

Lista de Exercícios #06 - 2024.1

O objetivo desta lista é testar as soluções de **sincronização** e **coordenação** entre *tasks* no **FreeRTOS** (e consequentemente no CMSIS-RTOS) utilizando a **STM32 CubeIDE**. Além disso, o uso do conversor AD e o uso de display de 7-seguimentos e a transferência de dados entre *threads* no **FreeRTOS** utilizando *Queues*.

1 Instruções Gerais

- Sempre comente seu código e identifique o que ele faz e quem foi que fez (você, no caso);
- Coloque todas as pastas dos projetos criados em uma única pasta “LE06_SEU_NOME”, em que SEU_NOME é o seu nome;
- Compacte a pasta e inclua como resposta da atividade.

2 Uso de *flags* para coordenar atividades

Neste exercício iremos ver como coordenar *tasks* com o auxílio de *flags*, ou seja, sem a necessidade que nenhuma *task* entre em espera por algum evento. O FreeRTOS não provê um tipo nativo *condition flag*, mas criaremos o nosso.

2.1 O problema a ser resolvido

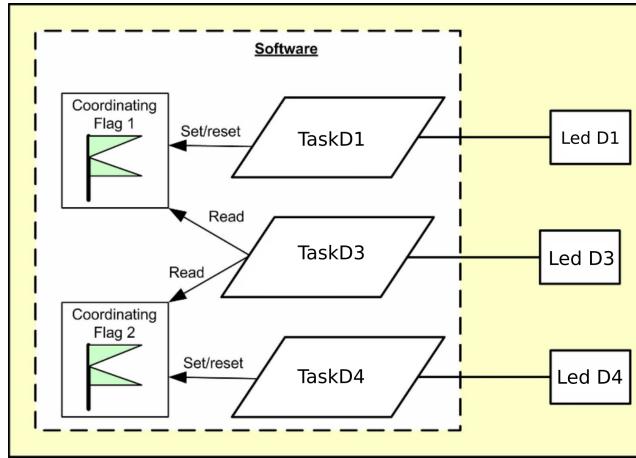
Neste primeiro exercício (LE06_Q1_SEU_NOME), estamos interessados em implementar um sistema com três *tasks*: **TaskD1**, **TaskD4** e **TaskD3**; cuja interação é mostrada na Figura 1 e cujos comportamentos em detalhados são dados nas subseções seguintes:

2.1.1 TaskD1

Comportamento da **TaskD1**:

1. Inverte o Led D1 com uma frequência de 10 Hz por 10s;
2. Seta a coordinatingFlag1;
3. Inverte o Led D1 com uma frequência de 1 Hz por 10s;
4. Reseta a coordinatingFlag1;
5. Inverte o Led D1 com uma frequência de 10 Hz por 10s;
6. Suspende a *task*.

Figura 1: Diagrama de *Tasks* do Sistema da Questão 1.



Fonte - Adaptado de [1].

2.1.2 TaskD4

Comportamento da **TaskD4**:

1. Inverte o Led D4 com uma frequência de 10 Hz por 15s;
2. Seta a coordinatingFlag2;
3. Inverte o Led D4 com uma frequência de 1 Hz por 10s;
4. Reseta a coordinatingFlag2;
5. Inverte o Led D4 com uma frequência de 10 Hz por 5s;
6. Suspende a *task*.

2.1.3 TaskD3

Comportamento da **TaskD3**:

1. Quando iniciada, inverte o Led D3 com uma frequência de 10Hz;
2. Quando ambas as *flags* estão setadas, passa a inverter o Led D3 com uma frequência de 1Hz;
3. Quando ambas as *flags* voltam a ficar resetadas, volta a inverter o Led D3 com uma frequência de 10Hz.

2.2 Criando as *Conditions Flags*

Como dito anteriormente, cabe a nós criamos as *flags* (coordinatingFlag1 e coordinatingFlag2). Mas para isso, é interessante encapsularmos elas em um arquivo .c e disponibilizarmos uma interface para cada uma através de um .h e as funções:

- **Set flag**;
- **Reset flag**;
- **Check flag**.

é possível fazer um .c/.h para cada uma das *coordinating Flags* ou criar as duas e dentro de um único par de arquivos e suas respectivas funções. Note que iremos criar essas *flags* dentro do arquivo .c a partir de um tipo enumerado:

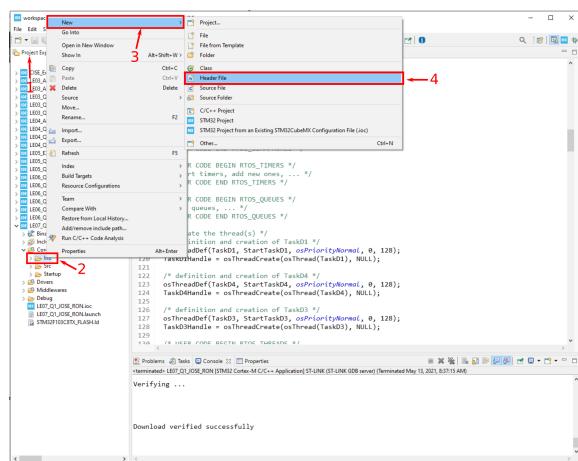
```
1 typedef enum { Set , Reset } Flag ;
```

criado no .h.

2.2.1 Criando um .h na STM32 CubeIDE

Para deixar o código organizado, é interessante adicionar nossos arquivos de cabeçalho juntos com os demais arquivos do projeto (**LE06_Q1_SEU_NOME → Core → Inc**). Para isso, basta seguir o passo-a-passo mostrado na Figura 2, no qual:

Figura 2: Criando um arquivo de cabeçalho no projeto.



Fonte - produzido pelo autor.

1. Na aba **Project Explorer**;

2. caminhe até a pasta **Inc** (**LE06_Q1_SEU_NOME → Core → Inc**) e clique com o botão direito do mouse;

3. vá em **New**;

4. e clique em **Header File**.

Será aberta uma janela *pop-up*, como mostrada na Figura 3. Nessa janela, coloque o nome do arquivo (condition_flag.h, por exemplo) em **Header file** e clique em **Finish**. Lembre-se de colocar .h no nome, para gerar o arquivo do tipo cabeçalho.

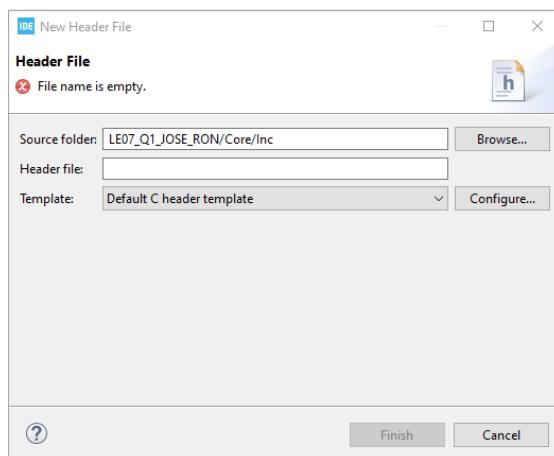
Abra o arquivo criado e veja que já foi colocado um cabeçalho com a data de criação e o nome do criador (nome do usuário do computador) e a proteção para que não seja adicionado mais de uma vez o arquivo no projeto:

```

1  /*
2   * condition_flag.h
3   *
4   * Created on: May 13, 2021
5   * Author: Jose Neto.
6   */
7
8 #ifndef INC_CONDITION_FLAG_H_
9 #define INC_CONDITION_FLAG_H_
10
11#endif /* INC_CONDITION_FLAG_H_ */

```

Figura 3: Nomeando o arquivo de cabeçalho.



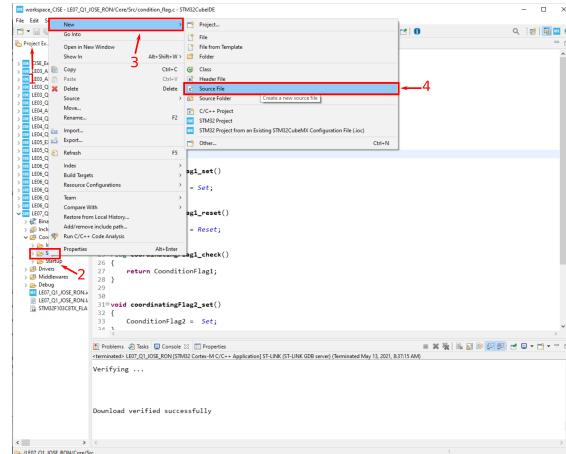
Fonte - produzido pelo autor.

2.2.2 Criando um .c na STM32 CubeIDE

O passo-a-passo criar o arquivo .c correspondente ao arquivo .h segue uma estrutura muito parecida, como mostrado na Figura 4 e Figura 5. Apenas note de que o caminho agora é: **LE06_Q1_SEU_NOME → Core → Src** e que o nome do arquivo deve terminar com .c.

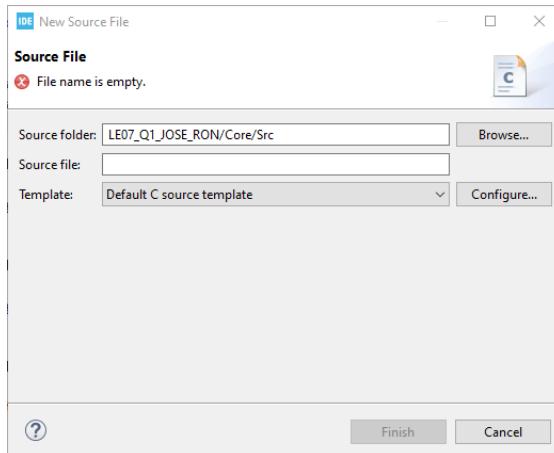
OBS: Voce passar o código das funções para acelerar o processo.

Figura 4: Criando um arquivo .c no projeto.



Fonte - produzido pelo autor.

Figura 5: Nomeando o arquivo de cabeçalho.



Fonte - produzido pelo autor.

3 Uso de *event flags* para sincronização unilateral

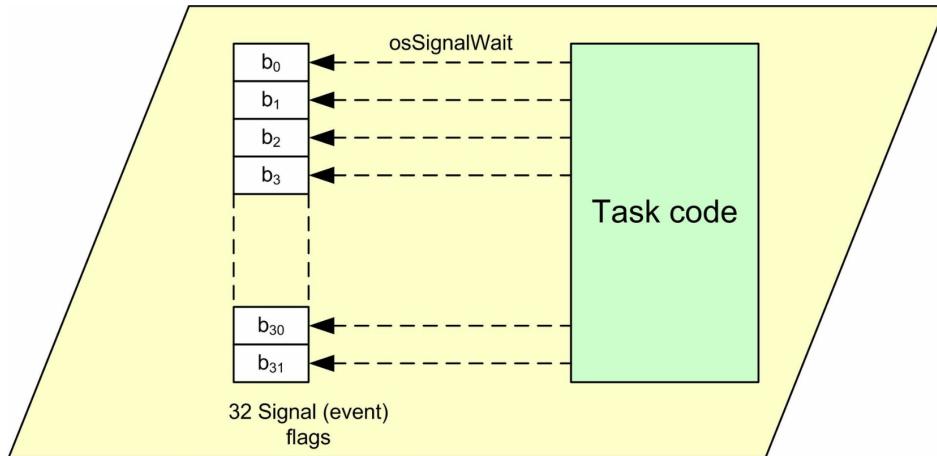
Neste segundo exercício (LE06_Q2_SEU_NOME) iremos ver como sincronizar *tasks* unilateralmente com o auxílio de estruturas nativas do FreeRTOS+CMSIS-RTOS, as *Signal (event) flags*. Infelizmente, dada não uniformidade na nomenclatura dos RTOS, essa estrutura acaba tendo um nome bem parecido com a estrutura utilizada para sincronização bilateral (**Signals**), mas fica aqui explicitado que **não é**. As *Signal (event) flags* trabalham como uma *event group flag* atrelada diretamente a uma *task*.

3.1 Trabalhando com *Signal (event) Flags*

Como dito, as *Signal (event) flags* são parte da estrutura de cada *thread* criada no FreeRTOS, como mostrado na Figura 6.

Cada *task* possui um conjunto de *signal flags* atribuídas a ela (um máximo possível de 32) e são identificados apenas por um número (bits em uma palavra). Portanto, elas não são itens definidos explicitamente, mas podem ser consideradas uma parte implícita da construção da *task*. O máximo real de *signal flags* por *thread* é especificado no arquivo *cmsis_os.h*, como por exemplo: `#define`

Figura 6: Estrutura das *Signal (event) Flags* relacionadas a uma *Task*.



Fonte - [1].

`osFeature_Signals 8`. As funções relacionadas as *signal flags* são descritas a seguir ¹.

3.1.1 osSignalWait

```
2 osEvent osSignalWait( int32_t signals ,
4 )
```

- **Parâmetros:**

- **signals**: espera até que todas *signal flags* especificadas estejam setadas ou 0 se for para qualquer *signal flag* acordar a *task*.
- **millisec**: tempo que o sistema esperará pela *signal flag* ser setada. Pode ter os valores:
 - * millisec = 0: a função retorna instantaneamente;
 - * millisec == osWaitForever: a função esperará por um tempo “infinito” até que a *signal flag* seja setada.
 - * 0 < millisec < osWaitForever: tempo que a função ficará esperando a *signal flag* ser setada.

- **Retorno:**

- Código de status que indica como foi a execução da função:
 - * **osOK**: nenhuma *signal flag* foi recebida e que o *timeout* passado foi 0.
 - * **osEventTimeout**: *signal flag* não foi setada dentro do *timeout* passado.
 - * **osEventSignal**: a *signal flag* foi setada, value.signals contém as *signal flags*; **estas *signal flags* são limpas**;
 - * **osErrorValue**: ao valor passado em *signal* está fora dos valores permitidos;
 - * **osErrorISR**: essa função não pode ser chamada de dentro de uma rotina de tratamento de interrupção.

¹https://www.keil.com/pack/doc/cmsis/RTOS/html/group__CMSIS__RTOS__SignalMgmt.html#ga38860acda96df47da6923348d96fc4c9

3.1.2 osSignalSet

```
1 int32_t osSignalSet( osThreadId thread_id ,  
2                     int32_t signals  
3 )
```

- **Parâmetros:**

- **thread_id:** ponteiro para *task*.
 - **signals:** especifica os bits (*signal flags*) que serão setados.

- **Retorno:**

- O valor prévio das *signal flags* ou `0x80000000` caso os parâmetros passados estejam incorretos.

3.1.3 osSignalClear

ESSA FUNÇÃO NÃO ESTÁ IMPLEMENTADA NA VERSÃO DO CMSIS-RTOS QUE ESTAMOS UTILIZANDO!

```
1 int32_t osSignalClear( osThreadId thread_id ,  
2                      int32_t signals  
3 )
```

- **Parâmetros:**

- **thread_id:** ponteiro para *task*.
 - **signals:** especifica os bits (*signal flags*) que serão resetados.

- **Retorno:**

- O valor prévio das *signal flags* ou `0x80000000` caso os parâmetros passados estejam incorretos ou se for chamado de uma ISR (*interrupt service routine*).

3.2 O problema a ser resolvido

Iremos revisitar nosso problema de detectar a borda de descida dos botões a partir de uma interrupção dos botões SW1, SW2 e SW3. Para isso, revise a LE04 para ver como configurar esses pinos como geradores de interrupção. Logo, continuaremos tendo 3 *tasks* (**TaskD1**, **TaskD4** e **TaskD3**) cujos comportamentos são descritos a seguir, junto com as interrupções relacionadas aos botões SW1, SW2 e SW3.

OBS: BIT_0 é um *define* que aponta para o bit 0 da palavra de 32 bit , ou seja, igual a ‘1’.

3.2.1 TaskD1

Comportamento da **TaskD1**:

- **LOOP:**

1. Espera pela *flag* ser setada: `osSignalWait(BIT_0, osWaitForever);;`
2. Inverte o valor do Led D1;
3. Espera 1000 ms;

- **END OF LOOP**

3.2.2 TaskD2

Comportamento da **TaskD2**:

- **LOOP:**

1. Espera pela *flag* ser setada: `osSignalWait(BIT_0, osWaitForever);;`
2. Inverte o valor do Led D2;
3. Espera 1000 ms;

- **END OF LOOP**

3.2.3 TaskD3

Comportamento da **TaskD3**:

- **LOOP:**

1. Espera pela *flag* ser setada: `osSignalWait(BIT_0, osWaitForever);;`
2. Inverte o valor do Led D3;
3. Espera 1000 ms;

- **END OF LOOP**

3.2.4 Interrupção de SW1

- Seta o bit 0 das *signal flags* da **TaskD1**: `osSignalSet(TaskD1Handle, BIT_0);`

3.2.5 Interrupção de SW2

- Seta o bit 0 das *signal flags* da **TaskD2**: `osSignalSet(TaskD2Handle, BIT_0);`

3.2.6 Interrupção de SW3

- Seta o bit 0 das *signal flags* da **TaskD3**: `osSignalSet(TaskD3Handle, BIT_0);`

4 Uso de *semaphores* para prover sincronização unilateral

Neste terceiro exercício (LE06_Q3_SEU_NOME), estamos interessados em mostrar como usar um *semaphore* para sincronização unilateral. Para criar semáforos na STM32 CubeIDE, revisite a LE05. O intuito é refazer o exercício da Seção 3 trocando o uso da *signal flag* por um semáforo. Para isso crie três semáforos, **SemaforoD1**, **SemaforoD2** e **SemaforoD3**.

Para que as interrupções sirvam para liberar o recurso, basta:

1. chamar `osSemaphoreWait(SemaforoDXHandle, osWaitForever)`; para os três semáforos antes da chamada do `osKernelStart()`;
- Outra solução:** chamar `osSemaphoreWait(SemaforoDXHandle, osWaitForever)`; em cada *task* correspondente antes de entrar no *loop* infinito. Ou seja, na área de definição de variáveis da *task*;
2. substituir `osSignalWait(BIT_0, osWaitForever)`; por `osSemaphoreWait(SemaforoDXHandle, osWaitForever)`; em cada uma das *tasks* (**substituindo SemaforoDXHandle pelo ponteiro correspondente**);
3. substituir `osSignalSet(TaskDXHandle, BIT_0)`; por `osSemaphoreRelease(SemaforoDXHandle)`; em cada uma das 3 interrupções dos botões (**substituindo SemaforoDXHandle pelo ponteiro correspondente**);

5 Uso de *semaphores* para prover sincronização bilateral

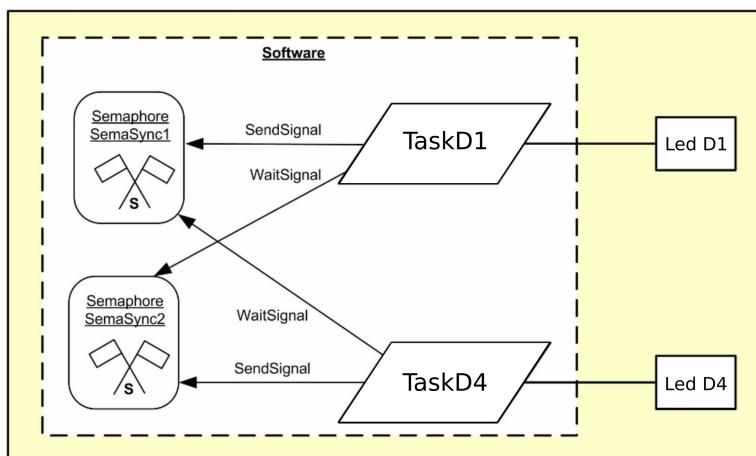
Neste quarto exercício (LE06_Q4_SEU_NOME) queremos mostrar como *semaphores* podem ser utilizados para sincronização bilateral de *tasks*.

Como mostrado na aula teórica anterior, isso pode ser alcançado utilizando dois semáforos, uma vez que normalmente não conseguíramos prever qual *task* chegará primeiro ao ponto de encontro.

5.1 O problema a ser resolvido

O sistema da quarta questão é mostrado na Figura 7 e o comportamento que desejamos implementar é o descrito na sequência:

Figura 7: Diagrama de *Tasks* do Sistema da Questão 4.



Fonte - Adaptado de [1].

- Ambas as *tasks* **TaskD1** e **TaskD4** iniciam ‘simultaneamente’ em $t = 0\text{s}$.
- Após a **TaskD1** inicializar, ela roda até $t = 10\text{s}$, invertendo o valor do Led D1 com uma frequência de 1 Hz.
- Após a **TaskD4** inicializar, ela roda até $t = 5\text{s}$, invertendo o valor do Led D4 com uma frequência de 10 Hz.
- Em $t = 5\text{s}$, **TaskD4** faz um pedido de sincronismo (*synchronizing call*) e nesse ponto entra em suspensão.
- Em $t = 10\text{s}$ **TaskD1** responde ao pedido de sincronismo, e então roda pelos próximos 5s (até $t = 15\text{s}$) invertendo o Led D1 com uma taxa de 10Hz.
- Em $t = 15\text{s}$ a **TaskD1** faz uma chamada de sincronização, o que causa a sua suspensão.
- Em $t = 20\text{s}$ a **TaskD4** responde a sincronização, o que acordará a **TaskD1**. Neste ponto todo o processo se repete.

OBS: Lembre-se de tomar os semáforos antes de iniciar o *scheduler* no começo do código.

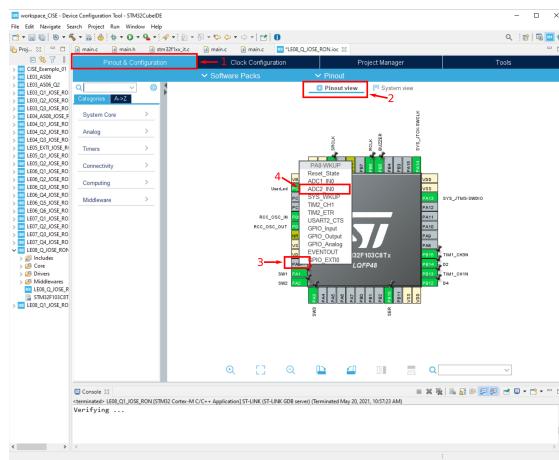
6 Configurando o Conversor Analógico Digital

Neste quinto exercício (LE06_Q5_SEU_NOME) iremos ver como configurar o conversor Analógico Digital para converter valores vindo do potenciômetro do **Shield Multifunções** que está ligado no pino PA0 da nossa placa **Blue Pill**.

6.1 Configurando um pino para captura do conversor AD

Para configurar um pino de entrada como canal de captura do conversor AD, basta seguir o passo-a-passo indicado na Figura 8, no qual:

Figura 8: Configuração do pino de entrada PA0 como canal de captura do conversor ADC2.



Fonte - Produzido pelo Autor.

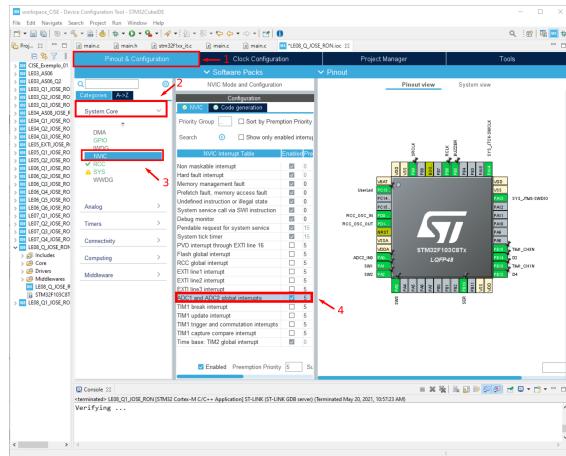
1. No ambiente de configuração vá em **Pinout & Configuration**;
2. No **Pinout view**;

3. Selecione o pino **PA0**;
4. Na lista suspensa que irá se abrir, selecione a opção **ADC2_IN0**.

6.2 Habilitando a interrupção do conversor AD

Para habilitar a interrupção dos conversores AD, basta seguir o passo-a-passo indicado na Figura 9, no qual:

Figura 9: Habilitando a interrupção do ADC2.



Fonte - Produzido pelo Autor.

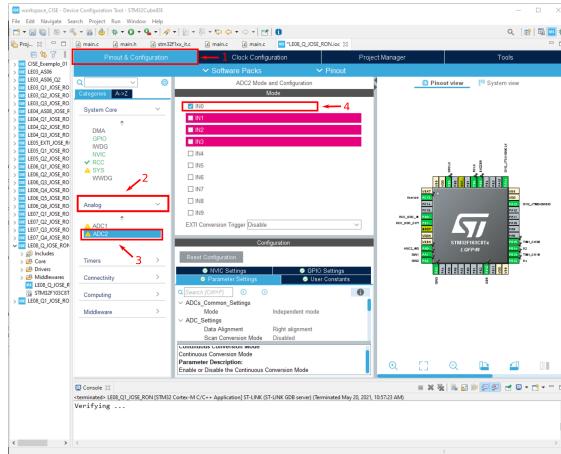
1. No ambiente de configuração vá em **Pinout & Configuration**;
2. Na seção **System Core**;
3. Selecione a opção **NVIC**;
4. Na coluna **Enable**, selecione a linha **ADC1 and ACD3 global interrupts**.

6.3 Habilitando o canal de captura IN0 do conversor AD

Para habilitar ou desabilitar um canal de captura do ADC2, basta seguir o passo-a-passo indicado na Figura 10, no qual:

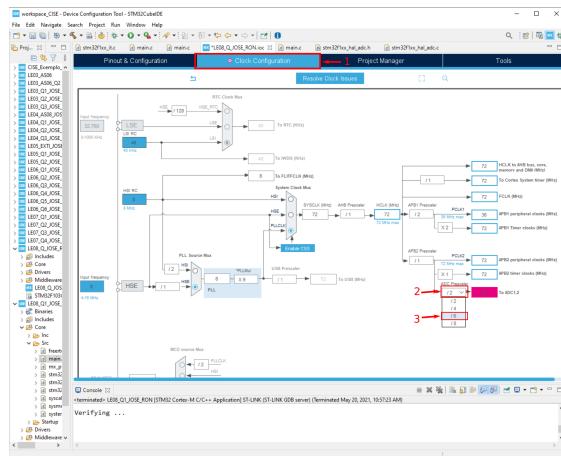
1. No ambiente de configuração vá em **Pinout & Configuration**;
2. Na seção **Analog**;
3. Selecione a opção **ADC2**;
4. Na seção **ADC2 Mode and Configuration**, marque o canal **IN0**.

Figura 10: Habilitando o canal de captura do ADC2.



Fonte - Produzido pelo Autor.

Figura 11: Configurando o clock do ADC2.



Fonte - Produzido pelo Autor.

6.4 Configurando o Clock para o Conversor AD

Assim que configurarmos o ADC2, será possível ver que a IDE estará reclamando (apontando um erro) na geração do *clock*. O *clock* para os conversores está configurado para 36 MHz e é muito alto para esse periférico. Para configurá-lo com uma frequência menor, basta seguir o passo-a-passo indicado na Figura 11, no qual:

1. No ambiente de configuração vá em **Clock Configuration**;
2. Clique em **ADC Prescaler**;
3. Selecione a opção **/6**.

6.5 Funções para trabalhar com o conversor AD

Para inicializar uma captura do ADC que gerará uma interrupção, é utilizada a função:

```
1 HAL_ADC_Start_IT(&hadc2);
```

O ADC2 irá gerar uma interrupção quando a conversão terminar. Logo, precisamos “resgatar” esse valor convertido na rotina de tratamento da interrupção. Para isso, nos escreveremos no main.c a função:

```
1 void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
3     if(hadc->Instance == ADC2) // Verifica se o ADC passado e o que estamos
4         trabalhando
5     {
6         // NOSSO CODIGO
7     }
}
```

Esta função já existe no arquivo stm32f1xx_hal.c, pois ela é chamada dentro da rotina de tratamento de interrupção (função ADC1_2_IRQHandler() dentro do arquivo stm32f1xx_it.c), no entanto ela é criada __weak para podermos reescrevê-la.

OBS: Copie a função do arquivo stm32f1xx_hal.c para garantir que o nome está correto e os espaços e variáveis também estão corretos.

A função para pegar o valor atual do conversor ADC é:

```
1 valor_adc = HAL_ADC_GetValue(&hadc2); // capta valor adc
```

em que valor_adc é uma variável que está recebendo o valor convertido pelo ADC e hadc2 é a estrutura relacionada ao ADC2 que foi criado pela IDE após a configuração do ADC2. Logo, caso tenhamos utilizado o ADC1 no lugar do ADC2, a única coisa que precisamos mudar é o argumento passado na função.

Lembrando que os ADCs da BluePill devolvem um número sem sinal de 12 bits.

7 Passando valores a partir de uma Queue

Nosso intuito aqui é criar uma *Queue* que irá passar o valor convertido pelo conversor AD para ser utilizado por uma *Task*.

7.1 Criando uma Queue da STM32CubeIDE

Para criar uma *Queue*, basta seguir o passo-a-passou indicado na Figura 12, no qual:

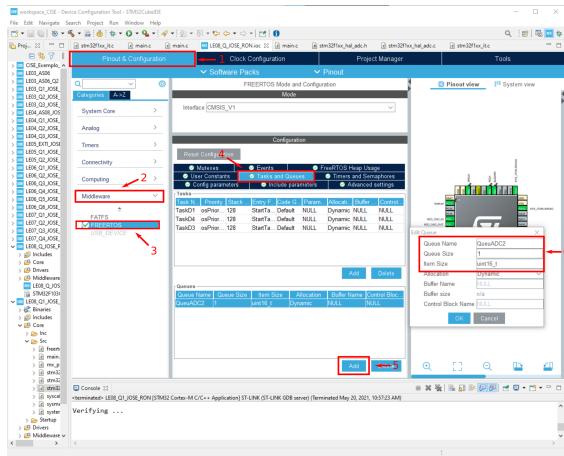
1. No ambiente de configuração, vá em **Pinout & Configuration**;
2. Na seção **Middleware**;
3. Selecione o **FREERTOS**;
4. Na seção **Configuration**, selecione a aba **Tasks and Queues**;
5. Em **Queues** clique em **ADD**;
6. Irá abrir uma janela pop-up, nela coloque o nome da *Queue*, o tamanho e o tipo de dado, como mostrado na Figura 12.

7.2 Funções para trabalhar com uma Queue

Basicamente existem duas funções para trabalhar com as filas²:

²https://www.keil.com/pack/doc/cmsis/RTOS/html/group__CMSIS__RTOS__Message.html

Figura 12: Configurando o clock do ADC2.



Fonte - Produzido pelo Autor.

7.2.1 osMessageGet

```
1 osEvent osMessageGet(osMessageQId queue_id,
                      uint32_t millisec
3 )
```

- **Parâmetros:**

- **queue_id:** ponteiro para a *Queue*.
- **millisec:** tempo que o sistema esperará por algum valor. Pode ter os valores:
 - * **millisec = 0:** a função retorna instantaneamente;
 - * **millisec == osWaitForever:** a função esperará por um tempo “infinito” até que a *signal flag* seja setada.
 - * **0 < millisec < osWaitForever:** tempo que a função ficará esperando a *signal flag* ser setada.

- **Retorno:**

- Estrutura que retorna entre outras coisas o valor da fila, além do status:
 - * **osOK:** não existe uma mensagem na *Queue* e nenhum *Timeout* foi passado.
 - * **osEventTimeout:** nenhuma mensagem chegou na *Queue* dentro do *Timeout* passado.
 - * **osEventMessage:** mensagem recebida.
 - * **osErrorParameter:** parâmetros inválidos;

7.2.2 osMessagePut

```
1 osStatus osMessagePut(osMessageQId queue_id,
                      uint32_t info,
                      uint32_t millisec
3 )
```

- **Parâmetros:**

- **queue_id:** ponteiro para a *Queue*.
- **info** informação da mensagem.
- **millisec:** tempo que o sistema esperará por algum valor. Pode ter os valores:
 - * millisec = 0: a função retorna instantaneamente;
 - * millisec == osWaitForever: a função esperará por um tempo “infinito” até que a *signal flag* seja setada.
 - * 0 < millisec < osWaitForever: tempo que a função ficará esperando a *signal flag* ser setada.

- **Retorno:**

- status de retorno da função:
 - * **osOK:** mensagem foi colocada dentro da *Queue*.
 - * **osErrorResource:** não há memória disponível na *Queue*;
 - * **osErrorTimeoutResource:** não há memória disponível na *Queue* dentro do *Timeout* passado.
 - * **osErrorParameter:** parâmetros inválidos;

OsEvent tem sua definição no arquivo cmsis_os.h dada por:

```

1 // Event structure contains detailed information about an event.
2 // \note MUST REMAIN UNCHANGED: \b os_event shall be consistent in every CMSIS-
3 // RTOS.
4 // However the struct may be extended at the end.
5 typedef struct {
6     osStatus           status;          ///< status code: event or error
7     information
8     union {
9         uint32_t        v;              ///< message as 32-bit value
10        void            *p;             ///< message or mail as void pointer
11        int32_t         signals;       ///< signal flags
12    } value;
13    union {
14        osMailQId      mail_id;       ///< mail id obtained by \ref
15        osMailCreate
16        osMessageQId   message_id;   ///< message id obtained by \ref
17        osMessageCreate
18    } def;                         ///< event definition
19 } osEvent;

```

7.2.3 Exemplo de uso

Para ajudar a entender como utilizar a *Queue* observe o exemplo a seguir em que nossa função de *callback* da interrupção de ADC é:

```

1 // fn que atende ao callback da ISR do conversor ADC2
2 void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
3 {
4     uint16_t val_adc = 0;
5     if(hadc->Instance == ADC2) // Verifica se o ADC passado e o que estamos
6         trabalhando

```

```

7     {
8         val_adc = HAL_ADC_GetValue(&hadc2); // capta valor adc
9         osMessagePut(QueuADC2Handle, val_adc, 1000);
10    }
}

```

e nossa *Task* que irá receber o valor convertido

```

2 void StartTaskD1(void const * argument)
{
3     /* USER CODE BEGIN 5 */
4     uint16_t ultimoValor = 0; //Guarda o valor lido da Queue
5     uint16_t miliVolt = 0;    //valor lido do potenciometro
6     osEvent QreadState;      //retorno da função que le a fila
7
8     /* Infinite loop */
9     for(;;)
10    {
11        HAL_ADC_Start_IT(&hadc2); //Inicia uma conversão do ADC
12        QreadState = osMessageGet(QueuADC2Handle, 1000); //Espera por 1s ter algo na
13        fila
14        ultimoValor = QreadState.value.v;                  //carrega o valor da fila
15        para uma variável
16        miliVolt = ultimoValor * 3300 / 4095;           //transforma o valor de
17        0-4095 de 0-3300;
18        osDelay(1000);
19    }
20    /* USER CODE END 5 */
21 }

```

8 Utilizando os *Displays* de 7 seguimentos

Para podermos ver os valores convertidos sem a necessidade do debugger, podemos usar os 4 *displays* de 7 seguimentos que temos no *Shield Multifunção*. Para isso vamos usar uma biblioteca já pronta para facilitar (os arquivos `mx_prat_05_funcoes.c` e `mx_prat_05_funcoes.h` disponibilizados com esse texto) [2]. Adicione os arquivos no projeto movendo-os para as pastas **Core → Src** e **Core → Inc** o `.c` e o `.h`, respectivamente.

Esta biblioteca nos disponibiliza duas funções que ajudam muito a enviar dados aos *displays*: `conv_7_seg()` e `serializar`:

8.1 `conv_7_seg`

```
int16_t conv_7_seg(int NumHex);
```

- **Parâmetros:**

- **NumHex:** deve receber um número em hexadecimal a ser convertido nos bits que deverão ligar no display de 7 seguimentos.

- **Retorno:**

- Se $0 \leq \text{NumHex} \leq 15$, a função retorna os bits que devem ser ligados para que apareça no *display*;
- Se $\text{NumHex} = 16$: retorna todos os leds apagados;
- Se $\text{NumHex} > 16$: retorna erro 0xBF00;

8.2 serializar

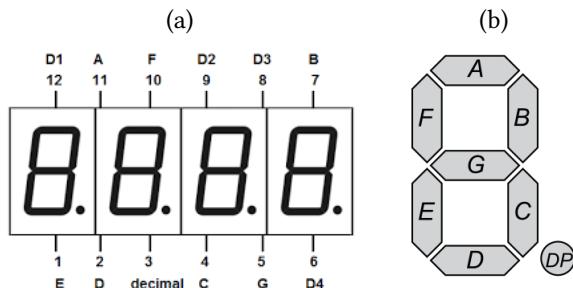
```
1 void serializar(int ser_data);
```

- **Parâmetros:**

- **ser_data:** dado a ser enviado para o chip 74HC595.

A função envia 16 bits, os 4 primeiros bits de *ser_data* (menos significativos) dizem quais os *displays* serão carregados com o valor passado, se o bit 0 = 1, é carregado no display 1, se o bit 1 = 1 é carregado no display 2 e assim por diante. Logo, é possível carregar o mesmo valor em todos os *displays* de uma só vez, ou na combinação que desejar. Os 12 mais significativos segue a ordem mostrada na Figura 13.

Figura 13: Esquemático: (a) 4 displays de 7 segmentos; (b) Posição dos LEDS a-g no display de 7 segmentos.



Fonte - https://pt.wikipedia.org/wiki/Ficheiro:7_segment_display_labeled.svg.

8.3 Exemplo de Uso

Para ajudar o entendimento de como trabalhar com os *displays*, analise o código mostrado a seguir e veja seu comportamento a placa:

```
1 void StartTaskD4(void const * argument)
{
3  /* USER CODE BEGIN StartTaskD4 */
4  uint16_t serial_data = 0; //Guarda o dado a se passado para os displays
5  uint16_t val7seg = 0;      //Guarda o valor convertido de HEX para os leds do
   display
6  int8_t   contador = 0;    //contador usado para mostrar um valor diferente no
   display
7  int8_t   qual_display = 0; //para dizer qual o display que vai ser mostrado
8  TickType_t TempoDelay = osKernelSysTick();
9  /* Infinite loop */
10 for(;;)
11 {
```

```

13     if( qual_display < 4)
14     {
15         qual_display++;
16     }
17     else
18     {
19         qual_display = 0;
20         if( contador < 17)
21         {
22             contador++;
23         }
24     else
25     {
26         contador = 0;
27     }
28 }
29
30 switch( qual_display)
31 {
32     case 0:
33     {
34         serial_data = DISPLAY_1; // display #1
35         break;
36     }
37     case 1:
38     {
39         serial_data = DISPLAY_2; // display #2
40         break;
41     }
42     case 2:
43     {
44         serial_data = DISPLAY_3; // display #3
45         break;
46     }
47     case 3:
48     {
49         serial_data = DISPLAY_4; // display #4
50         break;
51     }
52     case 4:
53     {
54         serial_data = DISPLAY_1|DISPLAY_2|DISPLAY_3|DISPLAY_4; // todos
55         break;
56     }
57     default:
58     {
59         serial_data = DISPLAY_1|DISPLAY_2|DISPLAY_3|DISPLAY_4; // todos
60         break;
61     }
62 }
63 val7seg = conv_7_seg(contador);
64 serial_data |= val7seg; // OR com val7seg = dado a serializar
65 serializar(serial_data); // serializa dado p/74HC595 (shift reg)
66 osDelayUntil(&TempoDelay, 500);
67 }
68 /* USER CODE END StartTaskD4 */
69 }
```

9 Exercício Proposto

O Exercício Proposto nesta quinta questão será fazer com que o valor convertido pelo ADC2 seja guardado em uma *Queue* e que uma *Task* leia esse valor dessa *Queue* e o imprima nos displays 7 segmentos. Em que o display 4 é o número mais significativo e o *display 1* o menos significativo. Logo, caso será mostrado de 0mV até próximo a 3.300 mV, dependendo do estado do potenciômetro.

Referências

- [1] J. Cooling, *Real-time Operating Systems: Book 2 - The Practice (Using STM Cube, FreeRTOS and the STM32 Discovery Board)*. Lindentree Associates, 2018.
- [2] João Ranhel, *Apostila: Sistemas Microprocessados – ARM CORTEX*, ver 5 ed., 2019.