Micriμm

New Features and Services since

μC/OS-II V2.00

(Current Version: V2.92.11)

www.Micrium.com

Introduction

This document describes all the features and services added to **µC/OS-II** since the introduction of the hard cover book *MicroC/OS-II, The Real-Time Kernel*, ISBN 0-87930-543-6. The software provided with the book was version 2.00 or V2.04. The version number of the change is shown when appropriate.

Added 'TLS' (V2.92.08)

µC/OS-II now supports 'Thread Local Storage' (TLS) for Analog Devices' CrossCore Embedded Studio (CCES) and IAR's Embedded Workbench.

µC/OS-II now provides built-in support for run-time library thread safety through the use of Task Local Storage (TLS) for storage of task-specific run-time library static data and mutual exclusion semaphores to protect accesses to shared resources.

The run-time environment consists of the run-time library, which contains the functions defined by the C and the C++ standards, and includes files that define the library interface (the system header files). Compilers provide complete libraries that are compliant with Standard C and C++. These libraries also supports floating-point numbers in IEEE 754 format and can be configured to include different levels of support for locale, file descriptors, multi-byte characters, etc. Most parts of the libraries are reentrant, but some functionality and parts are not reentrant because they require the use of static data. Different compilers provide different methods to add reentrancy to their libraries through an API defined by the tool chain supplier.

The current version supports TLS for Analog Devices' CrossCore Embedded Studio and IAR's Embedded Workbench.

In a multi-threaded environment the C/C++ library has to handle all library objects with a global state differently. Either an object is a true global object, then any updates of its state has to be guarded by some locking mechanism to make sure that only one task can update it at any one time, or an object is local to each task, then the static variables containing the objects state must reside in a variable area local for the task. This area is commonly named thread local storage or, TLS.

The run-time library may also need to use multiple types of locks. For example, a lock could be necessary to ensure exclusive access to the file stream, another one to the heap, etc. It is thus common to protect the following functions through one or more mutual exclusion semaphores (mutex):

- The heap through the usage of `malloc()`, `free()`, `realloc()`, and `calloc()`.
- The file system through the usage of `fopen()`, `fclose()`, `fdopen()`, `fflush()`, and `freopen()`.
- The signal system through the usage of `signal()`.
- The tempfile system through the usage of `tmpnam()`.
- Initialization of static function objects.
- Thread-local storage is typically needed for the following library objects:

- Error functions through `errno` and `strerror`
- Locale functions through the usage of `localeconv()` and `setlocale()`
- Time functions through the usage of `asctime()`, `localtime()`, `gmtime()`, and `mktime()`
- Multibyte functions through the usage of `mbrlen()`, `mbrtowc()`, `mbsrtowc()`, mbtowc(), `wcrtomb()`, `wcsrtomb()`, and `wctomb()`
- Random functions through the usage of `rand()` and `srand()`
- Other functions through the usage of `atexit()` and `strtok()`
- C++ exception engine

In order to enable thread safety, you need to do the following:

- Set `OS_TLS_TBL_SIZE` in `os_cfg.h` to a value greater than `1`. The actual value depends on the number of entries needed by the compiler used. In most cases you would only need to set this to 1 but you should consult the `os_tls.c` that you plan to use for additional information.
- Add to your build, the `os_tls.c` file that corresponds to the compiler you are using. See `\Micrium\Software\uCOS-II\TLS\*`
- Depending on the compiler and how TLS is allocated, you may also need to make sure that you have a heap. Consult your compiler documentation on how you can enable the heap and determine its size.
- Most likely, `os_tls.c` will make use of semaphores to guard access to shared resources (such as the heap or files) then you need to make sure `OS_SEM_EN` is set to 1 in `os_cfg.h`. Also, the run-time library may already define APIs to lock and unlock sections of code. The implementation of these functions should also be part of `os_tls.c`.

You should note that these are the only steps you need to make to enable thread safety.

Added 'Task Registers' (V2.92.07)

µC/OS-II now allows the user to store task-specific values in 'task registers'. Task registers are different than CPU registers and are used to save such information as "errno" which is common in software components. Task registers can also store task-related data to be associated with the task at run time such as I/O register settings, configuration values, etc. A task may have as many as `OS_TASK_REG_TBL_SIZE` registers, and all registers have a data type of `INT32U`. A task register is changed by calling `OSTaskRegSet()` and read by calling `OSTaskRegGet()`. The desired task register is specified as an argument to these functions and can take a value between `0` and `OS_TASK_REG_TBL_SIZE-1`.

```
OSTaskRegGet()
OSTaskRegGetID()
OSTaskRegSet()
```

`OSVersion()` now returns a version number scaled by `10,000`. So, if `OSVersion()` returns `29207` then it means V2.92.07.

Version number skipped (V2.91)

There is no version 2.91.

Added 'OSSafetyCriticalStart()' (V2.90)

µC/OS-II now allows your application to 'tell' **µC/OS-II** that your application is no longer allowed to delete kernel objects such as tasks, semaphores, queues, etc.

Delete task on incorrect return (V2.89)

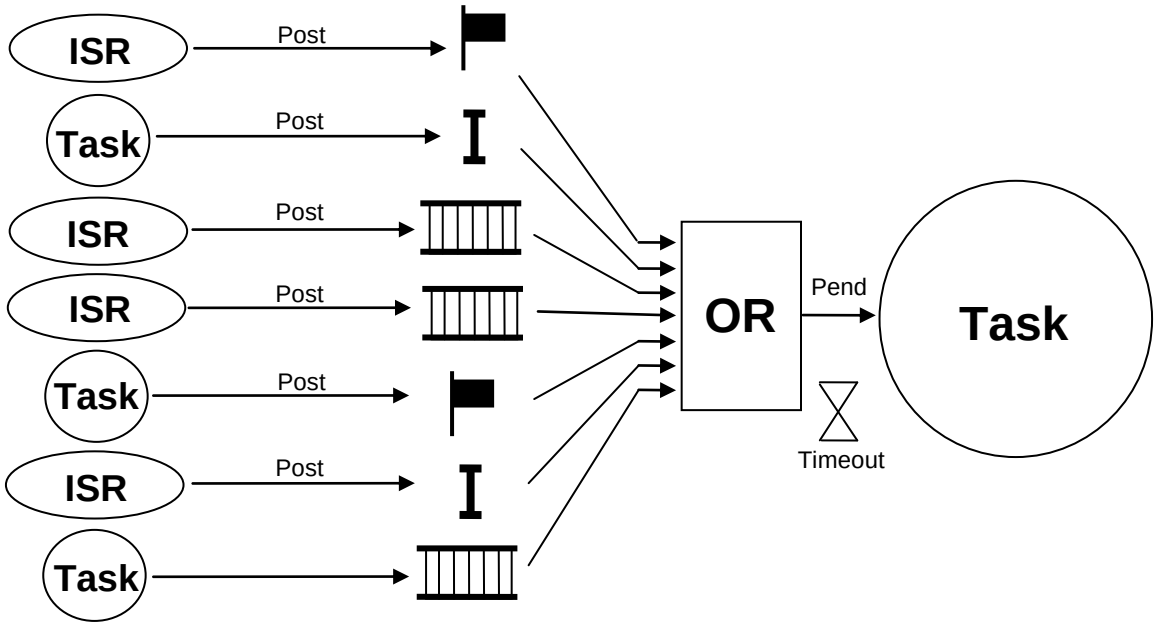
µC/OS-II now contains a new function called `OS_TaskReturn()`. All µC/OS-II tasks are not allowed to return. If a task returns by mistake, `OS_TaskReturn()` catches those and deletes the task.

`OS_TaskReturn()` calls `OSTaskReturnHook()` which in turn calls `App_TaskReturnHook()`.

Pend on Multiple Events (V2.86)

µC/OS-II now contains a new function called `OSEventPendMulti()` which allows a task to pend on multiple events (semaphores, mailboxes and queues) in any combination (see example diagram below). This new function is found in `OS_CORE.C` and is enabled by setting `OS_EVENT_MULTI_EN` to `1` in `OS_CFG.H`.

With `OSEventPendMulti()` it's possible to pend on any number of semaphores, mailboxes and message queues at the same time (we don't support Mutex and Event Flags at this time). If a task pends on a combination of the above 'events' then, as soon an event is posted (and the pending task is the highest priority task pending on the event), the waiting task will wake up and be 'handed' the event. If events are present as the task pends then ALL the available events will be provided to the task.



Timer Manager (V2.81)

µC/OS-II now provides support for periodic as well as one-shot timers. This functionality is found in `OS_TMR.C` and is enabled by setting `OS_TMR_EN` to `1` in `OS_CFG.H`. Your application can have any number of timers (up to 65500). When a timer times out, an optional callback function can be called allowing you to perform any action (signal a task, turn on/off a light, etc.). Each timer has its own callback function.

IMPORTANT

The APIs for the Timer Manager were changed in V2.83 from what they were in V2.81 and V2.82. This was necessary to correct some issues with the Timer Manager. Please consult the Reference Manual for the new APIs.

When timer management is enabled, **µC/OS-II** creates a timer task (`OSTmrTask()`) which is responsible for updating all the timers. The priority of this task is determined by `OS_TASK_TMR_PRIO` which should be defined in your application's `APP_CFG.H`.

The timer manager provides a number of services to your applications. Specifically, you can call one of the following functions (see the **µC/OS-II** reference manual for a description of these functions) from your tasks:

`OSTmrCreate()`	Create a timer
`OSTmrDel()`	Delete a timer
`OSTmrRemainGet()`	Determine how much time before a timer expires
`OSTmrNameGet()`	Get the name of a timer
`OSTmrStateGet()`	Get the state of a timer (UNUSED, STOPPED, RUNNING, COMPLETED)
`OSTmrStart()`	Start a timer
`OSTmrStop()`	Stop a timer

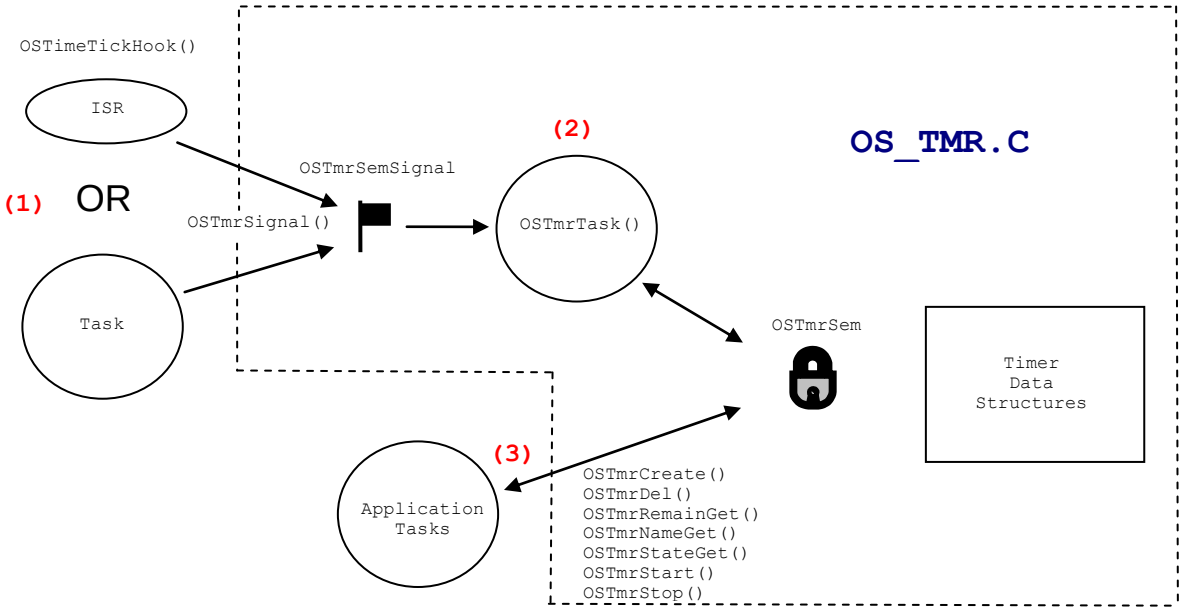
You should note that you **CANNOT** call these functions from ISRs.

The drawing below shows the task model of the Timer Manager. You should note that semaphore management needs to be enabled (you need to set `OS_SEM_EN` to `1` in `OS_CFG.H`) for the timer manager to work. The timer manager requires two (2) semaphores.

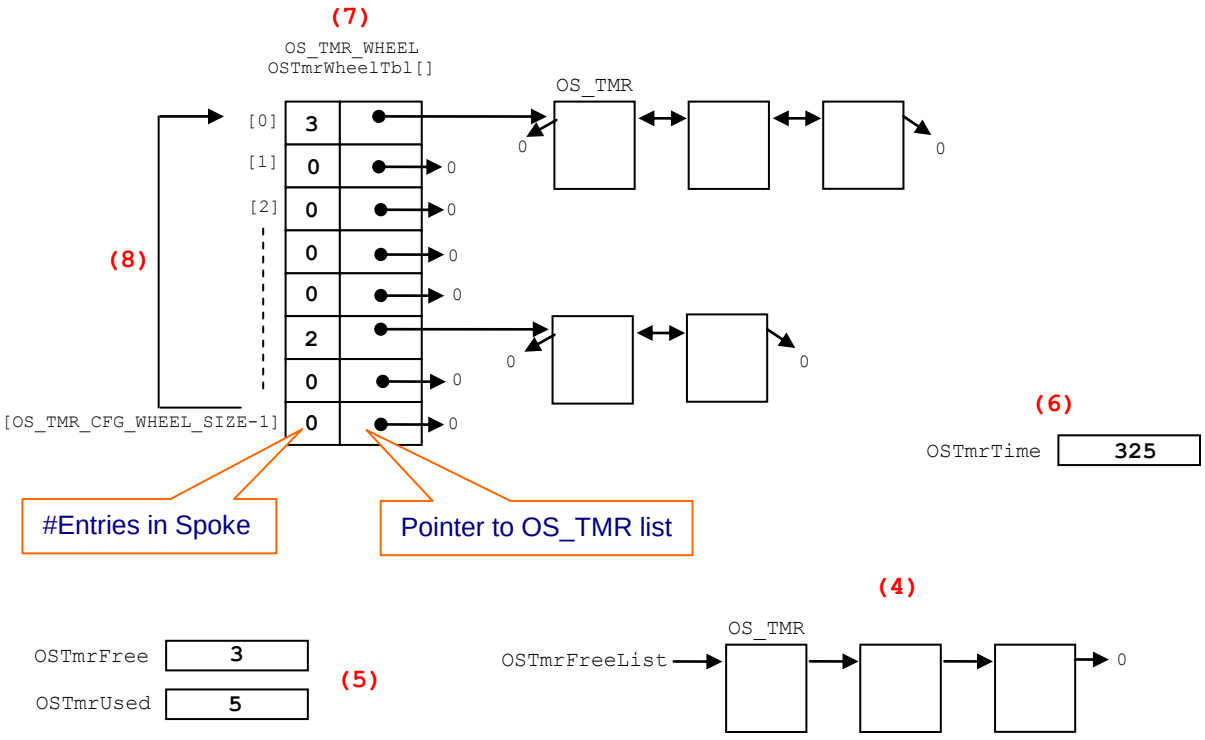
(1) An ISR or an application task needs to 'signal' a counting semaphore by calling `OSTmrSignal()` at the rate specified in `OS_TMR_CFG_TICK_RATE` (see `OS_CFG.H`). The counting semaphore is called `OSTmrSemSignal` that is initialized to `0` by **µC/OS-II** when `OSInit()` is called. You should note that you should ONLY call `OSTmrSignal()` and not worry about the semaphore; it's encapsulated by `OSTmrSignal()`.

(2) The timer management task (`OSTmrTask()`) pends forever on the counting semaphore waiting for it to be signaled. When the semaphore is signaled, `OSTmrTask()` acquires another semaphore (a binary semaphore in this case, `OSTmrSem`) to gain exclusive access to timer data structures. When `OSTmrTask()` is the owner of the semaphore it updates all the timers created by your application.

(3) Your application accesses timer data structure via interface functions. These functions allow you to create, delete, start and stop timers as well as examine the amount of time remaining before a timer times out.



The drawing below shows the data structures used in the timer manager.



(4) Each timer is characterized by a data structure of type `OS_TMR` (see `ucos_ii.h`). Each timer contains the 'period' of the timer (if the timer is to operate in periodic mode), the name of the timer, a timer 'match' value (described later) and other fields used to link the timer. Free timers are placed in a singly linked list of 'unused' timers pointed to by `OSTmrFreeList`.

(5) The number of free timers is held in `OSTmrFree` and the number of used (or allocated) timers is held in `OSTmrUsed`. Of course, the total number of timers is the sum of these two fields and, unless you don't properly use the timer management services, that sum should always equal `OS_TMR_CFG_MAX`.

(6) Every time `OSTmrSignal()` is called, the unsigned 32-bit variable `OSTmrTime` is incremented by one and used to see if timers have expired.

(7) The timer manager keeps track of which timer it needs to update using a 'timer wheel'. The wheel is basically an array of structures of type `OS_TMR_WHEEL` (see `ucos_ii.h`) that wraps around. This structure contains two fields: a pointer to a doubly-linked list of `OS_TMR` structures and, the number of entries in that list.

(8) The 'wheel' contains `OS_TMR_CFG_WHEEL_SIZE` entries or spokes.

`OS_TMR` structures are inserted in the wheel when you call `OSTmrStart()`. The position (i.e. spoke) in `OSTmrWheelTbl[]` for a specific timer is given by:

```
match = OSTmrTime + period;
spoke = match % OS_TMR_CFG_WHEEL_SIZE;
```

The 'match' corresponds to the value that `OSTmrTime` needs to reach before the timer expires. For example, let's say that `OSTmrTime` is `0` (just initialized) and we want to create a timer that will expire every second (assuming `OS_TMR_CFG_TICKS_PER_SEC` is set to `10`). Also, let's assume that `OS_TMR_CFG_WHEEL_SIZE` is `8` (as shown in the diagram above).

```
match = OSTmrTime + period;
match = 0 + 10;
match = 10;

spoke = match % OS_TMR_CFG_WHEEL_SIZE;
spoke = 10 % 8;
spoke = 2;
```

This means that `OSTmrStart()` will obtain a free `OS_TMR` data structure from the free list of timers and the place this data structure in `OSTmrWheelTbl[]` at position #2 in the table. `OSTmrStart()` will then store the 'match' value in the `OS_TMR` data structure.

Every time `OSTmrTime` is incremented by `OSTmrTask()`, `OSTmrTask()` goes through ALL the `OS_TMR` structures placed at spoke (`OSTmrTime % OS_TMR_CFG_WHEEL_SIZE`) to see if `OSTmrTime` 'matches' the value store in the `OS_TMR` structure. If a match occurs, the timer is removed from the list. If the timer was started by `OSTmrStart()` with a 'periodic' option then, the `OS_TMR` structure is placed in the `OSTmrWheelTbl[]` by calculating its new position, again using `OSTmrTime + period`. In our example, the new 'spoke' would be:

```
match = OSTmrTime + period;
match = 10 + 10;
match = 20;

spoke = match % OS_TMR_CFG_WHEEL_SIZE;
spoke = 20 % 8;
spoke = 4;
```

The use of a timer wheel basically reduces the execution time of the timer task so that it only handles a few of the timers. Of course, the worst case is such that all timers are placed in the same spoke of the timer wheel. However, statistically, this will occur rarely. It's generally recommended to keep the size of the wheel a fraction of the total number of times. In other words, you should set:

```
OS_TMR_CFG_WHEEL_SIZE <= Fraction of (OS_TMR_CFG_MAX)
```

A fraction of 2 to 8 should work well.

RAM usage (in bytes) for the timer manager is shown below:

```
2 * sizeof(INT16U)  +
1 * sizeof(INT32U)  +
3 * sizeof(POINTER) +
OS_TASK_TMR_STK_SIZE * sizeof(OS_STK) +
OS_TMR_CFG_WHEEL_SIZE * (sizeof(INT16U) + sizeof(POINTER)) +
OS_TMR_CFG_MAX * (4 * sizeof(POINTER) +
                  2 * sizeof(INT32U)  +
                  3 * sizeof(INT8U)   +
                  OS_TMR_CFG_NAME_SIZE * sizeof(INT8U))
```

Because `INT8U`s and `BOOLEAN`s are typically 1 byte, `INT16U`s are 2 bytes and `INT32U`s are 4 bytes, we can simplify the above equation as follows:

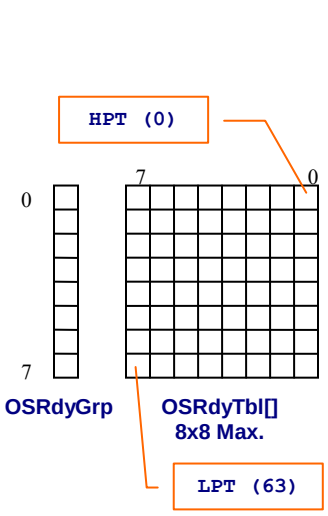
```
2 * 2  +
1 * 4  +
3 * sizeof(POINTER) +
OS_TASK_TMR_STK_SIZE * sizeof(OS_STK) +
OS_TMR_CFG_WHEEL_SIZE * (2 + sizeof(POINTER)) +
OS_TMR_CFG_MAX * (4 * sizeof(POINTER) +
                  2 * 4  +
                  3      +
                  OS_TMR_CFG_NAME_SIZE)
```

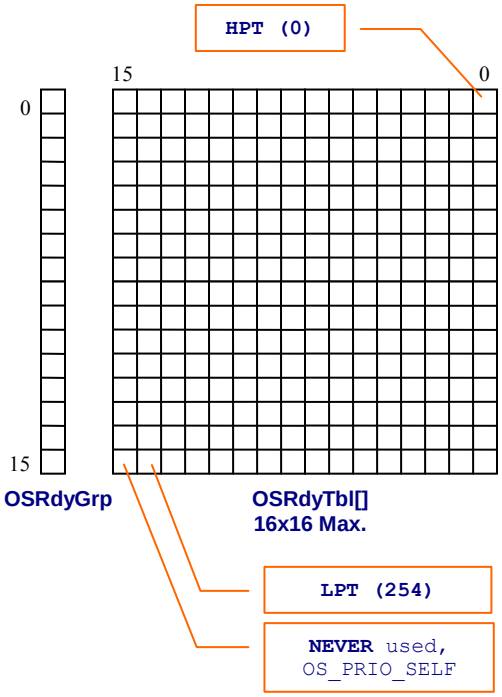
Or,

```
8                                          +
3 * sizeof(POINTER)                        +
OS_TASK_TMR_STK_SIZE * sizeof(OS_STK)      +
OS_TMR_CFG_WHEEL_SIZE * (2 + sizeof(POINTER)) +
OS_TMR_CFG_MAX * (4 * sizeof(POINTER) + 11 + OS_TMR_CFG_NAME_SIZE)
```

Support for 255 tasks (V2.80)

µC/OS-II can now support up to **255** tasks. To support up to 255 tasks, we simply increased the ready list and event wait lists to a matrix of 16x16 instead of 8x8. In fact, the actual size of the matrix (whether 8x8 or 16x16) depends on the value of `OS_LOWEST_PRIO` in `OS_CFG.H`. If `OS_LOWEST_PRIO` is less than or equal to `63`, we use an 8x8 matrix and thus **µC/OS-II** behaves exactly the same as it used to. If you specify a value for `OS_LOWEST_PRIO` to be greater than `63`, we use the 16x16 matrix as show below.





OS_LOWEST_PRIO <= 63 **OS_LOWEST_PRIO > 63**

You should note that the actual size of the matrix depends on `OS_LOWEST_PRIO`. For example, if `OS_LOWEST_PRIO` is `10` then the matrix is actually 2x8 (two bytes of 8 bits). Similarly, if `OS_LOWEST_PRIO` is set to `47`, the matrix will be 6x8. When `OS_LOWEST_PRIO` is above `63`, we use 16-bit wide entries. For example, if you specify `OS_LOWEST_PRIO` to be `100` then the matrix will be 7x16 (7 entries of 16 bits each). You **CANNOT** set `OS_LOWEST_PRIO` to `255` because this value is reserved for `OS_PRIO_SELF`.

New Files

APP_CFG.H (Added in V2.80)

We now assume the presence of a file called APP_CFG.H which is declared in your application. The purpose of this file is to assign task priorities, stack sizes and other configuration information for your application.

OS_CFG_R.H (Added in V2.70)

This file is 'reference' file so that you don't have to create this file from scratch. OS_CFG_R.H has been added in V2.70 and is found in the 'Source' directory of the microprocessor independent portion of **µC/OS-II**. It is recommended that you *copy* OS_CFG_R.H to OS_CFG.H of your project directory.

OS_TMR.C (Added in V2.81, revised in V2.83)

We added a timer manager function in **µC/OS-II**. You can now define any number of timers. The timers can be periodic or one-shots. A user definable function can be called when the timer expires. One such function is definable for each timer in your application.

New Port Files

OS_DBG.C (Added in V2.62 but renamed from OS_DEBUG.C in V2.70)
OS_DBG_R.C (Added in V2.70)

This file should be placed in the same directory as OS_CPU_C.C, OS_CPU.H and OS_CPU_A.ASM of the port you are using. OS_DBG.C defines a series of variables that are placed in ROM (code space). These variables are used by some Kernel Aware Debuggers to get information about **µC/OS-II** and its configuration. If you DON'T use a Kernel Aware Debugger that requires this file, you DON'T need to have it. Check you Kernel Aware Debugger documentation. OS_DBG.C used to be called OS_DEBUG.C in V2.62.

OS_DBG_R.C is a 'reference' file so that you don't have to create this file from scratch. OS_DBG_R.C has been added in V2.70 and is found in the 'Source' directory of the microprocessor independent portion of **µC/OS-II**.

Changes

`uCOS_II.H` (Changed in V2.70, V2.80 and V2.84)

This file now includes `#include` statements to include `APP_CFG.H`, `OS_CPU.H` and `OS_CFG.H`. This allows you to compile **µC/OS-II** without the needs of any other library functions.

Chaned error codes to make them more consistent. Specifically, all error codes start with `OS_ERR_`. The old error codes have been kept for backward compatibility but you should now use and check for the new error codes.

`Time delays and Timeouts` (Changed in V2.87)

All time delays and timeouts are now implemented using an unsigned 32-bit variable. This actually simplified `OSTimeDlyHMSM()` and allowed `OSTimeDlyResume()` to work with any delay. Of course, this means that additional storage is needed in the tasks `OS_TCB`s (2 extra bytes) but that should not be a problem with most applications. It turns out that close to 90% of µC/OS-II users use 32-bit CPUs.

`Names of objects stored as pointers` (Changed in V2.87)

Names of objects were previously stored in RAM inside the different kernel objects. Specifically, RAM storage was allocated in the OS_TCB, for example, to store the name of a task. As of V2.87, all such names are now referenced using pointers. This drastically reduces the amount of RAM needed to store ASCII names since names are now typically allocated by the compiler as constant strings and thus placed in ROM. This was done to reduce the amount of RAM needed (a processor typically has more ROM than RAM) and also to lift the limit of the length of a kernel object name.

`Stack check usage in number of entries` (Changed in V2.92)

To be consistent with `OSTaskCreate()` and `OSTaskCreateExt()`, `OSTaskStkChk()` reports stack usage in number of stack entries (or elements) as opposed to number of bytes. To convert number of entries in number of bytes, multiply the value by `sizeof(OS_STK)`.

New #define Constants and Macros

OS_APP_HOOKS_EN (OS_CFG.H, V2.85)

This constant specifies whether **µC/OS-II**'s hook functions will call application defined hooks. Specifically, when set to 1 …

The µC/OS-II hook …	Calls the Application-define hook …
OSTaskCreateHook()	App_TaskCreateHook()
OSTaskDelHook()	App_TaskDelHook()
OSTaskIdleHook()	App_TaskIdleHook()
OSTaskStatHook()	App_TaskStatHook()
OSTaskSwHook()	App_TaskSwHook()
OSTCBInitHook()	App_TCBInitHook()
OSTimeTickHook()	App_TimeTickHook()

OS_ARG_CHK_EN (OS_CFG.H, V2.04)

This constant is used to specify whether argument checking will be performed at the beginning of most of **µC/OS-II** services. You should always choose to turn this feature on (when set to 1) unless you need to get the best performance possible out of **µC/OS-II** or, you need to reduce code size.

OS_CRITICAL_METHOD #3 (OS_CPU.H, V2.04)

This constant specifies the method used to disable and enable interrupts during critical sections of code. Prior to V2.04, OS_CRITICAL_METHOD could be set to either 1 or 2. In V2.04, I added a local variable (i.e. cpu_sr) in most function calls to save the processor status register which generally holds the state of the interrupt disable flag(s). You would then declare the two critical section macros as follows:

```
#define OS_ENTER_CRITICAL()   (cpu_sr = OS_CPUSR_Save())
#define OS_EXIT_CRITICAL()    (OS_CPU_SR_Restore(cpu_sr))
```

Note that the functions OS_CPU_SR_Save() and OS_CPU_SR_Restore() would be written in assembly language and would typically be found in OS_CPU_A.ASM (or equivalent).

OS_DEBUG_EN (OS_CFG.H, V2.60)

This constant is used to enable ROM constants used for debugging using a kernel aware debugger. The constants are found in OS_CORE.C.

OS_EVENT_MULTI_EN (OS_CFG.H, V2.86)

This constant determines whether the code to support pending on multiple events will be enabled (1) or not (0). This constant thus enables code for the function OSEventPendMulti(). This #define was added in V2.86.

OS_EVENT_NAME_EN (OS_CFG.H, V2.60 and changed in V2.87)

This constant determines whether names can be assigned to a semaphore, a mutex, a mailbox or a message queue. If OS_EVENT_NAME_EN is set to 0, this feature is disabled.

OS_FLAG_EN (OS_CFG.H, V2.51)

This constant is used to specify whether you will enable (when 1) code generation for the event flags.

OS_FLAG_NAME_EN (OS_CFG.H, V2.60 and changed in V2.87)

This constant determines whether names can be assigned to event flag groups. If OS_FLAG_NAME_EN is set to 0, this feature is disabled.

OS_FLAG_WAIT_CLR_EN (OS_CFG.H, V2.51)

This constant is used to enable code generation (when 1) to allow to wait on cleared event flags.

OS_MAX_FLAGS (OS_CFG.H, V2.51)

This constant is used to determine how many event flags your application will support.

OS_MBOX_PEND_ABORT_EN (OS_CFG.H, V2.84)

This constant is used to determine whether you will enable (when 1) code generation for the OSMboxPendAbort() function.

OS_MEM_NAME_EN (OS_CFG.H, V2.60 and changed in V2.87)

This constant determines whether names can be assigned to memory partitions. If OS_MEM_NAME_EN is set to 0, this feature is disabled and no RAM is used in the OS_MEM for the memory partition.

OS_MUTEX_EN (OS_CFG.H, V2.04)

This constant is used to specify whether you will enable (when 1) code generation for mutual exclusion semaphores.

OS_Q_PEND_ABORT_EN (OS_CFG.H, V2.84)

This constant is used to determine whether you will enable (when 1) code generation for the OSQPendAbort() function.

OS_SEM_PEND_ABORT_EN (OS_CFG.H, V2.84)

This constant is used to determine whether you will enable (when 1) code generation for the OSSemPendAbort() function.

OS_TASK_NAME_EN (OS_CFG.H, V2.60 and changed in V2.87)

This constant determines whether names can be assigned to tasks. If OS_TASK_NAME_EN is set to 0, this feature is disabled and no RAM is used in the OS_TCB for the task name.

OS_TASK_PROFILE_EN (OS_CFG.H, V2.60)

This constant allows variables to be allocated in each task's OS_TCB that hold performance data about each task. Specifically, if OS_TASK_PROFILE_EN is set to 1, each task will have a variable to keep track of the number of context switches, the task execution time, the number of bytes used by the task and more.

OS_TASK_STAT_STK_CHK_EN (OS_CFG.H, V2.60)

This constant allows the statistic task to determine the actual stack usage of each active task. If OS_TASK_STAT_EN is set to 0 (the statistic task is not enabled), you can call OS_TaskStatStkChk() yourself from one of your tasks. . If OS_TASK_STAT_EN is set to 1, stack sizes will be determined every second.

OS_TASK_SW_HOOK_EN (OS_CFG.H, V2.60)

Normally, **µC/OS-II** requires that you have a context switch hook function called OSTaskSwHook(). When set to 0, this constant allows you to omit OSTaskSwHook() from your code. This configuration constant was added to reduce the amount of overhead during a context switch in applications that doesn't require the context switch hook. Of course, you will also need to remove the calls to OSTaskSwHook() from OSTaskStartHighRdy(), OSCtxSw() and OSIntCtxSw() in OS_CPU_A.ASM.

OS_TASK_TMR_STK_SIZE (OS_CFG.H, V2.81)

This #define determines the stack size (in number of stack-size elements, i.e. OS_STK) of the timer task. The size of the timer task's stack greatly depends on the processor architecture and the functions that are called when timers expire. Note that if you set OS_TMR_EN to 0 in OS_CFG.H then the value you set for OS_TASK_TMR_STK_SIZE is irrelevant because the timer functionality would be disabled.

OS_TICK_STEP_EN (OS_CFG.H, V2.60)

µC/OS-View can now 'halt' **µC/OS-II**'s tick processing and allow you to issue 'step' commands from **µC/OS-View**. In other words, **µC/OS-View** can prevent **µC/OS-II** from calling OSTimeTick() so that timeouts and time delays are no longer processed. However, though a keystroke from **µC/OS-View**, you can execute a single tick at a time. If OS_TIME_TICK_HOOK_EN (see below) is set to 1, OSTimeTickHook() is still executed at the regular tick rate in case you have time critical items to take care of in your application.

OS_TIME_TICK_HOOK_EN (OS_CFG.H, V2.60)

Normally, **µC/OS-II** requires the presence of a function called OSTimeTickHook() which is called at the very beginning of the tick ISR. When set to 0, this constant allows you to omit OSTimeTickHook() from your code. This configuration constant was added to reduce the amount of overhead during a tick ISR in applications that doesn't require this hook.

OS_TMR_EN (OS_CFG.H, V2.81)

This #define enables (when set to 1) or disables (when set to 0) the timer management code.

OS_TMR_CFG_MAX (OS_CFG.H, V2.81)

This #define determines the maximum number of timers that can exist in the application. If OS_TMR_EN is set to 1, you should declare **AT LEAST** two (**2**) timers.

OS_TMR_CFG_NAME_EN (OS_CFG.H, V2.81 and changed in V2.87)

This #define determines whether names can be assigned to timers.

OS_TMR_CFG_WHEEL_SIZE (OS_CFG.H, V2.81)

This #define determines the number of entries in the timer wheel. This value should be a number between 2 and 1024. Timer management overhead is somewhat determined by the size of the wheel. A large number of entries might reduce the overhead for timer management but would require more RAM. Each entry requires a pointer and a count (16-bit value). We recommend a number that is NOT a multiple of the tick rate. If your application has many timers then it's recommended that you have a high value. As a starting value, you could use OS_TMR_CFG_MAX / 4.

OS_TMR_CFG_TICKS_PER_SEC (OS_CFG.H, V2.81)

This #define determines the rate at which timers will be updated. You would typically set to a fraction of the tick rate (i.e. OS_TICKS_PER_SEC). We recommend that you set OS_TMR_CFG_TICKS_PER_SEC to 10 (i.e. 10 Hz).

The following table summarizes some of the new `#define` constants in `OS_CFG.H` which were all added in since V2.00.

`#define` name in `OS_CFG.H`	**... to enable the function(s):**

`OS_APP_HOOKS_EN`	`App_TaskCreateHook()`
	`App_TaskDelHook()`
	`App_TaskIdleHook()`
	`App_TaskStatHook()`
	`App_TaskSwHook()`
	`App_TCBInitHook()`
	`App_TimeTickHook()`

`OS_DEBUG_EN`	Enable debug constants in `OS_CORE.C`. If you are using a kernel aware debugger, you should enable this feature.

`OS_EVENT_NAME_EN`	`OSEventNameGet()`
	`OSEventNameSet()`
	And, to allow naming semaphores, mutexes, mailboxes and message queues.

`OS_EVENT_MULTI_EN`	`OSEventPendMulti()`
`OS_FLAG_ACCEPT_EN`	`OSFlagAccept()`
`OS_FLAG_DEL_EN`	`OSFlagDel()`

`OS_FLAG_NAME_EN`	`OSFlagNameGet()`
	`OSFlagNameSet()`
	And, to allow naming event flag groups.

`OS_FLAG_QUERY_EN`	`OSFlagQuery()`
`OS_MBOX_ACCEPT_EN`	`OSMboxAccept()`
`OS_MBOX_DEL_EN`	`OSMboxDel()`
`OS_MBOX_PEND_ABORT_EN`	`OSMboxPendAbort()`
`OS_MBOX_POST_EN`	`OSMboxPost()`
`OS_MBOX_POST_OPT_EN`	`OSMboxPostOpt()`
`OS_MBOX_QUERY_EN`	`OSMBoxQuery()`
`OS_MEM_NAME_EN`	`OSMemNameGet()` `OSMemNameSet()`
`OS_MEM_QUERY_EN`	`OSMemQuery()`
`OS_MUTEX_ACCEPT_EN`	`OSMutexAccept()`
`OS_MUTEX_DEL_EN`	`OSMutexDel()`
`OS_MUTEX_QUERY_EN`	`OSMutexQuery()`

OS_Q_ACCEPT_EN	OSQAccept()
OS_Q_DEL_EN	OSQDel()
OS_Q_FLUSH_EN	OSQFlush()
OS_Q_PEND_ABORT_EN	OSQPendAbort()
OS_Q_POST_EN	OSQPost()
OS_Q_POST_FRONT_EN	OSQPostFront()
OS_Q_POST_OPT_EN	OSQPostOpt()
OS_Q_QUERY_EN	OSQQuery()
OS_SEM_ACCEPT_EN	OSSemAccept()
OS_SEM_DEL_EN	OSSemDel()
OS_SEM_PEND_ABORT_EN	OSSemPendAbort()
OS_SEM_QUERY_EN	OSSemQuery()
OS_SEM_SET_EN	OSSemSet()
OS_TASK_NAME_EN	OSTaskNameGet() OSTaskNameSet() And, to allow naming tasks.
OS_TASK_PROFILE_EN	To allocate variables in OS_TCB for performance monitoring of each task at run-time.
OS_TASK_QUERY_EN	OSTaskQuery()
OS_TASK_STAT_STK_CHK_EN	OS_TaskStatStkChk()
OS_TASK_SW_HOOK_EN	OSTaskSwHook()
OS_TASK_TMR_STK_SIZE	Size in OS_STK elements of the Timer Management task.
OS_TICK_STEP_EN	To support the stepping feature of **µC/OS-View**.
OS_TIME_DLY_HMSM_EN	OSTimeDlyHMSM()
OS_TIME_DLY_RESUME_EN	OSTimeDlyResume()
OS_TIME_GET_SET_EN	OSTimeGet() and OSTimeSet()
OS_TIME_TICK_HOOK_EN	OSTimeTickHook()
OS_TMR_EN	Enables (1) or Disables (0) timer management functions.
OS_TMR_CFG_MAX	Determines the maximum number of timers in your application.
OS_TMR_CFG_NAME_EN	Determines whether names can be assigned to timers.
OS_TMR_CFG_WHEEL_SIZE	Determines the size of the timer wheel (in number of entries).
OS_TMR_CFG_TICKS_PER_SEC	Rate at which timers will be updated (Hz)
OS_SCHED_LOCK_EN	OSSchedLock() and OSSchedUnlock()

New Data Types

`OS_CPU_SR` (OS_CPU.H, V2.04)

This data type is used to specify the size of the CPU status register which is used in conjunction with `OS_CRITICAL_METHOD` #3 (see above). For example, if the CPU status register is 16-bit wide then you would `typedef` accordingly.

`OS_FLAGS` (uCOS_II.H, V2.51)

This data type determines how many bits an event flag group will have. You can thus `typedef` this data type to either `INT8U`, `INT16U` or `INT32U` to give event flags either 8, 16 or 32 bits, respectively.

`OS_TMR` (uCOS_II.H, V2.81)

This data type is a timer object which contains information about a specific timer that you started (see `OS_TMR.C`).

New Hook Functions

`void OSInitHookBegin(void)` (OS_CPU.C, V2.04)

This function is called at the very beginning of `OSInit()` to allow for port specific initialization BEFORE **µC/OS-II** gets initialized.

`void OSInitHookEnd(void)` (OS_CPU.C, V2.04)

This function is called at the end of `OSInit()` to allow for port specific initialization AFTER **µC/OS-II** gets initialized.

`void OSTCBInitHook(OS_TCB *ptcb)` (OS_CPU.C, V2.04)

This function is called by `OSTCBInit()` during initialization of the TCB assigned to a newly created task. It allows port specific initialization of the TCB.

`void OSTaskIdleHook(void)` (OS_CPU.C, V2.51)

This function is called by `OSTaskIdle()`. This allows you to STOP the CPU and thus reduce power consumption while there is nothing to do.

New Functions

The following table provides a list of all the new functions (i.e. services) that YOUR application can call. The list also includes functions that used to exist but, if these are in this list, it's because their API may have changed.

Refer to the ***Reference Manual*** of the current release for a description of these functions.

Function Name	File	Enabled By …
OSEventNameGet()	OS_CORE.C	OS_EVENT_NAME_EN
OSEventNameSet()	OS_CORE.C	OS_EVENT_NAME_EN
OSEventPendMulti()	OS_CORE.C	OS_EVENT_MULTI_EN
OSFlagAccept()	OS_FLAG.C	OS_FLAG_EN && OS_FLAG_ACCEPT_EN
OSFlagCreate()	OS_FLAG.C	OS_FLAG_EN
OSFlagDel()	OS_FLAG.C	OS_FLAG_EN && OS_FLAG_DEL_EN
OSFlagNameGet()	OS_FLAG.C	OS_FLAG_NAME_EN
OSFlagNameSet()	OS_FLAG.C	OS_FLAG_NAME_EN
OSFlagPend()	OS_FLAG.C	OS_FLAG_EN
OSFlagPendGetFlagsRdy()	OS_FLAG.C	OS_FLAG_EN
OSFlagPost()	OS_FLAG.C	OS_FLAG_EN
OSFlagQuery()	OS_FLAG.C	OS_FLAG_EN
OSMboxDel()	OS_MBOX.C	OS_MBOX_EN && OS_MBOX_DEL_EN
OSMboxPendAbort()	OS_MBOX.C	OS_MBOX_EN && OS_MBOX_PEND_ABORT_EN
OSMboxPostOpt()	OS_MBOX.C	OS_MBOX_EN && OS_MBOX_POST_OPT_EN
OSMutexAccept()	OS_MUTEX.C	OS_MUTEX_EN && OS_MUTEX_ACCEPT_EN
OSMutexCreate()	OS_MUTEX.C	OS_MUTEX_EN
OSMutexDel()	OS_MUTEX.C	OS_MUTEX_EN && OS_MUTEX_DEL_EN
OSMutexPend()	OS_MUTEX.C	OS_MUTEX_EN
OSMutexPost()	OS_MUTEX.C	OS_MUTEX_EN
OSMutexQuery()	OS_MUTEX.C	OS_MUTEX_EN && OS_MUTEX_QUERY_EN
OSQAccept()	OS_Q.C	OS_Q_EN && OS_Q_ACCEPT_EN
OSQDel()	OS_Q.C	OS_Q_EN && OS_Q_DEL_EN
OSQFlush()	OS_Q.C	OS_Q_EN && OS_Q_FLUSH_EN
OSQPend()	OS_Q.C	OS_Q_EN
OSQPendAbort()	OS_Q.C	OS_Q_EN && OS_Q_PEND_ABORT_EN
OSQPost()	OS_Q.C	OS_Q_EN
OSQPostFront()	OS_Q.C	OS_Q_EN && OS_Q_POST_FRONT_EN
OSQPostOpt()	OS_Q.C	OS_Q_EN && OS_Q_POST_OPT_EN
OSSafetyCriticalStart()	OS_CORE.C	OS_SAFETY_CRITICAL_IEC61508
OSSemDel()	OS_SEM.C	OS_SEM_EN && OS_SEM_DEL_EN
OSSemPendAbort()	OS_SEM.C	OS_SEM_EN && OS_SEM_PEND_ABORT_EN
OSSemSet()	OS_SEM.C	OS_SEM_EN && OS_SEM_SET_EN
OSTaskNameGet()	OS_TASK.C	OS_TASK_NAME_EN
OSTaskNameSet()	OS_TASK.C	OS_TASK_NAME_EN
OSTmrGetName()	OS_TMR.C	OS_TMR_EN
OSTmrGetRemain()	OS_TMR.C	OS_TMR_EN
OSTmrStart()	OS_TMR.C	OS_TMR_EN
OSTmrStop()	OS_TMR.C	OS_TMR_EN
OSTmrSignal()	OS_TMR.C	OS_TMR_EN

References

µC/OS-II, The Real-Time Kernel, 2nd Edition
Jean J. Labrosse
CMP Books, 2002
ISBN 1-57820-103-9

Contacts

Micriμm
949 Crestview Circle
Weston, FL 33327
954-217-2036
954-217-2037 (FAX)
e-mail: Jean.Labrosse@Micrium.com
WEB: www.Micrium.com

CMP Books, Inc.
1601 W. 23rd St., Suite 200
Lawrence, KS 66046-9950
(785) 841-1631
(785) 841-2624 (FAX)
WEB: http://www.cmpbooks.com
e-mail: rdorders@cmpbooks.com