Disciplina:	ME587 - Sistemas Digitais 2
Professor:	José Rodrigues de Oliveira Neto

Universidade Federal de Pernambuco
Centro de Tecnologia e Geociências
Departamento de Engenharia Mecânica

Lista de Exercícios #05 - 2024.1

O objetivo desta lista é testar as soluções de **Exclusão Mútua** do **FreeRTOS** (e consequentemente do CMSIS-RTOS) utilizando a **STM32 CubeIDE**.

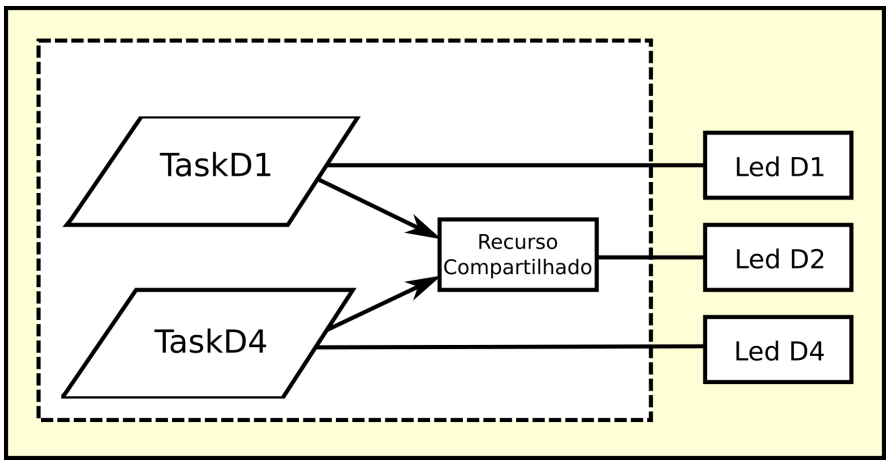
1 Instruções Gerais

- 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 "`LE05_SEU_NOME`", em que `SEU_NOME` é o seu nome;
- Compacte a pasta e inclua como resposta da atividade.

2 Introdução ao Problema

Como nossa intenção para essa prática é simular o uso de um recurso compartilhado, tentaremos fazer isso de uma forma que seja mais previsível o comportamento do sistema para tentar forçar a tentativa pelas *tasks* de acesso simultâneo ao recurso. Por isso, iremos trabalhar com o sistema mostrado na Figura 1. Nela é mostrado duas *threads*, `TaskD1` e `TaskD4` que terão como recurso compartilhado o acesso a um recurso fictício e o led `D2` será usado par nos indicar quando mais de uma *task* tentar acessar esse recurso. Simularemos o tempo de uso do recurso com *delays* para forçar a espera pelo recurso.

Figura 1: Diagrama de *Tasks* do Sistema.



Fonte - Adaptado de [1].

Tendo isso em mente, já é possível criar um projeto (`LE05_Q1_SEU_NOME`) e configura-lo para ter duas *tasks* com os nomes indicados. Lembrem-se de habilitar o `osDelayUntil()`.

OBS: Lembre-se de sempre adicionar seus projetos uma *task* que faz piscar o `UserLed` (pino PC13) para indicar que a placa está funcionando corretamente.

3 Uso de uma *Flag* para Exclusão Mútua

Como é sabido, o jeito mais simples, mas não tão seguro e eficiente, de implementar a exclusão mútua é através de uma *flag* criada como variável global e que as *tasks* manipulam para saber se podem ou não acessar o recurso.

Dado isso, crie essa variável global `flagD2` para funcionar como *flag*. Uma sugestão é criar um **`typedef enum`** com dois valores:

- **`Up`:** indicando que o recurso está disponível e
- **`Down`:** indicando que o recurso está em uso.

Com o uso dessa variável escreva um código cujo o comportamento de cada *thread* é explicado na sequência:

3.1 `TaskD1`

Essa *task* deve:

- **`Loop`:**
    1. Liga o led *D1*;
    2. **Tenta acessar o recurso compartilhado;**
    3. Desliga o led `D1`;
    4. Espera com 500ms;
- **`End loop`**

3.2 `TaskD4`

Essa *task* deve:

- **`Loop`:**
    1. Liga o led *D4*;
    2. **Tenta acessar o recurso compartilhado;**
    3. Desliga o led `D4`;
    4. Espera com 100ms;
- **`End loop`**

3.3 Acesso ao recurso compartilhado

O acesso ao recurso compartilhado para as duas *tasks* se dará da mesma forma:

1. Cheque se `flagD2` está `Up`;
2. Se `UP`:
   (a) Coloque a `flagD2` para `Down`;
   (b) force um delay de 500ms para simular o uso do recurso compartilhado;
   (c) Coloque a `flagD2` para `Up`;
3. Caso contrário, ligue o `D2`;

3.4 Resultado Esperado e Modificação Sugerida

Se tudo der certo, você verá que o led `D2` não demorará a ser ligado, indicando houve a primeira contenção do uso do recurso. No entanto esse led permanecerá ligado e com isso não saberemos com que frequência essa contensão ocorre. Para vermos isso, é possível modificar o código e ao invés de apenas ligar o led `D2`, caso a *task* ache a *flag* abaixada, o led `D2` deve ficar 10ms acesso e depois seja apagado.

Crie um novo projeto (`LE05_Q2_SEU_NOME`) e faça a modificação indicada e veja o comportamento do sistema.

4 Suspendendo o *Scheduler*

Uma maneira simples e segura de eliminar a contensão de um recurso é a partir da suspensão do *scheduler*. Pois, se conseguirmos garantir que uma e apenas uma *task* tem acesso ao recurso compartilhado por vez, é impossível que a interferência ocorra. E a maneira mais simples de garantir isso é desabilitando o *scheduler* enquanto a *task* está usando o recurso. Na prática isso é feito desabilitando as interrupções durante o processo, usando as funções do próprio RTOS.

Essas funções são[1]:

- **`taskENTER_CRITICAL( );`** - desabilita as interrupções e com isso evita que o *scheduler* tente mudar o contexto;
- **`taskEXIT_CRITICAL( );`** - habilita novamente as interrupções indicando que a *task* saiu da parte crítica do código;

4.1 Exercício Proposto

Crie um novo projeto (`LE05_Q3_SEU_NOME`) e modifique cada uma das *tasks* do exercício anterior para:

1. Desabilite as interrupções;
2. Acesse o recurso compartilhado;
3. Habilite as interrupções;

[1] https://www.freertos.org/a00020.html

O resto do comportamento deve ser o mesmo. No entanto, uma vez que desabilitamos as interrupções é esperado que a função `osDelay()` não funcione como esperado, já que ela utiliza o estouro do timer do sistema. Logo, faça uma função `meuDelay(TEMPO)` utilizando um *loop* `for()` que não faz nada para gerar os *delays*.

Os tempos dos *delays* não precisam ser tão precisos, mas é mais fácil fazer a função e testá-la em uma *task* que apenas inverte um led com um tempo grande (10 segundos por exemplo), calibrar o *delay*, e só depois refazer o projeto utilizando ela.

4.2 Resultado Esperando

Caso o sistema tenha sido implementado de forma correta, será possível ver que o led `D2` não ligará mais, já que o *scheduler* está desligado sempre que uma das *tasks* consegue o acesso ao recurso e não haverá a preempção até que o recurso seja liberado.

Embora essa técnica seja segura e fácil de implementar, o lado negativo dela é que ela simplesmente desliga o *multitasking* e nenhuma outra *task* vai ter acesso ao processador, as *tasks* que precisam rodar de forma periódica terão seu comportamento prejudicado e principalmente as *tasks* habilitadas por eventos terão seu comportamento comprometido.

Logo,

Se você precisar utilizar essa técnica, você deve garantir que a suspensão do scheduler seja tão curta quanto possível.

5 Mostrando a deterioração do Sistema

Para verificarmos o que foi dito anteriormente, caso ainda não tenha colocado, adicione uma *task* para ficar ligando o `UserLed` da BluePill por 100 ms a cada 2s segundo. **Depois de testar com diferentes valores de tempo ligado/desligado**. Teste também utilizando a função `osDelay()` e `meuDelay()`.

5.1 Resultado Esperado

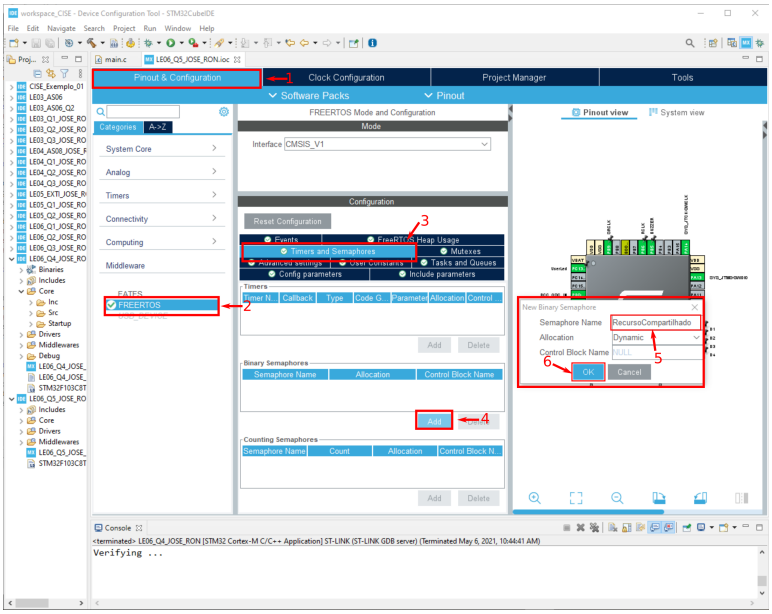
Se o sistema for implementado como pedido, vemos que o `UserLed` não fica o tempo esperado em um ou em zero. Este exemplo simples já mostra o quanto o comportamento dinâmico do sistema é comprometido pela suspensão do *scheduler*.

6 Uso de Semáforos

Vamos agora usar uma solução do RTOS para a exclusão mútua, um **Semáforo Binário**. Logo, será necessário criar um novo projeto (`LE05_Q4_SEU_NOME`). Uma vez criado o novo projeto, para criar um semáforo basta seguir o passo-a-passo mostrado na Figura 2, no qual:

1. Selecione a aba **Pinout & Configuration** no ambiente de configuração;
2. Na aba **Middleware** selecione a opção **FREERTOS**;
3. Na seção **Configuration** que irá abrir, selecione a aba **Timers and Semaphores**;
4. Na seção **Binary Semaphores**, que irá abrir, clique em **Add**;
5. Será aberta uma janela *pop up* intitulada **New Binary Semaphore**, no campo referente ao **Semaphore Name** escreva o nome que dará ao semáforo (`RecursoCompartilhado`);

Figura 2: Criando um Semáforo Binário.



Fonte - produzido pelo autor.

6. Para terminar de criar o semáforo, clique em **Ok**.

Uma vez que o projeto seja salvo e compilado para gerar o código, é possível ver que foi criado um ponteiro para o semáforo:

```
/* Private variables ---------------------------------------------------------*/
osThreadId TaskD1Handle;
osThreadId TaskD4Handle;
osThreadId TaskD3Handle;
osSemaphoreId RecursoCompartilhadoHandle;
/* USER CODE BEGIN PV */
```

e na `main()` foi criado o semáforo e passado para o ponteiro:

```
/* Create the semaphores(s) */
/* definition and creation of RecursoCompartilhado */
osSemaphoreDef(RecursoCompartilhado);
RecursoCompartilhadoHandle = osSemaphoreCreate(osSemaphore(RecursoCompartilhado)
    , 1);

/* USER CODE BEGIN RTOS_SEMAPHORES */
```

6.1 Funções para lidar com o Semáforo

Para usar o semáforo necessitamos das funções *Wait* e *Release*, no CMSIS-RTOS essas funções são[2]:

6.1.1 osSemaphoreWait

[2] https://www.keil.com/pack/doc/cmsis/RTOS/html/group__CMSIS__RTOS__SemaphoreMgmt.html#gacc15b0fc8ce1167fe43da33042e62098

```
int32_t osSemaphoreWait(osSemaphoreId    semaphore_id,
                        uint32_t   millisec
                        )
```

- **Parâmetros:**
  - **semaphore_id:** ponteiro para o semáforo.
  - **millisec:** tempo que o sistema esperará pela liberação do semáforo. Pode ter os valores:
    - `millisec = 0`: a função retorna instantaneamente;
    - `millisec == osWaitForever`: a função esperará por um tempo "infinito" até que ao semáforo seja liberado.
    - `0 < millisec < osWaitForever`: tempo que a função ficará esperando pela liberação do semáforo.
- **Retorno:**
  - Número de *tokens* disponíveis (lembre-se que o semáforo binário é só um caso particular do semáforo geral), ou -1 em caso de parâmetros errados.

6.1.2 osSemaphoreRelease

```
osStatus osSemaphoreRelease(osSemaphoreId    semaphore_id)
```

- **Parâmetros:**
  - **semaphore_id:** ponteiro para o semáforo.
- **Retorno:**
  - Código de status que indica como foi a execução da função:
    - **osOK:** o semáforo foi liberado.
    - **osErrorResource:** todos os *tokens* já tinham sido liberados.
    - **osErrorParameter:** o parâmetro `semaphore_id` passado está incorreto.

6.2 Exercício Proposto

O código terá as mesmas três *threads*: `TaskD1`, `TaskD4` e `TaskUserLed` em que:

6.2.1 TaskD1

- **Loop:**
  1. Tenta acessar o recurso (use `osSemaphoreWait(RecursoCompartilhadoHandle, osWaitForever);`)
  2. Liga o led `D1`;
  3. Espera com 500ms (use o `meuDelay`);
  4. Desliga o led `D1`;
  5. Libera o recurso (`osSemaphoreRelease(RecursoCompartilhadoHandle);`);
  6. Espera com 500ms (use o `meuDelay`);
- **End loop**

6.2.2 TaskD2

- **Loop:**
    1. Tenta acessar o recurso (use `osSemaphoreWait(RecursoCompartilhadoHandle, osWaitForever);`)
    2. Liga o led `D2`;
    3. Espera com 500ms (use o `meuDelay`);
    4. Desliga o led `D2`;
    5. Libera o recurso (`osSemaphoreRelease(RecursoCompartilhadoHandle);`);
    6. Espera com 100ms (use o `meuDelay`);
- **End loop**

6.2.3 TaskUserLed

- **Loop:**
    1. Inverte o valor do led `UserLed`;
    2. Espera com 50ms (use o `osDelay`);
- **End loop**

7 Uso de Mutex

O uso do mutex é bem parecido com o uso do semáforo, de fato, para o FreeRTOS o mutex é apenas um dos tipos de semáforos implementados no sistema[3]. No entanto, o mutex garante que apenas a *task* que possui o recurso pode liberar o recurso e, uma coisa que ainda não vimos mas que é importante para evitar inversão de prioridade de *tasks*, o mutex possui herança de prioridade, mas falaremos disso em aulas posteriores.

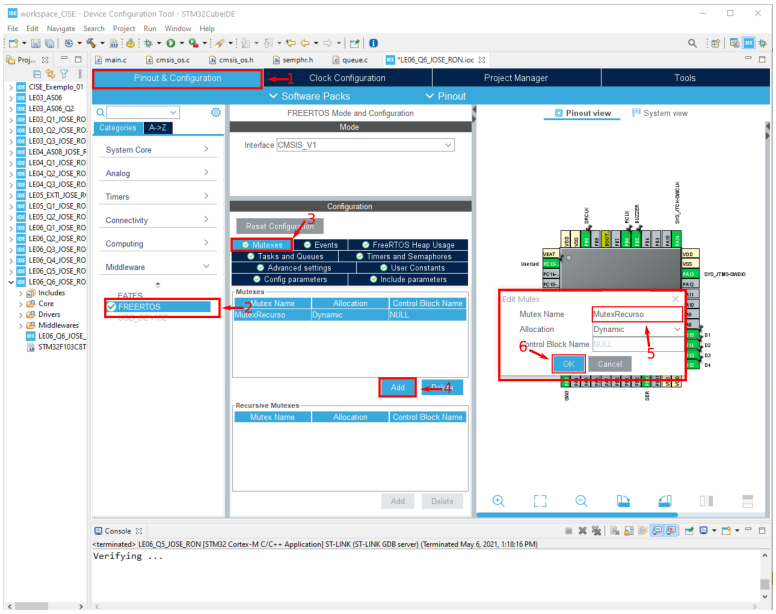
Agora, para usar o mutex precisamos primeiramente criar um novo projeto (`LE05_Q5_SEU_NOME`), para não perdermos o código passado e seguir o passo-a-passo da Figura 3, em que:

1. Selecione a aba **Pinout & Configuration** no ambiente de configuração;
2. Na aba **Middleware** selecione a opção **FREERTOS**;
3. Na seção **Configuration** que irá abrir, selecione a aba **Mutexes**;
4. Na seção **Mutexes**, que irá abrir, clique em **Add**;
5. Será aberta uma janela *pop up* intitulada **Edit Mutex**, no campo referente ao **Mutex Name** escreva o nome que dará ao semáforo (`MutexRecurso`);
6. Para terminar de criar o mutex, clique em **Ok**.

Uma vez que o projeto seja salvo e compilado para gerar o código, é possível ver que foi criado um ponteiro para o mutex:

[3] https://www.freertos.org/a00113.html

Figura 3: Criando um Mutex.



Fonte - produzido pelo autor.

```
/* Private variables ---------------------------------------------------------*/
osThreadId TaskD1Handle;
osThreadId TaskD4Handle;
osThreadId TaskD3Handle;
osMutexId MutexRecursoHandle;
/* USER CODE BEGIN PV */
```

e na `main()` foi criado o mutex e passado para o ponteiro:

```
  /* Create the mutex(es) */
  /* definition and creation of MutexRecurso */
  osMutexDef(MutexRecurso);
  MutexRecursoHandle = osMutexCreate(osMutex(MutexRecurso));

  /* USER CODE BEGIN RTOS_MUTEX */
```

7.1 Funções para lidar com o Mutex

Para usar o mutex necessitamos das funções *Lock* e *Unlock*, no CMSIS-RTOS essas funções são[4]:

7.1.1 osMutexWait

```
osStatus osMutexWait(osMutexId   mutex_id,
                     uint32_t    millisec
)
```

[4] https://www.keil.com/pack/doc/cmsis/RTOS/html/group__CMSIS__RTOS__MutexMgmt.html

- **Parâmetros:**
  - **`mutex_id`:** ponteiro para o semáforo.
  - **`millisec`:** tempo que o sistemas esperará pela liberação do mutex. Pode ter os valores:
    * `millisec = 0`: a função retorna instantaneamente;
    * `millisec == osWaitForever`: a função esperará por um tempo "infinito" até que ao mutex seja liberado.
    * `0 < millisec < osWaitForever`: tempo que a função ficará esperando pela liberação do mutex.
- **Retorno:**
  - Código de status que indica como foi a execução da função:
    * **`osOK`:** o mutex foi obtido.
    * **`osErrorTimeoutResource`:** o mutex não pode ser obtido no tempo dado.
    * **`osErrorResource`:** o mutex não foi obtido quando nenhum *timeout* foi passado.
    * **`osErrorParameter`:** o parâmetro `mutex_id` passado está errado.
    * **`osErrorISR`:** essa função não pode ser chamada de dentro de uma rotina de tratamento de interrupção.

7.1.2 osMutexRelease

```
1 osStatus osMutexRelease(osMutexId    mutex_id)
```

- **Parâmetros:**
  - **`mutex_id`:** ponteiro para o mutex.
- **Retorno:**
  - Código de status que indica como foi a execução da função:
    * **`osOK`:** o mutex foi liberado corretamente.
    * **`osErrorResource`:** o mutex não tinha sido obtido previamente.
    * **`osErrorParameter`:** o parâmetro `mutex_id` passado está incorreto.
    * **`osErrorISR`:** essa função não pode ser chamada de dentro de uma rotina de tratamento de interrupção.

7.2 Exercício Proposto

Refaça o Exercício Proposto na Seção 6.2 trocando ao semáforo pelo mutex.

Referências

[1] J. Cooling, *Real-time Operating Systems: Book 2 - The Practice (Using STM Cube, FreeRTOS and the STM32 Discovery Board)*. Lindentree Associates, 2018.