



Módulo 2: I/O, Temporizadores e Interrupções

ORIENTAÇÕES:

O segundo módulo do laboratório de Sistemas Microprocessados faz uso da linguagem C e agrupa exercícios de entrada/saída de dados, temporizadores e interrupções. O módulo é composto por 17 exercícios e pelo Problema 2. Os exercícios são propostos para permitir o desenvolvimento modular do Problema 2. Não há a necessidade de apresentar vistos para os exercícios, **apenas o Problema 2 deverá receber visto** e será corrigido mediante o envio do código fonte pelo ambiente de ensino online “Aprender”.

OBJETIVOS DESTE MÓDULO:

- Praticar o controle das portas de entrada e saída (GPIO).
- Gerar atrasos com laços de programa e entender o uso de temporizadores.
- Compreender o funcionamento de interrupções.

Configuração de I/O

Configurações de microcontroladores são feitas alterando-se individualmente bits de registros de configuração. É o caso, por exemplo, dos pinos de entrada e saída, com seus registros de configuração PxDIR, PxIN, PxOUT, PxREN. Em assembly, usamos as instruções de acesso direto aos bits, como BIC (bit clear) BIS (bit set) e XOR (ou exclusivo). Na linguagem de programação C, utilizaremos atribuições com base nas mesmas operações lógicas.

```
/* Configurar a porta P3.5 como entrada com resistor de pull-down */
P3DIR = P3DIR & ~(BIT5); // Configura pino como entrada (zera o bit 5 de P3DIR)
P3REN = P3REN | (BIT5); // Ativa o resistor (habilita o bit 5 de P3REN)
P3OUT = P3OUT & ~(BIT5); // Seleciona o resistor de pull-down
```

Perceba que as instruções que habilitam o bit (colocam o bit em ‘1’) usam a operação lógica OU e as instruções que limpam o bit (colocam o bit em ‘0’) usam a operação lógica E e invertem a máscara. Essas mesmas instruções podem ser reescritas na forma compacta. **É recomendado o uso da forma compacta.**

```
// Configura a porta P3.5
P3DIR &= ~BIT5; // Pino de entrada (P3.5)
P3REN |= BIT5; // Habilita o resistor
P3OUT &= ~BIT5; // Pull-down
```

Para alternar o valor de um bit, use a operação XOR

```
P1OUT ^= BIT5; // Alterna o bit P1.0 (LED vermelho).
// Se for 0 vai para 1, Se for 1 vai para 0.
```

Bit Set		Bit Clear		Bit Toggle		
0000.1111		0000.1111		0000.1111		← Registro
(OU)	0010.0000	(E) &	1111.1011	(XOR) ⊕	0001.1000	← Máscara
0010.1111		0000.1011		0001.0111		← Resultado

Da mesma forma que fizemos em assembly, será necessário tratar os rebotes das chaves mecânicas, vide figura 1. A estratégia usada em microcontroladores é, ao perceber uma transição da chave, esperar um certo tempo para que os rebotes acabem e só então prosseguir com o programa. O fluxograma da figura 2, ilustra esse conceito. Note que é necessário gastar tempo tanto no momento que pressionamos o botão quanto no momento que soltamos o mesmo. A caixa denominada debounce apenas consome um tempo pré-determinado (empiricamente).

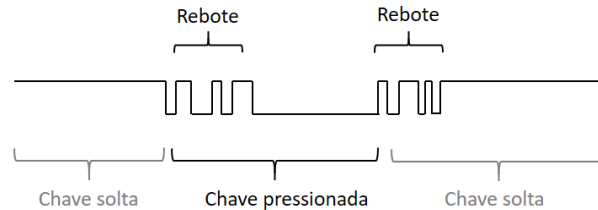


Figura 1 – Rebotes gerados pelas chaves mecânicas

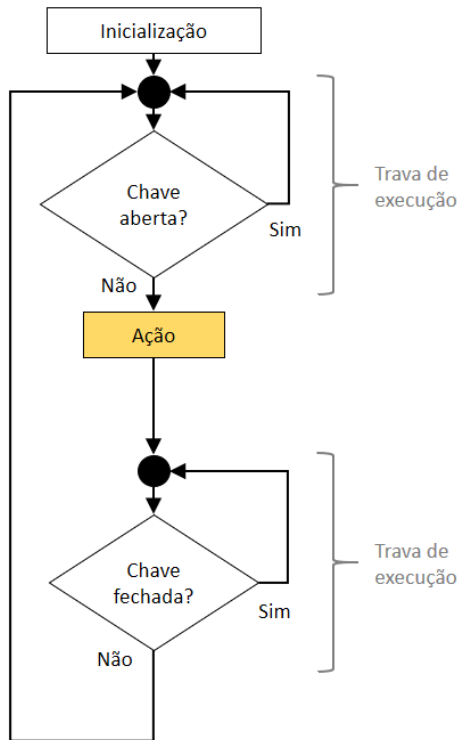


Figura 2a – Fluxograma de ação sem remoção de rebotes (sem debounce).

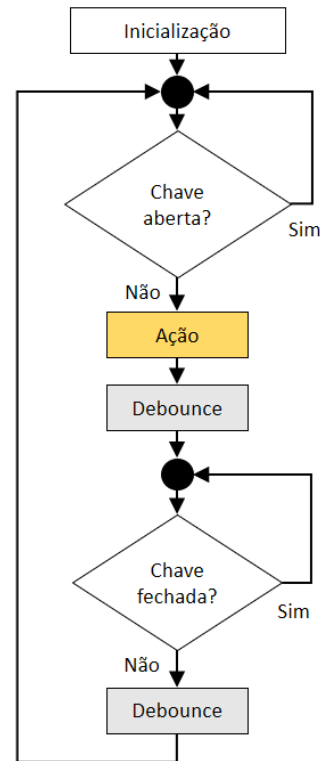


Figura 2b – Fluxograma de ação com remoção de rebotes (com debounce)

Os fluxogramas apresentados na figura 2 são pouco eficientes, pois “prendem” o processador esperando a chave abrir ou fechar. Nenhuma outra tarefa pode ser executada pela CPU. Uma situação mais realista é a de quando se precisa monitorar uma chave **e também** executar uma certa tarefa de tempos em tempos. Isto significa que o processador não pode “ficar preso” esperando pela alteração na chave. A solução para este caso está apresentada na figura 3. Note que são levados em conta dois parâmetros: o estado atual da chave e o estado anterior da chave.

O fluxograma abaixo permite executar a tarefa e monitorar a chave simultaneamente. O programador decide onde ele vai realizar suas ações em função dos estados da chave.

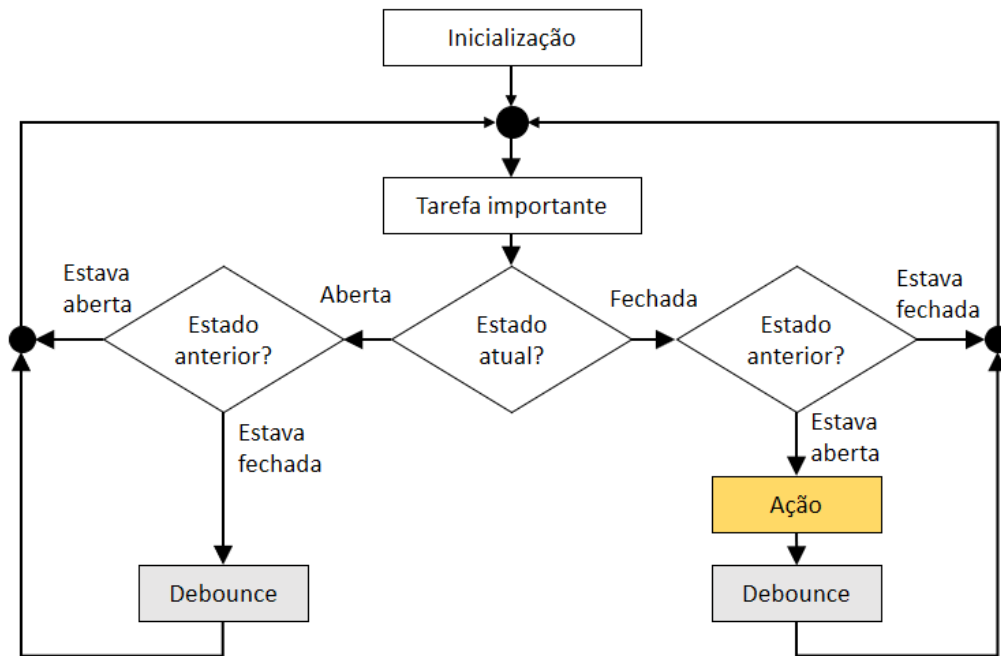


Figura 3 – Fluxograma para monitorar uma chave sem “prender a execução”.

Exercícios:

Exercício 1: Configuração dos pinos

Escreva um programa que faça o LED verde imitar o estado da chave S1. Ou seja, se S1 estiver pressionado o LED deverá estar aceso, se estiver solto, o LED deve apagar.

Exercício 2: Ruído de chaves mecânicas

Escreva na função `main()` uma rotina que alterne o estado do LED **vermelho** toda vez que o usuário apertar o botão S1. Não remova os rebotes.

Exercício 3: Remoção de rebotes

Refaça o exercício 2 removendo os rebotes das chaves. Para isso, defina uma função `debounce()` que consome tempo do processador através de um loop que decremente uma variável. Não deixe de declará-la como **volatile** para evitar que o recurso de otimização do compilador a remova.

Exercício 4: Trava de execução com condições mais elaboradas.

Escreva na função `main()` uma rotina que alterne o estado do LED **vermelho** toda vez que o usuário apertar o botão **S1** ou **S2**.

Interrupções das Portas de Entrada/Saída

Interrupções, como o próprio nome já indica, interrompem o funcionamento do programa principal e executam uma rotina de tratamento de interrupção (ISR – Interrupt Service Routine). O modelo de programação que usamos até agora é chamado de “loop infinito”, onde toda a funcionalidade se concentra na rotina principal (main). O modelo de programação à base de interrupções implica em distribuir funcionalidades para diferentes rotinas como na figura 4. As interrupções das portas de entrada e saída são configuradas nos registros PxIES (Interrupt Edge Select) e PxIFG (Interrupt FlaG) e PxIE (Interrupt Enable) nessa exata ordem.

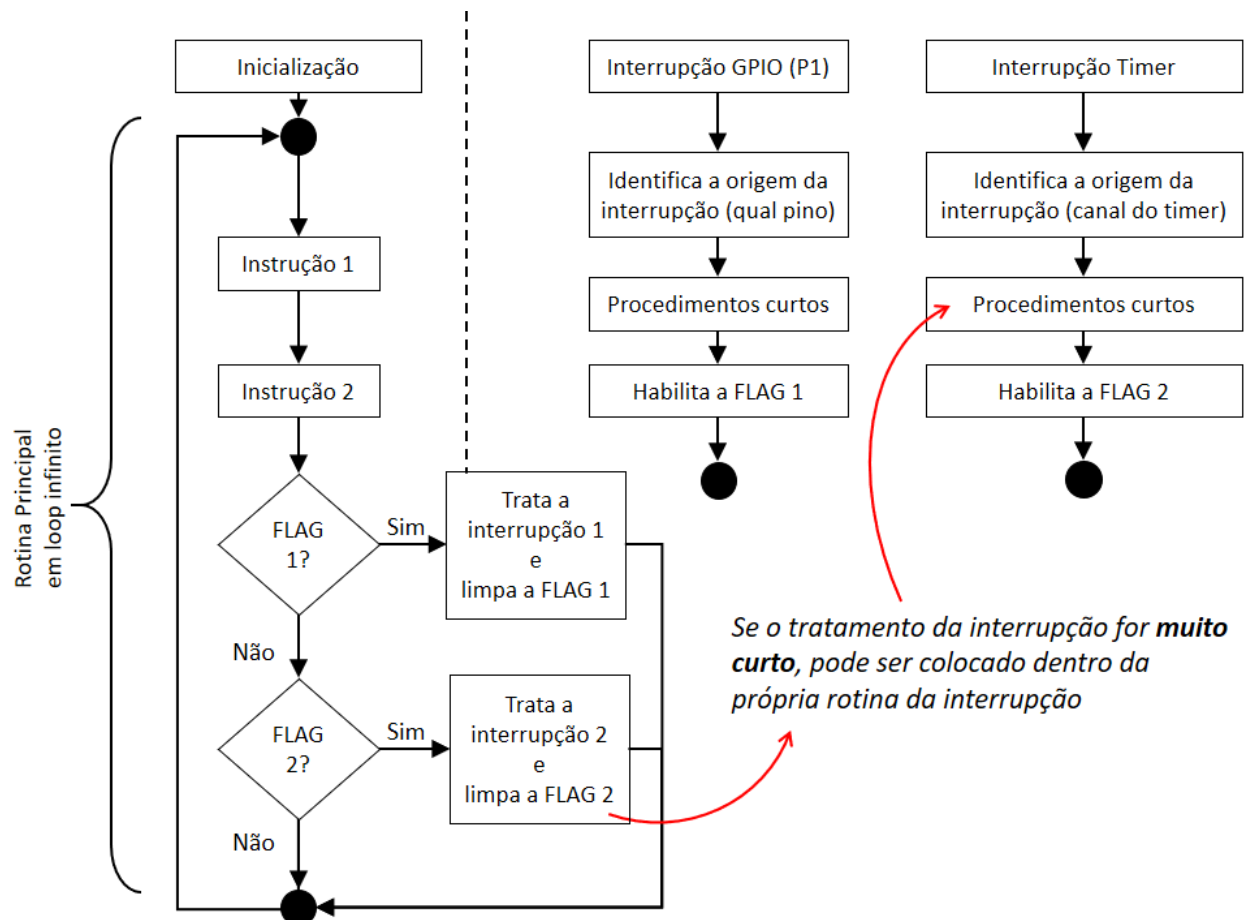


Figura 4: Modelo de programação a base de interrupções

A função que trata a interrupção é declarada de maneira especial. Ela possui uma linha antes do nome da função que o compilador usa para mapear a função para a devida interrupção. Além disso, note que a função foi declarada com o modificador “`__interrupt`”. Esse modificador impede que o compilador remova essa função do código, já que ela não é usada na rotina principal. Interrupções não são chamadas pelo usuário/programa. **Interrupções são chamadas por eventos de hardware.** As rotinas de interrupção são declaradas em C da seguinte forma:

```
#pragma vector=PORT1_VECTOR // Diretiva usada pelo compilador
```



```
__interrupt void P1ISR() {    // Port 1 Interrupt Service Routine
    P1OUT |= BIT0;           // Acende o LED vermelho
    P1IFG &= ~BIT1;          // Clear P1.1 IFG (botão S2)
}
```

A rotina acima mostrada acende o LED vermelho quando o botão S2 for pressionado. Este exemplo, entretanto, só funciona se apenas um pino da porta P1 for utilizado. No MSP430, as interrupções das portas são compartilhadas. Dessa forma, as interrupções de todos os pinos da porta P1 (P1.0, P1.1, P1.2, etc..) serão redirecionados para essa função. Para saber de onde veio a interrupção é necessário decodificar a origem da interrupção através do registro P1IV. Uma versão mais completa da rotina de interrupções é mostrada a seguir:

```
// Port 1 Interrupt Service Routine
#pragma vector=PORT1_VECTOR
__interrupt void P1ISR() {
    switch (P1IV) {           // A leitura de P1IV limpa a P1IFG correspondente
        case 0x0: break;      // Nada aconteceu
        case 0x2: break;      // P1.0 (maior prioridade)
        case 0x4:             // P1.1
            P1OUT ^= BIT0;    // Faz o LED vermelho alternar de estado
            break;
        ...                   // ...
        case 0xE: break;      // P1.7 (menor prioridade)
        default: break;
    }
}
```

Note a presença de um switch-case estruturado ao redor do registro P1IV (P1 Interrupt Vector). Este registro indica o número da interrupção de maior prioridade. Consulte o capítulo de GPIOs do User's Guide para saber qual número corresponde à qual pino e porque o vetor de interrupções sempre retorna um número par. *Para refletir: Porque a primeira rotina mostrada limpa o bit 1 do registro P1IFG enquanto a segunda rotina não?*

Exercícios:

Exercício 5: Configuração de interrupções

Usando interrupções, escreva um código que alterne o estado do LED **vermelho** ao pressionar o botão S1.

Exercício 6: Múltiplas interrupções

Usando interrupções, programe um contador binário de 2 bits, visualizado através dos LEDs. Use o LED verde como LSB e o vermelho como MSB. Os LEDs iniciam apagados. O contador deve incrementar a cada acionamento de S2 e decrementar com S1. É necessário eliminar os rebotes das chaves.

Exercício 7: Mini-problema

Neste exercício, iremos programar um jogo de adivinhação. O jogador deve adivinhar um número de 1 a 6. A cada rodada, o jogo gera um novo número e o jogador deve tentar adivinhar apertando N vezes o botão S1. Tendo feito a escolha, o jogador aperta o botão S2 e verifica se acertou ou não. Se o LED vermelho acender, significa que ele errou. Se o LED verde acender, significa que

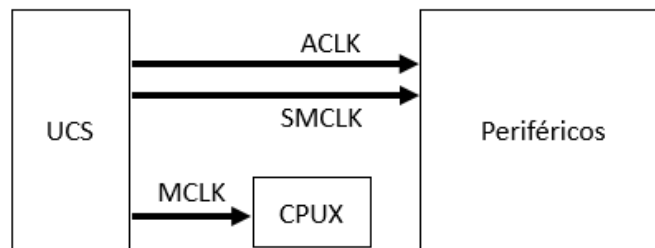


o jogador acertou e nesse momento o jogo gera um novo número. Use a rotina abaixo para gerar um número pseudo-aleatório.

```
unsigned char rand() {  
    static unsigned char num = 5;  
    num = (num * 17) % 7;  
    return num;  
}
```

Temporizadores (Timers)

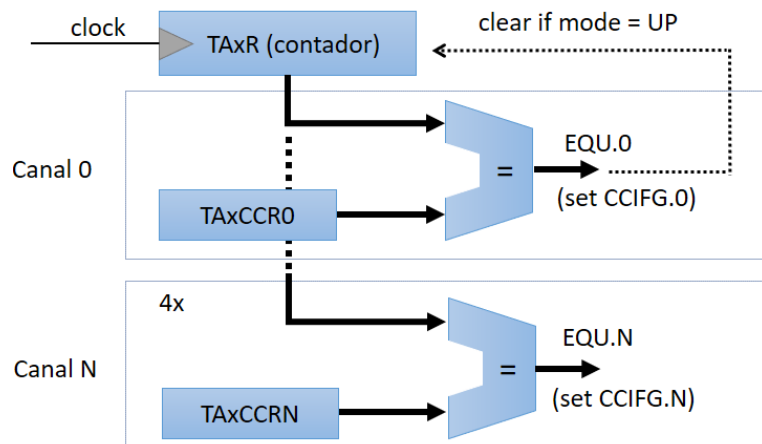
Contadores, temporizadores, ou simplesmente *timers* em inglês, são peças fundamentais em sistemas embarcados. *Timers* permitem que uma referência de tempo seja usada em aplicações que demandam sincronismo ou precisão temporal. A linguagem de programação C não possui a noção de tempo. Cada linha de código é executada sequencialmente, uma após a outra, e não há nada na linguagem que indique a velocidade ou tempo de execução do programa. A única forma de sabermos o tempo que leva para cada instrução executar é conhecendo o hardware em que o programa está rodando e principalmente a velocidade do clock que bate e dá o ritmo de execução das instruções. O MSP430 possui um módulo específico para a configuração dos clocks que são distribuídos no sistema chamado de *Unified Clock System* (UCS).



É no UCS que configuramos a frequência de cada clock. Para este módulo a configuração do UCS será dada. Leia atentamente o código fornecido para entender o seu funcionamento. Você irá notar que MCLK está sintonizado em 16MHz, SMCLK em 1MHz e ACLK em 32kHz.

Timer A: Modo de Comparação

O timer A possui um contador (**TAxR**) que incrementa o seu valor a cada batida do clock. O valor do contador é comparado com limiares definidos nos registros de comparação **TAxCCRN**. No MSP430F5529, temos 5 canais de comparação no timer A0. Quando o valor do contador chega num limiar pré-estabelecido, o sinal de igualdade (EQU.N) se torna '1' e ativa a flag de interrupção do canal correspondente. A interrupção só irá acontecer se a mesma estiver habilitada através do bit **CCIE** (Capture/Compare Interrupt Enable) do registro de controle do canal **TAxCTLN**.



Deve-se notar que o canal 0 do timer é especial. O sinal EQU.0 é usado para zerar o contador quando o limiar estabelecido em CCR0 for atingido. Ou seja, CCR0 funciona como limitador do contador principal. Além disso, o canal 0 gera uma interrupção separada dos outros canais. Todos os outros canais incluindo o overflow do contador são agrupados numa única interrupção. Interrupções agrupadas devem ser decodificadas pelo código do registro **TAxIV** (da mesma forma que usamos nas interrupções agrupadas das portas). Esse registro sempre indica a interrupção de maior prioridade que aconteceu dentre os eventos agrupados. As rotinas de tratamento de interrupção dos timers são declaradas da seguinte forma:

```
#define TA0_CCR0_INT 53          // Timer A0 CCR0 interruption priority
#define TA0_CCRN_INT 52          // Timer A0 CCR1,2,3,4 interruption priority
...
// A linha "#pragma vector =..." indica que o endereço da próxima função
// deve ser copiado na tabela de endereços de interrupções, na posição 53 (p/ o CCR0)
#pragma vector=TA0_CCR0_INT
__interrupt void TA0_CCR0_ISR() { // Interrupt Handler
...                               // Código executado quando TA0R = CCR0
}
```

Exercícios:

Exercício 8: Temporização imprecisa

Ainda sem usar timers (use laços de programa) faça o LED piscar em aproximadamente 1Hz, ou seja, fique 500ms apagado e 500ms aceso.

Você irá perceber que o uso de laços para consumir tempo não produz um resultado muito preciso. Isso acontece porque não temos precisão fina sobre a quantidade de ciclos que o código gerado pelo compilador consome. Podemos melhorar o controle do tempo se utilizarmos timers.

Exercício 9: Amostrando flags do timer

Escreva um programa em C que faça piscar o LED verde (P4.7) em exatamente 1Hz (0,5s apagado e 0,5s aceso). Use a técnica de amostragem (polling) da flag de overflow (TAIFG) do timer para saber quando o timer atingiu o valor máximo.



Exercício 10: Interrupção dedicadas

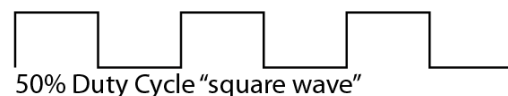
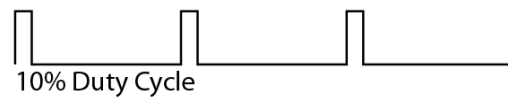
Repita o exercício 9 usando a interrupção do comparador CCR0.

Exercício 11: Interrupções agrupadas

Usando as interrupções agrupadas do timer A0, faça o LED vermelho piscar em 1Hz e o LED verde piscar duas vezes mais rápido, ou seja, em 2Hz. Use o registro CCR1 para alternar o valor do LED vermelho e CCR1 e CCR2 para acender e apagar o LED verde.

Pulse Width Modulation (PWM)

Modulação por largura de pulso é uma técnica bastante utilizada quando queremos controlar a energia transmitida entre dispositivos. Essa técnica é usada para controlar a velocidade de motores DC, luminosidade de LEDs, ou até regular níveis de tensão em controladores de carga. A técnica consiste em enviar pulsos de largura controlável num intervalo de tempo regular. A relação entre a largura do pulso e o período é chamada de *duty cycle* em inglês. Quanto maior for o *duty cycle*, maior será a energia média do sinal.



O olho humano não consegue perceber variações de luminosidade muito rápidas. Em média, uma população de indivíduos consegue enxergar variações de até 20 a 30Hz. Podemos utilizar essa "limitação" do olho humano para controlar a luminosidade de um LED.

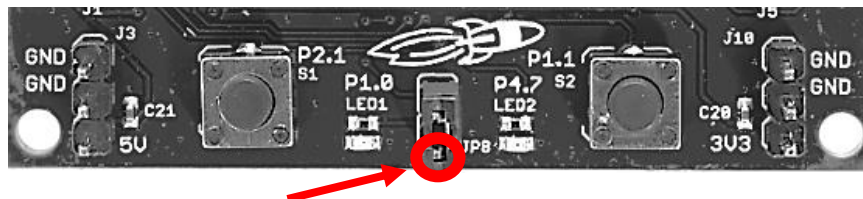
Exercícios:

Exercício 12: PWM com *duty cycle* fixo

Usando as interrupções do timer, faça o LED vermelho piscar em 50Hz com *duty cycle* de 50%. É recomendado o uso das interrupções de overflow e CCR1.

Exercício 13: PWM por hardware

Repita o programa 12 usando a saída do timer. Procure no datasheet qual pino está conectado à saída do canal desejado. Ligue a saída do timer no LED vermelho com um cabo removendo o jumper JP8 e conectando a saída do timer no pino de baixo do jumper (pino mais próximo da extremidade da placa). **Não conecte no outro pino! Você irá queimar a porta do seu MSP.**



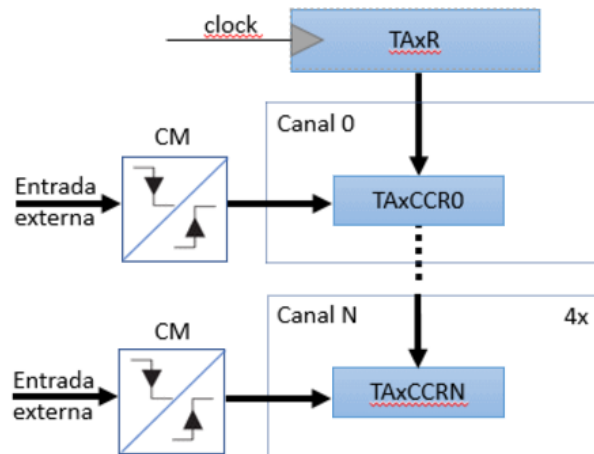
Exercício 14: PWM variável

Repita o programa 13, agora usando os botões S1 e S2 para aumentar e diminuir o *duty cycle* em passos de 10% de CCR0.



Timer A: Modo de Captura

No modo de captura, os registros CCR funcionam como variáveis para guardar o valor do contador quando um evento externo acontece. Esse evento pode ser devido a um flanco de subida, descida ou ambos, dependendo do modo de captura (CM). Esse modo é útil para medir tempo entre eventos externos ao microcontrolador.



Exercícios:

Exercício 15: Captura de um evento de hardware

Use o timer em modo de captura para medir o tempo entre o acionamento dos botões S1 e S2. Ou seja, quando o usuário pressionar o botão S1 o timer inicia. Quando pressionar o botão S2 o timer captura o tempo que passou. Se for menor que 1 segundo, acende o LED vermelho, se estiver entre 1 e 3 segundos, acende os dois LEDs, se for maior que 3 segundos, acende apenas o LED verde.

Exercício 16: Medir o intervalo entre os rebotes de uma chave

Use o timer em modo de captura de ambos os flancos para medir os intervalos entre os diversos rebotes que surgem quando a chave S1 é acionada. O programa deve armazenar os intervalos entre os rebotes em um vetor de nome "rebotes" e trabalhar com precisão de microssegundos. Para limitar o tempo de captura, faça a medição apenas durante uma janela de 1 segundo que começa a ser contada após o acionamento da chave S1. O programa termina após essa janela de tempo e o programador pode então consultar o vetor "rebotes" com o CCS.

Exercício 17: Captura por software.

Meça o tempo de execução das duas linhas de código abaixo:

```
volatile double hardVar = 128.43984610923f;  
hardVar = (sqrt(hardVar * 3.14159265359) + 30.3245)/1020.2331556 - 0.11923;
```



PROBLEMA 2: Utilizando o sensor de Proximidade

Este problema deve receber o visto (nota), que só será validado após seu upload. Faça o upload de apenas um programa por equipe. O programa solução entregue deve estar completo e pronto para ser “carregado” e executado no Code Composer. Será verificado o correto funcionamento de todo programa. **Caso o programa solução entregue não funcione, a nota será zerada.**

ÉTICA E HONESTIDADE ESTUDANTIL:

Será verificada a similaridade entre os programas entregues e os demais programas, incluindo os dos semestres anteriores. Caso a similaridade entre dois programas seja grande o suficiente para caracterizar a “cola”, as equipes estão **automaticamente reprovadas**. Oportunamente serão especificadas as datas limite para o visto e para o upload da solução.

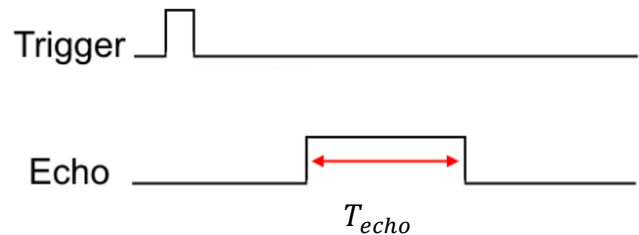
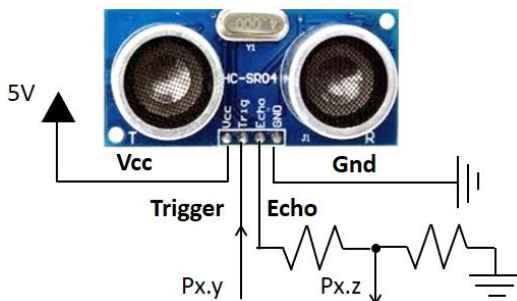
OBJETIVO:

Implementar uma “trena” digital utilizando o sensor de proximidade e o módulo HC-SR04.

DADOS:

O sensor de proximidade HC-SR04 é equipado de um autofalante e um microfone ultrassônicos. Quando acionado por um pulso positivo na entrada *trigger* o sensor envia pulsos sonoros ultrassônicos inaudíveis ao ouvido humano. Uma resposta do tempo de propagação é devolvida no sinal *echo*. O sinal *trigger* deve ter duração mínima de 10us e o sinal *echo* irá retornar um pulso com duração máxima de 12ms (qual é a maior distância que podemos medir com esse dispositivo?).

Atenção, o sensor é alimentado com 5V. Use então um **divisor resistivo no retorno do echo** para não queimar o pino de entrada da sua launchpad.



Pedido (programa para receber visto):

Escreva uma aplicação embarcada que ao pressionar o botão S1 realize uma medida de distância. Se a distância medida for menor que 20cm o LED vermelho deve acender, se estiver entre 20 e 40cm, o LED verde deve acender e, se estiver acima de 40 cm, ambos os LEDs devem acender.

Desafio opcional: Usando o recurso do “break point” do CCS, mostre a distância em tempo real.