Sistemas Tempo-Real

O sistema operativo de tempo-real FreeRTOS

UA/DETI, 2012
Paulo Pedreiras
pbrp@ua.pt

V1.0, Set/2012

Agenda

1. Introdução ao FreeRTOS
2. Gestão de tarefas
3. Filas
4. Interrupções e Sincronização
5. Gestão de recursos
6. Gestão de memória

1 – Introdução

Introdução ao FreeRTOS

- FreeRTOS
  - Open Source (http://www.freertos.org/)
  - Simples, portável e conciso
  - Diversas famílias de processadores suportados (e.g. PIC18, PIC32, ARM7, AVR, 8051, x86, ...)
  - Ferramentas de desenvolvimento para Windows e Linux (dependendo do tipo de processador)
  - Política de escalonamento configurável
    - Preemptiva (tarefas da mesma prioridade executam em ”roud robin”)
  - Cooperativa
    - Tarefas cedem o CPU por invocação explicita de uma primitiva (taskYIELD()).

Introdução ao FreeRTOS

- IPC
  - Semáforos
  - Message Queues
- Kernel minimalista, mas muito eficiente e portável
  - Maioria do código é comum às várias famílias de processadores
  - Executa em micro-controladores/processadores com recursos muito limitados (memória, capacidade de processamento – processadores 8,16,32 bit/ Flash >= 32K/RAM>=16KB)
- Gestão de alocação dinâmica de memória flexível

Introdução ao FreeRTOS

- "Type casting" excessivo
  - Forma de lidar com os requisitos de diversos compiladores
    - e.g. tipo "char" sem qualificador é tratado como *signed* por alguns compiladores e como *unsigned* por outros
- Regras:
  - Nomes de **variáveis** são precedidas por
    - c: tipo char / s: tipo short / l: tipo long / v:void p: pointer e ...
    - x: portBASE_TYPE (tipo que depende da arquitetura – 8/16/32 bit)

Introdução ao FreeRTOS

- Regras (cont.):
  - Nomes de **funções**
    - tipo+ficheiro em que estão definidas
      - e.g. vTaskPrioritySet()
        - devolve um void e está definida em task.c
  - Macros
    - Nomes em maiúsculas e com sufixos em minúsculas identificando onde estão definidas
      - e.g. portMAX_DELAY
        - Definida em port.h
        - Chama-se MAX_DELAY
  - Algumas macros comuns
    - pdTRUE (1) ; pdFALSE (0) ; pdPASS (1) ; pdFAIL (0)

Introdução ao FreeRTOS

- Criar projetos
  - Recomendável **partir de uma demonstração** e adaptar de acordo com as necessidades
    - Abrir demo e verificar que compila sem erros
    - Remover os ficheiros fonte específicos da demo (todos os ficheiros na pasta em ".\Demo\Common" podem ser removidos
    - Remover todas as funções do main.c, exceto prvSetupHardware()
    - Colocar a "0" as seguintes constantes em FreeRTOSConfig.h
      - config{USE_IDEL_HOOK;USE_TICK_HOOK;USE_MALLOC_FAILED_HOOK;CHECK_FOR_STACK_OVERFLOW}

Introdução ao FreeRTOS

- Criar projetos (cont)
  - Criar uma nova função main() de acordo com a template abaixo
  - Compilar
  - Adicionar tarefas

```
short main( void )
{
	/* Init hardware */
	prvSetupHardware();

	/* Create application tasks */
	...

	/* Set the scheduler running.  This function will not return unless a task calls
                                vTaskEndScheduler(). Or there is no hep start the scheduler*/
	vTaskStartScheduler();

	return 0;
}
```

2 – Gestão de Tarefas

Gestão de tarefas

- No FreeRTOS uma tarefa é implementada como uma função em linguagem “C”
- As tarefas apresentam a seguinte estrutura
  - Cabeçalho
    - Designação da função + parâmetros de entrada
    - Função devolve mandatoriamente “void”
  - Variáveis locais
    - Variáveis locais à função. “Stack” de cada tarefa deve acomodar espaço para as variáveis locais e salvaguarda de contexto
  - Inicialização
    - Código que é executado apenas uma vez

Gestão de tarefas

- As tarefas apresentam a seguinte estrutura (cont.)
  - Cabeçalho
  - Variáveis locais
  - Inicialização
  - Ciclo infinito
    - "Corpo" da tarefa (processamento/funcionalidade).
    - Cada execução denomina-se instância ou "job"
  - Eliminação da tarefa
    - As tarefas nunca devem sair da função associada à sua implementação – não há um "return"
    - Quando se pretende terminar uma tarefa, tal deve ser efetuado explicitamente

Gestão de tarefas

- Exemplo

```
void vTaskName(void *pvParameters) {
    int i;
    i=0;
    PORTAbits.RA0=0;
    while(1) {
        PORTAbits.RA0=~    PORTAbits.RA0;
        for(i=0;i<20000;i++);
        vTaskDelay(1000/portTICK_RATE_MS);
    }
    vTaskDelete(NULL);
}
```

Cabeçalho

Variáveis locais

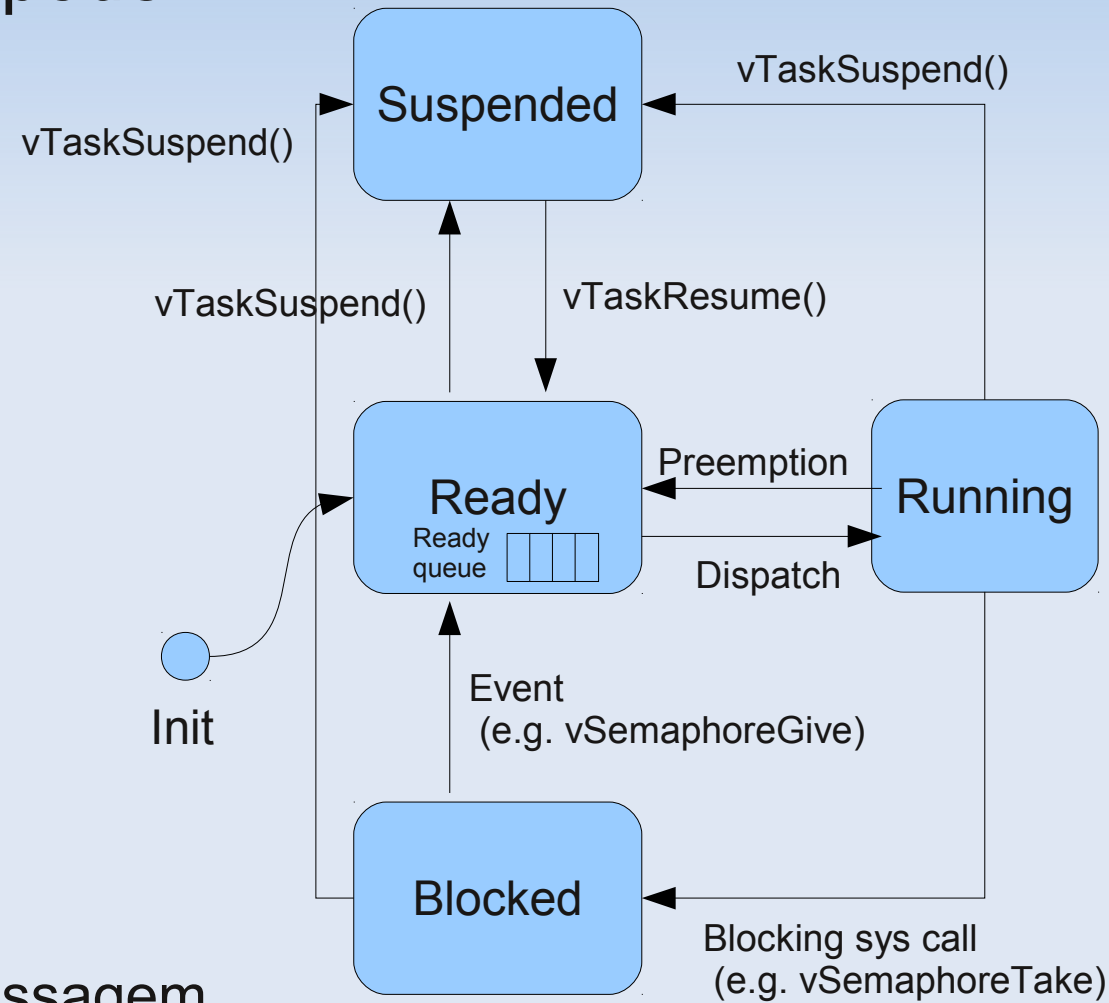
Inicializações

Corpo da tarefa
- Ciclo
- Processamento/funcionalidade
- Bloqueio até novo período

Eliminar tarefa

Gestão de tarefas

- No FreeRTOS uma tarefa pode estar num de 4 estados
  - **Running**
    - Código da tarefas está em execução
  - **Ready**
    - Pronta a executar;
    - Aguarda alocação do CPU
  - **Suspended**
    - Execução suspensa
  - **Blocked**
    - Aguarda um recurso ou passagem do tempo para poder executar



Gestão de tarefas

- Criação de tarefas
  - A criação de uma tarefa é feita por via da função xTaskCreate(), a qual possui como argumentos
    - pvTaskCode: ponteiro para a função
    - pcName: string com designação da função
      - Puramente descritivo.
    - usStackDepth: dimensão da stack.
      - Cada tarefa tem a sua stack individual
      - O valor indica o numero de “words” que deve ter
      - Valor difícil de estimar devido à chamada de funções, etc.
      - FreeRTOS define um valor configMINIMAL_STACK_SIZE (valor mínimo recomendado para qualquer aplicação)

Gestão de tarefas

- função xTaskCreate() - continuação
  - *pvParameters: argumentos para a tarefa
  - uxPriority: prioridade da tarefa
    - Valor entre 0 (menor prioridade) e configMAX_PRIORITIES-1 (maior prio.)
    - Maior número de prioridades implica mais consumo de RAM
  - *pxCreatedTask: “handle” para a tarefa
    - Permite referenciar a tarefa em outras tarefas para e.g. alterar a prioridade, eliminar, ...
- Retorno
  - pdTRUE: sucesso
  - err_COULD_NOT_ALLOCATE_REQUIRED_MEMORY
    - Tarefa não foi criada por falta de memória (TCB e stack)

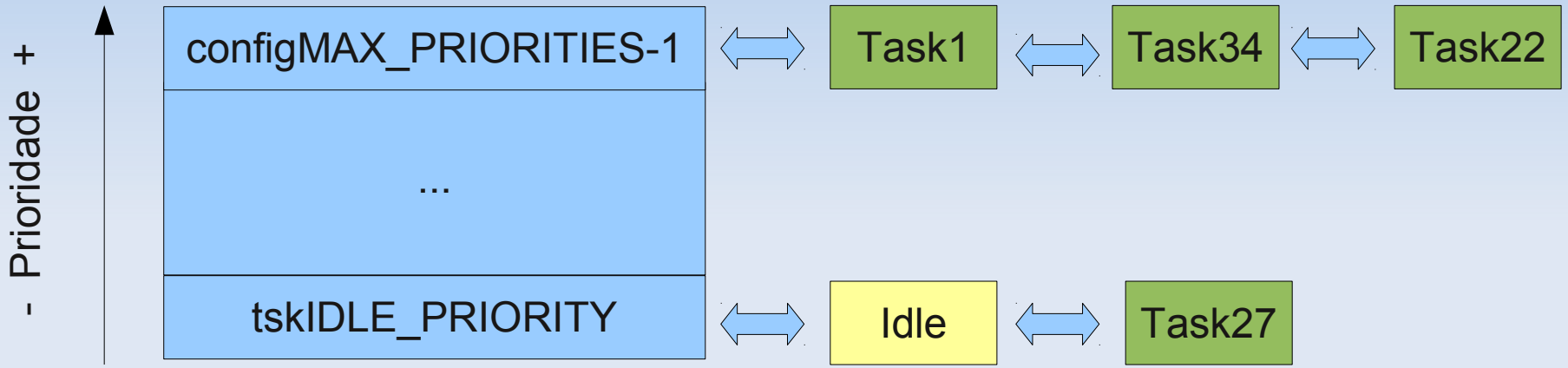
Gestão de tarefas

- Criação de tarefas
  - A partir da função main(); posteriormente inicia-se o escalonador
  - Exemplo:

```
void main( void ) {
    int i,j; ... /Local vars

    /* Inicializações */
    TRISA = 0x00;    /* Ports A all output */
    ...
    /* Create tasks */
    xTaskCreate(vTask1, (const portCHAR * const) "vTask1",
                configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 1, NULL);
    xTaskCreate(vTask2, ...);

    /* Start scheduler */
    vTaskStartScheduler();

    /* Main can't terminate */
    while(1);
}
```

Gestão de tarefas

- Prioridades no FreeRTOS



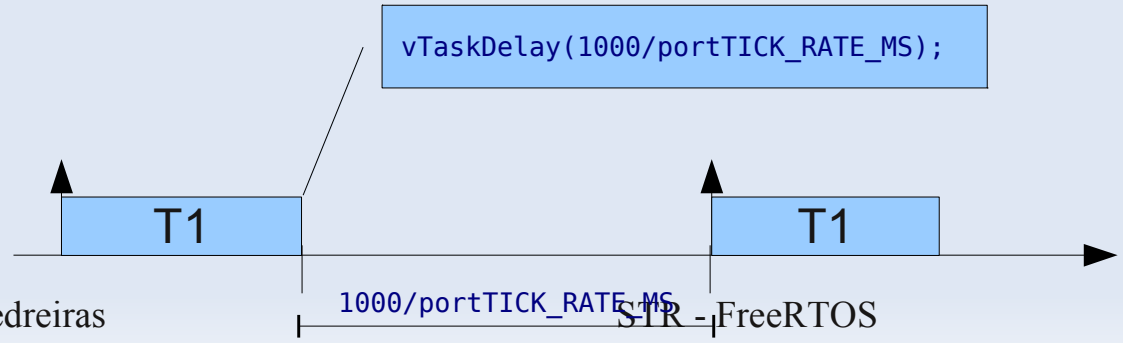
- Quando há várias tarefas “ready”:
  - É executada a de maior prioridade
  - Dentro de cada nível de prioridade há Round Robin
  - Duração do slice configurável (configTICK_RATE_HZ)
- As tarefas do utilizador podem partilhar o nível de prioridade “idle”

Gestão de tarefas

- Tarefas periódicas
  - Em aplicações de controlo muitas tarefas são periódicas (e.g. malhas de controlo)

    A função vTaskDelay permitir colocar uma tarefa no estado bloqueado durante um certo período de tempo
  - E.g.

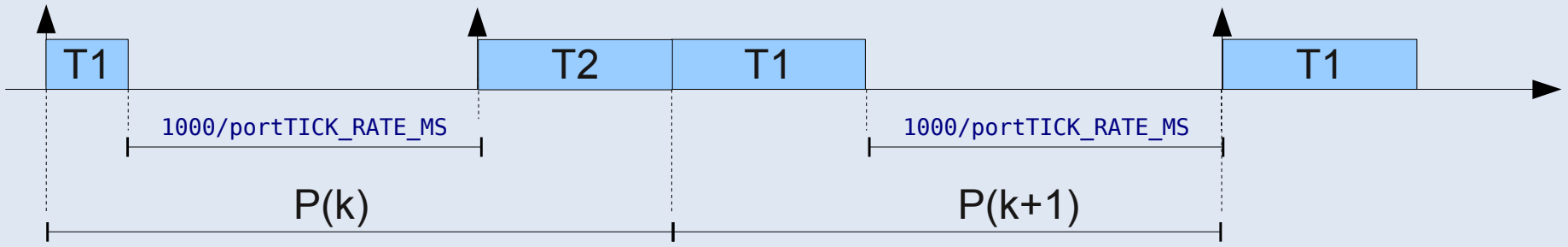
```
while(1) {
        Funcionalidade;
        vTaskDelay(1000/portTICK_RATE_MS);
    }
```



Gestão de tarefas

- Tarefas periódicas (continuação)
  - Problemas
    - Período pouco rigoroso. Dependente de variações do tempo de execução da tarefa, interferência de outras tarefas, interrupções, ...

```
while(1) {
        Funcionalidade;
        vTaskDelay(1000/portTICK_RATE_MS);
    }
```



P(k) <> P(k+1) !!!!!

Gestão de tarefas

- Tarefas periódicas (continuação)
  - Solução: vTaskDelayUntil()
    - Indica o tempo absoluto em que a ativação deve ocorrer
    - Referenciada à ativação anterior e não ao instante de invocação

```
void vTask1(void *pvParameters) {
    ...
    portTickType xLastWakeTime;
    ...
    xLastWakeTime=xTaskGetTickCount();
    while(1) {
        Funcionalidade...
        vTaskDelayUntil(&xLastWakeTime,(1000/portTICK_RATE_MS));
    }
}
```

Gestão de tarefas

- Parâmetros de entrada
  - Por vezes é útil passar parâmetros a tarefas
    - E.g. tarefas que leiam valores de uma ADC multiplexada têm um código idêntico, variando apenas o ID do canal

```
void vTask1(void *pvParameters) {
    ...
    int adc_chan;
    ...
    adc_chan=*((int *)pvParameters);
    ...
    while(1) {
        Funcionalidade...
        vTaskDelayUntil(&xLastWakeTime,(1000/portTICK_RATE_MS));
    }
}
```

Gestão de tarefas

- Alterar a prioridade de uma tarefa
  - A prioridade de uma tarefa pode ser alterada por meio da função

```
vTaskPrioritySet(xTaskHandle pxTask,
                 unsigned portBASE_TYPE, uxNewPriority)
```

    - pxTask: handle da tarefa que se pretende modificar
      - NULL refere a própria tarefa
    - UxNewPriority: valor da prioridade desejada
  - A prioridade de uma tarefa pode ser obtida com a system call vTaskPriorityGet()

Gestão de tarefas

- Eliminar uma tarefa
  - Uma tarefa pode ser eliminada por meio da função

    vTaskDelete(xTaskHandle pxTaskToDelete)

    - pxTaskToDelete: handle da tarefa que se pretende modificar
      - NULL refere a própria tarefa
  - Quando uma tarefa é eliminada o TCB e stack são libertados
  - Todavia podem **não ficar** disponíveis para reutilização
    - Depende do gestor de memória dinâmica

Gestão de tarefas

- Suspender uma tarefa
  - A execução de uma tarefa pode ser suspensa eliminada por meio da função

    vTaskSuspend(xTaskHandle pxTaskToSuspend)

    - pxTaskToSuspend: handle da tarefa alvo
      - NULL refere a própria tarefa
  - A saída do estado de suspensão dá-se por invocação de

    vTaskResume(xTaskHandle pxTaskToResume)

    - pxTaskToResume: handle da tarefa alvo

Gestão de tarefas

- Suspender uma tarefa (continuação)
  - Quando a suspensão é terminada a partir de uma ISR a system call é:

    portBASE_TYPE vTaskResumeFromISR(xTaskHandle pxTaskToResume)

    - pxTaskToResume: handle da tarefa alvo
  - Valor retornado:
    - pdTRUE: caso resulte da operação uma mudança de contexto
    - pdFALSE: caso contrário

Gestão de tarefas

- Outras funções
  - portTickType xTaskGetTickCount(void)
    - Devolve o número de ticks que decorreram desde que o scheduler foi iniciado (tempo absoluto)
    - Valor em “interrupt ticks”; dependente da plataforma
      - A constante `portTICK_RATE_MS` permite efetuar a conversão entre o tempo real e ticks
  - portBASE_TYPE uxTaskGetNumberOfTasks(void)
    - Devolve o número de tarefas no sistema
    - A eliminação de tarefas não decrementa imediatamente este valor (parcialmente executado por código associado à idle task)
  - TaskYELD(void)
    - Passa o controlo a outra tarefa sem esperar pelo final do time slot

Gestão de tarefas

- O “Idle Task Hook”
  - Permite executar código do utilizador na tarefa “idle”
  - Usos habituais:
    - Executar processamento contínuo de background
    - Colocar o processador em modo de poupança de energia
  - Restrições
    - Nunca deve bloquear ou suspender a sua execução
  - E.g.

```
...
unsigned long UidleCycleCount=0UL;
...
/* Idle task hook */
void vApplicationIdleHook( void)
{
      UIdleCycleCount++;
}
```

3 - Filas

Filas

- As aplicações em FreeRTOS são estruturadas como tarefas semi-independentes
- Para atingir o objetivo da aplicação as tarefas têm de cooperar/partilhar informação
  - E.g. sensor tem de enviar dados para controlador
- Em FreeRTOS as filas são o mecanismo privilegiado de comunicação entre tarefas
- Deste ponto em diante passa a usar-se a designação “Queue”

Filas

- Características das Queues
  - Armazenamento de dados
    - Uma queue pode armazenar um conjunto finito de items de um dado tipo (tamanho fixo)
    - Normalmente é usada uma política de acesso do tipo FIFO.
      - O FreeRTOS permite métodos alternativos.
    - Escrever numa queue implica cópia dos dados para a queue
    - Ler de uma queue implica cópia e remoção dos dados da queue
      - Leitura "**consome**" os dados

Filas

- Características das Queues (continuação)
  - Acesso
    - As queues são uma entidade de direito próprio
      - Não pertencem a nenhuma tarefa específica
    - Várias tarefas podem escrever na mesma queue
    - Várias tarefas podem ler da mesma queue
  - Sincronização - leitura
    - Leitura de uma queue vazia causa passagem ao estado “bloqued”
    - A escrita na queue desbloqueia automaticamente uma tarefa que esteja bloqueada
    - Caso haja várias tarefas bloqueadas, a de maior prioridade é desbloqueada

Filas

- Características das Queues (continuação)
  - Sincronização - leitura (continuação)
    - As tarefas podem especificar um tempo máximo de espera (timeout)
      - Se este tempo expirar a tarefa passa a ready, havendo uma falha na leitura da queue
  - Sincronização – escrita
    - A escrita numa fila cheia casa a passagem ao estado “blocked”
    - Quando é libertado espaço na queue e há várias tarefas bloqueadas a de maior prioridade é desbloqueada
    - Tal como na leitura, é possível especificar um “timeout”

Filas

- Criação de uma fila
  - Antes de ser usada, uma queue tem de ser explicitamente criada

    xQueueHandle xQueueCreate(unsigned portBASE_TYPE uxQueueLength, unsigned portBASE_TYPE uxItemSize);

    - uxQueueLength: máximo numero de items que a queue comporta
    - uxItemSize: tamanho, em bytes, de cada item
    - Retorno:
      - NULL - erro. Queue não criada
      - !NULL - Handle para acesso à fila

Filas

- Escrita numa queue

```
portBASE_TYPE xQueueSend( xQueueHandle xQueue,
        const void *pvItemToQueue, portTickType xTicksToWait);
```

  - xQueue: handle da queue
  - pvItem: ponteiro para dados a escrever na fila
  - XticksToWait: tempo máximo que tarefa poderá ficar bloqueada na escrita na fila
    - "0" (zero): retorno imediato
    - portMAX_DELAY: timeout infinito
    - Valor especificado em ticks. Usar **`portTICK_RATE_MS`** para efetuar conversão para tempo real

Filas

- Escrita numa queue (continuação)
  - Retorno
    - pdPASS – escrita com sucesso
    - errQUEUE_FULL – dados não foram escritos por queue estar cheia (timeout expirado)
  - Variantes
    - xQueueSendToBack()
      - Equivalente a xQueueSend
    - xQueueSendToFront()
      - Dados são escritos no início da fila

Filas

- Leitura duma queue

```
portBASE_TYPE xQueueReceive( xQueueHandle xQueue,
        const void *pvBuffer, portTickType xTicksToWait);
```

  - xQueue: handle da queue
  - pvBuffer: ponteiro para zona de memória para onde os dados serão copiados
  - XticksToWait: timeout para bloqueio
- Retorno
  - pdPASS – Leitura com sucesso
  - errQUEUE_EMPTY – dados não lidos devido a queue estar vazia (timeout expirado)

Filas

- Estado duma queue

```
unsigned portBASE_TYPE uxQueueMessagesWaiting(
                xQueueHandle xQueue);
```

  - xQueue: handle da queue
  - Retorno
    - Número de items correntemente na fila.

Exemplo

```
...
/* Application global vars */
xQueueHandle xQueue;
...

Void main(void){
...
xQueue = xQueueCreate(QUEUE_LENGTH, sizeof(char));
...
```

Tarefa periódica

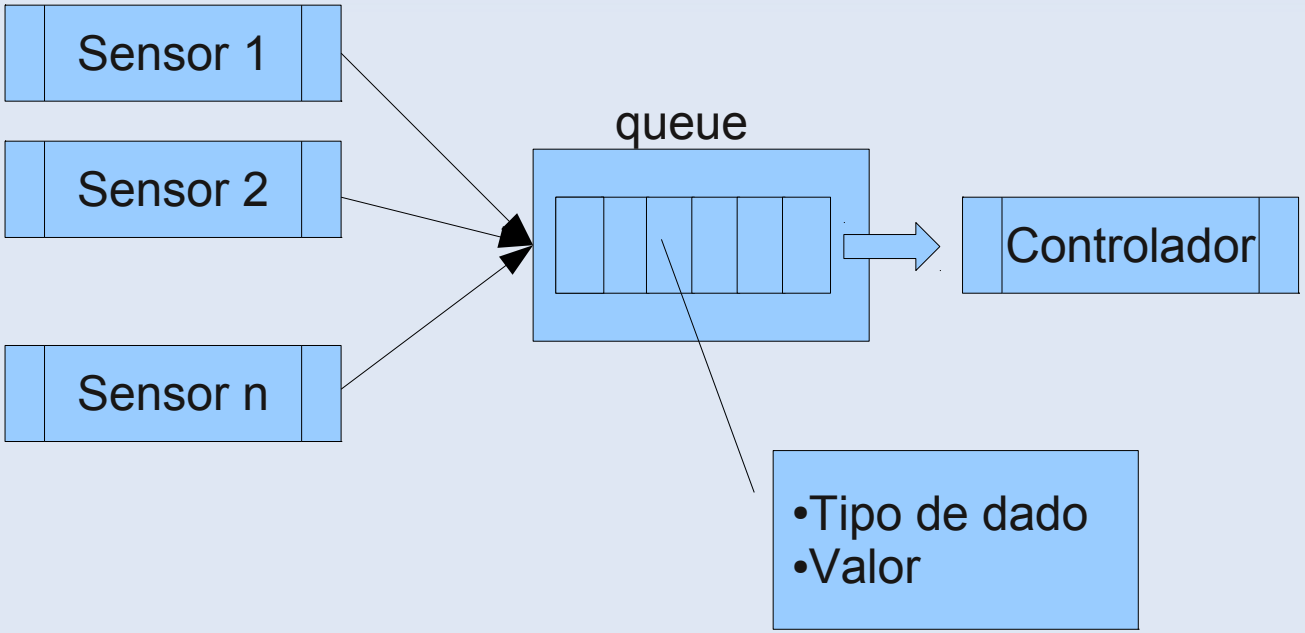
```
void vTask1(void *pvParameters) {
    ...
    while(1) {
        ...
        xStatus = xQueueSend(xQueue,(void *)(&ledStat),
                  portMAX_DELAY); // Send ledStat to queue
        ...
        vTaskDelay(TASK1_PERIOD_MS/portTICK_RATE_MS);
    }
}
```

Tarefa esporádica

```
void vTask2(void *pvParameters) {
    ...
    while(1) {
        xStatus=xQueueReceive(xQueue, (void *)(&ledStat),
                  portMAX_DELAY); // Read Queue (blocking)
        if(xStatus == pdPASS)  //Read successfuly
            PORTAbits.RA1=ledStat;
    }
}
```

Filas

- Passagem de tipos de dados complexos
  - E.g. quando uma fila contém diferentes tipos de dados



Filas

- Passagem de tipos de dados complexos (cont.)
  - A passagem por ponteiros permite estruturas de dados arbitrárias

```
...
/* Application global vars */
typedef struct {
    unsigned char ucType;
    unsigned char ucValue;
} xData;
...

void vTask1(void *pvParameters) {
    xData xReceiveData;
    ...
    xStatus=xQueueReceive(xQueue, &xReceiveData,0)
    ...
    if(xReceiveData.ucType == TEMP_SENS) {
        ...
}
...
```

4 - Interrupções e Sincronização

Interrupções

- Os sistemas embutidos têm frequentemente que responder a eventos de diversas origens que ocorrem no ambiente onde estão integrados
  - Chegada de um caracter, alteração do estado de uma porta, pacote que chegou a uma interface de comunicação, etc.
  - Estes eventos estão associados a diferentes overheads de processamento, importância e requisitos de tempo de resposta
  - Questões:
    - Que métodos de detecção de eventos
    - Qual a quantidade de processamento que deve ser efetuada na ISR?
    - Como devem ser comunicados às tarefas?

Interrupções e Sincronização

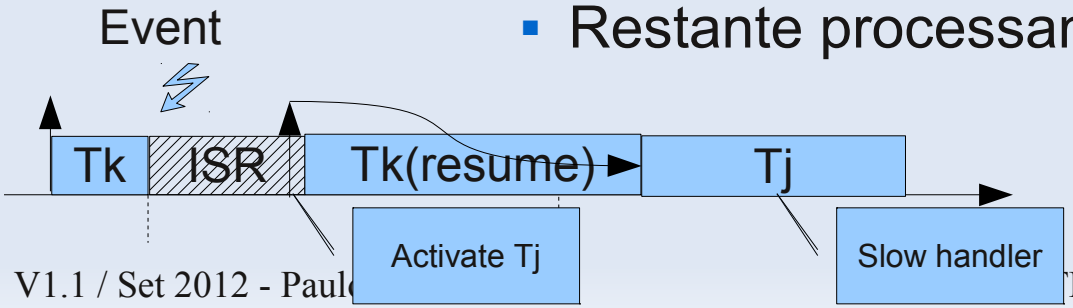
- Deteção de eventos
  - Em caso muito simples é possível efetuar poll - uma tarefa verifica periodicamente se um dado evento ocorreu (e.g. lê o estado da USART para saber se chegou algum caracter)

```
void vTask1(void *pvParameters) {
     ...
     while(1) {
          ...
          if(USART_STAT.RxC == 1) {
               ...
          }
          ...
          vTaskDelay(TASK1_PERIOD_MS/portTICK_RATE_MS);
     }
}
```

    - Muito ineficiente, pois normalmente os eventos ocorrem muito esparsamente
    - Difícil determinar o período ótimo
      - Curto: grande overhead
      - Longo: latência elevada

Interrupções e Sincronização

- Processamento diferido de interrupções
  - Para colmatar os problemas anteriores pode efetuar-se a deteção de eventos por interrupção (se o hardware o permitir)
  - O processamento de uma interrupção inibe o processamento de outras interrupções (pelo menos parcialmente)
    - A quantidade de trabalho dentro duma ISR deve ser tão reduzido quanto possível
  - Mecanismo vulgarmente usado: processamento diferido de interrupções
    - ISR faz processamento mínimo (*fast handler*) e
    - Restante processamento efetuado a nível de tarefa (*slow handler*)



Interrupções e Sincronização

- Semáforos binários para sincronização
  - Uma das formas de ativar o “slow handler” consiste no uso de semáforos binários

    void vSemaphoreCreateBinary(xSemaphoreHandle xSemaphore)

    - xSempahore – handle para o semáforo
    - Nota: esta system call é implementada por meio de uma macro. xSemaphore deve passar-se diretamente e não por referência

Interrupções e Sincronização

- Semáforos binários para sincronização (cont.)
  - Tomar o semáforo (operação “P()” na notação clássica)

    portBASE_TYPE xSemaphoreTake(xSemaphoreHandle xSemaphore, portTickTime xTicksToWait)

    - xSemaphore – *handle* para o semáforo
    - XticksToWait – Tempo máximo de bloqueio
      - Uso equivalente às queues
    - Retorno
      - pdPASS – retornou antes do *timeout*
      - pdFALSE – *timeout* expirou

Interrupções e Sincronização

- Semáforos binários para sincronização (cont.)
  - Dar o semáforo (operação “V()” na notação clássica)

    portBASE_TYPE xSemaphoreGiveFromISR(xSemaphoreHandle xSemaphore, portBASE_TYPE *pxHigherPriorityTaskWoken)

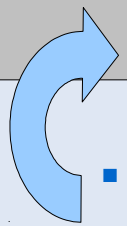
    - xSemaphore – handle para o semáforo
    - pxHigherPriorityTaskWoken – Indica se em resultado da operação uma tarefa com prioridade superior à da tarefa correntemente em execução passou a ready. Neste caso esta system call coloca *pxHigherPriorityTaskWoken com pdTRUE.
    - Retorno
      - pdPASS – retornou antes do *timeout*
      - pdFALSE – *timeout* expirou

Interrupções e Sincronização

- Semáforos binários para sincronização (cont.)
  - Exemplo

```
static void vHandlerTask( void *pvParameters )
{
    while(1) {
        xSemaphoreTake(xBinarySemaphore, portMAX_DELAY);
        /* Ocorreu evento! Fazer o seu processamento */
        Event_Proc ...
    }
}
```

```
static void __interrupt vExampleInterruptHandler( void )
{
     static portBASE_TYPE xHigherPriorityTaskWoken;

     xHigherPriorityTaskWoken = pdFALSE;

     /* 'Give' the semaphore to unblock the task. */
     xSemaphoreGiveFromISR( xBinarySemaphore, &xHigherPriorityTaskWoken );

     if( xHigherPriorityTaskWoken == pdTRUE )
     {
          ...
          portSWITCH_CONTEXT();
```



- Invocação do escalonador dentro da ISR faz com que retorno da ISR seja feito diretamente para o *slow handler*
- No limite o tempo de resposta pode ser equivalente ao processamento integral dentro da ISR

Interrupções e Sincronização

- Semáforos de contagem
  - O uso de semáforos binários pode levar à perda de eventos
    - Se vários eventos se sucederem antes de o slow handler terminar serão perdidos
      - Semáforo binário tem apenas dois estados!
  - FreeRTOS disponibiliza semáforos de contagem
    - Semáforos “convencionais”
    - Para além da contagem de eventos, são também usados para gestão de recursos
      - Inicializados com um certo valor (número de recursos). Tarefas bloqueiam se o contador tiver valor nulo.

Interrupções e Sincronização

- Semáforos de contagem
  - Antes de serem usados têm de ser criados

    xSemaphoreHandle xSemaphoreCreateCounting(portBASE_TYPE uxMaxCount, portBASE_TYPE uxInitialCount)

    - uxMaxCount – máximo valor que o semáforo pode contar (máximo número de eventos que podem ser mantidos em espera / máximo número de recursos)
    - usInitialCount – valor inicial (deve sre zero no caso de eventos / uxMAxCount no caso de recursos)
  - Retorno
    - NULL – erro (falta de memória heap)
    - !NULL - sucesso

Interrupções e Sincronização

- Uso de queues a partir de ISR
  - Por vezes pode ser desejável o uso de *queues* a partir de ISR
    - Combina sincronização com passagem de dados!
  - As seguintes *system calls* podem ser usadas
    - portBASE_TYPE xQueueSendFromISR()
    - portBASE_TYPE xQueueSendToFrontFromISR()
    - portBASE_TYPE xQueueSendToBackFromISR()
  - Utilização semelhante às suas pares , mas podem ser usadas a partir de ISR

Interrupções e Sincronização

- Uso eficiente de queues no contexto de execução diferida de ISR
  - O acesso a queues e eventual bloqueio/desbloqueio de tarefas são operações com custos computacionais elevados
  - Pode ser efetuado algum processamento dentro da ISR para otimizar o consumo de recursos
  - Exemplo – receção de comandos via USART
    - Opção 1: envia cada caracter recebido
    - Opção 2: pré-processar localmente à ISR os caracteres e colocar na queue apenas comandos inteiros
      - Muito mais eficiente ...
      - Mas atenção ao impacto nas outras interrupções e tarefas

5 – Gestão de recursos

Gestão de recursos

- As tarefas executam concorrentemente e podem partilhar recursos
  - Hardware - portas de I/O, buffers, LCD, USART, ...
  - Software – variáveis partilhadas, funções não reentrantes, ...
- Se uma tarefa sofre preempção durante o acesso a um recurso pode ocorrer uma situação de inconsistência/corrupção de dados
  - Necessário garantir exclusão mutua durante o acesso a recursos partilhados

```
void vTask1( void *pvParameters )
{
    ...
    while(1) {
        ...
        tmp=PORTA;
        tmp |=0x01;
        PORTA=tmp;
        ...
    }
}
```

```
void vTask2( void *pvParameters )
{
    ...
    while(1) {
        ...
        tmp=PORTA;
        tmp |=0x02;
        PORTA=tmp;
        ...
    }
}
```

Resultado ????

Gestão de recursos

- *Critical Sections*
  - Secção de código em que a preempção de tarefas é inibida
  - Define-se colocando o código entre as primitivas taskENTER_CRITICAL() e taskEXIT_CRITICAL()
    - Nota: inibe as interrupções que possuam prioridade acima de configMAX_SYSCALL_INTERRUPT_PRIORITY
  - O escalonador pode ser diretamente ativado e desativado
    - vTaskSuspendAll(); / xTaskResumeAll()
    - Interrupções não são afetadas

Gestão de recursos

- *Critical Sections* (cont)
  - Outras formas de implementar regiões criticas
    - Mutex (semáforo binário)
      - Apenas afeta as tarefas que partilham o recurso
      - Atenção a possíveis deadlocks!!!!
    - Gatekeeper
      - Acesso ao recurso é efetuado por uma única tarefa
      - Tarefas que pretendam aceder ao recurso comunicam com a Gatekeeper, e.g. via mensagens

6 – Gestão de memória

Gestão de memória

- Funções específicas para alocação dinâmica de memória
  - pvPortMalloc()
  - pvPortFree()
- Três implementações
  - Heap_1.c: versão básica de pvPortMalloc() e não implementa pvPortFree()
    - Comportamento determinístico
    - **Não permite libertação de memória!**
      - Adequado quando todos os recursos (tarefas, message queues, ...) são reservados inicialmente e se mantêm durante o funcionamento do sistema

Gestão de memória

- Três implementações (cont)
  - Heap_2.c: usa algoritmo “best fit” para alocação e implementa pvPortFree()
    - Não combina segmentos adjacentes
      - Eventuais problemas de fragmentação
    - Pode ser usado em tarefas que criam e removem tarefas frequentemente desde que a stack seja constante
  - Heap_3.c: usa código derivado das bibliotecas standard malloc() e free()
    - Menos determinístico
    - Não sofre de problemas de fragmentação

- Bibliografia
  - Manual do FreeRTOS
  - Links
    - http://www.slideshare.net/amraldo/free-freertos-coursetask-management
    - http://embedded-tips.blogspot.com/2010/07/freertos-course-semaphoremutex.html