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 vale 20 pontos. Toda semana será selecionado um exercício para receber visto. Serão 3 vistos ao total. Cada visto vale 3 pontos. No final do módulo, um problema será proposto para sumarizar o conteúdo do módulo. O problema vale 11 pontos.

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.

General Purpose Input/Output (GPIO)

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

Chaves mecânicas apresentam um ruído característico quando alteram de estado. Esse ruído é conhecido na literatura como rebote (*bounce*), vide figura 1. A estratégia usada para tratar rebotes em microcontroladores é 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. A caixa denominada debounce apenas consome um tempo pré-determinado empiricamente.

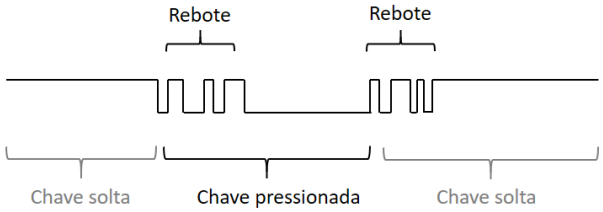


Figura 1 – Rebotes gerados pelas chaves mecânicas

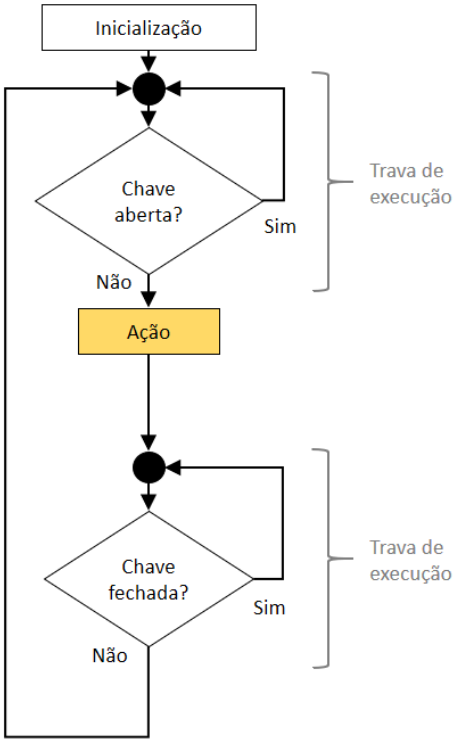


Figura 2a – Fluxograma de ação sem remoção de rebotes (sem debounce).

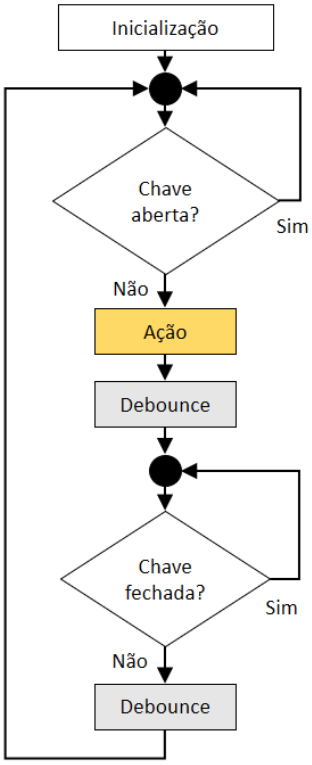


Figura 2b – Fluxograma de ação com remoção de rebotes (com debounce)

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.

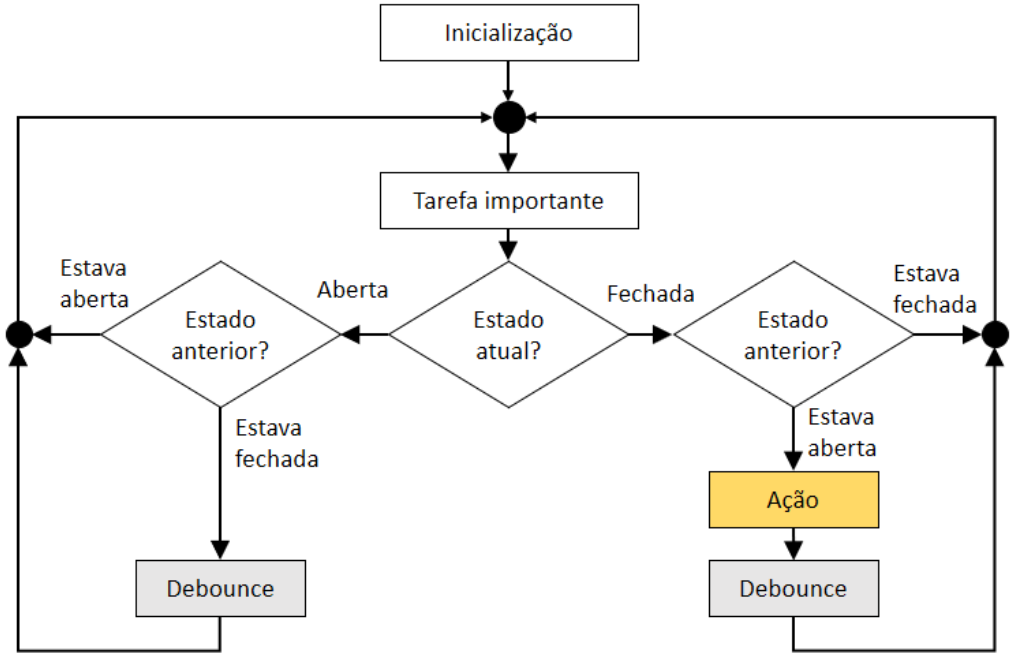


Figura 3 – Fluxograma para monitorar uma chave sem "prender a execução".

Exercício 1: Configuração dos pinos
Escreva um programa que faça o LED verde imitar o estado da chave S1. Se S1 estiver pressionado o LED deverá estar aceso, se estiver solto, o LED deve apagar.

Exercício 2: Ruído (rebotes) 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 acionar a chave **S1 ou S2**. Por acionamento de uma chave, entende-se sua passagem do estado aberta para o estado fechada (A → F).

Temporizadores (Timers)

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. O MSP430 possui três *clocks* que são distribuídos no microcontrolador. O módulo para a configuração dos *clocks* é chamado de *Unified Clock System* (UCS). ACLK e SMCLK são usados nos periféricos. Apenas o MCLK é usado para o processador. ACLK é considerado como *clock* lento e bate numa frequência de 32.768Hz (2^15). SMCLK e MCLK estão sintonizados em 1.048.576 Hz (2^20).

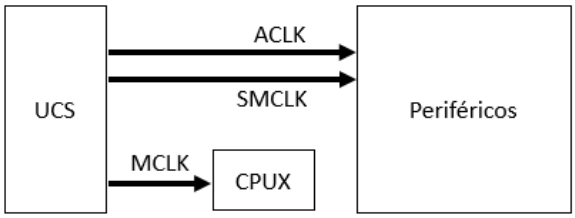


Figura 4 – Distrubuição do Sistema de clocks

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 (CCIFG) do canal correspondente.

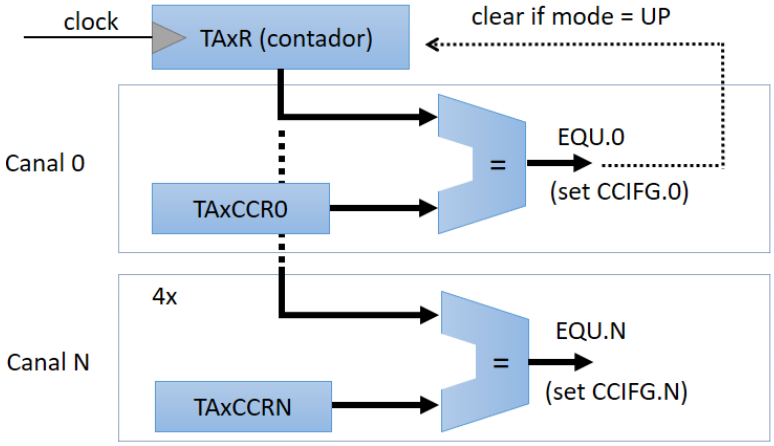


Figura 5 – Canais do timer

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.

Exercício 5: 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.

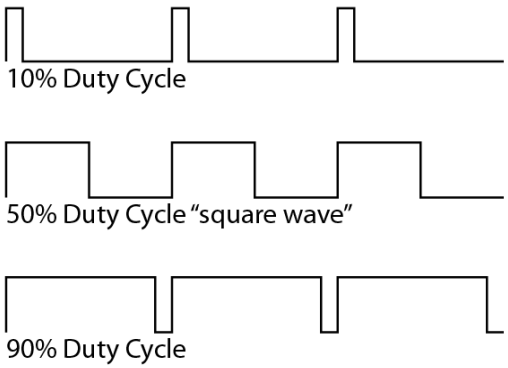
Exercício 6: Amostrando flags do timer
Escreva um programa em C que faça piscar o LED verde (P4.7) em exatamente 1Hz (0,5s apagado e 0,5s aceso). Use a técnica de amostragem (polling) da flag de overflow (TAIFG) ou a flag do canal 0 (CCIFG) para saber quando o timer atingiu o valor máximo.

Exercício 7: Medindo tempo.
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;
```

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.

Exercício 8: PWM com *duty cycle* fixo
Faça o LED vermelho piscar em 128Hz com *duty cycle* de 50%. É recomendado o uso das interrupções de overflow e CCR1.

Exercício 9: PWM por hardware
Repita o programa anterior 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). N**ão conecte no outro pino! Você irá queimar a porta do seu MSP.**

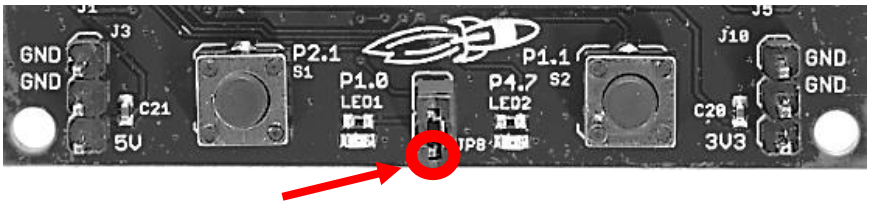


Figura 6 – Jumper do LED vermelho

Exercício 10: PWM variável

Incremente o programa anterior, acrescentando a possibilidade de ajustar o *duty* cycle usando os botões S1 e S2. O botão S2 aumenta e o botão S1 diminui o *duty cycle* em passos de 12,5% (1/8) de CCR0.

Exercício 11: Usando um servo motor

O servo motor é um dispositivo eletromecânico cuja rotação pode ser comandada por um sinal PWM. Ele não gira continuamente, apenas muda o ângulo da sua posição. A posição angular é codificada num pulso que varia de 0,5ms a 2,5ms. Vamos usar o SG90 como mostrado na figura 7a. O sinal de controle é mostrado na figura 7b. Use o canal 2 do timer TA2, cuja saída está conectada no pino P2.5, para controlar esse motor. Configure um sinal PWM com período de 20 ms (50 Hz) e faça o tamanho do pulso variar de 0,5ms a 2,5ms usando S1 e S2 como no exercício anterior. Use passos de 0,1ms.

Figura 7a – Servo motor

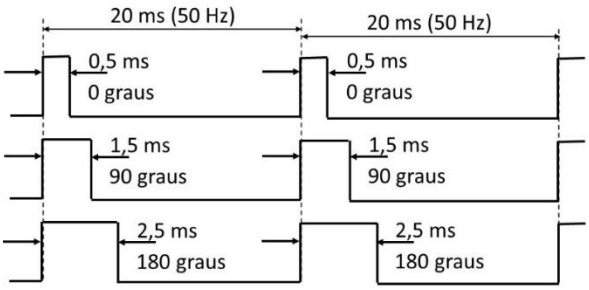


Figura 7b – Comando do motor

Conexão do servo motor: fio marrom = GND, fio vermelho = +5V e fio laranja = P2.5 (TA2.2).

Recomendamos encapsular a atuação do motor numa função que receba como parâmetro o ângulo de atuação do motor e altera o sinal PWM de acordo.

Interrupções

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 a base de interrupções implica em distribuir funcionalidades para diferentes rotinas como na figura 7.

Para habilitar uma interrupção é necessário habilitá-la através das configurações de *Interrupt Enable* (IE). As interrupções dos pinos de GPIO são configuradas nos registros PxIES (Interrupt Edge Select) e PxIFG (Interrupt FlaG) e PxIE (Interrupt Enable). As interrupções dos timers são mais simples e podem ser detectadas pelas flags dos canais (**CCIFG**), mas só irão gerar a execução de uma ISR se estiverem habilitadas através do bit **CCIE** (Capture/Compare Interrupt Enable) do registro de controle do canal **TAxCCTLn**. A flag de overflow fica no registro de configuração do canal TAxCTL e chama-se **TAIFG**. A sua ISR executa se **TAIE** estiver habilitada, localizada também no mesmo registro.

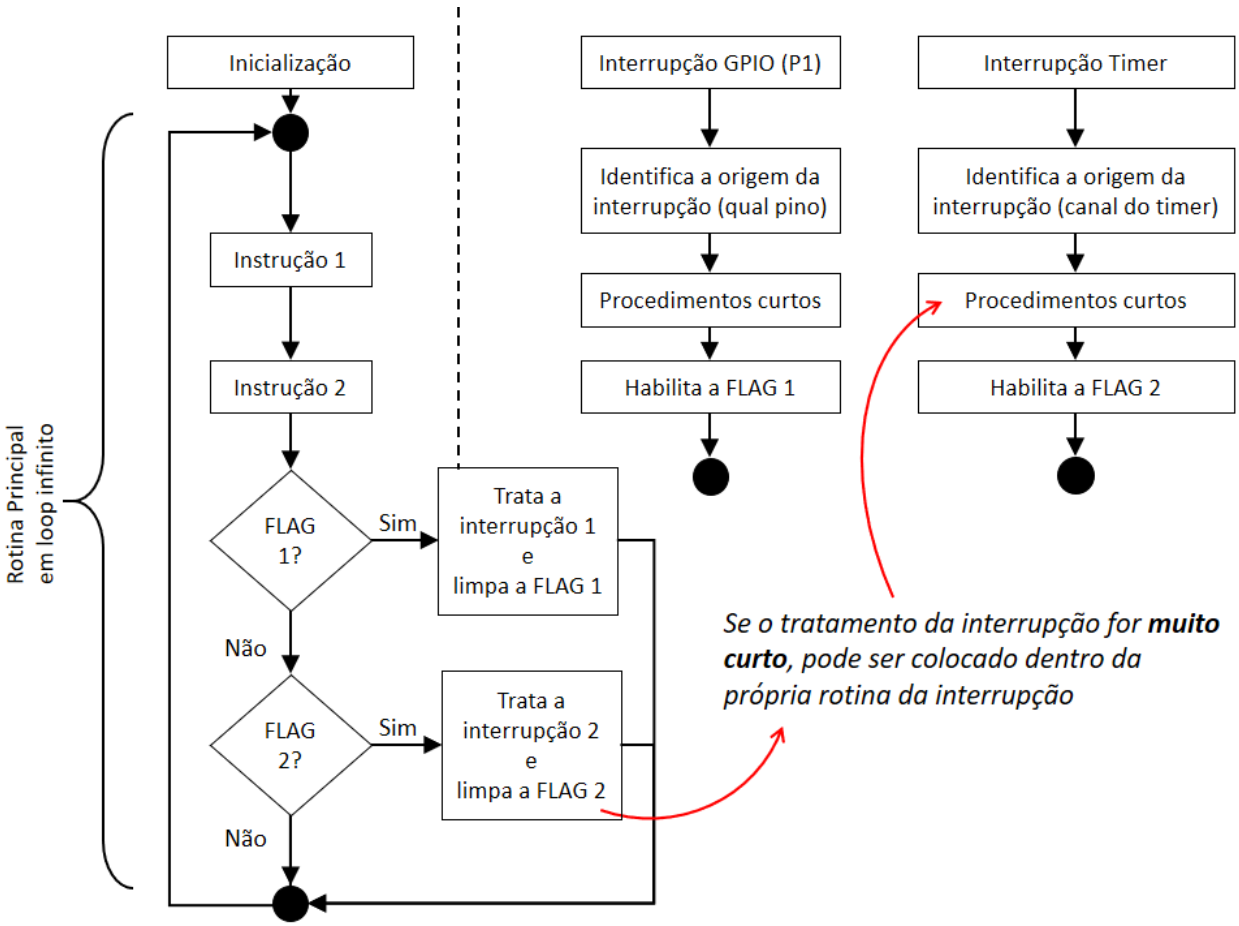


Figura 7: Modelo de programação a base de interrupções

A função que trata a interrupção (ISR) 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. Além disso, o modificador "**__interrupt**" faz a rotina de interrupção retornar com a instrução RETI ao invés de RET. 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:

```
#define TA0_CCR0_INT 53              // Timer A0 CCR0 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
}
```

Interrupções agrupadas vs dedicadas

O canal 0 gera uma interrupção separada dos outros canais, por isso dizemos que a interrupção do canal 0 é uma interrupção dedicada. Todos os outros canais incluindo o overflow do contador são agrupados numa outra rotina de tratamento de interrupções (ISR). Interrupções agrupadas devem ser decodificadas pelo código do registro **TAxIV**. Esse registro sempre indica a interrupção de maior prioridade que aconteceu dentre os eventos agrupados. As rotinas de tratamento de interrupção agrupadas são declaradas da seguinte forma:

```
#define TA0_CCRN_INT 52         // Timer A0 CCR1,2,3,4 interruption priority
...

// Rotina de tratamento de interrupção do timer A0 p/ o canal 1 em diante e overflow
#pragma vector = TA0_CCRN_INT
__interrupt void TA0_CCRN_ISR()
{
    switch (TA0IV)              // A leitura de TA0IV limpa a CCIFG correspondente
    {
        case 0x0: break;        // Nada aconteceu
        case 0x2: break;        // CCR1 (maior prioridade)
        case 0x4:               // CCR2
            P1OUT ^=  BIT0;     // Faz o LED vermelho alternar de estado
            break;              // quando o contador chegar em CCR2
        ...                     // ...
        case 0xE: break;        // Overflow (menor prioridade)
        default:  break;
    }
}
```

Note a presença de um switch-case estruturado ao redor do registro TA0IV (TA0 Interrupt Vector). Este registro indica o número da interrupção de maior prioridade. Consulte o capítulo do Timer A do User's Guide para saber qual número corresponde à qual canal do timer e porque o vetor de interrupções sempre retorna um número par. *Para refletir: Porque não se zera a flag IFG nas rotinas de tratamento de interrupção? Quando isso acontece de fato?*

Exercício 12: [GPIO] Configuração de interrupções
Usando interrupções, escreva um código que alterne o estado do LED vermelho ao pressionar o botão S1.

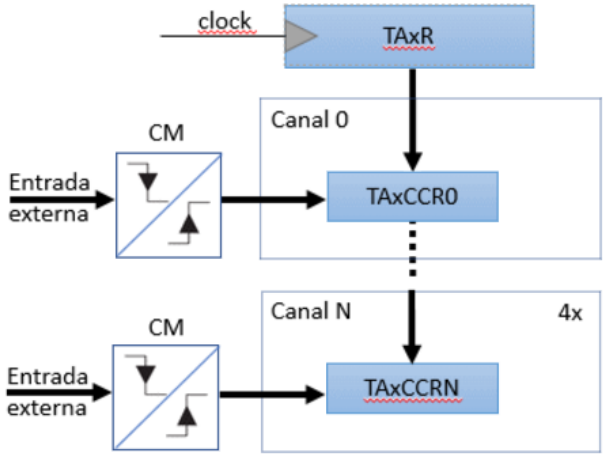
Exercício 13: [GPIO] Múltiplas interrupções
Usando interrupções, programe um contador binário de 2 bits, visualizado através dos LEDs. Use o LED verde como LSB e o vermelho como MSB. Os LEDs iniciam apagados. O contador deve incrementar a cada acionamento de S1 e decrementar com S2. É necessário eliminar os rebotes das chaves.

Exercício 14: [Timer] Interrupção dedicadas
Repita o exercício 6 (fazer piscar o LED a 1Hz) usando a interrupção do comparador CCR0.

Exercício 15: [Timer] Interrupções agrupadas
Usando as interrupções agrupadas do timer A0 e PWM, faça o LED vermelho ascender a 30% de *duty cycle* e o LED verde em 70%. Use o registro CCR1 para controlar o LED vermelho e CCR2 para controlar o LED verde.

Modo de captura do Timer

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



Use o timer em modo de captura para medir os intervalos entre os diversos rebotes que surgem quando a chave S1 é acionada. Configure o modo de acionamento p/ ambos os flancos. 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: Utilizando o sensor de Proximidade

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 10 µs e o sinal echo irá retornar um pulso com duração máxima de 12 ms. *Pergunta: 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. Sugestão: 4K7 e 10K. *Pergunta: qual desses dois resistores deve estar ligado ao terra?*