0){
        led_VM();
        while(TRUE);           //Fila cheia
    }
    else{
        vet[ix++]=dado;
        fila0[pin0++]=dado;   //Colar dado na fila
        if (pin0==FILA0_TAM)  pin0=0; //Fim da fila?
        return TRUE;
    }
}

// Retirar um dado da fila 0
// Retorna TRUE se conseguiu retirar *pt=dado
// Retorna FALSE se a fila está vazia
char fila0_tira(char *pt){
    unsigned int pout_aux;
    __disable_interrupt();
    pout_aux=pout0+1;
    if (pout_aux==FILA0_TAM)  pout_aux=0; //Fim da fila
    if (pout_aux == pin0){
        __enable_interrupt();
        return(FALSE);         //Fila vazia
    }
    else{
        *pt=fila0[pout_aux]; //Retirar dado da fila
        pout0=pout_aux;      //Atualizar ponteiro pout
        __enable_interrupt();
        return TRUE;
    }
}

```

```

////////////////// FILA 1 ///////////////////////////////
// Inicializar fila 1
void fila1_inic(void){
    pin1=1;
    pout1=0;
}

// Colocar dado na fila 1
// Retorna TRUE se conseguiu colocar na fila
// Retorna FALSE se a fila está cheia
char fila1_poe(char dado){
    if (pin1 == pout1){
        led_VM();
        while(TRUE);                                //Fila cheia
    }
    else{
        fila1[pin1++]=dado;                      //Colar dado na fila
        if (pin1==FILA1_TAM)  pin1=0;              //Fim da fila?
        return TRUE;
    }
}

// Retirar um dado da fila 1
// Retorna TRUE se conseguiu retirar *pt=dado
// Retorna FALSE se a fila está vazia
char fila1_tira(char *pt){
    unsigned int pout_aux;
    __disable_interrupt();
    pout_aux=pout1+1;
    if (pout_aux==FILA1_TAM)  pout_aux=0;      //Fim da fila
    if (pout_aux == pin1){
        __enable_interrupt();
        return(FALSE);                          //Fila vazia
    }
    else{
        *pt=fila1[pout_aux];   //Retirar dado da fila
        pout1=pout_aux;          //Atualizar ponteiro pout
        __enable_interrupt();
        return TRUE;
    }
}

// Configurar USCI_A0 em 9.600
void USCI_A0_config(void){
    UCA0CTL1 = UCSWRST;           //RST=1 para USCI_A0
    UCA0CTL0 = 0;                 //sem paridade, 8 bits, 1 stop, modo UART
    UCA0BRW = 6;                  //Divisor
    UCA0MCTL = UCBRF_13 | UCOS16; //Modulador = 13 e UCOS16=1
    P3SEL |= BIT4 | BIT3;         //Disponibilizar pinos
    UCA0CTL1 = UCSSEL_2;          //RST=0, SMCLK
}

```

```

        UCA0IE = UCRXIE;           //Interrupção por recepção
    }

// Configurar USCI_A1 em 38.400
void USCI_A1_config(void){
    UCA1CTL1 = UCSWRST;       //RST=1 para USCI_A0
    UCA1CTL0 = 0;              //sem paridade, 8 bits, 1 stop, modo UART
    UCA1BRW = 27;              // Divisor
    UCA1MCTL = UCBRS_2;        //Moduladores 2, UCOS16=0
    P4SEL |= BIT3 | BIT0;      //Disponibilizar P4.3 e P4.0
    PMAPKEYID = 0X02D52;       //Liberar mapeamento de P4
    P4MAP0 = PM_UCA1TXD;       //P4.0 = TXD
    P4MAP3 = PM_UCA1RXD;       //P4.3 = RXD
    UCA1CTL1 = UCSSEL_2;       //RST=0 e Selecionar SMCLK
    UCA1IE = UCRXIE;          //Interrupção por recepção
}

void bt_str(char *vet){...}   //Copiar do ER 8.6
void bt_char(char c)  {...}  //Copiar do ER 8.6

// Copiar do ER 8.1 as funções abaixo:
void led_VM(void),void led_vm(void),void led_VD(void),
void led_vd(void) e void leds_config(void)

```

ER 8.8. Para apresentar mais um exemplo do uso da UART, conectamos agora um dispositivo GPS serial. Vamos usar o GPS GY-NEO6MV2 que é muito comum, barato e fácil de ser comprado no mercado doméstico. Sua conexão é muito simples e uma sugestão para uso com a USCI_A0 está apresentada na Figura 8.45. Ele usa os 5V para alimentar um regulador interno de 3,3 V que fornece tensão para toda a placa. Assim, ela pode ser conectada diretamente ao MSP430. Esse módulo opera a 9.600 bauds, 8 bits de dados, sem paridade. As informações de localização são enviadas por mensagens de texto.

Pedido: usando uma solução semelhante à do ER 8.7, escreva um programa que exiba num terminal serial do PC todas as mensagens de texto geradas pelo GY-NEO6MV2.

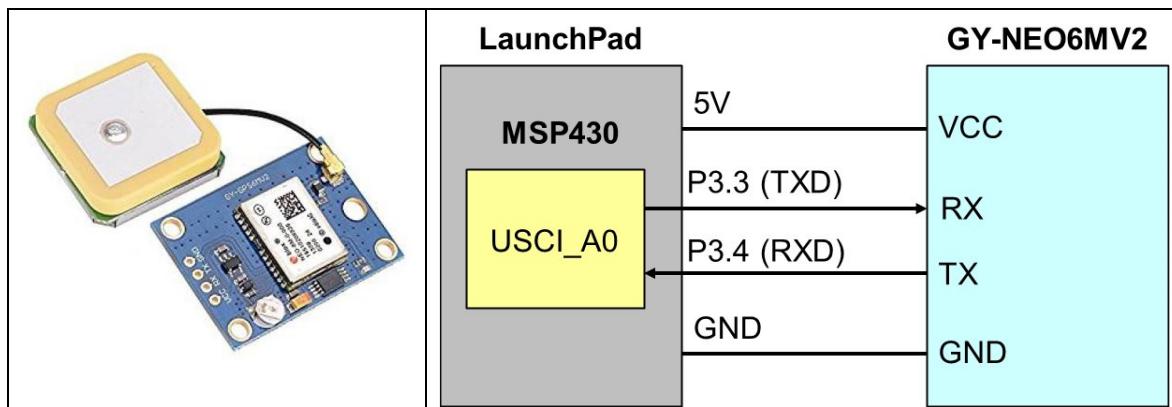


Figura 8.45. Foto do Módulo GPS GY-NEO6MV2 e uma sua conexão com a LaunchPad, neste caso usando a USCI_A0.

Solução:

A solução é muito semelhante à do exercício anterior e a conexão sugerida está apresentada na Figura 8.46. Agora, as duas portas seriais operam em 9.600 bauds e existe apenas uma direção para os dados que é do módulo GPS para o módulo Bluetooth. Assim, vamos usar apenas uma fila circular. A conexão do pino P4.0 com o RX do módulo GPS foi indicada apenas para o caso de o leitor precisar enviar alguma mensagem de configuração para o GPS. Ela não será usada neste exercício. De forma bem simples, o MSP envia pela USCI_A0 tudo o que recebe pela USCI_A1. A listagem abaixo indica a solução.

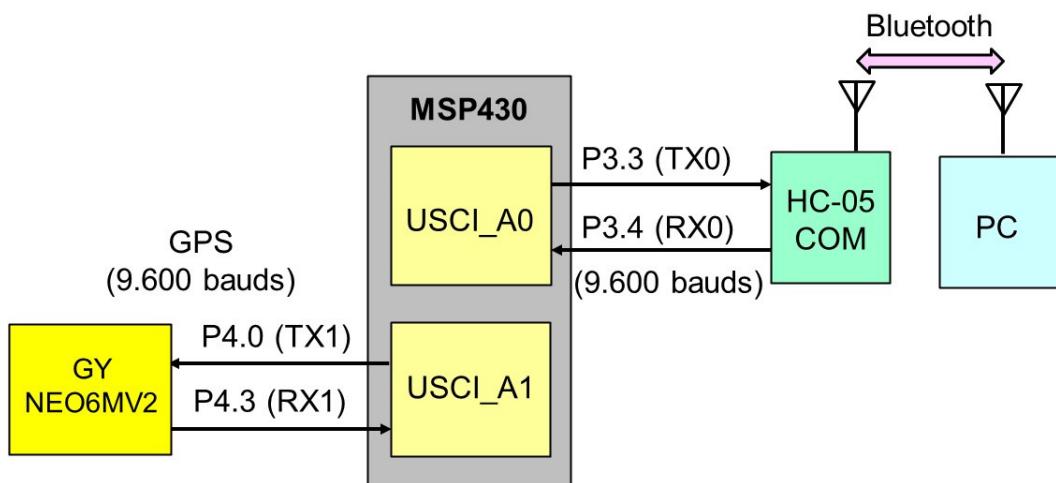


Figura 8.46. Conexões dos módulos GPS e Bluetooth de forma a atender ao pedido do presente exercício resolvido.

Assim que o programa começa a rodar, ele imprime no terminal serial a mensagem Receber dados do GPS GY-NEO6MV2. Essa mensagem serve apenas para confirmar o funcionamento do Bluetooth. Depois, passa a imprimir tudo que recebe do módulo GPS. É claro que o módulo GPS precisa se sincronizar com os satélites e, dependendo do local, pode demorar bastante ou nunca acontecer. Um led começa a piscar quando o módulo GPS se sincroniza com os satélites. Por exemplo, se o leitor estiver trabalhando dentro de casa, é provável que o GPS nunca se sincronize com os satélites. Pode ser que funcione melhor próximo de uma janela. O ideal é num ambiente aberto.

Abaixo está o que se espera receber no Terminal Serial. As linhas em branco foram adicionadas para facilitar a leitura. Note que são várias mensagens, cada uma iniciando com o cifrão (\$). A atualização é feita uma vez por segundo. A interpretação dessas mensagens pode ser conseguida no *link* abaixo.

https://www.u-blox.com/sites/default/files/products/documents/u-blox6_ReceiverDescrProtSpec_%28GPS.G6-SW-10018%29_Public.pdf

No caso deste exercício, o ensaio foi realizado no dia 13/08/2020 por volta das 15 horas. Em vermelho estão marcados a hora (18:14:29 GMT) e a data. A latitude e a longitude estão em negrito (1548.62002, S, 04748.65167, W). A letra “A”, que aparece logo após a mensagem \$GPGSA é muito importante pois indica que o GPS localizou uma quantidade suficiente de satélites. Uma letra “V” nesta posição, indica que não localizou os satélites. Ainda na mensagem \$GPGSA os números 3, 15, 24, 17, 19, 3 são os identificadores dos satélites, ou seja, significa que o GPS localizou 6 satélites. Entretanto, a mensagem \$GPGGA indica que está usando apenas 5 satélites. Note que esta mensagem \$GPGGA também exibe as coordenadas.

Exemplo das mensagens que se espera receber do módulo GPS GY-NEO6MV2

```
$GPRMC,181429.00,A,1548.62002,S,04748.65167,W,0.065,,130820,,,A*71
$GPVTG,,T,,M,0.065,N,0.120,K,A*23
$GPGGA,181429.00,1548.62002,S,04748.65167,W,1,05,7.30,1027.9,M,-11.8,M,,*4F
$GPGSA,A,3,15,24,17,19,13,,,,,,11.62,7.30,9.03*36
$GPGSV,3,1,12,02,18,001,,05,09,336,,06,31,044,,07,04,061,*78
$GPGSV,3,2,12,12,14,269,24,13,71,295,35,15,41,250,38,17,54,144,31*73
$GPGSV,3,3,12,19,74,134,25,24,12,220,27,28,21,141,20,30,29,077,12*74
$GPGLL,1548.62002,S,04748.65167,W,181429.00,A,A*63

$GPRMC,181430.00,A,1548.62003,S,04748.65167,W,0.106,,130820,,,A*7C
$GPVTG,,T,,M,0.106,N,0.196,K,A*2A
$GPGGA,181430.00,1548.62003,S,04748.65167,W,1,05,7.30,1028.0,M,-11.8,M,,*40
$GPGSA,A,3,15,24,17,19,13,,,,,,11.61,7.30,9.03*35
$GPGSV,3,1,12,02,18,001,,05,09,336,,06,31,044,,07,04,061,*78
$GPGSV,3,2,12,12,14,269,24,13,71,295,35,15,41,250,38,17,54,144,31*73
$GPGSV,3,3,12,19,74,134,25,24,12,220,27,28,21,141,20,30,29,077,12*74
$GPGLL,1548.62003,S,04748.65167,W,181430.00,A,A*6A

$GPRMC,181431.00,A,1548.62003,S,04748.65167,W,0.126,,130820,,,A*7F
$GPVTG,,T,,M,0.126,N,0.234,K,A*23
```

```
$GPGGA,181431.00,1548.62003,S,04748.65167,W,1,05,7.30,1028.0,M,-11.8,M,,*41
$GPGSA,A,3,15,24,17,19,13,,,,,,11.61,7.30,9.03*35
$GPGSV,3,1,12,02,18,001,,05,09,336,,06,31,044,,07,04,061,*78
$GPGSV,3,2,12,12,14,269,24,13,71,295,35,15,41,250,38,17,54,144,31*73
$GPGSV,3,3,12,19,74,134,25,24,12,220,27,28,21,141,20,30,29,077,14*72
$GPGLL,1548.62003,S,04748.65167,W,181431.00,A,A*6B

$GPRMC,181432.00,A,1548.62001,S,04748.65169,W,0.031,,130820,,,A*77
$GPVTG,,T,,M,0.031,N,0.057,K,A*23
$GPGGA,181432.00,1548.62001,S,04748.65169,W,1,05,7.30,1027.9,M,-11.8,M,,*48
$GPGSA,A,3,15,24,17,19,13,,,,,,11.61,7.30,9.03*35
$GPGSV,3,1,12,02,18,001,,05,09,336,,06,31,044,,07,04,061,*78
$GPGSV,3,2,12,12,14,269,24,13,71,295,35,15,41,250,38,17,54,144,31*73
$GPGSV,3,3,12,19,74,134,25,24,12,220,27,28,21,141,20,30,29,077,15*73
$GPGLL,1548.62001,S,04748.65169,W,181432.00,A,A*64
```

Com este esquema de comunicação via Bluetooth, podemos fazer uso do programa **u-center**. A Figura 8.47 apresenta uma captura de tela deste programa. Ele oferece diversas ferramentas para interpretação das mensagens e para localização. O leitor interessado em geolocalização vai considerá-lo muito útil. Ele é gratuito e pode ser baixado no link a seguir.

<https://www.u-blox.com/en/product/u-center>

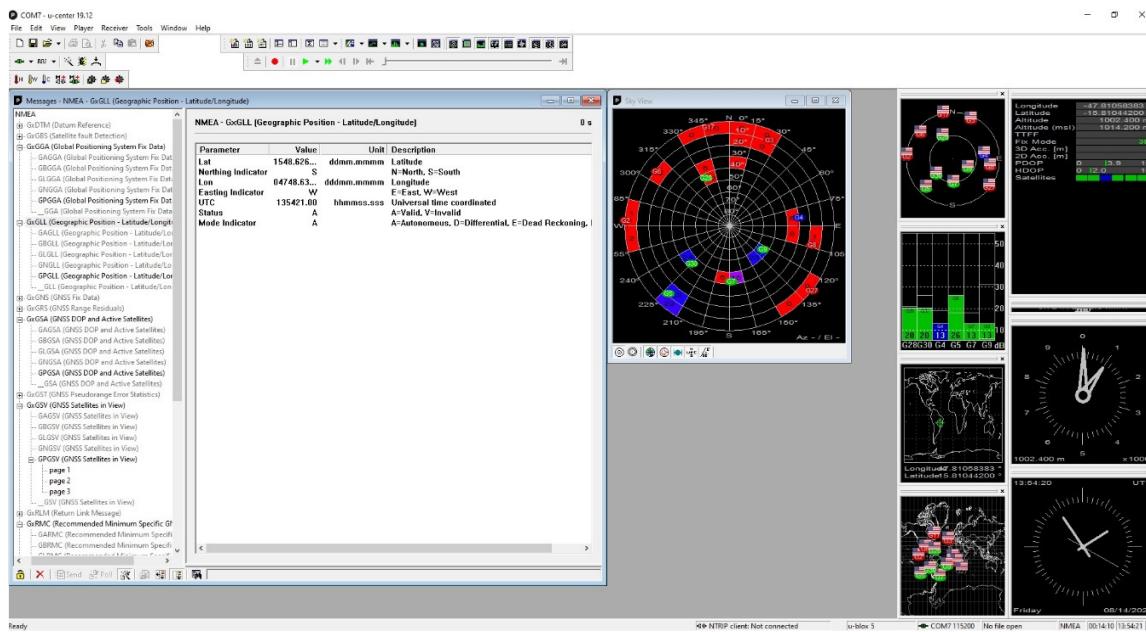


Figura 8.47. Um instantâneo da programa u-center.

Listagem da solução do ER 8.8

Ricardo Zelenovsky, Daniel Café e Eduardo Peixoto

```

// ER8.8 Enviar para o PC os dados do GPS GY-NEO6MV2
//
// USCI_A0    P3.3 (TXD) ==> (RX) HC-05
// 9.600      P3.4 (RXD) <== (TX)
//
// USCI_A1    P4.0 (TXD) ==> (RX) GPS GY-NEO6MV2
// 9.600      P4.3 (RXD) <== (TX)

#include <msp430.h>

#define TRUE      1
#define FALSE     0
#define FILA0_TAM 128 //Tamanho da fila 0
#define FILA1_TAM 128 //Tamanho da fila 1

void fila1_inic(void);
char fila1_poe(char dado);
char fila1_tira(char *pt);
void USCI_A0_config(void);
void USCI_A1_config(void);
void bt_str(char *vet);
void bt_char(char c);
void led_VM(void);
void led_vm(void);
void led_VD(void);
void led_vd(void);
void leds_config(void);

// Variáveis para as filas
volatile char fila1[FILA1_TAM];      //Buffers para as filas
volatile unsigned char pin1,pout1;    //Ponteiros fila1

int main(void) {
    char x;
    WDTCTL = WDTPW | WDTHOLD;      // stop watchdog timer
    leds_config();
    USCI_A0_config();
    USCI_A1_config();
    bt_str("Receber dados do GPS GY-NEO6MV2.\n");
    fila1_inic();
    __enable_interrupt();
    while(TRUE){                  //Laço principal
        if ( (UCA0IFG&UCTXIFG) == UCTXIFG){ //TX0 ocioso?
            if (fila1_tira(&x) == TRUE)       //Tirar da Fila 1
                UCA0TXBUF=x;                 //Enviar
        }
    }
    return 0;
}

```

```

// Interrupção da USCI_A1
//#pragma vector = 46
#pragma vector = USCI_A1_VECTOR
__interrupt void usci_a1_int(void){
    UCA1IV;                      //Apagar RXIFG
    fila1_poe(UCA1RXBUF);        //Por dado na fila
}

// Configurar USCI_A1 em 38.400
void USCI_A1_config(void){
    UCA1CTL1 = UCSWRST;          //RST=1 para USCI_A0
    UCA1CTL0 = 0;                //sem paridade, 8 bits, 1 stop, modo UART
    UCA1BRW = 6;                 // Divisor
    UCA1MCTL = UCBRF_13 | UCOS16; //Moduladores = 13, UCOS16=1
    P4SEL |= BIT3 | BIT0;         //Disponibilizar P4.3 e P4.0
    PMAPKEYID = 0X02D52;          //Liberar mapeamento de P4
    P4MAP0 = PM_UCA1TXD;          //P4.0 = TXD
    P4MAP3 = PM_UCA1RXD;          //P4.3 = RXD
    UCA1CTL1 = UCSSEL_2;          //RST=0 e Selecionar SMCLK
    UCA1IE = UCRXIE;             //Interrupção por recepção
}

void fila1_inic(void){ ... }           //Copiar do ER 8.7
char fila1_poe(char dado){ ... }       //Copiar do ER 8.7
char fila1_tira(char *pt){ ... }        //Copiar do ER 8.7
void USCI_A0_config(void){ ... }        //Copiar do ER 8.7
void bt_str(char *vet){ ... }           //Copiar do ER 8.6
void bt_char(char c){ ... }             //Copiar do ER 8.6

// Copiar do ER 8.1 as funções abaixo:
void led_VM(void),void led_vm(void),void led_VD(void),
void led_vd(void) e void leds_config(void)

```

Ainda faltam (se que devemos colocar ?):

ER 8.9. Exercício com a paridade.

ER 8.10. Exercício com a sinalização de break. Por exemplo, ao detectar a condição de Break acende o *led* verde.

ER 8.11. Exercício com a detecção automática de *baud-rate*.

ER 8.12. Exercício com modo multiprocessador usando idle.

ER 8.13. Exercício com modo multiprocessador usando bit de endereço.

ER 8.14. Algum outro mais ????

8.8. Exercícios Propostos

Apresentamos a seguir uma lista com diversos exercícios para que o leitor pratique o que foi estudado sobre UART. A lista é grande. Sempre que possível, verifique sua solução no LaunchPad.

EP 8.1. Vamos sofisticar um pouco o ER 8.4 para operar em 16 bits com as duas instâncias da USCI_A, conectadas como mostrado na Figura 8.38. A USCI_A1 trabalhará como unidade auxiliar, com a simples tarefa de retornar o complemento a 2 de todo número (16 bits com sinal) que receber. Para que fique transparente, a esta unidade USCI_A1 deverá operar por interrupção. Assim, o número (16 bits) que a USCI_A0 enviar pela porta serial, somado com a resposta serial que receber deve resultar em zero. Configuração serial: 38.400 bauds, 8 bits, sem paridade em com um bit de parada. Use o modo de super amostragem (UCOS = 1) com SMCLK (1.048.576 Hz).

EP 8.2. Vamos novamente usar a conexão da Figura 8.38. Encaramos a USCI_A0 como o mestre que envia comandos e a USCI_A1 como o escravo que os executa. Usando os comandos estão listados abaixo, faça um contador binário com contagens intervaladas de 1 segundo. Use um *timer* para criar o atraso de 1 segundo.

Tabela 8.15. Comandos para a USCI_A1 controlar remotamente os leds da LaunchPad

Cmdo	Ação		Cmdo	Ação
0	Apagar ambos		7	Acender ambos
1	Acender verde		4	Acender vermelho
2	Apagar verde		5	Apagar vermelho
3	Inverter verde		6	Inverter Vermelho

EP 8.3. Vamos agora ampliar o exercício anterior. Além dos comandos já listados, adicionamos perguntas que a USCI_A0 pode fazer à USCI_A1. As duas perguntas e as possíveis respostas (letras) são:

Pergunta: **S1** (qual estado da chave S1?). Respostas: **A** (aberta) ou **F** (fechada).

Pergunta: **S2** (qual estado da chave S2?). Respostas: **A** (aberta) ou **F** (fechada).

Com essas perguntas, a USCI_A0 tem como saber o estado das chaves. Este exercício pede então que se construa um contador binário acionado pelas chaves S1 e S2. O acionamento (passagem de aberta para fechada) da chave S1 incrementa o contador e o acionamento da chave S2 decrementa o contador. Em resumo, o programa principal faz tudo indiretamente pela USCI_A1.

EP 8.4. Uma forma de se verificar a confiabilidade de um canal serial é testando se por ele é possível trafegar sem erros um padrão conhecido. Como padrão vamos usar um vetor de números de 8 bits sem sinal, indo desde 0 até 255. Decidir fazendo um único ensaio é falho, assim vamos propor vários ensaios seguidos.

Usaremos a USCI_A0 para fazer o ensaio, assim, conecte com um cabo os pinos P3.3 (TXD) e P3.4 (RXD). Depois, escreva a função `char testa_ser(char qtd)` que recebe como argumento a quantidade de ensaios a ser feito e retorna quantos deles ocorreram sem erro. O programa principal que faz uso desta função deve acender o *led* verde se os acertos foram de 100% ou vermelho, caso oposto. Use os recursos do CCS para verificar o percentual de acertos. Especificação da porta serial: 38.400 bauds, 8 bits de dados, sem paridade de 1 bit de parada.

Sugestão: Crie dois vetores, um com o padrão a ser transmitido e outro para armazenar a recepção. Um ensaio acontece sem erros quando, ao final do ciclo de transmissão e recepção, ambos vetores são iguais.

EP 8.5. Ainda com o tema de se verificar a confiabilidade de um canal serial, vamos usar a USCI_A1 como eco, que deve usar as interrupções para operar de forma transparente. Faça então as conexões da Figura 8.39 e neste novo ambiente, repita do ensaio do EP 8.4.

EP 8.6. Neste exercício vamos trabalhar com o problema de variação de *baud-rate*. Repita o EP 8.5, mas sob as seguintes condições.

- USCI_A0 operando em 9.600 bauds no modo super amostragem (USCOS16 = 1) com SMCLK (1.048.576 Hz). Para este caso, o manual sugere UCBRX = 6, UCBRSx = 0, UCBRFx = 13.
- USCI_A1 operando em 9.600 bauds no modo baixa frequência (USCOS16 = 0) com ACLK (32.768 Hz). Para este caso, o manual sugere UCBRX = 3 e UCBRSx = 3.

É pedido que seja verificada a ocorrência de erros?

EP 8.7. Caso o leitor tenha disponibilidade de duas LaunchPad, repita o EP 8.6. Use sempre a USCI_A0 de cada LaunchPad sendo uma a mestre e a outra o eco. Verifique se surgiram erros. Caso positivo, tente identificar sua origem.

EP 8.8. Neste exercício vamos trabalhar com o problema de ajuste da geração de *baud-rate*. Usaremos as conexões da Figura 8.38. Quando trabalhamos no modo super amostragem (UCOS16 = 1) temos 3 parâmetros para ajustar: UCBR, USCRF e UCBRS. Verificaremos até que ponto podemos alterar o conteúdo de cada um deles e ainda ter comunicação serial.

Por simplificação, vamos considerar a hierarquia apresentada na listagem abaixo, de forma crescente:

- UCBRS contador cíclico de 0 até 7, quando volta a zero, soma 1 no UCBRF;
- USCBF contador cíclico de 0 até 15, quando volta a zero, soma 1 no UCBR e
- UCBR contador sequencial, não iremos ultrapassar seu limite.

Configure a unidade USCI_A1 (eco) para operar em 19.200 bauds usando os dados do manual: UCBR = 3, UCBRF = 6 e UCBRS = 1. A unidade USCI_A0 terá seu *baud-rate* alterado a cada ensaio (100 repetições do ensaio para cada configuração). Ela deve partir com um *baud-rate* muito elevado. A sugestão é iniciar com UCBR = 2, UCBRF = 0 e UCBRS = 0 e finalizar com UCBR = 5, UCBRF = 0 e UCBRS = 0.

Pedido: armazene em três vetores: `vet_ucbr`, `vet_ucbrf` e `vet_ucbrs` os valores das trincas que funcionaram sem erros. Inicialize os vetores com zeros e use a variável `total` para indicar o total de combinações que levaram a uma comunicação sem erros. Verifique os resultados com usando os recursos do CCS.

Observação: os resultados obtidos são otimistas, pois os relógios das USCI_A0 e USCI_A1 estão sincronizados (dica para explicar o EP 8.7).

EP 8.9. Se tiver possibilidade, repita o EP 8.8 empregando duas LaunchPad. Uma como mestre e a outra como eco. Compare os resultados.

EP 8.10. Vamos agora usar o módulo Bluetooth que foi descrito no ER 8.5. Faça as conexões especificadas na Figura 8.40. Usando a USCI_A0, escreva um programa para que a LaunchPad funcione como eco, ou seja, ela devolve pela porta serial tudo o que receber. Use um terminal serial no PC para enviar dados pelo Bluetooth e assim testar seu programa.

EP 8.11. Este exercício é um desafio. Tente refazer o programa do ER 8.7, mas agora sem usar interrupções.

EP 8.12. Vamos propor um problema semelhante ao EP 8.2. Agora a LaunchPad, é um escravo que responde aos comandos que chegam pelo HC-05 conectado à USCI_A0. Os comandos estão listados abaixo e são enviados pelo PC via Bluetooth com o emprego de um terminal serial. Faça vários testes para comprovar seu funcionamento.

Tabela 8.16. Comandos para controlar remotamente os leds da LaunchPad vai terminal serial no PC (atenção às letras maiúsculas e minúsculas)

Cmdo	Ação	Cmdo	Ação
vd	Apagar verde	vm	Apagar vermelho
VD	Acender verde	VM	Acender vermelho
Vd	Inverter verde	Vm	Inverter Vermelho
S1	Estado da chave S1. Resposta = A ou F	S2	Estado da chave S2. Resposta = A ou F

EP 8.13. Vamos melhorar o exercício anterior. Além de todos os comandos listados, a Launch_Pad sinaliza automaticamente, enviando a sequência de letras “S1” ou “S2”, quando a chave correspondente passar do estado de aberta para fechada.

EP 8.14. Com os recursos que criamos com os EP 8.12 e 8.13, podemos controlar a LaunchPad a partir de um PC. Este exercício pede para testar o script Matlab abaixo que deve fazer um contador binário com os dois *leds*. O programa termina automaticamente depois de 16 segundos. O leitor pode alterar como quiser esse script Matlab.

```
% EP 8.14 (Script para o Matlab)
% Comandar remotamente os leds da Launch Pad
% Usar canal Bluetooth criado com HC-05
% É preciso saber qual porta serial (COMxx) está dedicada ao Bluetooth

% Observação:
% Pode ser que algum erro trave a porta serial
% No Desktop Matlab use o comando "fclose(instrfind)" para fecha-la

br = 9600;           %Baud-rate (Alterar de acordo com seu caso)
nr = input('Qual o numero da serial ? ');
nome = sprintf('COM%d',nr);
sid = serial(nome,'Baudrate',br);
fopen(sid);
if (sid == -1)
    fprintf(1,'Nao abriu porta %s.\n',nome);
    break;
end
fprintf(1,'Abriu porta %s.\n',nome);

% Laço de 16 segundos (16 contagens)
for x = 0:16
    z = mod(x,4); %Resto da divisão por 4
    if      z == 0    fprintf(sid,'vm\r\nvd\r\n');
    elseif z == 1    fprintf(sid,'vm\r\nVD\r\n');
    elseif z == 2    fprintf(sid,'VM\r\nvd\r\n');
    elseif z == 3    fprintf(sid,'VM\r\nVD\r\n');
    end
    pause(1);       %Pausa de 1 segundo
end

fclose(sid);         %Fchar porta serial
fprintf(1,'Fechou porta %s.\nFinalizar.\n',nome);
```

EP 8.15. Usando o esquema do ER 8.7, altere a velocidade de operação do HC-05 para 115.200 bauds (AT+UART=115200,0,0). Nesta configuração conseguimos enviar até

11,5 kbytes por segundo. Isto porque 8 bits de dados, 1 de partida e 1 de parada resultam em 10 bits para cada byte. Para comprovar o funcionamento, refaça o EP 8.9 com esta nova velocidade.

Observação: Com o EP 8.15 construímos uma ferramenta muito útil para ser usada em nossos programas. Com essa ferramenta conseguimos imprimir dados no PC de forma rápida.

EP 8.16. Vamos aperfeiçoar o resultado do EP 8.15. Este exercício pede para construir a série de funções lista abaixo. Todas elas devem ter como base a função `bt_char()`. Tais funções irão facilitar o trabalho do programador. Teste exaustivamente o correto funcionamento de cada uma delas.

Tabela 8.16. Sugestão de funções para facilitar o uso do HC-05 como um terminal serial.

Função	Descrição
<code>void bt_char(char x)</code>	Imprime um caractere
<code>void bt_str(char *pt)</code>	Imprime a string apontada por <code>pt</code>
<code>void bt_crlf(char qtd)</code>	Envia <code>qtd</code> pares de CR ('\r') e LF ('\n')
<code>void bt_spc (char qtd)</code>	Envia <code>qtd</code> espaços (0x20)
<code>void bt_tab (char qtd)</code>	Envia <code>qtd</code> tabulações horizontais ('\t')
<code>void bt_hex(long x, char b)</code>	Imprime em hexadecimal os <code>b</code> bits menos significativos do argumento <code>x</code> .
<code>void bt_dec(long x, char b)</code>	Imprime em decimal os <code>b</code> bits menos significativos do argumento <code>x</code> .
<code>void bt_decu(long x, char b)</code>	Imprime em decimal sem sinal os <code>b</code> bits menos significativos do argumento <code>x</code> .
<code>void bt_decnz(long x, char b)</code>	Imprime em decimal os <code>b</code> bits menos significativos do argumento <code>x</code> , removendo os zeros à esquerda
<code>void bt_decnuz(long x, char b)</code>	Imprime em decimal sem sinal os <code>b</code> bits menos significativos do argumento <code>x</code> , removendo os zeros à esquerda

É preciso verificar se o conceito está correto. Não seria melhor criar funções já com o tamanho do argumento fixo? Exemplos: `bt_dec8(char x); bt_dec16(int x);`

Sugestões para teste:

```
bt_char('A');           bt_str("Teste\n");
bt_hex(0x55,8);        bt_hex(0x1234,16);      bt_hex(0x12345678,32);
bt_dec(87654,16);      bt_dec(-87654,16);       bt_decnz(87654,32);
```

EP 8.17. Usando as rotinas do exercício anterior, rode o seguinte programa para conferir seu funcionamento. Os resultados estão de acordo com o esperado?

```
void main(void) {
    unsigned int z=0;
    ... configurações ...
    bt_str("Teste das rotinas de impressão.\n");
    while (1) {
        bt_hex(z,16);      bt_tab(1);
        bt_dec(z,16);      bt_tab(1);
        bt_decnz(z,16);   bt_tab(1);
        bt_decu(z,16);    bt_tab(1);
        bt_decunz(z,16);  bt_tab(1);
        bt_crlf(1);
        z++;              //O leitor pode alterar o incremento de z
    }
}
```

EP 8.18. Este exercício propõe que a LaunchPad, usando as conexões indicadas abaixo, execute o jogo “Alunissagem com HP-25”. Este jogo está descrito no Apêndice H. Escreva um programa que execute os cálculos na LaunchPad. Use PC com um terminal serial, para enviar comandos e exibir as respostas, listadas abaixo.

- q = quantidade de unidades de combustível a serem queimadas;
- v = velocidade do módulo que está alunissando;
- h = altura do módulo e
- f = unidades restantes de combustível.

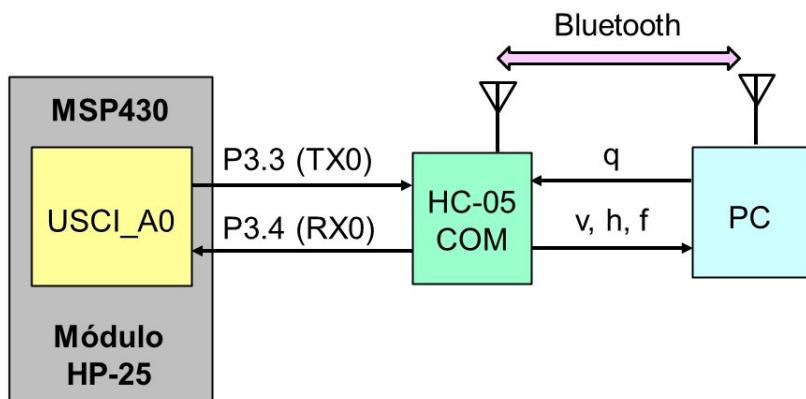


Figura 8.48. Esquema a ser usado para o jogo Alunissagem com a HP-25.

Sugestão: Seu programa deve imprimir no terminal serial algo parecido com a linha abaixo, onde a letra 'x' representa um número:

Velocidade = x ft/s Altitude = x ft Combustível = x unidades.

EP 8.19. Este exercício e os próximos fazem uso do Apêndice E que explica a máquina Enigma e apresenta alguns modelos simplificados. A conexão a ser usada é a mostrada na Figura 8.39. As portas seriais operam em 115.200 bauds, 8 bits de dados, sem paridade de um bit de parada, usando o SMCLK (1.048.576 Hz). Vamos repetir alguns exercícios que foram propostos no Capítulo 2.

Refaça o EP 2.44. Para testar, crie dois vetores, denominados de `char claro[]` e `char cifro[]`. O vetor `claro` contém a mensagem a ser cifrada e o vetor `cifro`, o resultado da cifragem. Use os recursos do CCS para verificar a correção da cifragem. É importante lembrar que o Enigma é uma máquina simétrica. Isto significa que se o vetor `cifro` for usado como entrada, ela retorna o vetor `claro`.

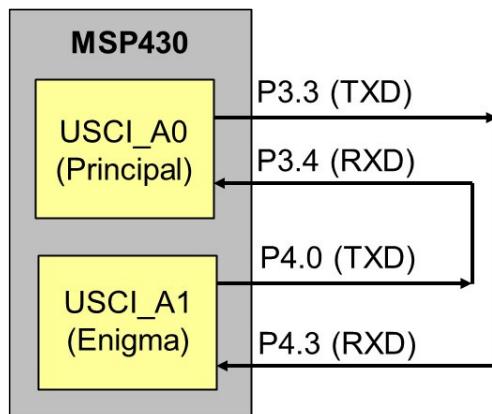


Figura 8.49. Sugestão para a USCI_A1 operar como se fosse a máquina Enigma.

EP 8.20. Se o leitor tiver a disponibilidade de um módulo Bluetooth HC-05, o exercício pode ficar mais interessante. Faça as conexões indicadas na figura 8.50. Usando as mesmas especificações do exercício anterior, programe a LaunchPad para simular a máquina Enigma 1. Será preciso usar um terminal serial. O usuário digita o texto em claro e recebe de volta o texto cifrado.

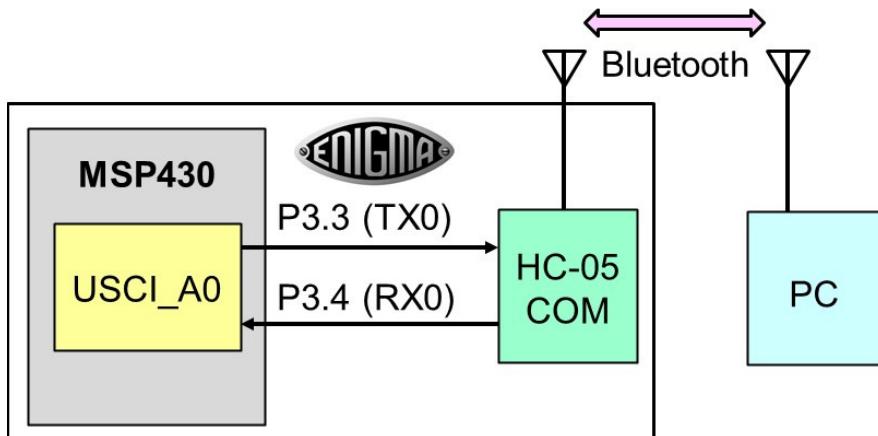


Figura 8.50. Sugestão do uso do canal Bluetooth (HC-05) para que a LaunchPad simile a máquina cifradora Enigma.

EP 8.21. Usando as mesmas especificações, do EP 8.19, resolva o EP 2.47.

Sugestão: o leitor pode resolver antes os exercícios EP 2.45 e EP 2.46, já que eles têm um menor nível de dificuldade. Se tiver disponibilidade do HC-05, tente o esquema da Figura 8.50.

EP 8.22. Usando as mesmas especificações, do EP 8.19, resolva o EP 2.59, que pede a construção completa de uma máquina Enigma. Se tiver disponibilidade do HC-05, tente o esquema da Figura 8.50.

EP 8.23. Vamos aperfeiçoar o EP 8.8, usando as conexões da Figura 8.46. Agora a LaunchPad vai enviar para o terminal serial apenas as informações de data, hora, latitude e longitude, de acordo com o formato abaixo. A última linha é o que deve ser impresso caso a informação do GPS não esteja disponível (GPS não achou satélites). Em geral, as informações de data e hora sempre estão disponíveis.

data	hora	latitude	longitude
20/08/2020	10:31:42	4717.11399 N	00833.91590 E
20/08/2020	10:31:43	4717.11399 N	00833.91590 E
dd/mm/aaaa	hh:mm:ss	ddmm.mmmmmm x	dddmm.mmmmmm x
...			

Observação: Na latitude e longitude, as letras “d” representam os dígitos dos graus e as letras “m” representam os dígitos dos minutos. Note que se usa parte fracionária dos minutos e que a longitude tem um dígito “d” a mais. Exemplo, 4717.11399 significa 47 graus e 17,11399 minutos.

EP 8.24. Vamos avançar com o EP 8.23. Agora a LaunchPad deve enviar apenas as informações de latitude e longitude, se elas estiverem disponíveis, de acordo com o formato abaixo, que é denominado “csv”. Note o formato diferente, agora cada

coordenada está representada em graus e fração de graus. O sinal (+/-) é dado pela seguinte convenção:

Latitude Norte (N) → positiva
Longitude Este (E) → positiva

Latitude Sul (S) → negativa
Longitude Oeste (W) → negativa

```
+47.285233; +8.5652650
-15.762953; -47,866988 (o que será que existe nestas coordenadas?)
```

Como fazer a conversão:

47 graus e 17,11399 minutos → $47 + (17,11399 / 60) = 47.285233$.

Se o leitor tiver um celular Androide, baixe um aplicativo terminal Bluetooth. Conecte-o ao HC-05 e seu celular passará a receber as informações de posicionamento. O interessante é que o Androide permite salvar em arquivo os dados do terminal serial.

Vamos então ao pedido deste exercício, que promete ser interessante. Faça a montagem da LaunchPad do HC-05 e do GPS em uma *protoboard*, para ter alguma robustez. Alimente a LaunchPad pela porta USB usando um Powerbank. Agora temos um GPS portátil que envia dados de posição para seu celular. Passeie com este *hardware* e depois use o Googlemaps para ver o caminho realizado.

EP 8.25. Repita o EP 8.24, mas agora envie imediatamente para o Googlemaps as informações de latitude e longitude. Temos então um GPS portátil que mostra na tela de seu celular o percurso realizado. É recomendado fazer uma pesquisa na Internet sobre a forma correta de se fazer isso. **Será que é realmente possível?**

Gabarito para configurar os registradores da USCI_Ax no Modo UART

Registradores de 8 bits

	7	6	5	4	3	2	1	0
UCAxCTL0	UCPEN	UCPAR	UCMSB	UC7BIT	UCSPB	UCMODE _x		UCSYNC
	0	0	0	0	0	0		0
UCAxCTL1	UCSSEL		UCRXEIE	UCBRKIE	UCDORM	UCTXADDR	UCTXBRK	UCWRST
	0		0	0	0	0	0	1
UCAxMCTL	UCBRFx				UCBRSx			UCOS16
	0				0			0
UCAxSTAT	UCLISTEN	UCFE	UCOE	UCPE	UCBRK	UCRXERR	UCADDR/ UCIDLE	UCBUSY
	0	0	0	0	0	0	0	0
UCAxABCTL	-	-	UCDELM _x		UCSTOE	UCBTOE	-	UCABDEN
	0	0	0		0	0	0	0
UCAxIE	-	-	-	-	-	-	UCTXIE	UCRXIE
	0	0	0	0	0	0	0	0
UCAxIFG	-	-	-	-	-	-	UCTXIFG	UCRXIFG
	0	0	0	0	0	0	1	0
UCAxRXBUF								
UCAxTXBUF								

Registradores de 16 bits

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
UCAxBRW	Divisor para gerar o <i>Baud-rate</i>															
	0															
UCAxIV	Vetores de interrupção															
	0															