Introduction to FreeRTOS
Part 2

Interrupt Management

Events

Embedded real-time systems have to take actions in response to events that originate from the environment.

How should they be detected? Interrupts, polling
What kind of processing needs to be done? Inside ISR, outside ISR

Interrupt priority vs task priority

Lowest priority interrupt pre-empt highest priority task

Interrupt Safe APIFunction

FreeRTOS provides two versions of some API functions:
one for use from tasks,
and one for use from ISRs (“FromISR” appended to their name).

Interrupt Management

Context Switching from ISR

Context switching is performed when the task that comes into running state when interrupts finish, is different from the task that went to bloqued state when interrupt occured.

Use **pxHigherPriorityTaskWoken** parameter

a variable to inform the application writer that a context switch should be performed

PdTRUE, PdFalse

Macros to **request a context switch from ISR**

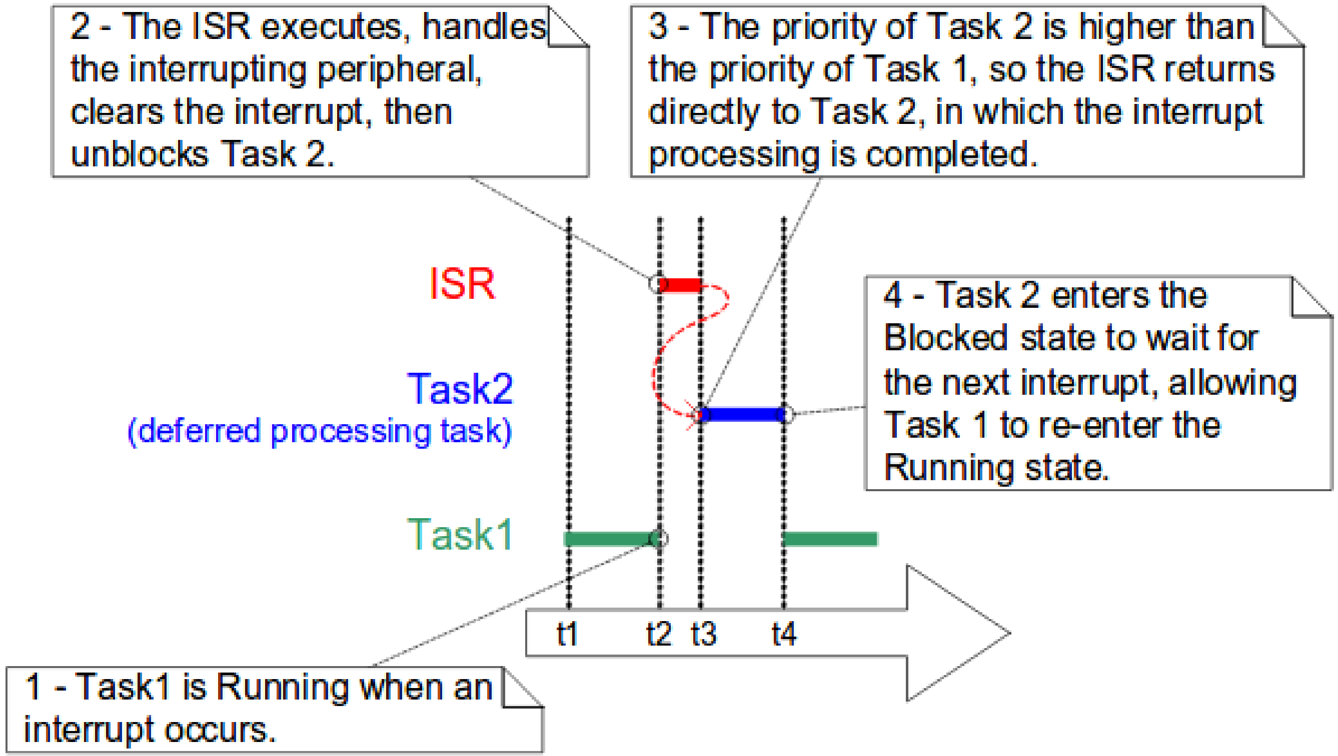
```
portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
```

Macro to **request context switching from a task**

```
taskYIELD()
```

Interrupt Management



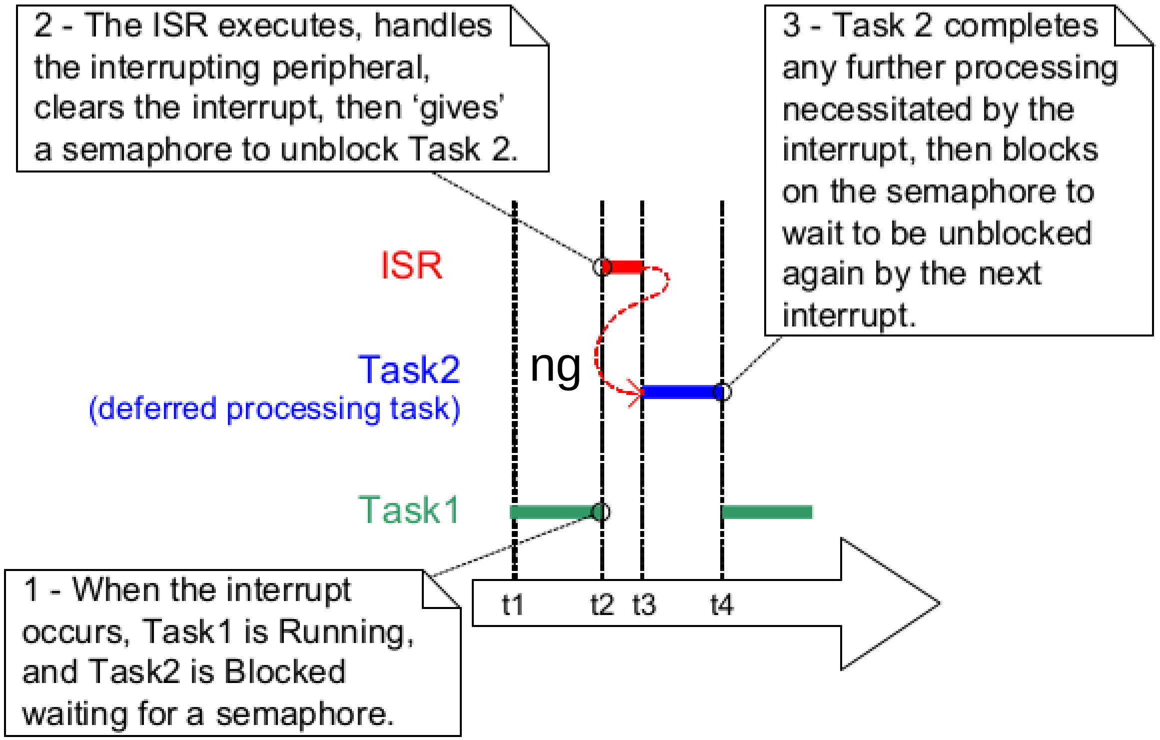
Interrupts should be deferred to a task

Interrupt Management

Binary Semaphores Used for Synchronization

The deferred processing task can be controlled using a ISR

- The deferred task is blocked with a “take” call for a semaphore.
- The ISR “gives” a semaphore to unblock the deferred task



Interrupt Management

APIFunctions for managing binary semaphores

Creating a semaphore

```
SemaphoreHandle_t  xSemaphoreCreateBinary(void);
```

Take

```
BaseType_t xSemaphoreTake(SemaphoreHandle_t   xSemaphore,
                          TickType_t   xTicksToWait);
```

Give

```
BaseType_t xSemaphoreGiveFromISR(SemaphoreHandle_t   xSemaphore,
                                 BaseType_t  *pxHigherPriorityTaskWoken);
```

Interrupt Management

Example Using binary semaphores (include “semphr.h”)

```
static uint32_t ulExampleInterruptHandler( void )
{
BaseType_t xHigherPriorityTaskWoken;

    /* The xHigherPriorityTaskWoken parameter must be initialized to pdFALSE as it will get set
     to pdTRUE inside the interrupt safe API function if a context switch is required. */
    xHigherPriorityTaskWoken = pdFALSE;

    /* This interrupt does nothing more than demonstrate how to synchronise a
    task with an interrupt.  A semaphore is used for this purpose.  Note
    lHigherPriorityTaskWoken is initialised to pdFALSE. */
    xSemaphoreGiveFromISR( xBinarySemaphore, &xHigherPriorityTaskWoken );

    /* If there was a task that was blocked on the semaphore, and giving the
    semaphore caused the task to unblock, and the unblocked task has a priority
    higher than or equal to the currently Running task (the task that this
    interrupt interrupted), then lHigherPriorityTaskWoken will have been set to
    pdTRUE internally within xSemaphoreGiveFromISR().  Passing pdTRUE into the
    portYIELD_FROM_ISR() macro will result in a context switch being pended to
    ensure this interrupt returns directly to the unblocked, higher priority,
    task.  Passing pdFALSE into portYIELD_FROM_ISR() has no effect. */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Interrupt Management

Example Using binary semaphores

```
static void vHandlerTask( void *pvParameters )
{
/* As per most tasks, this task is implemented within an infinite loop. */
     for( ;; )
     {
          /* Use the semaphore to wait for the event. The semaphore was created
          before the scheduler was started, so before this task ran for the first
          time. The task blocks indefinitely, meaning this function call will only
          return once the semaphore has been successfully obtained - so there is
          no need to check the value returned by xSemaphoreTake(). */
          xSemaphoreTake( xBinarySemaphore, portMAX_DELAY );
          /* To get here the event must have occurred. Process the event (in this case, just
          print out a message). */
          vPrintString( "Handler task - Processing event.\r\n" );
     }
}
```

Interrupt Management

Example: Create an application that executes a periodic task reading the buttons value of the board. If one button is pressed will trigger an interrupt. This interrupt will launch a deferred task using a semaphore. The deferred task will print a message indicating which button have been pressed.

Using 4 tasks
A periodic task reading buttons
An lower priority task showing a counter in leds
An interruptHandler task (ISR)
The Handlertask (deferred)

And the main() function

Use the following functions (in xgpiops.h and xScuGic.h)

```
void XGpioPs_IntrEnable(XGpioPs *InstancePtr, u8 Bank, u32 Mask);
void XGpioPs_SetIntrType(XGpioPs *InstancePtr, u8 Bank, u32 IntrType,
                        u32 IntrPolarity, u32 IntrOnAny);

void XGpioPs_SetDirection(XGpioPs *InstancePtr, u8 Bank, u32 Direction);
u32 XGpioPs_Read(XGpioPs *InstancePtr, u8 Bank);
xScuGic xInterruptController;
```

Interrupt Management

Example: Create an application that executes a periodic task reading the buttons value of the board. If one button is pressed will trigger an interrupt. This interrupt will launch a deferred task using a semaphore. The deferred task will print a message indicating which button have been pressed.

```
/* Scheduler includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"


/* Xilinx includes. */
#include "xgpio.h"
#include "xscugic.h"
#include "xil_exception.h"
```

#include "semphr.h"

Interrupt Management

Using **counting semaphores** (queues with length > 1)

It matters only the amount of items in the queue.

configUSE_COUNTING_SEMAPHORES = 1

Two functions: a) counting events
b) resource managements

```
SemaphoreHandle_t    xSemaphoreCreateCounting( UBaseType_t    uxMaxCount,
                                               UBaseType_t    uxInitialCount );
```

Returned value — If NULL is returned, the semaphore cannot be created because there is insufficient heap memory available

Interrupt Management

Using a queue (writing) from an interrupt

```
BaseType_t xQueueSendToFrontFromISR(QueueHandle_t xQueue,
                                    void *pvItemToQueue,
                                    BaseType_t *pxHigherPriorityTaskWoken );

BaseType_t xQueueSendToBackFromISR(QueueHandle_t xQueue,
                                   void *pvItemToQueue,
                                   BaseType_t *pxHigherPriorityTaskWoken );
```

- xQueue: The handle of the queue
- pvItemToQueue: A pointer to the data to be copied into the queue
- pxHigherPriorityTaskWoken : a variable to inform the application writer that a context switch should be performed

.

Return:

- pdPASS – OK
- errQUEUE_FULL – Error, queue full

Interrupt Management

Using a queue (reading) from an ISR

```
BaseType_t   xQueueReceiveFromISR( QueueHandle_t   xQueue,
                                   void *pvBuffer,
                              BaseType_t   *pxHigherPriorityTaskWoken );
```

- xQueue: The handle of the queue
- pBuffer: A pointer to the memory into which the data will be copied
- PxHigherPriorityTaskWoken: a variable to inform the application writer that a context switch should be performed

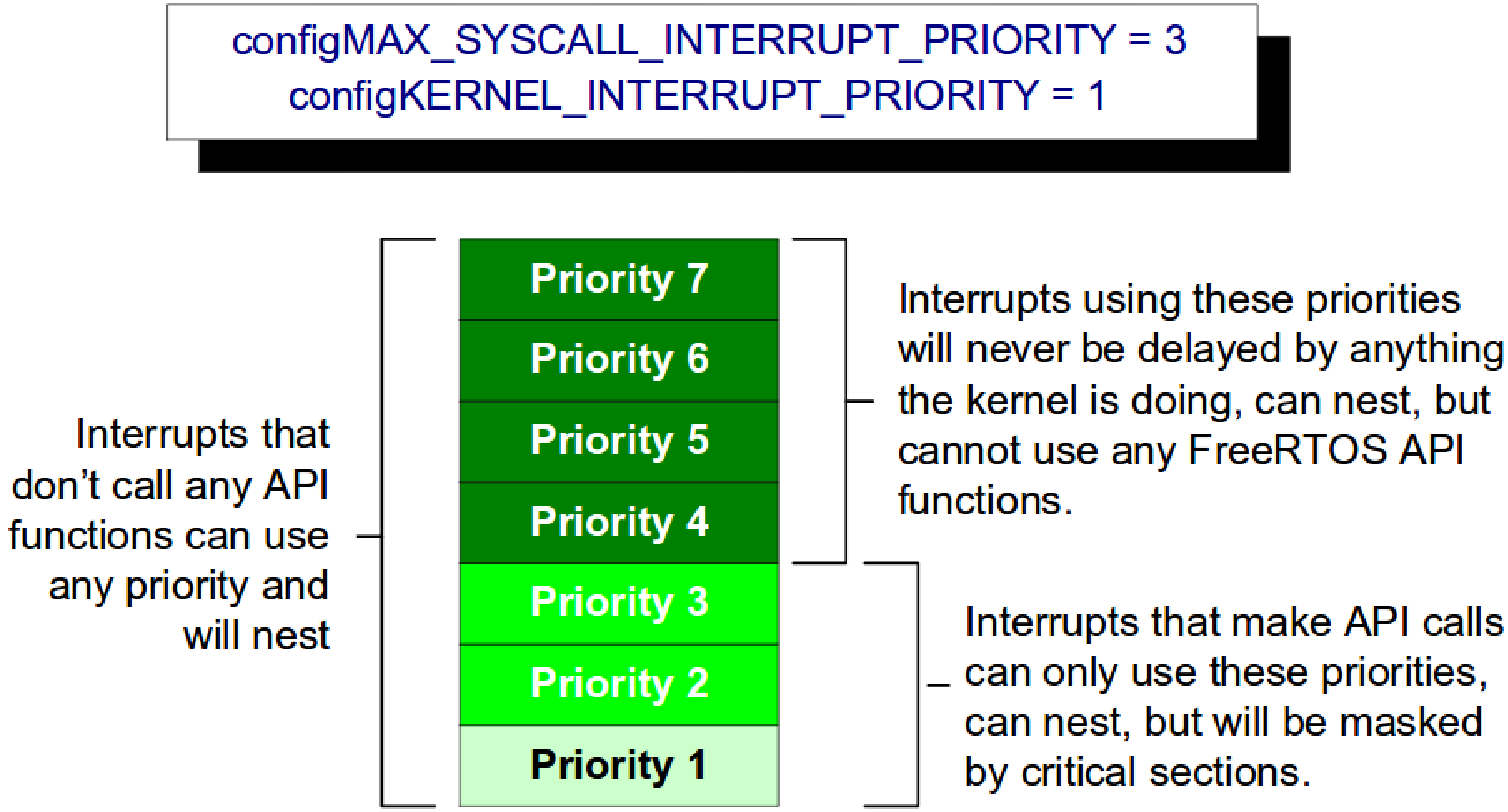
.

Return:

- pdPASS – OK
- errQUEUE_EMPTY – Error, queue full

Interrupt Management

Nested interrupts



Resources Management

Sharing resources between tasks
To ensure data consistency is maintained at all times access to a resource that is shared between tasks, or between tasks and interrupts, must be managed using a 'mutual exclusion' technique.

Critical sections: regions of code
Start with **taskENTER_CRITICAL()** and finish with **taskEXIT_CRITICAL()**

```
/* Ensure access to the PORTA register cannot be interrupted by placing it within
a critical section. Enter the critical section. */
taskENTER_CRITICAL();

PORTA |= 0x01;
/* Access to PORTA has finished, so it is safe to exit the critical section. */
taskEXIT_CRITICAL();
```

Disable interrupts

Resources Management

Sharing resources between tasks
To ensure data consistency is maintained at all times access to a resource that is shared between tasks, or between tasks and interrupts, must be managed using a ‘mutual exclusion’ technique.

Suspending the scheduler

Interrupts remain enabled

```
void vTaskSuspendAll( void );
```

The scheduler is suspended by calling vTaskSuspendAll().
Suspending the scheduler prevents a context switch from occurring

The scheduler is resumed (un-suspended) by calling **xTaskResumeAll().**

```
BaseType_t   xTaskResumeAll( void );
```

Resources Management

Sharing resources between tasks

To ensure data consistency is maintained at all times access to a resource that is shared between tasks, or between tasks and interrupts, must be managed using a ‘mutual exclusion’ technique.

Suspending the scheduler

```
void vPrintString( const char *pcString )
{
/* Write the string to stdout, suspending the scheduler as a method of mutual
exclusion. */
    vTaskSuspendScheduler();
    {
        printf( "%s", pcString );
        fflush( stdout );
    }
    xTaskResumeScheduler();
}
```

Resources Management

Sharing resources between tasks

To ensure data consistency is maintained at all times access to a resource that is shared between tasks, or between tasks and interrupts, must be managed using a ‘mutual exclusion’ technique.

Mutex

```
SemaphoreHandle_t xSemaphoreCreateMutex( void );
```

Returned value	If NULL is returned then the mutex could not be created because there is insufficient heap memory available.

Resources Management

Sharing resources between tasks

To ensure data consistency is maintained at all times access to a resource that is shared between tasks, or between tasks and interrupts, must be managed using a ‘mutual exclusion’ technique.

Mutex

```
static void prvNewPrintString( const char *pcString )
{
    xSemaphoreTake( xMutex, portMAX_DELAY );
    {
        /* The following line will only execute once the mutex has been successfully
        obtained. */
        printf( "%s", pcString );
        fflush( stdout );
        /* The mutex MUST be given back! */
    }
    xSemaphoreGive( xMutex );
}
```

Two risks: priority inversion or deadlock.

Resources Management

Sharing resources between tasks
To ensure data consistency is maintained at all times access to a resource that is shared between tasks, or between tasks and interrupts, must be managed using a ‘mutual exclusion’ technique.

Gatekeeper tasks: provide a clean method of implementing mutual exclusion without the risk of priority inversion or deadlock.

Only the gatekeeper task is allowed to access the resource directly.

Indirect use of resources from other tasks by using the services of the gatekeeper.

Resources Management

Sharing resources between tasks

To ensure data consistency is maintained at all times access to a resource that is shared between tasks, or between tasks and interrupts, must be managed using a 'mutual exclusion' technique.

Gatekeeper tasks

```
static void prvStdioGatekeeperTask( void *pvParameters )
{
char *pcMessageToPrint;
/* This is the only task that is allowed to write to standard out. */
    for( ;; )
    {
        /* The function will only return when a message has been successful
        received. */
        xQueueReceive( xPrintQueue, &pcMessageToPrint, portMAX_DELAY );
        /* Output the received string. */
        printf( "%s", pcMessageToPrint );
        fflush( stdout );
    /* Loop back to wait for the next message. */
    }
}
```

QUESTIONS?