Insper

11 - Handout - RTOS

Rafael Corsi - rafael.corsi@insper.edu.br

Maio - 2017

Introdução

Nesse Handout iremos trabalhar com o uso de um sistema operacional de tempo real (RTOS) para gerenciarmos três LED e três botões (vamos refazer a entrega do tickTackTock porém agora com o uso do SO).

O sistema operacional a ser utilizado é o FreeRtos (www.freertos.org), um sistema operacional muito utilizado pela industria, sendo o segundo sistema operacional (20%) mais utilizado em projetos embarcados, perdendo só para o Linux embarcado [1].

[1] https://m.eet.com/media/1246048/2017-embedded-market-study.pdf

Exemplo - Walk-through

(45 minutos)

Objetivo : Entender as tarefas de um RTOS e fazer pequenas modificações no código

Iremos trabalhar com o exemplo do FreeRTOS que a Atmel disponibiliza para a placa SAME70-XPLD, esse exemplo já inclui um projeto com as configurações iniciais do OS e um código exemplo que faz o LED da placa piscar a uma frequência determinada.

No AtmelStudio, vá em :
-> File -> New -> Example Project ->
Filtre pelo microcontrolador : SAME70
e busque pelo exemplo *FreeRTOS Basic Example on …*
Criei o projeto no seu repositório na pasta da aula 11.

Terminal

Esse exemplo faz uso da comunicação UART para debug de código (via printf), para acessar o terminal no atmel estúdio clique em :

-> View -> Terminal Window

(você deve ter instalado o pacote extra do atmel listado no documento inicial de infra)

Configure o terminal para a porta que (COM) correta (verificar no windiows) e para operar com um BaudRate de 115200.

```
Compile e grave o código no uC, abra o terminal e analise o output.
```

Tasks

Esse exemplo cria inicialmente 2 tarefas : **task_monitor**, **task_led**. A primeira serve como monitor do sistema (como o monitor de tarefas do Windows/Linux), enviando via printf informações sobre o estado das tarefas do sistema embarcado. A segunda serve para gerenciar o LED e o faz piscar a uma taxa de uma vez por segundo.

task_led

A task_led possui a implementação a seguir (limpei a parte que não é referente ao SAME70):

```
/**
 * \brief This task, when activated, make LED blink at a fixed rate
 */
static void task_led(void *pvParameters)
{
    UNUSED(pvParameters);
    for (;;) {
        LED_Toggle(LED0);
        vTaskDelay(1000);
    }
}
```

Notem que a função possui um laço infinito (for (;;){}), tasks em um rtos não devem retornar, elas executam como se estivessem exclusividade da CPU (assim como um código bare-metal que não deve retornar da função main).

A função LED_Toggle é na verdade um macro que faz o LED piscar, usando uma série de funções do PIO-ASF que não usamos no curso (podemos aqui usar a nossa função de pisca led).

A função vTaskDelay() faz com que a tarefa fique em estado de **blocked** (permite que outras tarefas utilizem a CPU) por um determinado número de *ticks*, essa função é diferente da delay_ms() que bloqueia a CPU para sua execução. Deve-se evitar o uso de funções de delay baseadas em "queimar" clocks na tarefas de um RTOS, já que elas agem como um trecho de código a ser executada.

A função dorme por um determinado número de ticks, podemos traduzir ticks para ms, usando o define portTICK_PERIOD_MS como no exemplo a seguir :

```
/**
 * \brief This task, when activated, make LED blink at a fixed rate
 */
static void task_led(void *pvParameters)
{
    /* Block for 500ms. */
    const TickType_t xDelay = 500 / portTICK_PERIOD_MS;

    for (;;) {
        LED_Toggle(LED0);
        vTaskDelay(xDelay);
    }
}
```

Modifique o firmware com o código anterior proposto e reprograme o uC.

O Tick de um RTOS define quantas fezes por segundo o escalonador irá executar o algoritmo de mudança de tarefas, no ARM o tick é implementado utilizando um timer do próprio CORE da ARM chamado de *system clock*, criado para essa função. Por exemplo, um RTOS que opera com um tick de 10ms irá decidir pelo chaveamento das tarefas 100 vezes por segundo, já um tick configurado para 1ms irá executar o escalonador a uma taxa de 1000 vezes por segundo. Trechos de código que necessitam executar a uma taxa maior que 1000 vezes por segundo (tick = 1ms) não devem ser implementados em tasks do RTOS mas sim via interrupção de timer.

A configuração da frequência do tick assim como o mesmo é implementando está no arquivo do projeto : config/FreeRTOSConfig.h :

....

```
#define configTICK_RATE_HZ                      ( 1000 )
....
#define xPortSysTickHandler SysTick_Handler
```

> **!** Impacto do tick na função vTaskDelay é que a mesma só pode ser chamada com múltiplos inteiros referente ao tick. Não temos tanta resolução comparado com o TimerCounter. Quanto maior a frequência mais vezes o OS necessita salvar e recuperar o contexto, diminuindo assim sua eficiência.

Modifique o configTICK_RATE_HZ para 500 (src/config/FreeRTOSConfig.h)

Task Monitor

Essa task é responsável por enviar pela serial (terminal) informações sobre o estado interno do sistema operacional e suas tarefas, ela possui um formato semelhante ao da task_led porém na sua execução (que acontece 1 vez por segundo já que nosso TICK_RATE_HZ é 1000) coleta o número de tarefas e suas listas e faz o envio via printf [2].

[2] : https://www.freertos.org/a00021.html

Note que a função também está em um laço infinito, nunca terminando de executar.

```
/**
 * \brief This task, when activated, send every ten seconds on debug UART
 * the whole report of free heap and total tasks status
 */
static void task_monitor(void *pvParameters)
{
    static portCHAR szList[256];
    UNUSED(pvParameters);

    for (;;) {
        printf("--- task ## %u", (unsigned int)uxTaskGetNumberOfTasks());
        vTaskList((signed portCHAR *)szList);
        printf(szList);
        vTaskDelay(1000);
    }
}
```

Essa tarefa produz uma saída como a seguir :

```
--- task ## 4 Monitor   R   0   77  1
IDLE          R         0   110 3
Led           B         0   231 2
Tmr Svc       B         4   225 4
```

Com a seguinte estrutura :

```
taskName Status Priority WaterMark Task ID
```

- taskName : nome dado a task na sua criação
- Status : status da task [3]:
  - Suspended
  - Ready
  - Running
  - Blocked

[3] : https://www.freertos.org/RTOS-task-states.html

> Modifique a task para executar a cada 2 segundos. Programe o uC com essa modificação.

Criando as tarefas

Criar uma tarefa é similar ao de inicializar um programa em um sistema operacional mas no caso devemos indicar para o RTOS quais "funções" irão se comportar como pequenos programas (tarefas), para isso devemos chamar a função **xTaskCreate** que possui a seguinte documentação [4]:

[4] : https://docs.aws.amazon.com/freertos/latest/lib-ref/group___x_task_create.html

---

```
/**
 * task. h
 *
 BaseType_t xTaskCreate(
                              TaskFunction_t pvTaskCode,
                              const char * const pcName,
                              uint16_t usStackDepth,
                              void *pvParameters,
                              UBaseType_t uxPriority,
                              TaskHandle_t *pvCreatedTask
                          );
 *
 * Create a new task and add it to the list of tasks that are ready to run.
 *
```

```
 * xTaskCreate() can only be used to create a task that has unrestricted
 * access to the entire microcontroller memory map.  Systems that include MPU
 * support can alternatively create an MPU constrained task using
 * xTaskCreateRestricted().
 *
*/
```

A criação das tasks monitor e LED são feitas da seguinte maneira :

```
    xTaskCreate(task_monitor, "Monitor", TASK_MONITOR_STACK_SIZE, NULL,
                TASK_MONITOR_STACK_PRIORITY, NULL);;

    xTaskCreate(task_led, "Led", TASK_LED_STACK_SIZE, NULL,
                TASK_LED_STACK_PRIORITY, NULL);
    }
```

O primeiro parâmetro da xTaskCreate é a função que será lidada como uma task, a segunda é o nome dessa tarefa, a terceira é o tamanho da stack que cada task vai possuir, o quarto seria um ponteiro para uam estrutura de dados que poderia ser passada para a task em sua criação, o quinto a sua prioridade e o último é um ponteiro e retorna uma variável que pode ser usada para gerencias a task (deletar, pausar).

A stack e a prioridade estão definidos no próprio main.c :

```
#define TASK_MONITOR_STACK_SIZE             (2048/sizeof(portSTACK_TYPE))
#define TASK_MONITOR_STACK_PRIORITY         (tskIDLE_PRIORITY)
#define TASK_LED_STACK_SIZE                 (1024/sizeof(portSTACK_TYPE))
#define TASK_LED_STACK_PRIORITY             (tskIDLE_PRIORITY)
```

> A cada tarefa pode ser atribuída uma prioridade que vai de *0 até (configMAX_PRIORITIES - 1)*, onde *configMAX_PRIORITIES* está definido no arquivo de configuração FreeRTOSConfig.h

Uma das dúvidas mais comum no uso de RTOS é o quanto de espaço devemos alocar para cada tarefa, e essa é uma pergunta que não existe um resposta correta, caso esse valor seja muito grande podemos estar alocando um espaço extra que nunca será utilizado e caso pequena, podemos ter um stack overflow e o firmware parar de funcionar.
A melhor solução é a de executar o programa e analisar o consumo da stack pelas tasks ao longo de sua execução, tendo assim maiores parâmetros para a sua configuração.

Modifique a prioridade da Task Led para prioridade máxima do FreeRTOS.

Power Save mode ?

Uma forma muito simples de conseguirmos diminuir o consumo energético de um sistema embarcado com RTOS é o de ativar os modos de baixo consumo energético (powersave/ sleep mode) quando o SO estiver na tarefa idle. A tarefa idle é aquela executada quando nenhuma outra tarefa está em execução. Sempre que essa tarefa idle for chamada a o RTOS irá executar a função a seguir já definida no main.c:

```
/**
 * \brief This function is called by FreeRTOS idle task
 */
extern void vApplicationIdleHook(void)
{
}
```

Porém ainda devemos ativar essa funcionalidade no arquivo de configuração, via o define : configUSE_IDLE_HOOK.

No arquivo de configuração FreeRTOSConfig.h
modifique :
`#define configUSE_IDLE_HOOK 0`
para
`#define configUSE_IDLE_HOOK 1`

Com isso podemos controlar o modo sleep na função vApplicationIdleHook.

Entre em modo sleep dentro da função vApplicationIdleHook (no main.c) chamando :
`pmc_sleep(SAM_PM_SMODE_SLEEP_WFI);`

! Devemos entrar em um modo de sleep que o timer utilizado pelo tick consiga ainda acordar a CPU executar, caso contrário o RTOS não irá operar corretamente já que o escalonador não será chamado.

API - Comunicação entre task / irq

(30 minutos)

Objetivo : Comunicar a tarefa do LED para ser executada via a interrupção (callback) do botão da placa .

Uma das principais vantagens de usar um sistema operacional é o de usar ferramentas de comunicação entre tarefas ou entre ISR e tarefas, em um código baremetal fazemos esse comunicação via variáveis globais (buffers, flags, ...), essa implementação carece de funcionalidades que o RTOS irá suprir, tais como :

- Semáforo (semaphore) É como uma flag binária, permitindo ou não a execução de uma task, funciona para sincronização de tarefas ou para exclusão mútua (multual exclusion), sem nenhum tipo de prioridade.
- Mutex : Similar aos semáforos porém com prioridade de execução (mutex alteram a prioridade da tarefa)
- MailBox ou Queues : Usado para enviar dados entre tarefas ou entre ISR e Tasks

[5] : https://www.freertos.org/Embedded-RTOS-Binary-Semaphores.html

Semaphore

Iremos implementar um semáforo para comunicação entre o call back do botão e a tarefa que faz o controle do LED, o callback do botão irá liberar o semáforo para a tarefa do LED executar em um formato : produtor-consumidor.

Para implementarmos um semáforo precisamos primeiramente definir uma variável global que será utilizada pelo sistema operacional para definir o endereço desse semáforo :

```
SemaphoreHandle_t xSemaphore;
```

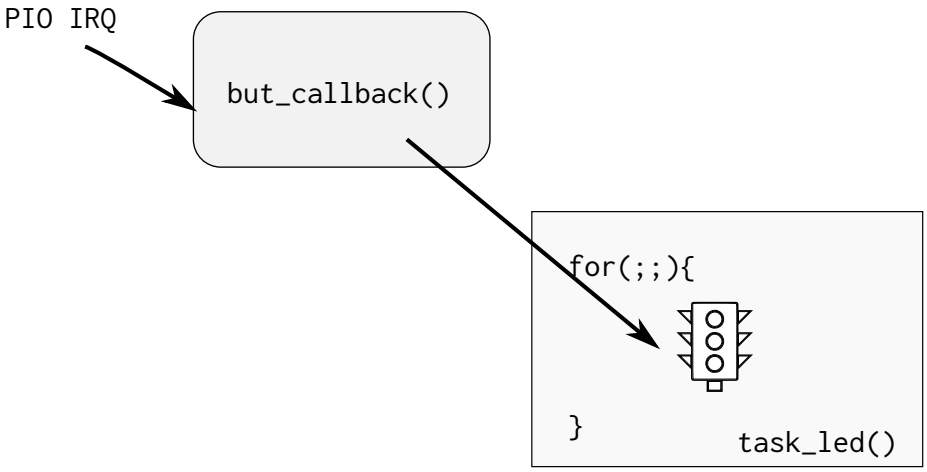

Figure 1: Semáforo

Devemos antes de usar o semáforo, fazermos sua criação/inicialização :

```
/* Attempt to create a semaphore. */
xSemaphore = xSemaphoreCreateBinary();
```

Uma vez criado o semáforo podemos esperar a liberação do semáforo via a função :

```
xSemaphoreTake(xSemaphore, Tick) ;
```

- xSemaphore : O semáforo a ser utilizado
- Tick : timeout (em ticks) que a função deve liberar caso o semáforo não chegue. Se passado o valor 0, a função irá bloquear até a chegada do semáforo.

Para liberarmos o semáforo devemos usar a função :

```
xSemaphoreGiveFromISR(...);
```

Note o ISR no final da função, isso quer dizer que estamos liberando um semáforo de dentro de uma interrupção. Caso a liberação do

semáforo não seja de dentro de uma interrupção, basta utilizar a função *xSemaphoreGive*

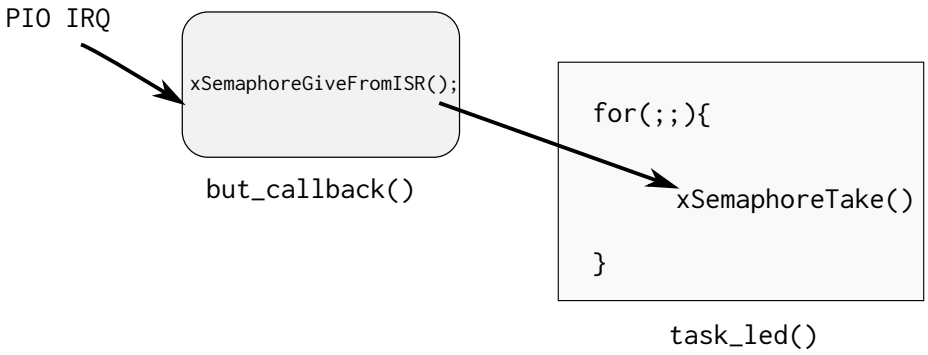

Figure 2: Semáforo

Adicionando o botão

Altere o conteúdo do arquivo main.c pelo fornecido no arquivo main_semaphore.c, que faz a criação e uso do semáforo na interrupção do botão e faz com que a task led seja executada somente quando liberada.

Interrupções devem possuir prioridade inferior ao do timer reservado para o tick (sys_tick), porém por algum motivo o RTOS está fazendo uma confusão quanto essas prioridades, iremos desabilitar essa verificação. Edite o arquivo de configuração FreeRTOSConfig.h e comente as linhas a seguir :

```
/* Normal assert() semantics without relying on the provision of an assert.h
header file. */
//#define configASSERT( x ) if( ( x ) == 0 ) { taskDISABLE_INTERRUPTS(); for( ;; ); }
```

Entrega Final

Usando a placa externa OLED faça com que cada LED pisque em uma frequência pré determinada e seu controle (parar de piscar) seja implementando em cada botão da placa. A entrega final deve possuir : - Uma task para cada LED - Um call back para cada botão - Um semáforo para cada botão