µC/OS-III™

The Real-Time Kernel

Reference Manual

Jean J. Labrosse

Micriµm Press
Weston, FL 33326

µC/OS-III User's Manual

uC-OS-III Reference Manual

The µC/OS-III Reference Manual includes the following sections:

- µC/OS-III Configuration Manual
- Migrating from µC/OS-II to µC/OS-III
- Bibliography
- Licensing Policy

uC-OS-III Configuration Manual

Three (3) files are used to configure µC/OS-III as highlighted in the figure below: `os_cfg.h`, `os_cfg_app.h` and `os_type.h`.

The table below shows where these files are typically located on your on a computer.

File	Directory
os_cfg.h	\Micrium\Software\uCOS-III\Cfg\Template
os_cfg_app.h	\Micrium\Software\uCOS-III\Cfg\Template
os_type.h	\Micrium\Software\uCOS-III\Source

Table - Configuration files and directories

Figure - µC/OS-III File Structure

(1) **µC/OS-III Features (** `os_cfg.h` **):**

`os_cfg.h` is used to determine which features are needed from µC/OS-III for an application (i.e., product). Specifically, this file allows a user to determine whether to include semaphores, mutexes, event flags, run-time argument checking, etc.

(2) **µC/OS-III Data Types (** `os_type.h` **):**

`os_type.h` establishes µC/OS-III-specific data types used when building an application. It specifies the size of variables used to represent task priorities, the size of a semaphore count, and more. This file contains recommended data types for µC/OS-III, however these can be altered to make better use of the CPU's natural word size. For example, on some 32-bit CPUs, it is better to declare boolean variables as 32-bit values for performance considerations, even though an 8-bit quantity is more space efficient (assuming performance is more important than footprint).

The port developer typically makes those decisions, since altering the contents of the file requires a deep understanding of the CPU and, most important, how data sizes affect µC/OS-III.

(3) **µC/OS-III Stacks, Pools and other data sizes (** `os_cfg_app.h` **):**

µC/OS-III can be configured at the application level through `#define` constants in `os_cfg_app.h`. The `#defines` allows a user to specify stack sizes for all µC/OS-III internal tasks: the idle task, statistic task, tick task, timer task, and the ISR handler task. `os_cfg_app.h` also allows users to specify task priorities (except for the idle task since it is always the lowest priority), the tick rate, and more.

The contents of the three configuration files will be described in the following sections.

uC-OS-III Features os_cfg.h

Compile-time configuration allows users to determine which features to enable and those features that are not needed. With compile-time configuration, the code and data sizes of µC/OS-III (i.e., its footprint) can be reduced by enabling only the desired functionality.

Compile-time configuration is accomplished by setting a number of `#define` constants in a file called `os_cfg.h` that the application is expected to provide. You simply copy `os_cfg.h` into the application directory and change the copied file to satisfy the application's requirements. This way, `os_cfg.h` is not recreated from scratch.

The compile-time configuration `#defines` are listed below as per the sections and order found in `os_cfg.h`.

Miscellaneous Options

OS_CFG_APP_HOOKS_EN

When set to `DEF_ENABLED`, this option specifies that application-defined hooks can be called from µC/OS-III's hooks. This allows the application code to extend the functionality of µC/OS-III. Specifically:

The µC/OS-III Hook	Calls the Application-defined Hook
`OSIdleTaskHook()`	`OS_AppIdleTaskHookPtr`
`OSInitHook()`	`None`
`OSStatTaskHook()`	`OS_AppStatTaskHookPtr`
`OSTaskCreateHook()`	`OS_AppTaskCreateHookPtr`
`OSTaskDelHook()`	`OS_AppTaskDelHookPtr`
`OSTaskReturnHook()`	`OS_AppTaskReturnHookPtr`
`OSTaskSwHook()`	`OS_AppTaskSwHookPtr`
`OSTimeTickHook()`	`OS_AppTimeTickHookPtr`

Application hook functions could be declared as shown in the code below.

```
void  App_OS_TaskCreateHook (OS_TCB *p_tcb)
```

```
{
    /* Your code here */
}


void  App_OS_TaskDelHook (OS_TCB *p_tcb)
{
    /* Your code here */
}


void  App_OS_TaskReturnHook (OS_TCB *p_tcb)
{
    /* Your code here */
}


void  App_OS_IdleTaskHook (void)
{
    /* Your code here */
}


void  App_OS_StatTaskHook (void)
{
    /* Your code here */
}


void  App_OS_TaskSwHook (void)
{
    /* Your code here */
}


void  App_OS_TimeTickHook (void)
{
    /* Your code here */
}
```

It's also up to a user to set the value of the pointers so that they point to the appropriate functions as shown below. The pointers do not have to be set in `main()` but, you can set them after calling `OSInit()`.

```
void  main (void)
{
     OS_ERR  err;


     OSInit(&err);
     :
     :
     OS_AppTaskCreateHookPtr = (OS_APP_HOOK_TCB )App_OS_TaskCreateHook;
     OS_AppTaskDelHookPtr    = (OS_APP_HOOK_TCB )App_OS_TaskDelHook;
     OS_AppTaskReturnHookPtr = (OS_APP_HOOK_TCB )App_OS_TaskReturnHook;
     OS_AppIdleTaskHookPtr   = (OS_APP_HOOK_VOID)App_OS_IdleTaskHook;
     OS_AppStatTaskHookPtr   = (OS_APP_HOOK_VOID)App_OS_StatTaskHook;
     OS_AppTaskSwHookPtr     = (OS_APP_HOOK_VOID)App_OS_TaskSwHook;
     OS_AppTimeTickHookPtr   = (OS_APP_HOOK_VOID)App_OS_TimeTickHook;
     :
```

```
    :
    OSStart(&err);
}
```

Note that not every hook function need to be defined, only the ones the user wants to place in the application code.

Also, if you don't intend to extend µC/OS-III's hook through these application hooks, you can set `OS_CFG_APP_HOOKS_EN` to `DEF_DISABLED` to save RAM (i.e., the pointers).

OS_CFG_ARG_CHK_EN

`OS_CFG_ARG_CHK_EN` determines whether the user wants most of µC/OS-III functions to perform argument checking. When set to `DEF_ENABLED`, µC/OS-III ensures that pointers passed to functions are non-`NULL`, that arguments passed are within allowable range, that options are valid, and more. When set to `DEF_DISABLED`, those arguments are not checked and the amount of code space and processing time required by µC/OS-III is reduced. You would set `OS_CFG_ARG_CHK_EN` to `DEF_DISABLED` if you are certain that the arguments will always be correct.

µC/OS-III performs argument checking in close to 50 functions. Therefore, you can save a few hundred bytes of code space by disabling this check. However, you should always enable argument checking until you are certain the code can be trusted.

OS_CFG_CALLED_FROM_ISR_CHK_EN

`OS_CFG_CALLED_FROM_ISR_CHK_EN` determines whether most of µC/OS-III functions are to confirm that the function is not called from an ISR. In other words, most of the functions from µC/OS-III should be called by task-level code except "post" type functions (which can also be called from ISRs). By setting this `DEF_ENABLED`, µC/OS-III is told to make sure that functions that are only supposed to be called by tasks are not called by ISRs. It's highly recommended to set this to `DEF_ENABLED` until you are absolutely sure that the code is behaving correctly and that task-level functions are always called from tasks. You can set this to `DEF_DISABLED` to save code space and, of course, processing time.

µC/OS-III performs this check in approximately 50 functions. Therefore, you can save a few hundred bytes of code space by disabling this check.

OS_CFG_DBG_EN

When set to `DEF_ENABLED`, this configuration adds ROM constants located in `os_dbg.c` to help support kernel aware debuggers. Specifically, a number of named ROM variables can be queried by a debugger to find out about compiled-in options. For example, a debugger can find out the size of a `OS_TCB`, µC/OS-III's version number, the size of an event flag group (`OS_FLAG_GRP`), and much more.

OS_CFG_DYN_TICK_EN

When set to `DEF_ENABLED`, µC/OS-III will use a dynamic ticking mechanism instead of the traditional continuous tick. This allows µC/OS-III to sleep until a task needs to be awakened, instead of waking up every `1/OS_CFG_TICK_RATE_HZ` seconds to find no that no tasks need to be awakened. This can be used to save power since the scheduler is run only when strictly necessary.

Note that the use of this feature requires a proper Board Support Package (BSP) that implements the API described in Board Support Package (BSP).

OS_CFG_INVALID_OS_CALLS_CHK_EN

When set to `DEF_ENABLED`, µC/OS-III will validate the call and check that the kernel is indeed running before performing the function. You would set `OS_CFG_INVALID_OS_CALLS_CHK_EN` to `DEF_DISABLED` if you are sure that the OS functions will be called only once `OSStart()` has been called.

µC/OS-III performs this check in more than 40 functions. Therefore, you can save a few hundred bytes of code space by disabling this check.

OS_CFG_OBJ_TYPE_CHK_EN

OS_CFG_OBJ_TYPE_CHK_EN determines whether most of µC/OS-III functions should check to see if the function is manipulating the proper object. In other words, if attempting to post to a semaphore, is the user in fact passing a semaphore object or another object by mistake? It is recommended to set OS_CFG_OBJ_TYPE_CHK_EN to DEF_ENABLED until absolutely certain that the code is behaving correctly and the user code is always pointing to the proper objects. You would set this to DEF_DISABLED to save code space as well as data space.

µC/OS-III object type checking is done nearly 40 times, and it is possible to save a few hundred bytes of code space and processing time by disabling this check.

OS_CFG_TS_EN

When OS_CFG_TS_EN is set to DEF_ENABLED, it enables the timestamp facilities provided by µC/CPU. This allows the user and the kernel to measure the time between various events. For example, the time spent by a task pending on an object, the maximum interrupt disable time (if CPU_CFG_INT_DIS_MEAS_EN is set to DEF_ENABLED), the time the scheduler is locked, etc. This option is mostly useful in profiling and performance measurement contexts. To save space and processing time, set this option to DEF_DISABLED .

Note that to use the timestamp facilities the µC/CPU Board Support Package should implement the functions described in cpu_bsp.c and cpu_bsp.h.

OS_CFG_PRIO_MAX

OS_CFG_PRIO_MAX specifies the maximum number of priorities available in the application. Specifying OS_CFG_PRIO_MAX to just the number of priorities the user intends to use, reduces the amount of RAM needed by µC/OS-III.

In µC/OS-III, task priorities can range from 0 (highest priority) to a maximum of 255 (lowest possible priority) when the data type OS_PRIO is defined as a CPU_INT08U. However, in µC/OS-III, there is no practical limit to the number of available priorities. Specifically, if defining OS_PRIO as a CPU_INT16U, there can be up to 65536 priority levels. It is recommended to leave OS_PRIO defined as a CPU_INT08U and use only 256 different priority levels (i.e., 0..255), which is generally sufficient for every application. You should always set the value of `OS_CFG_PRIO_MAX` to even multiples of 8 (`8`, `16`, `32`, `64`, `128`, `256`, etc.). The higher the number of different priorities, the more RAM µC/OS-III will consume.

An application cannot create tasks with a priority number higher than or equal to `OS_CFG_PRIO_MAX`. In fact, µC/OS-III reserves priority `OS_CFG_PRIO_MAX-2` and `OS_CFG_PRIO_MAX-1` for itself; `OS_CFG_PRIO_MAX-1` is reserved for the Idle Task `OS_IdleTask()`, if used. Additionally, do not use priority 0 for an application since it is reserved by µC/OS-III's ISR handler task. The priorities of the application tasks can therefore take a value between 2 and `OS_CFG_PRIO_MAX–3` (inclusive).

To ensure proper operation of µC/OS-III and it's services, care should be taken when setting the priorities of other system task such as the Tick Task, the Statistics Task and the Timer Task in `os_cfg_app.h`.

To summarize, there are two priority levels to avoid in an application:

Priority	Reserved by µC/OS-III for
0	The ISR Handler Task (`OS_IntQTask()`), if used
1	Reserved
`OS_CFG_PRIO_MAX-2`	Reserved
`OS_CFG_PRIO_MAX-1`	The Idle Task (`OS_IdleTask()`), if used

OS_CFG_SCHED_LOCK_TIME_MEAS_EN

When set to `DEF_ENABLED` , `OS_CFG_SCHED_LOCK_TIME_MEAS_EN` allows µC/OS-III to use the timestamp facilities (provided `OS_CFG_TS_EN` is also set to `DEF_ENABLED`) to measure the peak amount of time that the scheduler is locked. Use this feature to profile the application, the deployed application should set this to `DEF_DISABLED` .

OS_CFG_SCHED_ROUND_ROBIN_EN

Set `OS_CFG_SCHED_ROUND_ROBIN_EN` to `DEF_ENABLED` to use the Round Robin Scheduler. This is only useful when there is multiple tasks sharing the same priority, if this is not your case, set this option to `DEF_DISABLED` . See Round-Robin Scheduling for more information.

OS_CFG_STK_SIZE_MIN

`OS_CFG_STK_SIZE_MIN` specifies the minimum stack size (in `CPU_STK` elements) for each task. This is used by µC/OS-III to verify that sufficient stack space is provided for each task when the task is created. Suppose the full context of a processor consists of 16 registers of 32 bits. Also, suppose `CPU_STK` is declared as being of type `CPU_INT32U`, at a bare minimum, `OS_CFG_STK_SIZE_MIN` should be set to 16. However, it would be quite unwise to not accommodate for storage of local variables, function call returns, and possibly nested ISRs. Refer to the “port” of the processor used to see how to set this minimum. Again, this is a safeguard to make sure task stacks have sufficient space.

Event Flag Configuration

OS_CFG_FLAG_EN

When `OS_CFG_FLAG_EN` is set to `DEF_ENABLED`, it enables the event flag services and data structures. If event flags are not needed, set this to `DEF_DISABLED`. It reduces the amount of code and data space needed by µC/OS-III. Note than when `OS_CFG_FLAG_EN` is set to `DEF_DISABLED` , it is not necessary to enable or disable any of the other `OS_CFG_FLAG_xxx` options in this section.

OS_CFG_FLAG_DEL_EN

If your application needs to delete event flags with `OSFlagDel()` once they're created, set `OS_CFG_FLAG_DEL_EN` to `DEF_ENABLED`, if not, set this option to `DEF_DISABLED`. Critical applications should not delete kernel objects once the kernel is started.

OS_CFG_FLAG_MODE_CLR_EN

If your application requires to wait until a event is cleared, set `OS_CFG_FLAG_MODE_CLR_EN` to `DEF_ENABLED`, if not set this to `DEF_DISABLED`. Generally, you would wait for event flags to be set. However, the user may also want to wait for event flags to be clear and in this case, enable this option.

OS_CFG_FLAG_PEND_ABORT_EN

When OS_CFG_FLAG_PEND_ABORT_EN is set to `DEF_ENABLED`, it enables the generation of the function `OSFlagPendAbort()`. If your application does not require fault-aborts on event flags, set this option to `DEF_DISABLED`.

Memory Management Configuration

OS_CFG_MEM_EN

When `OS_CFG_MEM_EN` is set to `DEF_ENABLED`, it enables the µC/OS-III partition memory manager. If your application does not require the partitionned memory manager, set this to `DEF_DISABLE` to reduce µC/OS-III's code and data space usage.

Mutal Exclusion Semaphore Configuration

OS_CFG_MUTEX_EN

When `OS_CFG_MUTEX_EN` is set to `DEF_ENABLED`, it enables the mutual exclusion semaphore services and data structures. If your application does not require mutexes, set this option to `DEF_DISABLED` to reduce the amount of code and data space needed by µC/OS-III. When `OS_CFG_MUTEX_EN` is set to `DEF_DISABLED`, there is no need to enable or disable any of the other `OS_CFG_MUTEX_XXX` options in this section.

OS_CFG_MUTEX_DEL_EN

If your application needs to delete mutexes with `OSMutexDel ()` once they're created, set `OS_CFG_MUTEX_DEL_EN` to `DEF_ENABLED`, if not, set this option to `DEF_DISABLED`. Critical applications should not delete kernel objects once the kernel is started.

OS_CFG_MUTEX_PEND_ABORT_EN

When `OS_CFG_MUTEX_PEND_ABORT_EN` is set to `DEF_ENABLED`, it enables the generation of the function `OSMutexPendAbort()`. If your application does not require fault-aborts on mutexes, set this option to `DEF_DISABLED`.

Message Queue Configuration

OS_CFG_Q_EN

When `OS_CFG_Q_EN` is set to `DEF_ENABLED`, it enables the message queue services and data structures. If your application does not require mutexes, set this option to `DEF_DISABLED` to reduce the amount of code and data space needed by µC/OS-III. When `OS_CFG_Q_EN` is set to `DEF_DISABLED`, there is no need to enable or disable any of the other `OS_CFG_Q_XXX` options in this section.

OS_CFG_Q_DEL_EN

If your application needs to delete message queues with `OSQDel()` once they're created, set `OS_CFG_Q_DEL_EN` to `DEF_ENABLED`, if not, set this option to `DEF_DISABLED`. Critical applications should not delete kernel objects once the kernel is started.

OS_CFG_Q_FLUSH_EN

When `OS_CFG_Q_FLUSH_EN` is set to `DEF_ENABLED`, it allows your application to flush, or clear, a message queue with `OSQFlush()`. If this feature is not needed, set this option to `DEF_DISABLED`.

OS_CFG_Q_PEND_ABORT_EN

When `OS_CFG_Q_PEND_ABORT_EN` is set to `DEF_ENABLED`, it enables the generation of the function `OSQPendAbort()`. If your application does not require fault-aborts on message queues, set this option to `DEF_DISABLED`.

Semaphore Configuration

OS_CFG_SEM_EN

When `OS_CFG_SEM_EN` is set to `DEF_ENABLED`, it enables the semaphore services and data structures. If your application does not require semaphores, set this option to `DEF_DISABLED` to reduce the amount of code and data space needed by μC/OS-III. When `OS_CFG_SEM_EN` is set to `DEF_DISABLED`, there is no need to enable or disable any of the other `OS_CFG_SEM_XXX` options in this section.

OS_CFG_SEM_DEL_EN

If your application needs to delete semaphores with `OSSemDel ()` once they're created, set `OS_CFG_SEM_DEL_EN` to `DEF_ENABLED`, if not, set this option to `DEF_DISABLED`. Critical applications should not delete kernel objects once the kernel is started.

OS_CFG_SEM_PEND_ABORT_EN

When `OS_CFG_SEM_PEND_ABORT_EN` is set to `DEF_ENABLED`, it enables the generation of the function `OSSemPendAbort()`. If your application does not require fault-aborts on semaphores queues, set this option to `DEF_DISABLED`.

OS_CFG_SEM_SET_EN

If your application needs to explicitly set the value of a semaphore with `OSSemSet()` at another time than it's creation, set `OS_CFG_SEM_SET_EN` to `DEF_ENABLE`, if not, set this option to `DEF_DISABLED`.

Monitor Configuration

OS_CFG_MON_EN

When `OS_CFG_MON_EN` is set to `DEF_ENABLED`, it enables the monitor services and data structures. If your application does not require monitors, set this option to `DEF_DISABLED` to reduce the amount of code and data space needed by μC/OS-III.

OS_CFG_MON_DEL_EN

If your application needs to delete a monitor with `OSMonDel()` once they're created, set `OS_CFG_MON_DEL_EN` to `DEF_ENABLED`, if not, set this option to `DEF_DISABLED`. Critical applications should not delete kernel objects once the kernel is started.

Task Management Options

OS_CFG_STAT_TASK_EN

`OS_CFG_STAT_TASK_EN` specifies whether or not to enable µC/OS-III's statistic task, as well as its initialization function. When set to `DEF_ENABLED` , the statistic task `OS_StatTask()` and the statistic task initialization function are enabled. `OS_StatTask()` computes the CPU usage of an application, the stack usage of each task, the CPU usage of each task at run time and more.

When enabled, `OS_StatTask()` executes at a rate of `OS_CFG_STAT_TASK_RATE_HZ` (see `os_cfg_app.h`), and computes the value of `OSStatTaskCPUUsage`, which is a variable that contains the percentage of CPU used by the application. `OS_StatTask()` calls `OSStatTaskHook()` every time it executes so that the user can add his own statistics as needed. See `os_stat.c` for details on the statistic task. The priority of `OS_StatTask()` is configurable by the application code (see `os_cfg_app.h`).

`OS_StatTask()` computes stack usage of each task created when the option `OS_CFG_STAT_TASK_STK_CHK_EN` is set to `DEF_ENABLED` . In this case, `OS_StatTask()` calls `OSTaskStkChk()` for each task and the result is placed in the task's TCB. The `.StkFree` and `.StkUsed` fields of the task's TCB represent the amount of free space (in `CPU_STK` elements) and amount of used space (in `CPU_STK` elements), respectively.

When `OS_CFG_STAT_TASK_EN` is set to `DEF_DISABLED`, all variables used by the statistic task are not declared (see `os.h`). This, of course, reduces the amount of RAM needed by µC/OS-III when not enabling the statistic task.

OS_CFG_STAT_TASK_STK_CHK_EN

When set to `DEF_ENABLED`, this option allows the statistic task to call `OSTaskStkChk()` for each task created. Note that for this to happen, `OS_CFG_STAT_TASK_EN` must also be set to `DEF_ENABLED` . However, you can call `OSStatStkChk()` from one of the tasks to obtain this information about the tasks.

OS_CFG_TASK_CHANGE_PRIO_EN

If your application needs to dynamically change a task's priority using `OSTaskChangePrio()` , set `OS_CFG_TASK_CHANGE_PRIO_EN` to `DEF_ENABLED`. If not, set this option to `DEF_DISABLED`. Note that the new priority has to be available and not currently in-use by a kernel task.

OS_CFG_TASK_DEL_EN

If your application needs to delete tasks using `OSTaskDel()`, set `OS_CFG_TASK_DEL_EN` to `DEF_ENABLED`. If not, set this option to `DEF_DISABLED`. Note that critical applications should not delete tasks once the kernel is started.

OS_CFG_TASK_IDLE_EN

Setting `OS_CFG_TASK_IDLE_EN` to `DEF_ENABLED` allows µC/OS-III to create it's Idle Task at priority `OS_CFG_PRIO_MAX-1`. However, to save data space, it is possible to remove the Idle Task. To do so, set this option to `DEF_DISABLED`. Doing so will move the functionality of the Idle Task within the `OSSched()` function. The same counters will be incremented and the same hooks will be called under the same circumstances.

OS_CFG_TASK_PROFILE_EN

To enable the performance profiling tools within µC/OS-III, set `OS_CFG_TASK_PROFILE_EN` to `DEF_ENABLED`. Doing so allows variables to be allocated in each task's `OS_TCB` to hold performance data about each task. When enabled, each task will have variables to keep track of the number of times a task is switched in, the task execution time, the CPU usage percentage of the task relative to the other tasks and more. The information made available with this feature is highly useful when debugging, but requires extra RAM. To save data and code space, set this option to `DEF_DISABLED` after your are certain that you application is profiled and works correctly.

OS_CFG_TASK_Q_EN

When `OS_CFG_TASK_Q_EN` is set to `DEF_ENABLED`, it allows the generation of the `OSTaskQ???()` functions used to send and receive messages directly to and from tasks and ISRs. Sending messages directly to a task is more efficient than sending messages using a traditional message queue because there is no pend list associated with messages sent to a task. If your application does not require task-level message queues, set this option to `DEF_DISABLED`. Note that if this option is set to `DEF_DISABLED` , the `OS_CFG_TASK_Q_PEND_ABORT_EN` configuration option is ignored.

OS_CFG_TASK_Q_PEND_ABORT_EN

When `OS_CFG_TASK_Q_PEND_ABORT_EN` is set to `DEF_ENABLED`, it enables the generation of the function `OSTaskQPendAbort()`. If your application does not require fault-aborts on task-level message queues, set this option to `DEF_DISABLED`.

OS_CFG_TASK_REG_TBL_SIZE

This constant allows each task to have task context variables. Use task variables to store such elements as “`errno`”, task identifiers and other task-specific values. The number of variables that a task contains is set by `OS_CFG_TASK_REG_TBL_SIZE`. Each variable is identified by a unique identifier from `0` to `OS_CFG_TASK_REG_TBL_SIZE-1`. Also, each variable is declared as having an `OS_REG` data type (see `os_type.h`). If `OS_REG` is a `CPU_INT32U`, all variables in this table are of this type. To disable the usage of task context variables, set this option to `0u`.

OS_CFG_TASK_STK_REDZONE_EN

While debugging, it is useful to determine if a task overflowed it's stack space. To do so, set `OS_CFG_TASK_STK_REDZONE_EN` to `DEF_ENABLED`. Then, every time a task is switched in after an interrupt, it's stack is checked. If the monitored zone located at the end of a task's stack is corrupted, a software exception is thrown. To disable this feature, set this option to `DEF_DISABLED`. Note that the effectively usable stack space is the task stack size minus `OS_CFG_TASK_STK_REDZONE_DEPTH`.

OS_CFG_TASK_STK_REDZONE_DEPTH

The default monitored zone, located at the end of a task's stack, is `8` `CPU_STK` elements long. To change the size of the monitored zone, change this option accordingly. If `OS_CFG_TASK_STK_REDZONE_EN` is set to `DEF_DISABLED`, this value is ignored.

OS_CFG_TASK_SEM_PEND_ABORT_EN

When `OS_CFG_TASK_SEM_PEND_ABORT_EN` is set to `DEF_ENABLED`, it enables the generation of the function `OSTaskSemPendAbort()`. If your application does not require fault-aborts on task-level semaphores, set this option to `DEF_DISABLED`.

OS_CFG_TASK_SUSPEND_EN

If your application requires the ability to explicitly suspend and resume the execution of tasks, set `OS_CFG_TASK_SUSPEND_EN` to `DEF_ENABLED`. Doing so, allows the generation of the `OSTaskSuspend()` and `OSTaskResume()` functions used to suspend and resume tasks, respectively. Note that other effects are additive with the suspension. For example, if a suspended task is pending on a semaphore that becomes available, the task will not run until it's explicitly resumed with `OSTaskResume()`. Also, the suspension of a task can be nested. To resume a task, you must call `OSTaskResume()` the same number of times `OSTaskSuspend()` was called. If your application does not require this feature, set this option to `DEF_DISABLED`.

OS_CFG_TASK_TICK_EN

To keep the traditional behavior, set `OS_CFG_TASK_TICK_EN` to `DEF_ENABLED`. If your application does not require any form of timeouts or time keeping, either with timeouts on kernel objects or delayed execution times, you may set `OS_CFG_TASK_TICK_EN` to `DEF_DISABLED`. Doing so, removes all time keeping facilities from µC/OS-III. Removing the Tisk Task from µC/OS-III allows the user to save code and data space. However, the users looses the ability to use timeouts and delays.

Task Local Storage Configuration

OS_CFG_TLS_TBL_SIZE

If your application requires task local storage, set `OS_CFG_TLS_TBL_SIZE` to a non-null value. This value will determine the size of the Task Local Storage Table (`TLS_Tbl`, member of `OS_TCB`) present in each task. To disable TLS, set this option to `0u`.

Time Management Options

OS_CFG_TIME_DLY_HMSM_EN

If your application requires the ability to delay a task for a specified number of hours, minutes, seconds and milliseconds, set `OS_CFG_TIME_DLY_HMSM_EN` to `DEF_ENABLED`. This will allow the generation of the `OSTimeDlyHMSM()` function. Otherwise, set this option to `DEF_DISABLED`.

OS_CFG_TIME_DLY_RESUME_EN

When `OS_CFG_TIME_DLY_RESUME_EN` is set to `DEF_ENABLED`, it allows applications to resume a previously delayed task, using the function `OSTimeDlyResume()`, without waiting for the entire delay. If you do not require this feature, set this option to `DEF_DISABLED`.

Timer Management Options

OS_CFG_TMR_EN

When OS_CFG_TMR_EN is set to DEF_ENABLED, it enables the timer management services. If your application does no require programmable timers, set this option to DEF_DISABLED to reduce µC/OS-III's required code and data space.

OS_CFG_TMR_DEL_EN

If your application needs to delete timers with OSTmrDel() once they're created, set OS_CFG_TMR_DEL_EN to DEF_ENABLED, if not, set this option to DEF_DISABLED. Critical applications should not delete kernel objects once the kernel is started.

Trace Options

OS_CFG_TRACE_EN

µC/OS-III has built-in trace points throughout the code to record all the kernel events and interrupts in real-time. These trace calls are disabled by default and enabled when this constant is set to DEF_ENABLED. If you do not require this feature, set this option to DEF_DISABLED to not only reduce the required code and data space but also to eliminate the inherent overhead.

OS_CFG_TRACE_API_ENTER_EN

When this constant is set to DEF_ENABLED the beginning of each API call in µC/OS-III will be recorded as part of the trace (it applies to SEGGER's SystemView only).

OS_CFG_TRACE_API_EXIT_EN

When this constant is set to DEF_ENABLED the end of each API call in µC/OS-III will be recorded as part of the trace (it applies to SEGGER's SystemView only).

Data Types os_type.h

`os_type.h` contains the data types used by µC/OS-III, which should only be altered by the implementer of the µC/OS-III port. You can alter the contents of `os_type.h`. However, it is important to understand how each of the data types that are being changed will affect the operation of µC/OS-III-based applications.

The reason to change `os_type.h` is that processors may work better with specific word sizes. For example, a 16-bit processor will likely be more efficient at manipulating 16-bit values and a 32-bit processor more comfortable with 32-bit values, even at the cost of extra RAM. In other words, the user may need to choose between processor performance and RAM footprint.

If changing “any” of the data types, you should copy `os_type.h` in the project directory and change that file (not the original `os_type.h` that comes with the µC/OS-III release).

Recommended data type sizes are specified in comments in `os_type.h`.

uC-OS-III Stacks Pools and Other os_cfg_app.h

µC/OS-III allows the user to configure the sizes of the idle task stack, statistic task stack, message pool, debug tables, and more. This is done through `os_cfg_app.h`.

Miscellaneous

OS_CFG_ISR_STK_SIZE

This parameter specifies the size of µC/OS-III's interrupt stack (in `CPU_STK` elements). Note that the stack size needs to accommodate for worst case interrupt nesting, assuming the processor supports interrupt nesting. The ISR handler task stack is declared in `os_cfg_app.c` as follows:

```
CPU_STK  OSCfg_ISRStk[OS_CFG_ISR_STK_SIZE];
```

OS_CFG_MSG_POOL_SIZE

This entry specifies the number of `OS_MSG`s available in the pool of `OS_MSG`s. The size is specified in number of `OS_MSG` elements and the message pool is declared in `os_cfg_app.c` as follows:

```
OS_MSG  OSCfg_MsgPool[OS_CFG_MSG_POOL_SIZE];
```

OS_CFG_TASK_STK_LIMIT_PCT_EMPTY

This parameter sets the position (as a percentage of empty) of the stack limit for the idle, statistic, tick, interrupt queue handler, and timer tasks stacks. In other words, the amount of space to leave before the stack is full. For example if the stack contains 1000 `CPU_STK` entries and the user declares `OS_CFG_TASK_STK_LIMIT_PCT_EMPTY` to `10u`, the stack limit will be set when the stack reaches 90% full, or 10% empty.

If the stack of the processor grows from high memory to low memory, the limit would be set towards the "base address" of the stack, i.e., closer to element 0 of the stack.

If the processor used does not offer automatic stack limit checking, you should set `OS_CFG_TASK_STK_LIMIT_PCT_EMPTY` to `0u`.

Idle Task Configuration

OS_CFG_IDLE_TASK_STK_SIZE

This parameter sets the size of the idle task's stack (in `CPU_STK` elements) as follows:

```
CPU_STK  OSCfg_IdleTaskStk[OS_CFG_IDLE_TASK_STK_SIZE];
```

Note that `OS_CFG_IDLE_TASK_STK_SIZE` must be at least greater than `OS_CFG_STK_SIZE_MIN`.

Statistic Task Configuration

OS_CFG_STAT_TASK_PRIO

This parameter allows the user to specify the priority assigned to µC/OS-III's statistic task. It is recommended to make this task a very low priority such as one priority level above the idle task, or, `OS_CFG_PRIO_MAX-2`.

OS_CFG_STAT_TASK_RATE_HZ

This option defines the execution rate (in Hz) of µC/OS-III's statistic task. It is recommended to make this rate an even multiple of the tick rate (see `OS_CFG_TICK_RATE_HZ`).

OS_CFG_STAT_TASK_STK_SIZE

This parameter sets the size of the statistic task's stack (in `CPU_STK` elements). The statistic task stack is declared in `os_cfg_app.c` as follows:

```
CPU_STK  OSCfg_StatTaskStk[OS_CFG_STAT_TASK_STK_SIZE];
```

Note that `OS_CFG_STAT_TASK_STK_SIZE` must be at least greater than `OS_CFG_STK_SIZE_MIN`.

Tick Rate and Task Configuration

OS_CFG_TICK_RATE_HZ

This parameter defines the execution rate (in Hz) of µC/OS-III’s tick task. The tick rate should be set between 10 and 1000 Hz. The higher the rate, the more overhead it will impose on the processor. The desired rate depends on the granularity required for time delays and timeouts.

OS_CFG_TICK_TASK_PRIO

This option specifies the priority to assign to the µC/OS-III's tick task. It is recommended to make this task a fairly high priority, but not necessarily the highest. The priority assigned to this task must be greater than 0 and less than `OS_CFG_PRIO_MAX-1`.

OS_CFG_TICK_TASK_STK_SIZE

This parameter specifies the size of µC/OS-III’s tick task stack (in `CPU_STK` elements). The tick task stack is declared in `os_cfg_app.c` as follows:

```
CPU_STK  OSCfg_TickTaskStk[OS_CFG_TICK_TASK_STK_SIZE];
```

Note that `OS_CFG_TICK_TASK_STK_SIZE` must be at least greater than `OS_CFG_STK_SIZE_MIN`.

Timer Task Configuration

OS_CFG_TMR_TASK_PRIO

This parameter allows the user to specify the priority to assign to µC/OS-III's timer task. It is recommended to make this task a medium-to-low priority, depending on how fast the timer task will execute (see `OS_CFG_TMR_TASK_RATE_HZ`) and how many timers are running in the application. The priority assigned to this task must be greater than 0 and less than `OS_CFG_PRIO_MAX-1`.

You should start with these simple rules:

1. The faster the timer rate, the higher the priority to assign to this task.

2. The higher the number of timers in the system, the lower the priority.

In other words:

High Timer Rate — Higher Priority

High Number of Timers — Lower Priority

OS_CFG_TMR_TASK_RATE_HZ

This option defines the execution rate (in Hz) of µC/OS-III’s timer task. The timer task rate should typically be set to 10 Hz. However, timers can run at a faster rate at the price of higher processor overhead. Note that `OS_CFG_TMR_TASK_RATE_HZ` must be an integer multiple of `OS_CFG_TICK_TASK_RATE_HZ`.

OS_CFG_TMR_TASK_STK_SIZE

This parameter sets the size of the timer task’s stack (in `CPU_STK` elements). The timer task stack is declared in `os_cfg_app.c` as follows:

```
CPU_STK  OSCfg_TmrTaskStk[OS_CFG_TMR_TASK_STK_SIZE];
```

Note that `OS_CFG_TMR_TASK_STK_SIZE` must be at least greater than `OS_CFG_STK_SIZE_MIN`.

Migrating from uC-OS-II to uC-OS-III

µC/OS-III is a completely new real-time kernel with roots in µC/OS-II. Portions of the µC/OS-II Application Programming Interface (API) function names are the same, but the arguments passed to the functions have, in some places, drastically changed.

This section explains several differences between the two real-time kernels. However, access to µC/OS-II and µC/OS-III source files best highlights the differences.

The table below is a feature-comparison chart for µC/OS-II and µC/OS-III.

Feature	µC/OS-II	µC/OS-III
Year of introduction	1998	2009
Book	Yes	Yes
Source code available	Yes	Yes
Preemptive Multitasking	Yes	Yes
Maximum number of tasks	255	Unlimited
Number of tasks at each priority level	1	Unlimited
Round Robin Scheduling	No	Yes
Semaphores	Yes	Yes
Mutual Exclusion Semaphores	Yes	Yes (nestable)
Event Flags	Yes	Yes
Message Mailboxes	Yes	No (not needed)
Message Queues	Yes	Yes
Fixed Sized Memory Management	Yes	Yes
Signal a task without requiring a semaphore	No	Yes
Send messages to a task without requiring a message queue	No	Yes
Software Timers	Yes	Yes
Task suspend/resume	Yes	Yes (nestable)
Deadlock prevention	Yes	Yes
Scalable	Yes	Yes
Code Footprint	6K to 26K	6K to 24K
Data Footprint	1K+	1K+
ROMable	Yes	Yes
Run-time configurable	No	Yes
Catch a task that returns	No	Yes
Compile-time configurable	Yes	Yes
ASCII names for each kernel object	Yes	Yes
Optio to post without scheduling	No	Yes
Pend on multiple objects	Yes	Yes
Task registers	Yes	Yes
Built-in performance measurements	Limited	Extensive

User definable hook functions	Yes	Yes
Time stamps on posts	No	Yes
Built-in Kernel Awareness support	Yes	Yes
Optimizable Scheduler in assembly language	No	Yes
Tick handling at task level	No	Yes
Number of services	~90	~70
MISRA-C:1998	Yes	N/A
MISRA-C:2004	No	Yes
DO178B Level A and EUROCAE ED-12B	Yes	Yes
Medical FDA pre-market notification (510(k)) and pre-market approval (PMA) and IEC62304	Yes	Yes
SIL3 IEC for transportation and nuclear systems	Yes	Yes
IEC-61508	Yes	Yes

Table - μC/OS-II and μC/OS-III features comparison chart

Differences in Source File Names and Contents

he table below shows the source files used in both kernels. Note that a few of the files have the same or similar name.

µC/OS-II	µC/OS-III	Note
	os_app_hooks.c	(1)
	os_cfg_app.c	(2)
	os_cfg_app.h	(3)
os_cfg_r.h	os_cfg.h	(4)
os_core.c	os_core.c	
os_cpu.h	os_cpu.h	(5)
os_cpu_a.asm	os_cpu_a.asm	(5)
os_cpu_c.c	os_cpu_c.c	(5)
os_dbg_r.c	os_dbg.c	(6)
os_flag.c	os_flag.c	
	os_prio.c	(7)
os_mbox.c		(8)
os_mem.c	os_mem.c	
	os_msg.c	(9)
os_mutex.c	os_mutex.c	
os_q.c	os_q.c	
os_sem.c	os_sem.c	
	os_stat.c	(10)
os_task.c	os_task.c	
os_time.c	os_time.c	
os_tmr.c	os_tmr.c	
	os_var.c	(11)
	os_type.h	(12)
ucos_ii.h	os.h	(13)

Table - µC/OS-II and µC/OS-III files

(1) µC/OS-II does not have this file, which is now provided for convenience so you can add application hooks. You should copy this file to the application directory and edit the contents of the file to satisfy your application requirements.

(2) `os_cfg_app.c` did not exist in µC/OS-II. This file needs to be added to a project build for µC/OS-III.

(3) In µC/OS-II, all configuration constants were placed in `os_cfg.h`. In µC/OS-III, some of the configuration constants are placed in this file, while others are in `os_cfg_app.h`. `os_cfg_app.h` contains application-specific configurations such as the size of the idle task stack, tick rate, and others.

(4) In µC/OS-III, `os_cfg.h` is reserved for configuring certain features of the kernel. For example, are any of the semaphore services required, and will the application have fixed-sized memory partition management?

(5) These are the port files and a few variables and functions will need to be changed when using a µC/OS-II port as a starting point for the µC/OS-III port.

µC/OS-II variable changes from …	… to these in µC/OS-III
`OSIntNesting`	`OSIntNestingCtr`
`OSTCBCur`	`OSTCBCurPtr`
`OSTCBHighRdy`	`OSTCBHighRdyPtr`

µC/OS-II function changes from …	… to these in µC/OS-III
`OSInitHookBegin()`	`OSInitHook()`
`OSInitHookEnd()`	N/A
`OSTaskStatHook()`	`OSStatTaskHook()`
`OSTaskIdleHook()`	`OSIdleTaskHook()`
`OSTCBInitHook()`	N/A
`OSTaskStkInit()`	`OSTaskStkInit()`

The name of `OSTaskStkInit()` is the same but it is listed here since the code for it needs to be changed slightly as several arguments passed to this function are different. Specifically, instead of passing the top-of-stack as in µC/OS-II, `OSTaskStkInit()` is passed the base address and the size of the task stack.

(6) In µC/OS-III, `os_dbg.c` should always be part of the build. In µC/OS-II, the equivalent file (os_dbg_r.c) was optional.

(7) The code to determine the highest priority ready-to-run task is isolated in µC/OS-III and placed in `os_prio.c`. This allows the port developer to replace this file by an assembly language equivalent file, especially if the CPU used supports certain bit manipulation instructions and a count leading zeros (CLZ) instruction.

(8) µC/OS-II provides message mailbox services. A message mailbox is identical to a message queue of size one. µC/OS-III does not have these services since they can be easily emulated by message queues.

(9) Management of messages for message queues is encapsulated in `os_msg.c` in µC/OS-III.

(10) The statistics task and its support functions have been extracted out of `os_core.c` and placed in `os_stat.c` for µC/OS-III.

(11) All the µC/OS-III variables are instantiated in a file called `os_var.c`.

(12) In µC/OS-III, the size of most data types is better adapted to the CPU architecture used. In µC/OS-II, the size of a number of these data types was assumed.

(13) In µC/OS-II, the main header file is called ucos_ii.h. In µC/OS-III, it is renamed to `os.h`.

Convention Changes

There are a number of convention changes from µC/OS-II to µC/OS-III. The most notable is the use of CPU-specific data types. The table below shows the differences between the data types used in both kernels.

µC/OS-II (os_cpu.h)	µC/CPU (cpu.h)	Note
BOOLEAN	CPU_BOOLEAN	
INT8S	CPU_INT08S	
INT8U	CPU_INT08U	
INT16S	CPU_INT16S	
INT16U	CPU_INT16U	
INT32S	CPU_INT32S	
INT32U	CPU_INT32U	
OS_STK	CPU_STK	(1)
OS_CPU_SR µC/OS-II (os_cfg.h)	CPU_SR µC/CPU (cpu.h)	(2)
OS_STK_GROWTH	CPU_CFG_STK_GROWTH	(3)

Table - µC/OS-II vs. µC/OS-III basic data types

(1) A task stack in µC/OS-II is declared as an `OS_STK` , which is now replaced by a CPU specific data type `CPU_STK` . These two data types are equivalent, except that defining the width of the CPU stack in µC/CPU makes more sense.

(2) It also makes sense to declare the CPU's status register in µC/CPU.

(3) Stack growth (high-to-low or low-to-high memory) is declared in µC/CPU since stack growth is a CPU feature and not an OS one.

Another convention change is the use of the acronym " `CFG` " which stands for configuration. Now, all `#define` configuration constants and variables have the " `CFG` " or " `Cfg` " acronym in them as shown in the table below. This table shows the configuration constants that have been moved from `os_cfg.h` to `os_cfg_app.h` . This is done because µC/OS-III is configurable at the application level instead of just at compile time as with µC/OS-II.

µC/OS-II (os_cfg.h)	µC/OS-III (os_cfg_app.h)	Note
	OS_CFG_MSG_POOL_SIZE	
	OS_CFG_ISR_STK_SIZE	
	OS_CFG_TASK_STK_LIMIT_PCT_EMPTY	
OS_TASK_IDLE_STK_SIZE	OS_CFG_IDLE_TASK_STK_SIZE	
	OS_CFG_STAT_TASK_PRIO	
	OS_CFG_STAT_TASK_RATE_HZ	
OS_TASK_STAT_STK_SIZE	OS_CFG_STAT_TASK_STK_SIZE	
OS_TICKS_PER_SEC	OS_CFG_TICK_RATE_HZ	(1)
	OS_CFG_TICK_TASK_PRIO	
	OS_CFG_TICK_TASK_STK_SIZE	
	OS_CFG_TMR_TASK_PRIO	
OS_TMR_CFG_TICKS_PER_SEC	OS_CFG_TMR_TASK_RATE_HZ	
OS_TASK_TMR_STK_SIZE	OS_CFG_TMR_TASK_STK_SIZE	

Table - µC/OS-III uses “CFG” in configuration

(1) The very useful `OS_TICKS_PER_SEC` in µC/OS-II was renamed to `OS_CFG_TICK_RATE_HZ` in µC/OS-III. The “ `HZ` ” indicates that this `#define` represents Hertz (i.e., ticks per second).

The table belowshows additional configuration constants added to `os_cfg.h` , while several µC/OS-II constants were either removed or renamed.

µC/OS-II (os_cfg.h)	µC/OS-III (os_cfg.h)	Note
OS_APP_HOOKS_EN	OS_CFG_APP_HOOKS_EN	
OS_ARG_CHK_EN	OS_CFG_ARG_CHK_EN	
	OS_CFG_CALLED_FROM_ISR_CHK_EN	
OS_DEBUG_EN	OS_CFG_DBG_EN	(1)
OS_EVENT_MULTI_EN		
OS_EVENT_NAME_EN		(2)
OS_MAX_EVENTS		(3)
OS_MAX_FLAGS		(3)
OS_MAX_MEM_PART		(3)
OS_MAX_QS		(3)
OS_MAX_TASKS		(3)
	OS_CFG_OBJ_TYPE_CHK_EN	
OS_LOWEST_PRIO	OS_CFG_PRIO_MAX	
	OS_CFG_SCHED_LOCK_TIME_MEAS_EN	
	OS_CFG_SCHED_ROUND_ROBIN_EN	
	OS_CFG_STK_SIZE_MIN	
OS_FLAG_EN	OS_CFG_FLAG_EN	
OS_FLAG_ACCEPT_EN		(6)
OS_FLAG_DEL_EN	OS_CFG_FLAG_DEL_EN	
OS_FLAG_WAIT_CLR_EN	OS_CFG_FLAG_MODE_CLR_EN	
OS_FLAG_NAME_EN		(2)
OS_FLAG_NBITS		(4)
OS_FLAG_QUERY_EN		(5)
	OS_CFG_PEND_ABORT_EN	
OS_MBOX_EN		
OS_MBOX_ACCEPT_EN		(6)
OS_MBOX_DEL_EN		
OS_MBOX_PEND_ABORT_EN		
OS_MBOX_POST_EN		
OS_MBOX_POST_OPT_EN		

OS_MBOX_QUERY_EN		(5)
OS_MEM_EN	OS_CFG_MEM_EN	
OS_MEM_NAME_EN		(2)
OS_MEM_QUERY_EN		(5)
OS_MUTEX_EN	OS_CFG_MUTEX_EN	
OS_MUTEX_ACCEPT_EN		(6)
OS_MUTEX_DEL_EN	OS_CFG_MUTEX_DEL_EN	
	OS_CFG_MUTEX_PEND_ABORT_EN	
OS_MUTEX_QUERY_EN		(5)
OS_Q_EN	OS_CFG_Q_EN	
OS_Q_ACCEPT_EN		(6)
OS_Q_DEL_EN	OS_CFG_Q_DEL_EN	
OS_Q_FLUSH_EN	OS_CFG_Q_FLUSH_EN	
	OS_CFG_Q_PEND_ABORT_EN	
OS_Q_POST_EN		(7)
OS_Q_POST_FRONT_EN		(7)
OS_Q_POST_OPT_EN		(7)
OS_Q_QUERY_EN		(5)
OS_SCHED_LOCK_EN		
OS_SEM_EN	OS_CFG_SEM_EN	
OS_SEM_ACCEPT_EN		(6)
OS_SEM_DEL_EN	OS_CFG_SEM_DEL_EN	
OS_SEM_PEND_ABORT_EN	OS_CFG_SEM_PEND_ABORT_EN	
OS_SEM_QUERY_EN		(5)
OS_SEM_SET_EN	OS_CFG_SEM_SET_EN	
OS_TASK_STAT_EN	OS_CFG_STAT_TASK_EN	
OS_TASK_STK_CHK_EN	OS_CFG_STAT_TASK_STK_CHK_EN	
OS_TASK_CHANGE_PRIO_EN	OS_CFG_TASK_CHANGE_PRIO_EN	
OS_TASK_CREATE_EN		
OS_TASK_CREATE_EXT_EN		
OS_TASK_DEL_EN	OS_CFG_TASK_DEL_EN	
OS_TASK_NAME_EN		(2)
	OS_CFG_TASK_Q_EN	

	OS_CFG_TASK_Q_PEND_ABORT_EN	
OS_TASK_QUERY_EN		(5)
OS_TASK_PROFILE_EN	OS_CFG_TASK_PROFILE_EN	
	OS_CFG_TASK_REG_TBL_SIZE	
	OS_CFG_TASK_SEM_PEND_ABORT_EN	
OS_TASK_SUSPEND_EN	OS_CFG_TASK_SUSPEND_EN	
OS_TASK_SW_HOOK_EN		
OS_TICK_STEP_EN		(8)
OS_TIME_DLY_HMSM_EN	OS_CFG_TIME_DLY_HMSM_EN	
OS_TIME_DLY_RESUME_EN	OS_CFG_TIME_DLY_RESUME_EN	
OS_TIME_GET_SET_EN		
OS_TIME_TICK_HOOK_EN		
OS_TMR_EN	OS_CFG_TMR_EN	
OS_TMR_CFG_NAME_EN		(2)
OS_TMR_DEL_EN	OS_CFG_TMR_DEL_EN	

Table - µC/OS-III uses "CFG" in configuration

(1) DEBUG is replaced with DBG.

(2) In µC/OS-II, all kernel objects can be assigned ASCII names after creation. In µC/OS-III, ASCII names are assigned when the object is created.

(3) In µC/OS-II, it is necessary to declare the maximum number of kernel objects (number of tasks, number of event flag groups, message queues, etc.) at compile time. In µC/OS-III, all kernel objects are allocated at run time so it is no longer necessary to specify the maximum number of these objects. This feature saves valuable RAM as it is no longer necessary to over allocate objects.

(4) In µC/OS-II, event-flag width must be declared at compile time through OS_FLAG_NBITS. In µC/OS-III, this is accomplished by defining the width (i.e., number of bits) in os_type.h through the data type OS_FLAG. The default is typically 32 bits.

(5) µC/OS-III does not provide query services to the application.

(6) µC/OS-III does not directly provide "`accept`" function calls as with µC/OS-II. Instead, `OS???Pend()` functions provide an option that emulates the "`accept`" functionality by specifying `OS_OPT_PEND_NON_BLOCKING`.

(7) In µC/OS-II, there are a number of "`post`" functions. The features offered are now combined in the `OS???Post()` functions in µC/OS-III.

(8) The µC/OS-View feature `OS_TICK_STEP_EN` is not present in µC/OS-III since µC/OS-View is an obsolete product and in fact, was replaced by µC/Probe.

Variable Name Changes

Some of the variable names in µC/OS-II are changed for µC/OS-III to be more consistent with coding conventions. Significant variables are shown in the table below.

µC/OS-II (ucos_ii.h)	µC/OS-III (os.h)	Note
`OSCtxSwCtr`	`OSTaskCtxSwCtr`	
`OSCPUUsage`	`OSStatTaskCPUUsage`	(1)
`OSIdleCtr`	`OSIdleTaskCtr`	
`OSIdleCtrMax`	`OSIdleTaskCtrMax`	
`OSIntNesting`	`OSIntNestingCtr`	(2)
`OSPrioCur`	`OSPrioCur`	
`OSPrioHighRdy`	`OSPrioHighRdy`	
`OSRunning`	`OSRunning`	
`OSSchedNesting`	`OSSchedLockNestingCtr`	(3)
	`OSSchedLockTimeMax`	
`OSTaskCtr`	`OSTaskQty`	
`OSTCBCur`	`OSTCBCurPtr`	(4)
`OSTCBHighRdy`	`OSTCBHighRdyPtr`	(4)
`OSTime`	`OSTickCtr`	(5)
`OSTmrTime`	`OSTmrTickCtr`	

Table - Changes in variable naming

(1) In µC/OS-II, `OSCPUUsage` contains the total CPU utilization in percentage format. If the CPU is busy 12% of the time, `OSCPUUsage` has the value 12. In µC/OS-III, the same information is provided in `OSStatTaskCPUUsage`. However, as of µC/OS-III V3.03.00, the resolution of `OSStatTaskCPUUsage` is 1/100th of a percent or, 0.00% (value is `0`) to 100.00% (value is `10,000`).

(2) In µC/OS-II, `OSIntNesting` keeps track of the number of interrupts nesting. µC/OS-III uses `OSIntNestingCtr`. The “`Ctr`” has been added to indicate that this variable is a counter.

(3) `OSSchedNesting` represents the number of times `OSSchedLock()` is called. µC/OS-III renames this variable to `OSSchedLockNestingCtr` to better represent the variable's meaning.

(4) In µC/OS-II, `OSTCBCur` and `OSTCBHighRdy` are pointers to the `OS_TCB` of the current task, and to the `OS_TCB` of the highest-priority task that is ready-to-run. In µC/OS-III, these are renamed by adding the "`Ptr`" to indicate that they are pointers.

(5) The internal counter of the number of ticks since power up, or the last time the variable was changed through `OSTimeSet()`, has been renamed to better reflect its function.

API Changes

The most significant change from µC/OS-II to µC/OS-III occurs in the API. In order to port a µC/OS-II-based application to µC/OS-III, it is necessary to change the way services are invoked.

Table C-7 shows changes in the way critical sections in µC/OS-III are handled. Specifically, µC/OS-II defines macros to disable interrupts, and they are moved to µC/CPU withµC/OS-III since they are CPU specific functions.

µC/OS-II (`os_cpu.h`)	µC/CPU (`cpu.h`)	Note
`OS_ENTER_CRITICAL()`	`CPU_CRITICAL_ENTER()`	
`OS_EXIT_CRITICAL()`	`CPU_CRITICAL_EXIT()`	

Table - Changes in macro naming

One of the biggest changes in the µC/OS-III API is its consistency. In fact, based on the function performed, it is possible to guess which arguments are needed, and in what order. For example, "`*p_err`" is a pointer to an error-returned variable. When present, "`*p_err`" is always the last argument of a function. In µC/OS-II, error-returned values are at times returned as a "`*perr`," and at other times as the return value of the function. This inconsistency has been removed in µC/OS-III.

Event Flags API Changes

The table below shows the API for event-flag management.

µC/OS-II (os_flag.c)	µC/OS-III (os_flag.c)	Note

OS_FLAGS
OSFlagAccept(
    OS_FLAG_GRP *pgrp,
    OS_FLAGS flags,
    INT8U wait_type,
    INT8U *perr);
		(1)

OS_FLAG_GRP *
OSFlagCreate(
    OS_FLAGS   flags,
    INT8U   *perr);
	void
	OSFlagCreate(
	    OS_FLAG_GRP *p_grp,
	    CPU_CHAR *p_name,
	    OS_FLAGS flags,
	    OS_ERR *p_err);
		(2)

OS_FLAG_GRP *
OSFlagDel(
    OS_FLAG_GRP *pgrp,
    INT8U   opt,
    INT8U   *perr);
	OS_OBJ_QTY
	OSFlagDel(
	    OS_FLAG_GRP *p_grp,
	    OS_OPT opt,
	    OS_ERR *p_err);	

INT8U
OSFlagNameGet(
    OS_FLAG_GRP *pgrp,
    INT8U   **pname,
    INT8U *perr);		

void
OSFlagNameSet(
    OS_FLAG_GRP *pgrp,
    INT8U   *pname,
    INT8U   *perr);
		(3)

OS_FLAGS
OSFlagPend(
    OS_FLAG_GRP *pgrp,
    OS_FLAGS   flags,
    INT8U   wait_type,
    INT32U   timeout,
    INT8U   *perr);
	OS_FLAGS
	OSFlagPend(
	    OS_FLAG_GRP *p_grp,
	    OS_FLAGS flags,
	    OS_TICK timeout,
	    OS_OPT opt,
	    OS_TS *p_ts,
	    OS_ERR *p_err);	

OS_FLAGS
OSFlagPendGetFlagsRdy(
    void);
	OS_FLAGS
	OSFlagPendGetFlagsRdy(
	    OS_ERR *p_err);	

`OS_FLAGS`
`OSFlagPost(`
`    OS_FLAG_GRP  *pgrp,`
`    OS_FLAGS      flags,`
`    INT8U         opt,`
`    INT8U        *perr);`
	`OS_FLAGS`
	`OSFlagPost(`
	`    OS_FLAG_GRP   *p_grp,`
	`    OS_FLAGS       flags,`
	`    OS_OPT         opt,`
	`    OS_ERR        *p_err);`	

`OS_FLAGS`
`OSFlagQuery(`
`    OS_FLAG_GRP  *pgrp,`
`    INT8U        *perr);`
		(4)

Table - Event Flags API

(1) In µC/OS-III, there is no “accept” API. This feature is actually built-in the `OSFlagPend()` by specifying the `OS_OPT_PEND_NON_BLOCKING` option.

(2) In µC/OS-II, `OSFlagCreate()` returns the address of an `OS_FLAG_GRP`, which is used as the “handle” to the event-flag group. In µC/OS-III, the application must allocate storage for an `OS_FLAG_GRP`, which serves the same purpose as the `OS_EVENT`. The benefit in µC/OS-III is that it is not necessary to predetermine the number of event flags at compile time.

(3) In µC/OS-II, the user may assign a name to an event-flag group after the group is created. This functionality is built-into OSFlagCreate() for µC/OS-III.

(4) µC/OS-III does not provide query services, as they were rarely used in µC/OS-II.

Message Mailboxes API Changes

The table below shows the API for message mailbox management. Note that µC/OS-III does not directly provide services for managing message mailboxes. Given that a message mailbox is a message queue of size one, µC/OS-III can easily emulate message mailboxes.

µC/OS-II (os_mbox.c)	µC/OS-III (os_q.c)	Note
void * OSMboxAccept(OS_EVENT *pevent);		(1)

OS_EVENT *
OSMboxCreate(
    void *pmsg);
	void
	OSQCreate(
	    OS_Q *p_q,
	    CPU_CHAR *p_name,
	    OS_MSG_QTY max_qty,
	    OS_ERR *p_err);
		(2)

void *
OSMboxDel(
    OS_EVENT *pevent,
    INT8U opt,
    INT8U *perr);
	OS_OBJ_QTY,
	OSQDel(
	    OS_Q *p_q,
	    OS_OPT opt,
	    OS_ERR *p_err);	

void *
OSMboxPend(
    OS_EVENT *pevent,
    INT32U timeout,
    INT8U *perr);
	void *
	OSQPend(
	    OS_Q *p_q,
	    OS_TICK timeout,
	    OS_OPT opt,
	    OS_MSG_SIZE *p_msg_size,
	    CPU_TS *p_ts,
	    OS_ERR *p_err);
		(3)

INT8U
OSMBoxPendAbort(
    OS_EVENT *pevent,
    INT8U opt,
    INT8U *perr);
	OS_OBJ_QTY
	OSQPendAbort(
	    OS_Q *p_q,
	    OS_OPT opt
	    OS_ERR *p_err);	

INT8U
OSMboxPost(
    OS_EVENT *pevent,
    void *pmsg);
	void
	OSQPost(
	    OS_Q *p_q,
	    Void *p_void,
	    OS_MSG_SIZE msg_size,
	    OS_OPT opt,
	    OS_ERR *p_err);
		(4)

INT8U
OSMboxPostOpt(
    OS_EVENT *pevent,
    void *pmsg,
    INT8U opt);
		(4)

INT8U
OSMboxQuery(
    OS_EVENT *pevent,
    OS_MBOX_DATA *p_mbox_data);
		(5)

Table - Message Mailbox API

(1) In µC/OS-III, there is no "accept" API since this feature is built into the `OSQPend()` by specifying the `OS_OPT_PEND_NON_BLOCKING` option.

(2) In µC/OS-II, `OSMboxCreate()` returns the address of an `OS_EVENT`, which is used as the "handle" to the message mailbox. In µC/OS-III, the application must allocate storage for an `OS_Q`, which serves the same purpose as the `OS_EVENT`. The benefit in µC/OS-III is that it is not necessary to predetermine the number of message queues at compile time. Also, to create the equivalent of a message mailbox, you would specify `1` for the `max_qty` argument.

(3) µC/OS-III returns additional information about the message received. Specifically, the sender specifies the size of the message as a snapshot of the current timestamp is taken and stored as part of the message. The receiver of the message therefore knows when the message was sent.

(4) In µC/OS-III, `OSQPost()` offers a number of options that replaces the two post functions provided in µC/OS-II.

(5) µC/OS-III does not provide query services, as they were rarely used in µC/OS-II.

Memory Management API Changes

The table below shows the difference in API for memory management.

µC/OS-II (os_mem.c)	µC/OS-III (os_mem.c)	Note

OS_MEM *
OSMemCreate(
    void   *addr,
    INT32U   nblks,
    INT32U   blksize,
    INT8U   *perr);
	void
	OSMemCreate(
	    OS_MEM *p_mem,
	    CPU_CHAR *p_name,
	    void *p_addr,
	    OS_MEM_QTY n_blks,
	    OS_MEM_SIZE blk_size,
	    OS_ERR *p_err);
		(1)

void *
OSMemGet(
    OS_MEM   *pmem,
    INT8U   *perr);
	void *
	OSMemGet(
	    OS_MEM *p_mem,
	    OS_ERR *p_err);	

INT8U
OSMemNameGet(
    OS_MEM   *pmem,
    INT8U   **pname,
    INT8U *perr);		

void
OSMemNameSet(
    OS_MEM   *pmem,
    INT8U   *pname,
    INT8U   *perr);
	void
	OSMemPut(
	    OS_MEM *p_mem,
	    void *p_blk,
	    OS_ERR *p_err);
		(2)

INT8U
OSMemPut(
    OS_MEM   *pmem,
    void *pblk);		

INT8U
OSMemQuery(
    OS_MEM   *pmem,
    OS_MEM_DATA *p_mem_data);
		(3)

Table - Memory Management API

(1) In µC/OS-II, `OSMemCreate()` returns the address of an `OS_MEM` object, which is used as the "handle" to the newly created memory partition. In µC/OS-III, the application must

allocate storage for an `OS_MEM`, which serves the same purpose. The benefit in µC/OS-III is that it is not necessary to predetermine the number of memory partitions at compile time.

(2) µC/OS-III does not need an `OSMemNameSet()` since the name of the memory partition is passed as an argument to `OSMemCreate()`.

(3) µC/OS-III does not support query calls.

Mutual Exclusion Semaphores API Changes

The table below shows the difference in API for mutual exclusion semaphore management.

µC/OS-II (os_mutex.c)	µC/OS-III (os_mutex.c)	Note
BOOLEAN OSMutexAccept(OS_EVENT *pevent, INT8U *perr);		(1)
OS_EVENT * OSMutexCreate(INT8U prio, INT8U *perr);	void OSMutexCreate(OS_MUTEX *p_mutex, CPU_CHAR *p_name, OS_ERR *p_err);	(2)
OS_EVENT * OSMutexDel(OS_EVENT *pevent, INT8U opt, INT8U *perr);	void OSMutexDel(OS_MUTEX *p_mutex, OS_OPT opt, OS_ERR *p_err);	
void OSMutexPend(OS_EVENT *pevent, INT32U timeout, INT8U *perr);	void OSMutexPend(OS_MUTEX *p_mutex, OS_TICK timeout, OS_OPT opt, CPU_TS *p_ts, OS_ERR *p_err);	(3)
	OS_OBJ_QTY OSMutexPendAbort(OS_MUTEX *p_mutex, OS_OPT opt, OS_ERR *p_err);	
INT8U OSMutexPost(OS_EVENT *pevent);	void OSMutexPost(OS_MUTEX *p_mutex, OS_OPT opt, OS_ERR *p_err);	
INT8U OSMutexQuery(OS_EVENT *pevent, OS_MUTEX_DATA *p_mutex_data);		(4)

Table - Mutual Exclusion Semaphores API

(1) In µC/OS-III, there is no “accept” API, since this feature is built into the `OSMutexPend()` by specifying the `OS_OPT_PEND_NON_BLOCKING` option.

(2) In µC/OS-II, `OSMutexCreate()` returns the address of an `OS_EVENT`, which is used as the “handle” to the message mailbox. In µC/OS-III, the application must allocate storage for an `OS_MUTEX`, which serves the same purpose as the `OS_EVENT`. The benefit in µC/OS-III is that it is not necessary to predetermine the number of mutual-exclusion semaphores at compile time.

(3) µC/OS-III returns additional information when a mutex is released. The releaser takes a snapshot of the current time stamp and stores it in the `OS_MUTEX`. The new owner of the mutex therefore knows when the mutex was released.

(4) µC/OS-III does not provide query services as they were rarely used.

Message Queues API Changes

This table shows the difference in API for message-queue management.

µC/OS-II (os_q.c)	µC/OS-III (os_q.c)	Note
void * OSQAccept(OS_EVENT *pevent, INT8U *perr);		(1)
OS_EVENT * OSQCreate(void **start, INT16U size);	void OSQCreate(OS_Q *p_q, CPU_CHAR *p_name, OS_MSG_QTY max_qty, OS_ERR *p_err);	(2)
OS_EVENT * OSQDel(OS_EVENT *pevent, INT8U opt, INT8U *perr);	OS_OBJ_QTY, OSQDel(OS_Q *p_q, OS_OPT opt, OS_ERR *p_err);	
INT8U OSQFlush(OS_EVENT *pevent);	OS_MSG_QTY OSQFlush(OS_Q *p_q, OS_ERR *p_err);	
void * OSQPend(OS_EVENT *pevent, INT32U timeout, INT8U *perr);	void * OSQPend(OS_Q *p_q, OS_MSG_SIZE *p_msg_size, OS_TICK timeout, OS_OPT opt, CPU_TS *p_ts, OS_ERR *p_err);	(3)
INT8U OSQPendAbort(OS_EVENT *pevent, INT8U opt, INT8U *perr);	OS_OBJ_QTY OSQPendAbort(OS_Q *p_q, OS_OPT opt, OS_ERR *p_err);	
INT8U OSQPost(OS_EVENT *pevent, void *pmsg);	void OSQPost(OS_Q *p_q, void *p_void, OS_MSG_SIZE msg_size, OS_OPT opt, OS_ERR *p_err);	(4)
INT8U OSQPostFront(OS_EVENT *pevent, void *pmsg);		

`INT8U OSQPostOpt( OS_EVENT *pevent, void *pmsg, INT8U opt);`		(4)
`INT8U OSQQuery( OS_EVENT *pevent, OS_Q_DATA *p_q_data);`		(5)

Table - Mesage Queues API

(1) In µC/OS-III, there is no "accept" API as this feature is built into the `OSQPend()` by specifying the `OS_OPT_PEND_NON_BLOCKING` option.

(2) In µC/OS-II, `OSQCreate()` returns the address of an `OS_EVENT`, which is used as the "handle" to the message queue. In µC/OS-III, the application must allocate storage for an `OS_Q` object, which serves the same purpose as the `OS_EVENT`. The benefit in µC/OS-III is that it is not necessary to predetermine at compile time, the number of message queues.

(3) µC/OS-III returns additional information when a message queue is posted. Specifically, the sender includes the size of the message and takes a snapshot of the current timestamp and stores it in the message. The receiver of the message therefore knows when the message was posted.

(4) In µC/OS-III, `OSQPost()` offers a number of options that replaces the three post functions provided in µC/OS-II.

(5) µC/OS-III does not provide query services as they were rarely used.

Miscellaneous API Changes

The table below shows the difference in API for miscellaneous services.

<table>
<tr><th>µC/OS-II (<code>os_core.c</code>)</th><th>µC/OS-III (<code>os_core.c</code>)</th><th>Note</th></tr>
<tr><td><pre>INT8U
OSEventNameGet(
    OS_EVENT   *pevent,
    INT8U   **pname,
    INT8U   *perr);</pre></td><td></td><td>(1)</td></tr>
<tr><td><pre>void
OSEventNameSet(
    OS_EVENT   *pevent,
    INT8U   *pname,
    INT8U   *perr);</pre></td><td></td><td>(1)</td></tr>
<tr><td><pre>INT16U
OSEventPendMulti(
    OS_EVENT **pevent_pend,
    OS_EVENT **pevent_rdy,
    void   **pmsgs_rdy,
    INT32U   timeout,
    INT8U   *perr);</pre></td><td></td><td>(2)</td></tr>
<tr><td><pre>void
OSInit(void)</pre></td><td><pre>void
OSInit(
    OS_ERR   *p_err);</pre></td><td>(3)</td></tr>
<tr><td><pre>void
OSIntEnter(void)</pre></td><td><pre>void
OSIntEnter(void);</pre></td><td></td></tr>
<tr><td><pre>void
OSIntExit(void)</pre></td><td><pre>void
OSIntExit(void)</pre></td><td></td></tr>
<tr><td></td><td><pre>void
OSSched(void);</pre></td><td></td></tr>
<tr><td><pre>void
OSSchedLock(void)</pre></td><td><pre>void
OSSchedLock(
    OS_ERR   *p_err);</pre></td><td>(4)</td></tr>
<tr><td></td><td><pre>void
OSSchedRoundRobinCfg(
    CPU_BOOLEAN   en,
    OS_TICK
dflt_time_quanta,
    OS_ERR   *p_err);</pre></td><td>(5)</td></tr>
<tr><td></td><td><pre>void
OSSchedRoundRobinYield(
    OS_ERR   *p_err);</pre></td><td>(6)</td></tr>
<tr><td><pre>void
OSSchedUnlock(void)</pre></td><td><pre>void
OSSchedUnlock(
    OS_ERR   *p_err);</pre></td><td>(7)</td></tr>
</table>

void OSStart(void)	void OSStart(void);	
void OSStatInit(void)	void OSStatTaskCPUUsageInit(OS_ERR *p_err);	(8)
INT16U OSVersion(void)	CPU_INT16U OSVersion(OS_ERR *p_err);	(9)

Table - Miscellaneous API

(1) Objects in µC/OS-III are named when they are created and these functions are not required in µC/OS-III.

(2) Multipend no longer exist in µC/OS-III.

(3) µC/OS-III returns an error code for this function. Initialization is successful if `OS_ERR_NONE` is received from `OSInit()`. In µC/OS-II, there is no way of detecting an error in the configuration that caused `OSInit()` to fail.

(4) An error code is returned in µC/OS-III for this function.

(5) Enable or disable µC/OS-III's round-robin scheduling at run time, as well as change the default time quanta.

(6) A task that completes its work before its time quanta expires may yield the CPU to another task at the same priority.

(7) An error code is returned in µC/OS-III for this function.

(8) Note the change in name for the function that computes the "capacity" of the CPU for the purpose of computing CPU usage at run-time.

(9) An error code is returned in µC/OS-III for this function.

Hooks and Port API Changes

Table C-18 shows the difference in APIs used to port µC/OS-II to µC/OS-III.

µC/OS-II (OS_CPU*.C/H)	µC/OS-III (OS_CPU*.C/H)	Note
	`OS_GET_TS();`	(1)
`void` `OSInitHookBegin(void);`	`void` `OSInitHook(void);`	
`void` `OSInitHookEnd(void);`		

`void`
`OSTaskCreateHook(`
`    OS_TCB    *ptcb);`
	`void`
	`OSTaskCreateHook(`
	`    OS_TCB        *p_tcb);`	

`void`
`OSTaskDelHook(`
`    OS_TCB    *ptcb);`
	`void`
	`OSTaskDelHook(`
	`    OS_TCB        *p_tcb);`	

`void` `OSTaskIdleHook(void);`	`void` `OSIdleTaskHook(void);`	

	`void`
	`OSTaskReturnHook(`
	`    OS_TCB        *p_tcb);`
		(2)

`void` `OSTaskStatHook(void)`	`void` `OSStatTaskHook(void);`	

`void`
`OSTaskStkInit(`
`    void     (*task)(void`
`*p_arg),`
`    void      *p_arg,`
`    OS_STK    *ptos,`
`    INT16U     opt);`
	`CPU_STK *`
	`OSTaskStkInit(`
	`    OS_TASK_PTR    p_task,`
	`    void          *p_arg,`
	`    CPU_STK       *p_stk_base,`
	`    CPU_STK      *p_stk_limit,`
	`    CPU_STK_SIZE   size,`
	`    OS_OPT         opt);`
		(3)

`void` `OSTaskSwHook(void)`	`void` `OSTaskSwHook(void);`	

`void`
`OSTCBInitHook(`
`    OS_TCB    *ptcb);`
		(4)

`void` `OSTimeTickHook(void);`	`void` `OSTimeTickHook(void);`	
`void` `OSStartHighRdy(void);`	`void` `OSStartHighRdy(void);`	(5)
`void` `OSIntCtxSw(void);`	`void` `OSIntCtxSw(void);`	(5)

`void OSCtxSw(void);`	`void OSCtxSw(void);`	(5)

Table - Hooks and Port API

(1) µC/OS-III requires that the Board Support Package (BSP) provide a 32-bit free-running counter (from `0x00000000` to `0xFFFFFFFF` and rolls back to `0x00000000`) for the purpose of performing time measurements. When a signal is sent, or a message is posted, this counter is read and sent to the recipient. This allows the recipient to know when the message was sent. If a 32-bit free-running counter is not available, you can simulate one using a 16-bit counter but, this requires more code to keep track of overflows.

(2) µC/OS-III is able to terminate a task that returns. Recall that tasks should not return since they should be either implemented as an infinite loop, or deleted if implemented as run once.

(3) The code for `OSTaskStkInit()` must be changed slightly in µC/OS-III since several arguments passed to this function are different than in µC/OS-II. Instead of passing the top-of-stack as in µC/OS-II, `OSTaskStkInit()` is passed the base address of the task stack, as well as the size of the stack.

(4) This function is not needed in µC/OS-III.

(5) These functions are a part of `os_cpu_a.asm`, and should only require name changes for the following variables:

µC/OS-II variable changes from …	… to this in µC/OS-III
`OSIntNesting`	`OSIntNestingCtr`
`OSTCBCur`	`OSTCBCurPtr`
`OSTCBHighRdy`	`OSTCBHighRdyPtr`

Task Management API Changes

The table below shows the difference in API for task-management services.

µC/OS-II (os_task.c)	µC/OS-III (os_task.c)	Note
INT8U OSTaskChangePrio(INT8U oldprio, INT8U newprio);	void OSTaskChangePrio(OS_TCB *p_tcb, OS_PRIO prio, OS_ERR *p_err);	(1)
INT8U OSTaskCreate(void (*task)(void *p_arg), void *p_arg, OS_STK *ptos, INT8U prio);	void OSTaskCreate(OS_TCB *p_tcb, CPU_CHAR *p_name, OS_TASK_PTR *p_task, void *p_arg, OS_PRIO prio, CPU_STK *p_stk_base, CPU_STK_SIZE stk_limit, CPU_STK_SIZE stk_size, OS_MSG_QTY q_size, OS_TICK time_quanta, void *p_ext, OS_OPT opt, OS_ERR *p_err);	(2)
INT8U OSTaskCreateExt(void (*task)(void *p_arg), void *p_arg, OS_STK *ptos, INT8U prio, INT16U id, OS_STK *pbos, INT32U stk_size, void *pext, INT16U opt);	void OSTaskCreate(OS_TCB *p_tcb, CPU_CHAR *p_name, OS_TASK_PTR *p_task, void *p_arg, OS_PRIO prio, CPU_STK *p_stk_base, CPU_STK_SIZE stk_limit, CPU_STK_SIZE stk_size, OS_MSG_QTY q_size, OS_TICK time_quanta, void *p_ext, OS_OPT opt, OS_ERR *p_err);	(2)
INT8U OSTaskDel(INT8U prio);	void OSTaskDel(OS_TCB *p_tcb, OS_ERR *p_err);	
INT8U OSTaskDelReq(INT8U prio);		

INT8U OSTaskNameGet(INT8U prio, INT8U **pname, INT8U *perr);		
void OSTaskNameSet(INT8U prio, INT8U *pname, INT8U *perr);		(3)
	OS_MSG_QTY OSTaskQFlush(OS_TCB *p_tcb, OS_ERR *p_err);	(4)
	void * OSTaskQPend(OS_TICK timeout, OS_OPT opt, OS_MSG_SIZE *p_msg_size, CPU_TS *p_ts, OS_ERR *p_err);	(4)
	CPU_BOOLEAN OSTaskQPendAbort(OS_TCB *p_tcb, OS_OPT opt, OS_ERR *p_err);	(4)
	void OSTaskQPost(OS_TCB *p_tcb, void *p_void, OS_MSG_SIZE msg_size, OS_OPT opt, OS_ERR *p_err);	(4)
INT32U OSTaskRegGet(INT8U prio, INT8U id, INT8U *perr);	OS_REG OSTaskRegGet(OS_TCB *p_tcb, OS_REG_ID id, OS_ERR *p_err);	
void OSTaskRegSet(INT8U prio, INT8U id, INT32U value, INT8U *perr);	void OSTaskRegGet(OS_TCB *p_tcb, OS_REG_ID id, OS_REG value, OS_ERR *p_err);	

INT8U
OSTaskResume(
    INT8U   prio);
	void
	OSTaskResume(
	    OS_TCB *p_tcb,
	    OS_ERR *p_err);	

	OS_SEM_CTR
	OSTaskSemPend(
	    OS_TICK timeout,
	    OS_OPT opt,
	    CPU_TS *p_ts,
	    OS_ERR *p_err);
		(5)

	CPU_BOOLEAN
	OSTaskSemPendAbort(
	    OS_TCB *p_tcb,
	    OS_OPT opt,
	    OS_ERR *p_err);
		(5)

	CPU_BOOLEAN
	OSTaskSemPendAbort(
	    OS_TCB *p_tcb,
	    OS_OPT opt,
	    OS_ERR *p_err);
		(5)

	OS_SEM_CTR
	OSTaskSemPost(
	    OS_TCB *p_tcb,
	    OS_OPT opt,
	    OS_ERR *p_err);
		(5)

	OS_SEM_CTR
	OSTaskSemSet(
	    OS_TCB *p_tcb,
	    OS_SEM_CTR cnt,
	    OS_ERR *p_err);
		(5)

INT8U
OSTaskSuspend(
    INT8U   prio);
	void
	OSTaskSuspend(
	    OS_TCB *p_tcb,
	    OS_ERR *p_err);	

INT8U
OSTaskStkChk(
    INT8U   prio,
    OS_STK_DATA *p_stk_data);
	void
	OSTaskStkChk(
	    OS_TCB *p_tcb,
	    CPU_STK_SIZE *p_free,
	    CPU_STK_SIZE *p_used,
	    OS_ERR *p_err);
		(6)

	void
	OSTaskTimeQuantaSet(
	    OS_TCB *p_tcb,
	    OS_TICK time_quanta,
	    OS_ERR *p_err);
		(7)

`INT8U`
`OSTaskQuery(`
`    INT8U        prio,`
`    OS_TCB      *p_task_data);`
		(8)

Table - Task Management API

(1) In µC/OS-II, each task must have a unique priority. The priority of a task can be changed at run-time, however it can only be changed to an unused priority. This is generally not a problem since µC/OS-II supports up to 255 different priority levels and is rare for an application to require all levels. Since µC/OS-III supports an unlimited number of tasks at each priority, the user can change the priority of a task to any available level.

(2) µC/OS-II provides two functions to create a task: `OSTaskCreate()` and `OSTaskCreateExt()`. `OSTaskCreateExt()` is recommended since it offers more flexibility. In µC/OS-III, only one API is used to create a task, `OSTaskCreate()`, which offers similar features to `OSTaskCreateExt()` and provides additional ones.

(3) µC/OS-III does not need an `OSTaskNameSet()` since an ASCII name for the task is passed as an argument to `OSTaskCreate()`.

(4) µC/OS-III allows tasks or ISRs to send messages directly to a task instead of having to pass through a mailbox or a message queue as does µC/OS-II.

(5) µC/OS-III allows tasks or ISRs to directly signal a task instead of having to pass through a semaphore as does µC/OS-II.

(6) In µC/OS-II, the user must allocate storage for a special data structure called `OS_STK_DATA`, which is used to place the result of a stack check of a task. This data structure contains only two fields: `.OSFree` and `.OSUsed`. In µC/OS-III, it is required that the caller pass pointers to destination variables where those values will be placed.

(7) µC/OS-III allows users to specify the time quanta of each task on a per-task basis. This is available since µC/OS-III supports multiple tasks at the same priority, and allows for round robin scheduling. The time quanta for a task is specified when the task is created, but it can be changed by the API at run time.

(8) µC/OS-III does not provide query services as they were rarely used.

Semaphores API Changes

The table below shows the difference in API for semaphore management.

µC/OS-II (os_sem.c)	µC/OS-III (os_sem.c)	Note
INT16U OSSemAccept(OS_EVENT *pevent);		(1)

OS_EVENT *
OSSemCreate(
    INT16U   cnt);
	void
	OSSemCreate(
	    OS_SEM *p_sem,
	    CPU_CHAR *p_name,
	    OS_SEM_CTR cnt,
	    OS_ERR *p_err);
		(2)

OS_EVENT *
OSSemDel(
    OS_EVENT   *pevent,
    INT8U   opt,
    INT8U   *perr);
	OS_OBJ_QTY,
	OSSemDel(
	    OS_SEM *p_sem,
	    OS_OPT opt,
	    OS_ERR *p_err);	

void
OSSemPend(
    OS_EVENT   *pevent,
    INT32U   timeout,
    INT8U   *perr);
	OS_SEM_CTR
	OSSemPend(
	    OS_SEM *p_sem,
	    OS_TICK timeout,
	    OS_OPT opt,
	    CPU_TS *p_ts,
	    OS_ERR *p_err);
		(3)

INT8U
OSSemPendAbort(
    OS_EVENT   *pevent,
    INT8U   opt,
    INT8U   *perr);
	OS_OBJ_QTY
	OSSemPendAbort(
	    OS_SEM *p_sem,
	    OS_OPT opt,
	    OS_ERR *p_err);	

void
OSSemPost(
    OS_EVENT   *pevent);
	void
	OSSemPost(
	    OS_SEM *p_sem,
	    OS_OPT opt,
	    OS_ERR *p_err);	

INT8U
OSSemQuery(
    OS_EVENT   *pevent,
    OS_SEM_DATA *p_sem_data);
		(4)

`void`
`OSSemSet(`
`    OS_EVENT    *pevent,`
`    INT16U       cnt,`
`    INT8U       *perr);`
	`void`
	`OSSemSet(`
	`    OS_SEM       *p_sem,`
	`    OS_SEM_CTR    cnt,`
	`    OS_ERR       *p_err);`	

(1) In µC/OS-III, there is no “accept” API since this feature is built into the `OSSemPend()` by specifying the `OS_OPT_PEND_NON_BLOCKING` option.

(2) In µC/OS-II, `OSSemCreate()` returns the address of an `OS_EVENT`, which is used as the “handle” to the semaphore. In µC/OS-III, the application must allocate storage for an `OS_SEM` object, which serves the same purpose as the `OS_EVENT`. The benefit in µC/OS-III is that it is not necessary to predetermine the number of semaphores at compile time.

(3) µC/OS-III returns additional information when a semaphore is signaled. The ISR or task that signals the semaphore takes a snapshot of the current timestamp and stores this in the `OS_SEM` object signaled. The receiver of the signal therefore knows when the signal was sent.

(4) µC/OS-III does not provide query services, as they were rarely used.

MISRA-C2004 and uC-OS-III

MISRA-C2004 Rule 8.12 Required

MISRA-C2004 Rule 14.7 Required

MISRA-C2004 Rule 15.2 Required

MISRA-C2004 Rule 17.4 Required

MISRA-C:2004, Rule 8.5 (Required)

Rule Description

There shall be no definitions of objects or functions in a header file.

Offending code appears as

```
OS_EXT   OS_IDLE_CTR   OSIdleTaskCtr;
```

`OS_EXT` allows us to declare "extern" and storage using a single declaration in `os.h` but allocation of storage actually occurs in `os_var.c`.

Rule suppressed

The method used in µC/OS-III is an improved scheme as it avoids declaring variables in multiple files.

Occurs in

`os.h`

Bibliography

Bal Sathe, Dhananjay. 1988. *Fast Algorithm Determines Priority.* EDN (India), September, p. 237.

Comer, Douglas. 1984. *Operating System Design, The XINU Approach.* Englewood Cliffs, New Jersey: Prentice-Hall. ISBN 0-13-637539-1.

Kernighan, Brian W. and Dennis M. Ritchie. 1988. *The C Programming Language, 2nd edition.* Englewood Cliffs, New Jersey: Prentice Hall. ISBN 0-13-110362-8.

Klein, Mark H., Thomas Ralya, Bill Pollak, Ray Harbour Obenza, and Michael Gonzlez. 1993. *A Practioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems.* Norwell, Massachusetts: Kluwer Academic Publishers Group. ISBN 0-7923-9361-9.

Labrosse, Jean J. 2002, *MicroC/OS-II, The Real-Time Kernel*, CMP Books, 2002, ISBN 1-57820-103-9.

Li, Qing. *Real-Time Concepts for Embedded Systems*, CMP Books, July 2003, ISBN 1-57820-124-1.

The Motor Industry Software Reliability Association, *MISRA-C:2004*, Guidelines for the Use of the C Language in Critical Systems, October 2004. www.misra-c.com.

Licensing Policy

µC/OS-III is provided in source form for FREE short-term evaluation, for educational use or for peaceful research. If you plan or intend to use µC/OS-III in a commercial application/ product then, you need to contact Micrium to properly license µC/OS-III for its use in your application/product. We provide ALL the source code for your convenience and to help you experience µC/OS-III. The fact that the source is provided does NOT mean that you can use it commercially without paying a licensing fee.

It is necessary to purchase this license when the decision to use µC/OS-III in a design is made, not when the design is ready to go to production.

If you are unsure about whether you need to obtain a license for your application, please contact Micriµm and discuss the intended use with a sales representative.

Contact Micrium

1290 Weston Road, Suite 306
Weston, FL 33326
USA

Phone: +1 954 217 2036
Fax: +1 954 217 2037

E-mail: Licensing@Micrium.com
Web: www.Micrium.com