µC/OS-III is a scalable, ROMable, preemptive real-time kernel that manages an unlimited number of tasks. µC/OS-III is a third-generation kernel, offering all of the services expected from a modern real-time kernel including resource management, synchronization, inter-task communication, and more. However, µC/OS-III also offers many unique features not found in other real-time kernels, such as the ability to perform performance measurements at run time, directly signal or send messages to tasks, and pending (i.e., waiting) on such kernel objects as semaphores and message queues.

Here is a list of features provided by µC/OS-III:

**Source Code:** µC/OS-III is provided in ANSI-C source form. The source code for µC/OS-III is arguably the cleanest and most consistent kernel code available. Clean source is part of the corporate culture at Micrium. Although many commercial kernel vendors provide source code for their products, unless the code follows strict coding standards and is accompanied by complete documentation with examples to show how the code works, these products may be cumbersome and difficult to harness. With this book, you will gain a deep understanding of the inner workings of µC/OS-III, which will protect your investment.

**Intuitive Application Programming Interface (API):** µC/OS-III is highly intuitive. Once familiar with the consistent coding conventions used, it is simple to predict the functions to call for the services required, and even predict which arguments are needed. For example, a pointer to an object is always the first argument, and a pointer to an error code is always the last one.

**Preemptive multitasking:** µC/OS-III is a preemptive multi-tasking kernel and therefore, µC/OS-III always runs the most important ready-to-run task.

**Round robin scheduling of tasks at equal priority:** µC/OS-III allows multiple tasks to run at the same priority level. When multiple tasks at the same priority are ready-to-run, and that priority level is the most important level, µC/OS-III runs each task for a user-specified time called a time quanta. Each task can define its own time quanta, and a task can also give up the CPU to another task at the same priority if it does not require the full time quanta.

**Low interrupt disable time:** µC/OS-III has a number of internal data structures and variables that it needs to access atomically. To ensure this, µC/OS-III is able to protect these critical
regions by locking the scheduler instead of disabling interrupts. Interrupts are therefore
disabled for very little time. This ensures that µC/OS-III is able to respond to some of the
fastest interrupt sources.

**Deterministic:** Interrupt response with µC/OS-III is deterministic. Also, execution times of
most services provided by µC/OS-III are deterministic.

**Scalable:** The footprint (both code and data) can be adjusted based on the requirements of the
application. Adding and removing features (i.e., services) is performed at compile time through
approximately 60 `#defines` (see `os_cfg.h`). µC/OS-III also performs a number of run-time
checks on arguments passed to µC/OS-III services. Specifically, µC/OS-III verifies that the
user is not passing `NULL` pointers, not calling task level services from ISRs, that arguments are
within allowable range, and options specified are valid, etc. These checks can be disabled (at
compile time) to further reduce the code footprint and improve performance. The fact that
µC/OS-III is scalable allows it to be used in a wide range of applications and projects.

**Portable:** µC/OS-III can be ported to a large number of CPU architectures. Most µC/OS-II
ports are easily converted to work on µC/OS-III with minimal changes in just a matter of
minutes and therefore benefit from more than 45 CPU architectures already supported by
µC/OS-II.

**ROMable:** µC/OS-III was designed especially for embedded systems and can be ROMed
along with the application code.

**Run-time configurable:** µC/OS-III allows the user to configure the kernel at run time.
Specifically, all kernel objects such as tasks, stacks, semaphores, event-flag groups, message
queues, number of messages, mutual exclusion semaphores, memory partitions and timers, are
allocated by the user at run time. This prevents over-allocating resources at compile time.

**Unlimited number of tasks:** µC/OS-III supports an unlimited number of tasks. From a
practical standpoint, however, the number of tasks is actually limited by the amount of
memory (both code and data space) that the processor has access to. Each task requires its own
stack space and, µC/OS-III provides features to allow stack growth of the tasks to be monitored
at run-time.

µC/OS-III does not impose any limitations on the size of each task stack, except that there be a
minimum size based on the CPU used.
Unlimited number of priorities: µC/OS-III supports an unlimited number of priority levels. However, configuring µC/OS-III for between 32 and 256 different priority levels is more than adequate for most applications.

Unlimited number of kernel objects: µC/OS-III allows for any number of tasks, semaphores, mutual exclusion semaphores, event flags, message queues, timers, and memory partitions. The user allocates all kernel objects at run-time.

Services: µC/OS-III provides all the services expected from a high-end real-time kernel, such as task management, time management, semaphores, event flags, mutexes, message queues, software timers, fixed-size memory pools, etc.

Mutual Exclusion Semaphores (Mutexes): Mutexes are provided for resource management. Mutexes are special types of semaphores that have built-in priority inheritance, which eliminate unbounded priority inversions. Accesses to a mutex can be nested and therefore, a task can acquire the same mutex up to 250 times. Of course, the mutex owner needs to release the mutex an equal number of times.

Nested task suspension: µC/OS-III allows a task to suspend itself or another task. Suspending a task means that the task will not be allowed to execute until the task is resumed by another task. Suspension can be nested up to 250 levels deep. In other words, a task can suspend another task up to 250 times. Of course, the task must be resumed an equal number of times for it to become eligible to run on the CPU.

Software timers: You can define any number of “one-shot” and/or “periodic” timers. Timers are countdown counters that perform a user-definable action upon counting down to 0. Each timer can have its own action and, if a timer is periodic, the timer is automatically reloaded and the action is executed every time the countdown reaches zero.

Task Signals: µC/OS-III allows an ISR or task to directly signal a task. This avoids having to create an intermediate kernel object such as a semaphore or event flag just to signal a task, and results in better performance.

Task Messages: µC/OS-III allows an ISR or a task to send messages directly to a task. This avoids having to create and use a message queue, and also results in better performance.

Task registers: Each task can have a user-definable number of “task registers.” Task registers are different than CPU registers. Task registers can be used to hold “errno” type variable, IDs,
interrupt disable time measurement on a per-task basis, and more.

**Error checking:** µC/OS-III verifies that NULL pointers are not passed, that the user is not calling task-level services from ISRs, that arguments are within allowable range, that options specified are valid, that a pointer to the proper object is passed as part of the arguments to services that manipulate the desired object, and more. Each µC/OS-III API function returns an error code concerning the outcome of the function call.

**Built-in performance measurements:** µC/OS-III has built-in features to measure the execution time of each task, stack usage of each task, number of times a task executes, CPU usage, ISR-to-task and task-to-task response time, peak number of entries in certain lists, interrupt disable and scheduler lock time on a per-task basis, and more.

**Built-in trace points:** µC/OS-III has built-in trace points throughout the code to record all the kernel events and interrupts in real-time using some of the most popular tracing analysis tools (e.g. Percepio’s TraceAlyzer and SEGGER SystemView).

**Can easily be optimized:** µC/OS-III was designed so that it could easily be optimized based on the CPU architecture. Most data types used in µC/OS-III can be changed to make better use of the CPU’s natural word size. Also, the priority resolution algorithm can easily be written in assembly language to benefit from special instructions such as bit set and clear, as well as count-leading-zeros (CLZ), or find-first-one (FF1) instructions.

**Deadlock prevention:** All of the µC/OS-III “pend” services include timeouts, which help avoid deadlocks.

**Tick handling at task level:** The clock tick manager in µC/OS-III is accomplished by a task that receives a trigger from an ISR. Handling delays and timeouts by a task greatly reduces interrupt latency. Also, µC/OS-III uses a hashed delta list mechanism, which further reduces the amount of overhead in processing delays and timeouts of tasks.

**User definable hooks:** µC/OS-III allows the port and application programmer to define “hook” functions, which are called by µC/OS-III. A hook is simply a defined function that allows the user to extend the functionality of µC/OS-III. One such hook is called during a context switch, another when a task is created, yet another when a task is deleted, etc.

**Timestamps:** For time measurements, µC/OS-III requires that a 16-bit or 32-bit free running counter be made available. This counter can be read at run time to make time measurements of
certain events. For example, when an ISR posts a message to a task, the timestamp counter is automatically read and saved as part of the message posted. When the recipient receives the message, the timestamp is provided to the recipient, and by reading the current timestamp, the time it took for the message to be received can be determined.

**Built-in support for Kernel Awareness debuggers:** This feature allows kernel awareness debuggers to examine and display µC/OS-III variables and data structures in a user-friendly way. The kernel awareness support in µC/OS-III can be used by µC/Probe to display this information at run-time.

**Object names:** Each µC/OS-III kernel object can have a name associated with it. This makes it easy to recognize what the object is assigned to. You can thus assign an ASCII name to a task, a semaphore, a mutex, an event flag group, a message queue, a memory partition, and a timer. The object name can have any length, but must be NUL terminated.