Anotações Free RTOS

16/06/2012
Felipe Bandeira

Memoria:

Para o PIC24 é mais recomendado utiliza variaveis locais. Nos meus testes eu posso alocar 4096 bytes no local do que global.

O freertos utiliza 5120 no heap total, menos que isso o kernel trava.

No exemplo para pic24, utilizando o heap1 funciona normal, mais aloca muita memoria, utilizando heap3 o kernel trava, resetando o micro.

Configurando para "heap3":
Inclua o arquivo "heap_3.c" no projeto, coloque para 0 o "configTOTAL_HEAP_SIZE" Na configuração do mplab defina o heap size para 5200(funcionou para p25fj256gb106).

STACK:

Monitorando o stack:

#define configCHECK_FOR_STACK_OVERFLOW 2

Documentação do freertos:

Stack Overflow Detection - Method 1

It is likely that the stack will reach its greatest (deepest) value after the kernel has swapped the task out of the Running state because this is when the stack will contain the task context. At this point the kernel can check that the processor stack pointer remains within the valid stack space. The stack overflow hook function is called if the stack pointer contain a value that is outside of the valid stack range.

This method is quick but not guaranteed to catch all stack overflows. Set configCHECK_FOR_STACK_OVERFLOW to 1 to use this method only.

Stack Overflow Detection - Method 2

When a task is first created its stack is filled with a known value. When swapping a task out of the Running state the kernel can check the last 16 bytes within the valid stack range to ensure that these known values have not been overwritten by the task or interrupt activity. The stack overflow hook function is called should any of these 16 bytes not remain at their initial value.

This method is less efficient than method one, but still fairly fast. It is very likely to catch stack overflows but is still not guaranteed to catch all overflows.

To use this method in combination with method 1 set configCHECK_FOR_STACK_OVERFLOW to 2. It is not possible to use only this method.

Agora para monitora cria a função:

```
void vApplicationStackOverflowHook( void )

{

	/* Look at pxCurrentTCB to see which task overflowed its stack. */

	while (1) {

		/* faz algo */

	}

}
```

Observação: Caso o freeRtos identifique um possivel stack overflow, a processador ainda continuara funcionando.

MUTEX:

Exemplo retirado da microchip, AN1264, para a criação de um controle de recursos que são compartilhados entre os processos. Como exemplo a comunicacao SPI, se faz necessaria pois a API da microchip utiliza a mesma SPI para funcoes diferentes em pontos diferentes da execução. Como não é possivel prever aonde o codigo esta e qual o momento contexto foi alterado, é criado um **semaforo** para esse controle de acesso:

```
SPI2Semaphore = xSemaphoreCreateMutex();
```

Colocado em pontos estrategicos, a utilizacao do recurso, a função aguarda o recurso ser liberado, aguardando na fila.

```
xSemaphoreTake(SPI2Semaphore, portMAX_DELAY);
```

Quando o periferico é liberado, o que deve ser processado é processado. No fim é inserido a seguinte linha informando que o a função não utiliza mais esse recurso:

```
xSemaphoreGive(SPI2Semaphore);
```

Exemplo de utilizacao, disponivel do freertos:

```
 // A task that uses the semaphore.
 void vAnotherTask( void * pvParameters )
 {
    // ... Do other things.

    if( xSemaphore != NULL )
    {
        // See if we can obtain the semaphore.  If the semaphore is not available
        // wait 10 ticks to see if it becomes free.
        if( xSemaphoreTake( xSemaphore, ( portTickType ) 10 ) == pdTRUE )
        {
            // We were able to obtain the semaphore and can now access the
            // shared resource.

            // ...

            // We have finished accessing the shared resource.  Release the
            // semaphore.
            xSemaphoreGive( xSemaphore );
        }
        else
```

```
        {
            // We could not obtain the semaphore and can therefore not access
            // the shared resource safely.
        }
    }
 }
```

Exemplo da microchip:

```
BYTE SST25ReadByte(DWORD address)
{
    BYTE temp;

    xSemaphoreTake(SPI2Semaphore, portMAX_DELAY);

    SST25CSLow();

    SPI2Put(SST25_CMD_READ);
    SPI2Get();

    SPI2Put(((DWORD_VAL)address).v[2]);
    SPI2Get();

    SPI2Put(((DWORD_VAL)address).v[1]);
    SPI2Get();

    SPI2Put(((DWORD_VAL)address).v[0]);
    SPI2Get();

    SPI2Put(0);
    temp = SPI2Get();

    SST25CSHigh();
    xSemaphoreGive(SPI2Semaphore);

    return temp;
}
```

A documentação do freertos diz:

```
/**
 * semphr. h
 * <pre>xSemaphoreHandle xSemaphoreCreateMutex( void )</pre>
 *
 * <i>Macro</i> that implements a mutex semaphore by using the existing queue
 * mechanism.
 *
 * Mutexes created using this macro can be accessed using the xSemaphoreTake()
 * and xSemaphoreGive() macros.  The xSemaphoreTakeRecursive() and
 * xSemaphoreGiveRecursive() macros should not be used.
 *
 * This type of semaphore uses a priority inheritance mechanism so a task
 * 'taking' a semaphore MUST ALWAYS 'give' the semaphore back once the
 * semaphore it is no longer required.
 *
 * Mutex type semaphores cannot be used from within interrupt service routines.
 *
 * See vSemaphoreCreateBinary() for an alternative implementation that can be
 * used for pure synchronisation (where one task or interrupt always 'gives' the
 * semaphore and another always 'takes' the semaphore) and from within interrupt
 * service routines.
 *
 * @return xSemaphore Handle to the created mutex semaphore.  Should be of type
 *           xSemaphoreHandle.
```

MENSAGENS:

Tambem definidas como **Queue**(subst. = fila, ver. = enfileirar), podem ser usadas para enviar mensagens entre processos, com a vantagem de utiliza toda a logica de envio do FreeRtos.

Criando uma Queue:

```
usb_buffer_queue = xQueueCreate(4, sizeof(USB_BUFFER));
```

Utilizando:

```
xQueueSendToBack(usb_buffer_queue, &usb_buffer, portMAX_DELAY);
```

O **portMAX_DELAY** informa quanto tempo se deve esperar até que ocorra um erro de timeout. No caso o tempo vai ser "infinito".

Consideração de uso:
Se for o seguinte caso ocorre:

```
xQueueSendToBack(usb_buffer_queue, &usb_buffer, portMAX_DELAY);
                              ...faz algo...
xQueueSendToBack(usb_buffer_queue, &usb_buffer, portMAX_DELAY);
                              ..faz algo...
```

O primeiro Queue é processado e so no seu termino o segundo Queue é iniciado.

Documentação diz:

```
xQueueHandle xQueueCreate( unsigned portBASE_TYPE uxQueueLength,
                           unsigned portBASE_TYPE uxItemSize
                         );
* </pre>
*
* Creates a new queue instance.  This allocates the storage required by the
* new queue and returns a handle for the queue.
*
* @param uxQueueLength The maximum number of items that the queue can contain.
*
* @param uxItemSize The number of bytes each item in the queue will require.
* Items are queued by copy, not by reference, so this is the number of bytes
* that will be copied for each posted item.  Each item on the queue must be
* the same size.
*
* @return If the queue is successfully create then a handle to the newly
* created queue is returned.  If the queue cannot be created then 0 is
* returned.
```