Interrupt Management
(slides are based on Prof. Dr. Jian-Jia Chen and http://www.freertos.org)

Anas Toma

LS 12, TU Dortmund

November 29, 2018
How to Detect Events?

Polling:
- Simple: Constantly testing a port to see if data is available.
- Inefficient, as it requires CPU for busy-looping
  - Long polling interval: late event detection
  - Short polling interval: high CPU overhead

Interrupt:
- An interrupt is an external **hardware/software** event that causes the **CPU to interrupt** the current instruction sequence and call a special **interrupt service routine (ISR)**.
- **More efficient**, as the CPU can continue while it is waiting for I/O
Interrupt

ISR (Interrupt Service Routine) or interrupt handler: a piece of code that is executed in response to an interrupt.

- Why do we need to consider interrupts?
  - Use the operating system to run programs, manage files, access peripherals, etc.
  - Help peripherals “talk” to microprocessors
  - Help measure time and control the timing of certain tasks in microprocessors
    - Instead of implementing timing via loops

- Three types:
  - Software interrupts: initiated by the INT instruction in programs
  - Hardware interrupts: initiated by peripheral hardware
  - Exceptions: occur in response to error states in the processor or during debugging (trace, breakpoints etc.)
What to Notice for Interrupt Handling in RTOS?

• General:
  • The interrupt handler should be **fast, efficient**, and **predictable**
    • It is normally desirable to keep each ISR as short as possible with low overhead
    • The execution time of an interrupt handler should be bounded

• FreeRTOS:
  • There is no specific event processing strategy on the application designer
  • It provides **features** that allow the chosen strategy to be implemented in a simple way
  • Our scope
    • Which APIs can be used within an interrupt service routine?
    • How to defer interrupts?
    • How to deal with interrupt nesting?
Interrupt Handler: Rule of Thumb

• When a system call or API must be called in the interrupt service routine, make sure whether you can call it correctly.
  • If the call/API may result in blocking the execution, it must be handled carefully by calling the proper variations that can be used in ISR or are safe in ISR.
  • In FreeRTOS, these functions are ended with xxxxISR().

• For example, if printf is called in ISR, it must be guaranteed to be safe. Most of time, it is a mistake to do this.
  • **Reason**: if the interrupts are disabled before entering ISR (most cases) and the stdout is a device that requires interrupts for proper operations, it all result in a deadlock. The system will wait for an interrupt that never occurs because interrupts are disabled.

• Fast fast fast and fast.
Warm Up: Setup Interrupts

- At boot time, OS installs corresponding interrupt handlers into the interrupt vector.
- During I/O interrupt, the controller signals that device is ready.
- Interrupts are hardware-dependent. Please have a look at:
  - port.c
  - interrupts.c
  - bcm2835_intc.h
Start the Scheduler

```c
portBASE_TYPE xPortStartScheduler( void )
{
    /* Start the timer that generates the tick ISR. Interrupts are disabled here already. */
    prvSetupTimerInterrupt();

    /* Start the first task. */
    vPortISRStartFirstTask();

    /* Should not get here! */
    return 0;
}
```
Tick Interrupts Frequency

1 /*
2  * Setup the timer 0 to generate the tick interrupts ←
3  * at the required frequency.
4  */
5 static void prvSetupTimerInterrupt( void )
6 {
7   unsigned long ulCompareMatch;
8
9  /* Calculate the match value required for our wanted ←
10  * tick rate. */
11  ulCompareMatch = configCPU_CLOCK_HZ / ←
12     configTICK_RATE_HZ;
13 ...
14 }
Example

1 - Task 1 is Running when an interrupt occurs.

2 - The ISR executes, handles the interrupting peripheral, clears the interrupt, then unblocks Task 2.

3 - The priority of Task 2 is higher than the priority of Task 1, so the ISR returns directly to Task 2, in which the interrupt processing is completed.

4 - Task 2 enters the Blocked state to wait for the next interrupt, allowing Task 1 to re-enter the Running state.
Semaphore

Semaphores usually have two purposes

- Mutex: ensure threads don’t access critical section at same time
- Scheduling constraints: ensure tasks execute in specific order

Semaphore has two types

- Binary semaphore: either 0 or 1
- Counting semaphore: integer numbers

Semaphore operations

- Allocate and Initialize
  - Semaphore is initialized as a non-negative integer value
- Reading or writing to the semaphore can only be done by using specific functions: wait() or signal()
- Wait or Test
  - P() for “test” comes from Dutch (proberen)
  - Waits until value of the semaphore is $> 0$, then decrements semaphore value
- Signal or Increment or Post
  - V() for “increment” comes from Dutch (verhogen)
  - Increments value of semaphore
Binary Semaphore and Interrupts

1 - When the interrupt occurs, Task1 is Running, and Task2 is Blocked waiting for a semaphore.

2 - The ISR executes, handles the interrupting peripheral, clears the interrupt, then ‘gives’ a semaphore to unblock Task 2.

ISR

Task2
(deferred processing task)

Task1

t1 t2 t3 t4

3 - Task 2 completes any further processing necessitated by the interrupt, then blocks on the semaphore to wait to be unblocked again by the next interrupt.
Synchronizing a Task with an Interrupt

The semaphore is not available...

...so the task is blocked waiting for the semaphore.

An interrupt occurs...that 'gives' the semaphore....

...which unblocks the task (the semaphore is now available)....
Synchronizing a Task with an Interrupt cont.

...that now successfully ‘takes’ the semaphore, so it is unavailable once more.

The task can now perform its action, when complete it will once again attempt to ‘take’ the semaphore which will cause it to re-enter the Blocked state.
FreeRTOS Semaphores (semphr.h)

- **void vSemaphoreCreateBinary(xSemaphoreHandle xSemaphore)**
  - Implement a semaphore by using the existing queue mechanism.
  - The queue length is 1 as this is a binary semaphore.
  - This type of semaphore can be used for pure synchronization between tasks or between an interrupt and a task.

- **portBASE_TYPE xSemaphoreTake(xSemaphoreHandle xSemaphore, portTickType xTicksToWait)**
  - Wait for a semaphore (P())
  - **xTicksToWait**: Number of ticks the task should be blocked before getting the semaphore. Setting to `portMAX_DELAY` will cause the task wait infinitely (without timeout).

- **portBASE_TYPE xSemaphoreGive( xSemaphoreHandle xSemaphore )**
  - Increment a semaphore (V())
  - This must not be used from an ISR, as `xSemaphoreGiveFromISR()` is ISR safe
FreeRTOS Semaphores (semphr.h) cont.

- `portBASE_TYPE xSemaphoreGiveFromISR( xSemaphoreHandle xSemaphore, portBASE_TYPE *pxHigherPriorityTaskWoken )`
  - A special form of `xSemaphoreGive()` that is specifically for use within an interrupt service routine
  - `pxHigherPriorityTaskWoken`: A pointer parameter used when it is required to switch to a higher priority task
FreeRTOS Semaphores (semphr.h) cont.

If the priority of a task that is unblocked by a FreeRTOS API function (eg. `xQueueSendToBack()` or `xSemaphoreGiveFromISR()`) is higher than the priority of the task in the Running state, then a switch to the higher priority task should occur.

- If the API function was called from a task: The context switch occurs automatically within the API function, if `configUSE_PREEMPTION` is set to 1.
- If the API function was called from an interrupt: The `pxHigherPriorityTaskWoken` variable is set to `pdTRUE` in order to inform the application writer that a context switch should be performed.

  - The variable pointed to by `pxHigherPriorityTaskWoken` must be initialized to `pdFALSE` before it is used for the first time.
  - A context switch is performed before the interrupt is exited.
  - If a context switch from the ISR it is not requested, the higher priority task will remain in the Ready state until the next tick interrupt.
**One Interrupt Occurs before the Task has Finished**

The semaphore is not available... ...the task is blocked waiting for the semaphore.

Interrupt! xSemaphoreGiveFromISR()

An interrupt ‘gives’ the semaphore....

... which unblocks the task.

The task ‘takes’ the semaphore, so the semaphore is no longer available. The task then starts to process the first event.
One Interrupt Occurs before the Task has Finished cont.

**Interrupt!**

xSemaphoreGiveFromISR()

Another interrupt occurs while the task is still processing the first event. The ISR ‘gives’ the semaphore again, effectively latching the event so the event is not lost.

**Task**

vProcessEvent()

The task is still processing the first event.

**Task**

xSemaphoreTake()

When the task has finished processing the first event it calls xSemaphoreTake() again. Another interrupt has already occurred, so the semaphore is already available.

**Task**

vProcessEvent()

The task takes the semaphore (without entering the Blocked state), then processes the second event.
Two Interrupts Occur before the Task has Finished

The semaphore is not available...
...the task is blocked waiting for the semaphore.

An interrupt ‘gives’ the semaphore....

... which unblocks the task.

The task ‘takes’ the semaphore, so the semaphore is no longer available. The task then starts to process the first event.
Two Interrupts Occur before the Task has Finished cont.

A second interrupt occurs while the task is still processing the first event. The ISR ‘gives’ the semaphore again, effectively latching the event so the event is not lost.

A third interrupt occurs while the task is still processing the first event. The ISR cannot give the semaphore again, because the semaphore is already available, and the event is lost.

When the task has finished processing the first event it calls xSemaphoreTake() again. A second and third interrupt have already occurred, so the semaphore is already available.
Two Interrupts Occur before the Task has Finished cont.

The task takes the semaphore (without entering the Blocked state), so the semaphore is no longer available. The task then processes the second event.

When the task has finished processing the second event it calls xSemaphoreTake() again, but the semaphore is not available, and the task enters the Blocked state to wait for the next interrupt - even though the third event has not been processed.
Counting Semaphore

[The semaphore count is 0]

Task
xSemaphoreTake()

The task is blocked waiting for a semaphore

Interrupt!
xSemaphoreGiveFromISR()

[The semaphore count is 1]

Task
xSemaphoreTake()

An interrupt occurs...that 'gives' the semaphore....

Interrupt!
xSemaphoreGiveFromISR()

[The semaphore count is 1]

Task
xSemaphoreTake()

...which unblocks the task (the semaphore is now available)....
Counting Semaphore cont.

[The semaphore count is 0]

Task
vProcessEvent()

...that now successfully ‘takes’ the semaphore, so it is unavailable once more. The task now starts to process the event.

Interrupt!
xSemaphoreGiveFromISR()

[The semaphore count is 2]

Another two interrupts occur while the task is still processing the first event. Both ISRs ‘give’ the semaphore, effectively latching both events, so neither event is lost.

Task
vProcessEvent()

The task is still processing the first event.

[The semaphore count is 1]

Task
xSemaphoreTake()

When the task has finished processing the first event it calls xSemaphoreTake() again. Another two semaphores are already ‘available’, one is taken without the task ever entering the Blocked state, leaving one ‘latched’ semaphore still available.
Semaphore Example Implementation

```c
1 void vExampleInterruptHandler()
2 {
3     static portBASE_TYPE xHigherPriorityTaskWoken;
4
5     // Clear interrupt Flag.
6     GPIOPinIntClear(GPIO_PORTE_BASE, (UP | DOWN | LEFT | RIGHT));
7
8     xHigherPriorityTaskWoken = pdFALSE;
9     xSemaphoreGiveFromISR(xBinSemp, &xHigherPriorityTaskWoken);
10    if (xHigherPriorityTaskWoken == pdTRUE){
11       portYIELD();
12    }
13 }

1 void vHandlerTask( void *pvParameters ){
2     char str[]="Intensity Changed!";
3     int intensity=5;
4     for (;;){
5         xSemaphoreTake( xBinSemp, portMAX_DELAY );
6         intensity++;
7         print( str, 15, 45, intensity%16 );
8     }
9 }
```
Interrupt Nesting

- When an interrupt can occur during the ISR, an interrupt service routine may be interrupted. This is called interrupt nesting.
  - On most microprocessors, interrupt nesting happens automatically
  - The Intel x86 microprocessors disable all interrupts automatically whenever they enter any interrupt routine; therefore, the interrupt routines must reenable interrupts to allow interrupt nesting.
  - A higher-priority interrupt can interrupt a lower-priority interrupt routine
  - Proper settings should be done in FreeRTOS configuration to reflect the nesting
    - `#define configKernel_INTERRUPT_PRIORITY X`: Sets the interrupt priority used by the tick interrupt
    - `#define configMAX_SYSCALL_INTERRUPT_PRIORITY Y`: Sets the highest interrupt priority from which interrupt safe FreeRTOS API functions can be called.