Micrium®
Embedded Software

µC/OS-III® Performance Optimization ARM Cortex-M (Part 1)

June 23, 2017 V.1.0

Introduction

Micrium’s µC/OS-III is a real-time operating system (RTOS) that has been optimized for use in embedded applications.

This first part of a two-part paper will explore how you can configure and use µC/OS-III to its fullest potential. The first part will also look at things you can do to improve performance without losing any of the instrumentation capabilities built-into µC/OS-III.

The µC/OS-III instrumentation is described in a two-part blog:

https://www.micrium.com/exploring-ucos-iiis-built-in-performance-measurements-part-i/

https://www.micrium.com/exploring-ucos-iii-built-in-performance-measurements-part-ii/

We’ll assume you are using a Cortex-M in our examples, but some of these improvements can be made using other CPU architectures.

This document assumes that you are familiar with µC/OS-III.

Turn on Full Compilier Optimization

The easiest and possibly most obvious optimization you can do is to turn on compiler optimization. This will ensure that the µC/OS-III APIs will be as efficient as possible. The performance gains are highly compiler- specific, and you should experiment with different optimization levels to see just how much performance can be improved.

For µC/Probe to work, you must DISABLE compiler optimization for `os_dbg.c`. This ensures that variables needed for µC/OS-III kernel awareness don't get optimized out.

Use the Latest Version of µC/OS-III

In µC/OS-III V3.06.00, we removed the multipend feature to improve performance. Not only did we improve performance by some 10 to 15% on the overall µC/OS-III code but we also reduced the code and RAM sizes.

Multipend can easily be duplicated using an extra event flag group. This allows a task to pend on up to 32 different semaphores, event flags and/or message queues. In fact, the mutipend implementation in previous versions of µC/OS-III was not able to pend on event flags; just semaphores and message queues. It's also possible to wake up the multipended task when multiple objects have been posted; previously, the multipended task was made ready-to-run as soon as one of the objects was posted to. Figure 1 shows an example on how to pend on multiple objects. A 32-bit event flag group is used in the example, but you don't need to pend on that many objects.

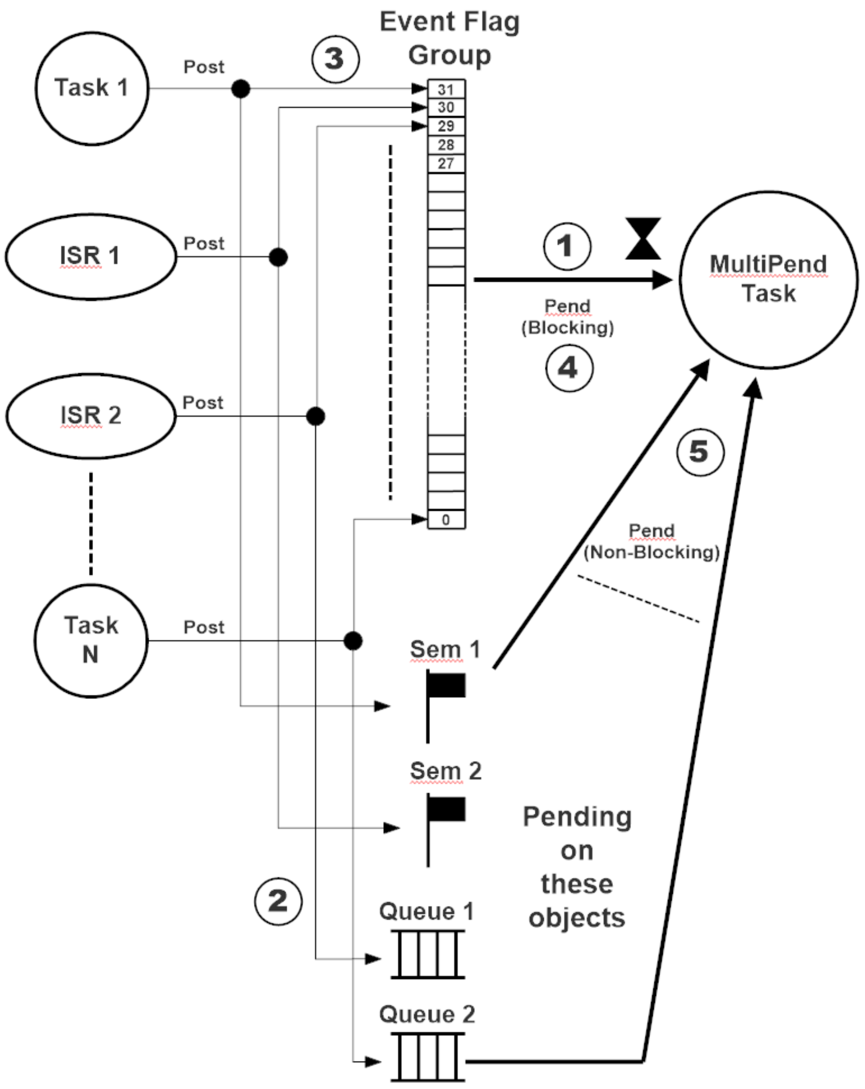


Figure 1: µC/OS-III V3.06 New Way to Pend on Multiple Objects

1) The task that wants to pend on multiple objects first performs a blocking pend on an event flag group. You can pend on "any" or "all" object(s) to be posted. You can of course specify a timeout as needed.

2) A task or ISR posts to a kernel object (semaphore, event flag or message queue).

3) The posting task must then post to the event flag group and specify the bit that corresponds to that object. Using Figure 1, Task 1 first posts to *Sem 1* and then to bit-31 of the event flag group. ISR 2 posts to *Queue 1* and then posts to bit-29 of the event flag group.

4) When the event flag group is posted and the condition of the pend is satisfied, the task simply determines which bit was posted to and can thus know which object has been posted. You can use the `CPU_CntLeadZeros()` function to give you the *ID* of the object posted. For example, 0 would indicate bit-31 (*Sem 1*), 1 would indicate bit-30 (*Sem 2*), 2 would indicate bit-29 (*Queue 1*) and so on. In other words: `31 - CPU_CntLeadZeros()`.

Alternatively, you can use the `CPU_CntTrailZeros()` to determine the first object counting from bit-0.

5) Once you know which object has been posted to, you can now do a non-blocking pend on that object.

To summarize, the sequence for the posting tasks or ISRs is:

- Post to Object
- Post to associated bit in Event Flag Group

For the task that is multipending:

- Blocking Pend on Event Flag Group with optional timeout
- Determine which object(s) was posted to
- Non-Blocking pend on posted to object

Keep the Number of Task Priorities At or Below 32 – OS_CFG_PRIO_MAX

The scheduling algorithm in µC/OS-III uses a bitmap table `OSPrioTbl[]` (see `os_prio.c`) to determine the highest-priority task that is ready to run. The size of the table depends on the natural word size of the CPU and the number of different priority levels you want to allow in your application. As shown in Figure 2, the natural word size of the Cortex-M is 32 bits; thus, each entry in `OSPrioTbl[]` maps 32 priority levels.

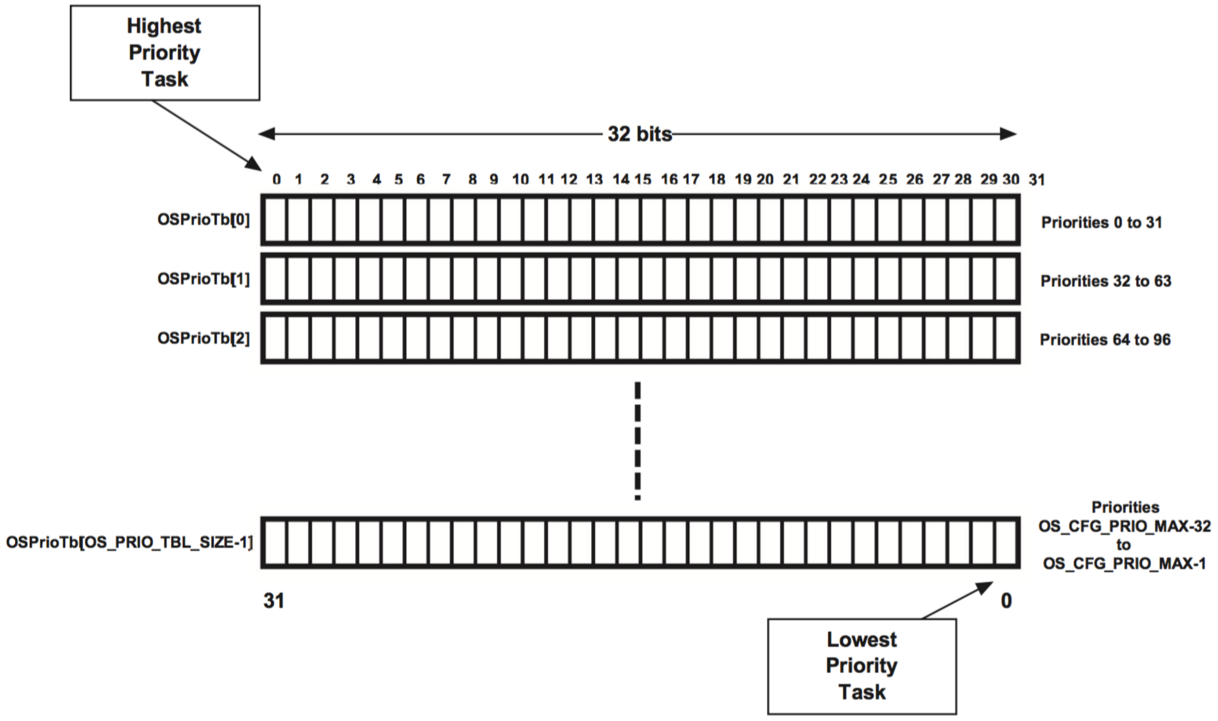


Figure 2: µC/OS-III Priority Table

The simplified code to find the highest priority that has at least one task ready to run is shown below.

os_prio.c:
```
OS_PRIO  OS_PrioGetHighest (void)
{
    CPU_DATA  *p_tbl;
    OS_PRIO    prio;


    prio  = 0u;
    p_tbl = &OSPrioTbl[0];
#if (OS_CFG_PRIO_MAX > 32)
    while (*p_tbl == 0u) {
        prio += 32;
        p_tbl++;
    }
#endif
    prio += (OS_PRIO)CPU_CntLeadZeros(*p_tbl);

    return (prio);
}
```

You will notice that the loop is eliminated if `OS_CFG_PRIO_MAX` (see os_cfg.h) is set to 32 or less, and `OS_PrioGetHighest()` is slightly faster in that case. In fact, in real life, we can actually make this function slightly faster yet, as follows:

```
OS_PRIO  OS_PrioGetHighest (void)
{
#if (OS_CFG_PRIO_MAX > 32)
    CPU_DATA  *p_tbl;
    OS_PRIO    prio;


    prio  = 0u;
    p_tbl = &OSPrioTbl[0];
while (*p_tbl == 0u) {
        prio += 32;
        p_tbl++;
    }
prio += (OS_PRIO)CPU_CntLeadZeros(*p_tbl);

    return (prio);
#else
    return (CPU_CntLeadZeros(OSPrioTbl[0]));
#endif
}
```

Summary:

Set …	In …	Saves …
`#define OS_CFG_PRIO_MAX 32u`	`os_cfg.h`	`~20 CPU instructions`

Use the Assembly Language Version of Count Leading Zeros (CLZ) – CPU_CFG_LEAD_ZEROS_ASM_PRESENT

The Cortex-M has an assembly language instruction that can optimize the `CPU_CntLeadZeros()` used in `os_prio.c`. The port file for the Cortex-M makes use of this feature by defining the `#define` constant `CPU_CFG_LEAD_ZEROS_ASM_PRESENT` in `cpu_cfg.h` as shown below:

```
cpu_cfg.h:
       #if 1
       #define  CPU_CFG_LEAD_ZEROS_ASM_PRESENT
       #endif

os_prio.c:
OS_PRIO  OS_PrioGetHighest (void)
{
#if (OS_CFG_PRIO_MAX > 32)
    CPU_DATA  *p_tbl;
    OS_PRIO    prio;


    prio  = 0u;
    p_tbl = &OSPrioTbl[0];
    while (*p_tbl == 0u) {
        prio += 32;
        p_tbl++;
    }
    prio += (OS_PRIO)CPU_CntLeadZeros(*p_tbl);

    return (prio);
#else
    return (CPU_CntLeadZeros(OSPrioTbl[0]));
#endif
}
```

This will invoke `CPU_CntLeadZeros()` declared in `cpu_a.asm` instead of a slower and thus less efficient C implementation found in `cpu_core.c`.

```
CPU_CntLeadZeros:
        CLZ     R0, R0                          ; Count leading zeros
        BX      LR
```

Summary:

Define …	In …	Saves …
`#define CPU_CFG_LEAD_ZEROS_ASM_PRESENT`	`cpu_cfg.h`	~ 20 CPU instructions

Disabling Round-Robin Scheduling – OS_CFG_SCHED_ROUND_ROBIN_EN

Although useful for some applications, round-robin scheduling is typically not used in most embedded systems.I If that's the case in your application, make sure this feature is disabled. When enabled, round-robin scheduling consumes between 50 and 100 CPU instructions, so if you don't need that feature, it's better to leave it turned off.

Summary:

Set …	In …	Saves …
`#define OS_CFG_SCHED_ROUND_ROBIN_EN DEF_DISABLED`	`os_cfg.h`	~50 to 100 instructions

Use Task Semaphores

Signaling a task using a semaphore is a very popular method of synchronization, and in µC/OS-III, each task has its own built-in semaphore. This feature not only simplifies code, but is also more efficient than using a separate semaphore object. In other words, there is no need to create a semaphore object if all you want to do is notify a task that an event occurred. The semaphore, which is built into each task, is shown in Figure 3. A task semaphore is about 10% more efficient than an external semaphore.

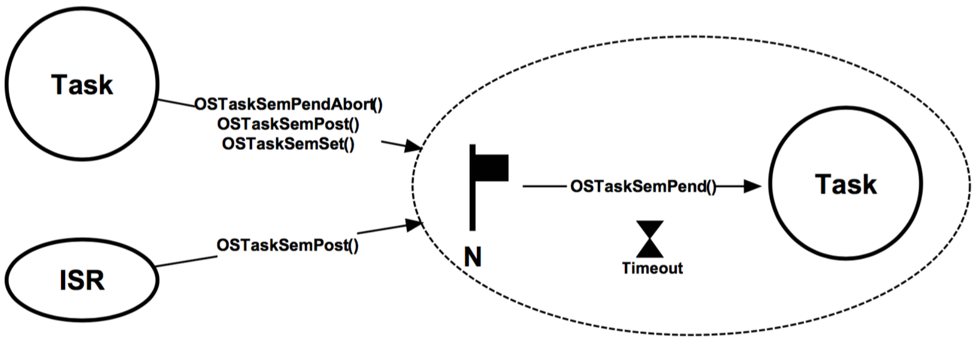


Figure 3: µC/OS-III's built-in task semaphore is more efficient than an external semaphore.

Use Task Message Queues – OS_CFG_TASK_Q_EN

If you need to send a message to a specific task, it's more efficient to use µC/OS-III's built-in message queues, as shown in Figure 4. This feature is not only more efficient, but it also avoids having to create a separate message queue object. To enable this feature, you will need to do three things:

Set `OS_CFG_TASK_Q_EN to DEF_ENABLED` in `os_cfg.h`

Specify a non-zero value for the `q_size` argument when you create the task that will be receiving these messages

Make sure you have allocated enough queue messages for the task message queue as well as other queues you have in your application. See `OS_CFG_MSG_POOL_SIZE` in `os_cfg_app.h`

A task message queue is about 10% more efficient than an external message queue

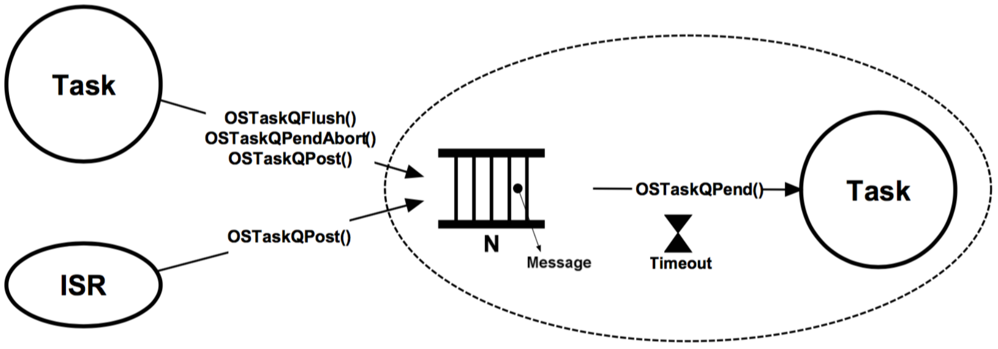


Figure 4: µC/OS-III's built-in task message queue is more efficient than an external message queue.

Summary:

`Set ...`	In ...
`#define OS_CFG_TASK_Q_EN       DEF_DISABLED`	`os_cfg.h`
`#define OS_CFG_MSG_POOL_SIZE   some_value`	`os_cfg_app.h`

`Specify ...`	In ...
A non-zero value for `q_size` in `OSTaskCreate()`	Your Code

Specify an Infinite Timeout on Pend Calls

Each µC/OS-III pend call (i.e., `OS???Pend()`) allows you to specify a timeout. If the object you are pending on has not been posted to, then the task will be inserted in that object's wait list and, if you specify a non-zero timeout, then the task will also be inserted in a list of tasks waiting to timeout. You can save a fair amount of CPU processing time if you don't need to use a timeout (i.e., wait forever). For example, specifying zero (`0`) for the `timeout` argument of `OSSemPend()` below tells µC/OS-III that the current task will only be inserted in the wait list of `MySem` and not the µC/OS-III timeout list. Depending on the number of entries in the timeout list, this could save between 20 and hundreds (100s) of CPU instructions.

YourCode.c:

```
       :
       :
       OSSemPend(&MySem,
                 0,                    // Zero means wait forever
                 OS_OPT_PEND_BLOCKING,
                 &ts,
                 &err);
       :
       :
```

Disable Argument Checking – OS_CFG_ARG_CHK_EN

Argument checking is performed on just about every API call in µC/OS-III to ensure that the caller passes proper arguments. For example:

- Did you pass a `NULL` pointer to a function that expects the address of a kernel object?
- Did you specify a proper option?
- Did you pass a value within allowable range?
- Etc.

Argument checking can easily be removed by setting the `#define` constant `OS_CFG_ARG_CHK_EN` in `os_cfg.h` to `DEF_DISABLED`. Of course, you would want to do that once you confirmed that your code doesn't pass invalid argument values to µC/OS-III APIs. Not only will you reduce processing time of APIs, but you will also reduce code size.

Summary:

Set …	**In …**	**Saves …**
`#define OS_CFG_ARG_CHK_EN DEF_DISABLED`	`os_cfg.h`	~10 to 100s CPU instructions

Turn OFF Checking of APIs Called from ISRs – OS_CFG_CALLED_FROM_ISR_CHK_EN

The following µC/OS-III APIs are not allowed to be called from an ISR:

```
OSFlagCreate()
OSFlagDel()
OSFlagPendAbort()
OSFlagPendGetFlagsRdy()
OSMemCreate()
OSMonCreate()
OSMonDel()
OSMutexCreate()
OSMutexDel()
OSMutexPend()
OSMutexPost()
OSMutexPendAbort()
OSQCreate()
OSQDel()
OSQFlush()
OSQPendAbort()
OSQPend()
OSSemCreate()
OSSemDel()
OSSemSet()
OSSemPendAbort()
OSSemPend()
OSSchedLock()
OSSchedUnlock()
OSTaskChangePrio()
OSTaskDel()
OSTaskQFlush()
OSTaskQPend()
OSTaskQPendAbort()
OSTaskSuspend()
OSTaskSemPend()
OSTaskSemPendAbort()
OSTimeDly()
OSTimeDlyHMSM()
OSTimeDlyResume()
OSTmrCreate()
OSTmrDel()
OSTmrTimeRemainGet()
OSTmrSet()
OSTmrStart()
OSTmrStateGet()
OSTmrStop()
```

Upon entry of those APIs, µC/OS-III checks the value of the interrupt nesting counter (`OSIntNestingCtr`) to ensure that it's zero. A non-zero value indicates you are incorrectly calling the function from an ISR. If you know ISRs are not calling any of the above functions, then you can remove the code that checks for this by setting `OS_CFG_CALLED_FROM_ISR_CHK_EN` to `DEF_DISABLED` in `os_cfg.h`.

Summary:

Set …	**In …**	**Saves …**
`#define OS_CFG_CALLED_FROM_ISR_CHK_EN DEF_DISABLED`	`os_cfg.h`	5 CPU instructions

Turn OFF Checking for Proper Object Type – OS_CFG_OBJ_TYPE_CHK_EN

µC/OS-III can check to ensure that you are passing the proper kernel object type to APIs. Some examples include: Are you passing a semaphore object to a semaphore management API? Are you passing a task control block to a task management API?

If you know you are passing the proper objects to µC/OS-III APIs, then you can turn this feature OFF by setting `OS_CFG_OBJ_TYPE_CHK_EN` to `DEF_DISABLED` in `os_cfg.h`.

Summary:

Set …	In …	Saves …
`#define OS_CFG_OBJ_TYPE_CHK_EN  DEF_DISABLED`	`os_cfg.h`	5 CPU instructions

Turn OFF Checking for Kernel Running – OS_CFG_INVALID_OS_CALLS_CHK_EN

Many of the µC/OS-III APIs ensure that the kernel is running in order to perform the desired function. This is done in case you call an API that is not supposed to be called prior to calling `OSStart()`.

Once you know your application is correctly calling the appropriate APIs before starting µC/OS-III, then you can turn this feature OFF by setting `OS_CFG_INVALID_OS_CALLS_CHK_EN` to `DEF_DISABLED` in `os_cfg.h`.

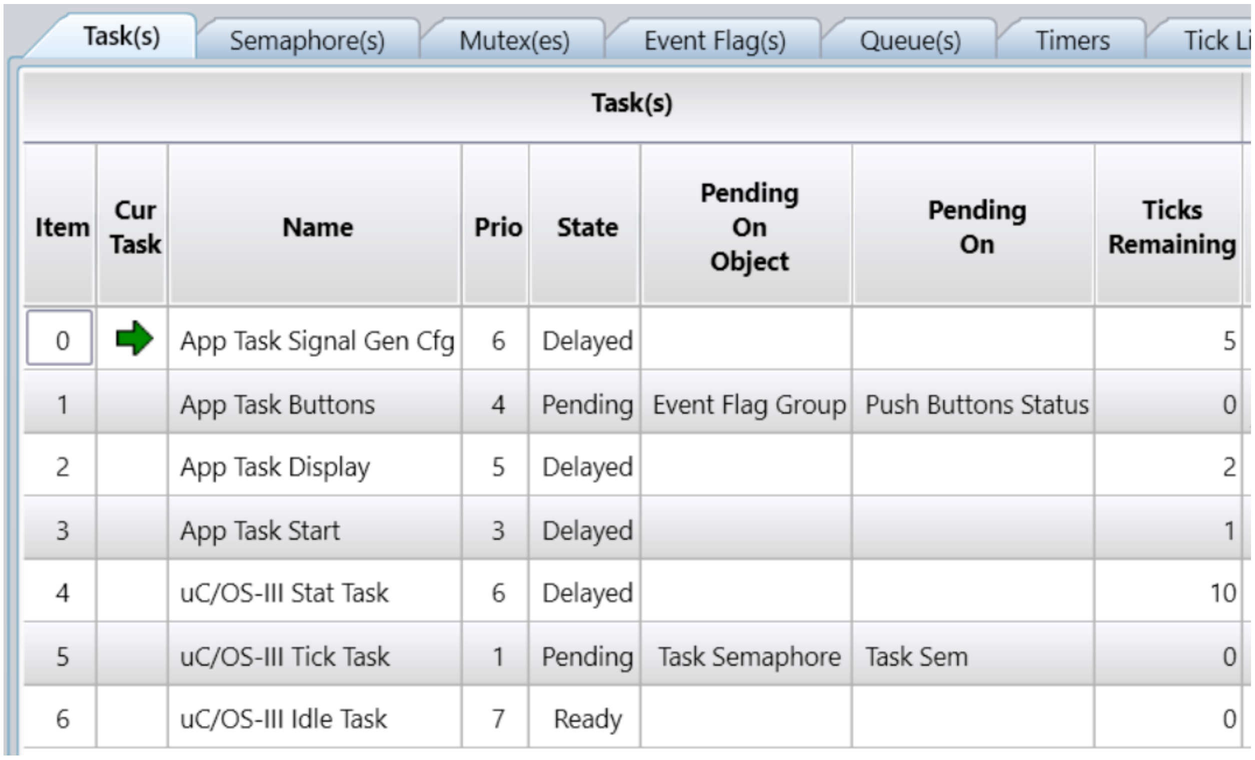
Summary:

Set …	In …	Saves …
`#define OS_CFG_INVALID_OS_CALLS_CHK_EN DEF_DISABLED`	`os_cfg.h`	5 CPU instructions

Do You Need Tick Interrupts? – OS_CFG_TASK_TICK_EN

Some low-power applications spend most of their time sleeping and are awoken exclusively by external events (i.e., interrupts) such as from a keyboard or an RTC that fires every second or even longer. In this case, you probably would not use `OSTimeDly()` or specify a timeout on pend calls, so you can completely disable the kernel tick by setting `OS_CFG_TASK_TICK_EN` to `DEF_DISABLE`.

If you disabled ticks and you accidentally used `OSTimeDly()` to suspend a task, then those tasks will stop running because there will be nothing to terminate the delay. You can always determine if you did this by looking at the kernel awareness screen in µC/Probe under the *Task(s)* tab, as shown in Figure 5. The *State* of the task will show *Delayed*.



Task(s)

Item	Cur Task	Name	Prio	State	Pending On Object	Pending On	Ticks Remaining
0	➡	App Task Signal Gen Cfg	6	Delayed			5
1		App Task Buttons	4	Pending	Event Flag Group	Push Buttons Status	0
2		App Task Display	5	Delayed			2
3		App Task Start	3	Delayed			1
4		uC/OS-III Stat Task	6	Delayed			10
5		uC/OS-III Tick Task	1	Pending	Task Semaphore	Task Sem	0
6		uC/OS-III Idle Task	7	Ready			0

Figure 5: µC/Probe Kernel Awareness

Also, when ticks are disabled, pend calls will behave as if you specified an infinite timeout regardless of whether or not you did. You can observe this by looking at the *Ticks Remaining* column and notice a non-zero value.

Disabling ticks will not only reduce power consumption but will make your application more responsive to events. In fact, you may not need ticks even if you are not overly concerned about CPU power usage. This is one of the biggest misconceptions about RTOSs; an RTOS is event-triggered, and a tick is just one of those events.

Summary:

Set …	In …
`#define OS_CFG_TASK_TICK_EN        DEF_DISABLED`	`os_cfg.h`

Use the CPU’s Natural Word Size

Processors perform best when data types use the CPU’s natural word size. For the Cortex-M, this is 32 bits. If you are willing to sacrifice RAM in exchange for performance, then you can set **ALL** the data types in `os_type.h` to `CPU_INT32U`.

Summary:

Set …	In …	Saves …
`typedef CPU_INT32U OS_???;`	`os_type.h`	1 to 2 CPU instructions / data access

Further Reading

To learn more about µC/OS-III including not only how the kernel works but also an API reference in detail, read the full µC/OS-III Documentation.

Getting Help

Don’t hesitate to contact us at Micrium: www.micrium.com.