# **Embedded Systems Essentials with Arm: Get Practical with Hardware**

## Module 4

## KV2: Using Interrupts and Dealing With Time in an RTOS Context

Now that we know how threads are developed and interact with each other within the Mbed RTOS, we can look at using interrupts to respond to external events and dealing with time in an RTOS context.

Let’s start with reviewing interrupts. In previous modules, we covered the underlying concepts and use of interrupts, and this introduced two microprocessor operating modes, which now also apply to the Mbed RTOS.​ They are:

1. Thread mode. The default application mode, and all user threads execute in this mode. This mode uses dedicated thread-specific stack memory.​
2. Handler mode. The interrupt mode, in which system code and interrupt handlers execute. It uses interrupt service routine, or ISR, stack memory.​

Here again is the table of interrupt API functions, with some RTOS-oriented adjustments.

|  |  |
| --- | --- |
| **Function Name​** | **Description ​** |
| **InterruptIn** (PinName pin)​ | Create an InterruptIn connected to the specified pin.​ |
| int **read**()​ | Read the input, represented as 0 or 1 (int)​ |
| void **rise** (Callback< void()> func)​ | Attach a function to call when a rising edge occurs on the input.​ |
| void **fall** (Callback< void()> func)​ | Attach a function to call when a falling edge occurs on the input.​ |
| void **enable\_irq**()​ | Enable the interrupt.​ |
| void **disable\_irq**()​ | Disable the interrupt.​ |

For more details on the use of these functions, please check the Mbed website. ​

Next is applying interrupts. The ISR, or interrupt service routine, should not disturb the flow

of the RTOS, so it should execute in the fastest possible time. This is why there are greater restrictions when working within an RTOS environment, and these restrictions can be summarized as follows:

* Avoid the use of wait or delay loops, it must be infinite while looping or blocking calls.​
* Avoid calls to bulky library functions, such as printf or malloc. These are non-reentrant functions, meaning they cannot safely be called, interrupted, and then recalled before the first call has finished without resulting in memory corruption.

Here is a table showing which functions can be used in an ISR.

|  |  |
| --- | --- |
| **Function Name​** | **Can it be used in ISR? ​** |
| Mutex()​ | X​ |
| lock()​ | X​ |
| unlock()​ | X​ |
| Semaphore()​ | X​ |
| acquire()​ | X​ |
| release()​ | √​ |
| try\_acquire()​ | √​ |
| Queue()​ | X​ |
| Empty()​ | √​ |
| full()​ | √​ |
| get()​ | √ if the ms param. is set to 0​ |
| put()​ | √ if the ms param. is set to 0​ |

Now let us cover dealing with time in the RTOS context. This table shows the API functions available in the Mbed RTOS that deal with time.

|  |  |
| --- | --- |
| **API** | **Full Profile** |
| RTC | √​ |
| Ticker | √​ |
| Time | √​ |
| Timeout | √​ |
| Timer | √​ |
| Wait | √​ |

Timer, timeout and ticker were mentioned in a previous module, and they remain available for use in the RTOS context. Timeout and ticker both result in interrupts, so when applying them bear in mind the interrupt constraints mentioned earlier.

A deterministic OS, program, or algorithm is one that always produces the same results for a given set of inputs. ​Certain operating systems however introduce timing uncertainties which can cause non-deterministic behavior. For this reason, the Mbed OS has been designed to ensure that it displays fully deterministic behavior, which is essential for high reliability and safety-critical embedded systems.​

Let’s look at a timing example​. This example shows a simple RTOS application, with two LEDs controlled by timeout and ticker, and a third by an external potentiometer.

|  |
| --- |
| /\*Simple demo of Timer and Ticker in RTOS context.​  For Nucleo STM42F401 board \*/​  ​  #include "mbed.h"​  ​  Ticker tickerflipper;​  Timeout timeoutflipper;​  DigitalOut led1(D6);​  DigitalOut led2(D7);​  PwmOut LED\_var(LED1); ​  AnalogIn pot1(A0); |

Both LED1 and LED2 start by being on, and after 1.5 seconds LED1 is turned off by the timeout, and it stays off.​

LED2 continues switching, every two seconds, controlled by the ticker. ​So, we can see that the variable LED is continuously adjustable.​  
  
We are already aware of the characteristics and limitations of working in a resource-controlled environment, so in the next knowledge video we will learn how to do so using the bare metal profile.

**Visit arm.com for more information.**