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

## Module 3

## KV4: RTOS on the Mbed Platform

Now that we have some underlying knowledge about the RTOS, let’s turn to the Mbed RTOS and how we implement RTOS concepts.

Arm provides an easy-to-use, powerful and highly effective RTOS to run on Mbed-enabled devices. The Mbed RTOS API is based on the Keil RTX RTOS and uses the CMSIS-RTOS API open standard. The advantage of this is that it provides a common API for real-time operating systems;​ and a standardized programming interface that is portable to many real-time operating systems. This allows it to enable software templates, middleware, libraries, and other components that can work across various supported RTOS systems.

It’s important to note that the Mbed RTOS uses the terminology of thread, rather than task. For the rest of this module, we’ll stick to this convention.

When in default mode, the Mbed RTOS applies time slices of 1 millisecond. At the end of any time slice, the thread of highest priority preempts all others. Threads of equal priority are executed using round robin scheduling, and a program always has at least one thread, which is main(). This is allocated 4KB of stack memory.

Other threads are added by the system, for example:​

* Idle: This thread is run by the scheduler when there's no other activity in the system (for example, if all other threads are waiting for some event). It's used to make sure the board is not burning empty processor cycles, but is put to sleep for as long as possible.​
* Timer: This thread handles system and user timer objects.​

As the programmer, you can create further threads (also allocated 4KB of stack) using the thread class. We’ll see examples of this later. ​Some drivers also create additional threads.

We have a number of categories for scheduling APIs. The scheduling capabilities of the Mbed OS give it its RTOS qualities. They fall into the categories shown in reverse alphabetical order. As always, visit the Mbed website for more details.

|  |  |
| --- | --- |
| **API​** | **Summary​** |
| **Thread ​** | The class that allows defining, creating and controlling parallel tasks.​ |
| **ThisThread ​** | The class with which you can control the current thread.​ |
| **Semaphore ​** | The class that manages thread access to a pool of shared resources of a certain type.​ |
| **Queue ​** | The class that allows you to queue pointers to data from producer threads to consumer threads.​ |
| **Mutex ​** | The class used to synchronize the execution of threads.​ |
| **Mail​** | The API that provides a queue combined with a memory pool for allocating messages.​ |
| **Kernel interface functions ​** | Kernel namespace implements functions to control or read RTOS information, such as tick count.​ |
| **IdleLoop ​** | Background system thread, executed when no other threads are ready to run.​ |
| **EventFlags​** | An event channel that provides a way of notifying other threads about conditions or events. ​ |
| **ConditionVariable​** | The class that provides a mechanism to wait safely for or signal a single state change. ​ |

Now let’s look at the Thread class. As we move forward, you’ll be able to see how some of these functions are used.

|  |  |
| --- | --- |
| **Function Name​** | **Description ​** |
| **Thread** (osPriority priority=osPriorityNormal, uint32\_t stack\_size=OS\_STACK\_SIZE, unsigned char \*stack\_mem=nullptr, const char \*name=nullptr)​ | Allocate a new thread without starting execution, setting priority, stack size, stack pointer and thread name.​ |
| osPriority **get\_priority** ()​ | Get priority of an active thread.​ |
| osStatus **set\_priority** (osPriority priority)​ | Set priority of an active thread.​ |
| osStatus **join** ()​ | Wait for thread to terminate (then calling function can continue).​ |
| osStatus **start** (mbed::Callback< void()> task)​ | Start a thread executing the function specified by task. Returns status code that indicates the execution status of the function.​ |
| State **get\_state** ()​ | Get the state of this thread. Returns code indicating thread status.​ |
| osStatus **terminate** ()​ | Terminate execution of a thread and remove it from active threads.​ |

This is another table of functions, this time for ThisThread. ThisThread allows you to manipulate a certain thread from within that thread. Here we see some of the functions available to you.

|  |  |
| --- | --- |
| **Function Name​** | **Description ​** |
| uint32\_t **flags\_clear** (uint32\_t flags)​ | Clears the specified Thread Flags of the currently running thread​ |
| uint32\_t **flags\_get** () ​ | Returns the Thread Flags currently set for the currently running thread.​ |
| uint32\_t **flags\_wait\_all** (uint32\_t flags, bool clear=true)​  ​ | Wait for all of the specified Thread Flags to become signaled for the current thread.​ |
| uint32\_t **flags\_wait\_any** (uint32\_t flags, bool clear=true)​ | Wait for any of the specified Thread Flags to become signaled for the current thread​ |
| void **sleep\_for** (uint32\_t millisec)​ | Sleep for a specified time period in millisec​ |
| void **sleep\_until** (uint64\_t millisec)​ | Sleep until a specified time in millisec ​ |
| void **yield** ()​ | Pass control to next equal-priority thread that is in state READY.​ |

At this point it’s useful to chat briefly about time delays. As the Mbed RTOS is designed for low-power applications, it aims to put the microcontroller into “sleep” mode whenever it’s not active. ​Thus, the older wait functions, which provided time delays by spinning the microcontroller in carefully defined timing loops, are being discontinued.

They are replaced with constructs such as Thread\_sleep\_for (time in ms), which give the opportunity for the microcontroller to be put to sleep, with just a background timer running. ​Where a time delay of less than 1 millisecond is wanted, then the wait\_us (time in us) is still available.

Now, let’s look at three examples of increasing complexity. We’ve seen similar examples in this course, but now we’re recognizing that we’re writing in an RTOS context. This example below shows the simplest possible RTOS-based program. The main() function is the only user-defined thread. Apart from use of ThisThread, you don’t see that it is RTOS-based.

SINGLE THREAD:

|  |
| --- |
| /\* mbed Microcontroller Library ​   * Copyright (c) 2019 ARM Limited ​ * \* SPDX-License-Identifier: Apache-2.0 ​ * \*/ ​   #include "mbed.h" ​  ​  // Blinking rate in milliseconds ​  #define BLINKING\_RATE 500ms ​  ​  int main() { ​  // Initialise the digital pin LED1 as an output ​  DigitalOut led(LED1); ​  ​  while (true) { ​  led = !led; ​  ThisThread::sleep\_for(BLINKING\_RATE); ​  }​  } |

This example shows a simple two-thread program. The first thread is main(). A second thread, called here thread2, is created as a function. It is immediately started from the first thread, and the two continue switching their LEDs indefinitely.

TWO THREADS:

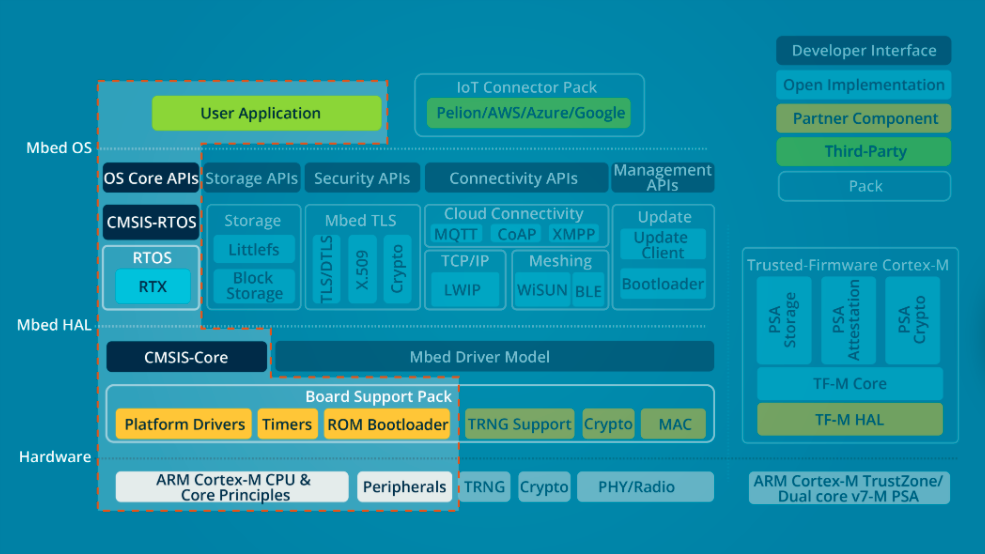
|  |
| --- |
| /\* Copyright (c) 2020 Arm Limited and affiliates.​  \* SPDX-License-Identifier: Apache-2.0 \*/​  #include "mbed.h"​  ​  DigitalOut led1(LED1);​  DigitalOut led2(LED2); ​  ​  Thread thread2;​  void led2\_thread()​  {​  while (true) {​  led1 = !led1;​  ThisThread::sleep\_for(1100);​  }​  }​  ​  int main()​  {​  thread2.start(led2\_thread);​  ​  while (true) {​  led2 = !led2;​  ThisThread::sleep\_for(500);​  }​  } |

This example shows how one thread can both start and terminate another. The Thread thread2 is started from the main() thread; after a wait of 10 seconds it is stopped, by setting the running variable to false. Notice the use of the Callback API, which provides a convenient way to pass arguments to spawned threads.

ONE THREAD CONTROLLING ANOTHER:

|  |
| --- |
| /\*From https://os.mbed.com/docs/mbed-os/v6.4/mbed-os-api-doxy/classrtos\_1\_1\_thread.html#details​  \*/​  ​  #include "mbed.h"​  ​  Thread thread2; //name thread2 chosen for this thread​  DigitalOut led1(LED1);​  volatile bool running = true;​  ​  // Blink function toggles the led in a long running loop​  void blink(DigitalOut \*led1) {​  while (running) {​  \*led1 = !\*led1;​  ThisThread::sleep\_for(1000); //this sets the blink rate​  }​  }​  // Spawns a thread to run blink for 5 seconds​  int main() {​  thread2.start(callback(blink, &led1)); //Starts thread2 executing.​  ThisThread::sleep\_for(10000); //this sets how long thread2 will be able to run​  running = false;​  thread2.join(); //Wait for thread2 to terminate.​  }​ |

Despite initially appearing complex, it is useful to examine the overall architecture of the Mbed RTOS. The things of immediate interest to us here are enclosed within the red dashed line.



To summarize,

* Operating systems provide the framework for program development in most computing systems of any complexity.​
* The RTOS is a special type of operating system, which is designed to deliver results according to clear timing deadlines. and its activities are structured into tasks or threads.​
* The Scheduler in the RTOS shares access to the CPU between the threads, according to their priority. This gives the illusion of simultaneous thread execution.​
* Threads can interact with each other, applying carefully managed techniques.​

What we have learned here about the RTOS is just the beginning. We’ll widen our knowledge and expertise into other areas of RTOS use, applying the Mbed RTOS.​

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