OSTaskCreate

Description

Tasks must be created in order for µC/OS-III to recognize them as tasks. You create a task by calling `OSTaskCreate()` and by providing arguments specifying to µC/OS-III how the task will be managed. Tasks are always created in the ready-to-run state.

Tasks can be created either prior to the start of multitasking (i.e., before calling `OSStart()`), or by a running task. A task cannot be created by an ISR. A task must either be written as an infinite loop, or delete itself once completed. If the task code returns by mistake, µC/OS-III will terminate the task by calling `OSTaskDel((OS_TCB *)0, &err)`.

At Micrium, we like the “while (DEF_ON)” to implement infinite loops because, by convention, we use a `while` loop when we don’t know how many iterations a loop will do. This is the case of an infinite loop. We prefer to use `for` loops when we know how many iterations a loop will do.

Files

`os.h/os_task.c`

Prototype

```c
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)
```

Task as an infinite loop:
void MyTask (void *p_arg)
{
    /* Local variables */

    /* Do something with 'p_arg' */
    /* Task initialization */
    while (DEF_ON) { /* Task body, as an infinite loop. */
    }
}

Listing - Task as an infinite loop

Run to completion task:

void MyTask (void *p_arg)
{
    OS_ERR  err;
    /* Local variables */

    /* Do something with 'p_arg' */
    /* Task initialization */
    /* Task body (do some work) */
    OSTaskDel((OS_TCB *)0, &err);
    /* Check 'err' ... your code should never end up here! */
}

Listing - Run to completion task

Arguments

p_tcb

is a pointer to the task’s OS_TCB to use. It is assumed that storage for the TCB of the task
will be allocated by the user code. You can declare a “global” variable as follows, and
pass a pointer to this variable to OSTaskCreate():

OS_TCB MyTaskTCB;

p_name

is a pointer to an ASCII string (NUL terminated) to assign a name to the task. The name
can be displayed by debuggers or by µC/Probe.
p_task

is a pointer to the task (i.e., the name of the function that defines the task).

p_arg

is a pointer to an optional data area which is used to pass parameters to the task when it is created. When µC/OS-III runs the task for the first time, the task will think that it was invoked, and passed the argument p_arg. For example, you could create a generic task that handles an asynchronous serial port. p_arg can be used to pass task information about the serial port it will manage: the port address, baud rate, number of bits, parity, and more. p_arg is the argument received by the task shown below.

```c
void MyTask (void *p_arg)
{
    while (DEF_ON) {
        Task code;
    }
}
```

prio

is the task priority. The lower the number, the higher the priority (i.e., the importance) of the task. If OS_CFG_ISR_POST_DEFERRED_EN is set to DEF_ENABLED in os_cfg.h, the user cannot use priority 0.

Task priority must also have a lower number than OS_CFG_PRIO_MAX-1. Priorities 0, 1, OS_CFG_PRIO_MAX-2 and OS_CFG_PRIO_MAX-1 are reserved. In other words, a task should have a priority between 2 and OS_CFG_PRIO_MAX-3, inclusively.

p_stk_base

is a pointer to the task’s stack base address. The task’s stack is used to store local variables, function parameters, return addresses, and possibly CPU registers during an interrupt.
The task stack must be declared as follows:

```c
CPU_STK MyTaskStk[???];
```

The user would then pass `p_stk_base` the address of the first element of this array or, `&MyTaskStk[0]` . "??" represents the size of the stack.

The size of this stack is determined by the task’s requirements and the anticipated interrupt nesting (unless the processor has a separate stack just for interrupts). Determining the size of the stack involves knowing how many bytes are required for storage of local variables for the task itself, all nested functions, as well as requirements for interrupts (accounting for nesting).

Note that you can allocate stack space for a task from the heap but, in this case, we don’t recommend to ever delete the task and free the stack space as this can cause the heap to fragment, which is not desirable in embedded systems.

`stk_limit` is used to locate, within the task’s stack, a watermark limit that can be used to monitor and ensure that the stack does not overflow.

If the processor does not have hardware stack overflow detection, or this feature is not
implemented in software by the port developer, this value may be used for other purposes. For example, some processors have two stacks, a hardware and a software stack. The hardware stack typically keeps track of function call nesting and the software stack is used to pass function arguments. stk_limit may be used to set the size of the hardware stack as shown below.

![Stack Diagram](image)

stk_size

specifies the size of the task’s stack in number of elements. If CPU_STK is set to CPU_INT08U (see os_type.h), stk_size corresponds to the number of bytes available on the stack. If CPU_STK is set to CPU_INT16U, then stk_size contains the number of 16-bit entries available on the stack. Finally, if CPU_STK is set to CPU_INT32U, stk_size contains the number of 32-bit entries available on the stack.

q_size

A µC/OS-III task contains an optional internal message queue (if OS_CFG_TASK_Q_EN is set to DEF_ENABLED in os_cfg.h). This argument specifies the maximum number of messages that the task can receive through this message queue. The user may specify that the task is unable to receive messages by setting this argument to 0.

time_quanta
the amount of time (in clock ticks) for the time quanta when round robin is enabled. If you specify 0, then the default time quanta will be used which is the tick rate divided by 10.

_p_ext_

is a pointer to a user-supplied memory location (typically a data structure) used as a TCB extension. For example, the user memory can hold the contents of floating-point registers during a context switch.

_opt_

contains task-specific options. Each option consists of one bit. The option is selected when the bit is set. The current version of µC/OS-III supports the following options:

_OS_OPT_TASK_NONE_

specifies that there are no options.

_OS_OPT_TASK_STK_CHK_

specifies whether stack checking is allowed for the task.

_OS_OPT_TASK_STK_CLR_

specifies whether the stack needs to be cleared.

_OS_OPT_TASK_SAVE_FP_

specifies whether floating-point registers are saved. This option is only valid if the processor has floating-point hardware and the processor-specific code saves the floating-point registers.

_OS_OPT_TASK_NO_TLS_

If the caller doesn’t want or need TLS (Thread Local Storage) support for the task being created. If you do not include this option, TLS will be supported by default, assuming Micrium supports TLS for the toolchain you are using. TLS support was
added in V3.03.00.

p_err

is a pointer to a variable that will receive an error code:

OS_ERR_NONE

If the function is successful.

OS_ERR_ILLEGAL_CREATE_RUN_TIME

If OS_SAFETY_CRITICAL_IEC61508 is defined: you called this after calling OSStart() and thus you are no longer allowed to create additional kernel objects.

OS_ERR_PRIO_INVALID

If OS_CFG_ARG_CHK_EN is set to DEF_ENABLED in os_cfg.h: if prio is higher than the maximum value allowed (i.e., > OS_PRIO_MAX-1). Or, if your tried to use any in-use priority by the kernel, see the prio argument.

OS_ERR_STAT_STK_SIZE_INVALID

If the task's stack overflowed during initialization.

OS_ERR_STK_INVALID

If OS_CFG_ARG_CHK_EN is set to DEF_ENABLED in os_cfg.h: if specifying a NULL pointer for p_stk_base.

OS_ERR_STK_SIZE_INVALID

If OS_CFG_ARG_CHK_EN is set to DEF_ENABLED in os_cfg.h: if specifying a stack size smaller than what is currently specified by OS_CFG_STK_SIZE_MIN (see the os_cfg.h).

OS_ERR_STK_LIMIT_INVALID
If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if specifying a stack limit greater than or equal to the stack size.

`OS_ERR_TASK_CREATE_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if attempting to create the task from an ISR.

`OS_ERR_TASK_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if specifying a NULL pointer for `p_task`.

`OS_ERR_TCB_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if specifying a NULL pointer for `p_tcb`.

**Returned Value**

None

**Required Configuration**

None

**Callers**

Application.
Notes/Warnings

1. The stack must be declared with the `CPU_STK` type.

2. A task must always invoke one of the services provided by µC/OS-III to wait for time to expire, suspend the task, or wait on an object (wait on a message queue, event flag, mutex, semaphore, a signal or a message to be sent directly to the task). This allows other tasks to gain control of the CPU.

3. You should not use task priorities 0, 1, `OS_CFG_PRIO_MAX-2` and `OS_CFG_PRIO_MAX-1` because they are reserved for use by µC/OS-III.

Example Usage

`OSTaskCreate()` can be called from `main()` (in C), or a previously created task.
void MyTask (void *p_arg)                /* (3) The address of the task is its name*/
{
    while (DEF_ON) {
        /* Wait for an event */
        /* My task body */
    }
}

Listing - OSTaskCreate() example usage

(1) In order to create a task, you need to allocate storage for a TCB and pass a pointer to this TCB to OSTaskCreate().

(2) You can assign an ASCII name to the task by passing a pointer to an ASCII string. The ASCII string may be allocated in code space (i.e., ROM), or data space (i.e., RAM). In either case, it is assumed that the code can access that memory. The ASCII string
must be NUL terminated.

(3) You pass the address of the task to OSTaskCreate(). In C, the address of a function is simply the name of that function.

(4) To provide additional data to MyTask(), you can pass a pointer to such data. In this case, MyTask() did not need such data and therefore, a NULL pointer is passed.

(5) The user must assign a priority to the task. The priority specifies the importance of this task with respect to other tasks. A low-priority value indicates a high priority. Priority 0 is the highest priority (reserved for an internal task) and a priority up to OS_CFG_PRIO_MAX-3 can be specified (see os_cfg.h). Note that OS_CFG_PRIO_MAX-1 is also reserved for an internal task, the idle task.

(6) The next argument specifies the “base address” of the task’s stack. In this case, it is simply the base address of the array MyTaskStk[]. Note that it is possible to simply specify the name of the array. We prefer to make it clear by writing &MyTaskStk[0].

(7) This argument sets the watermark limit for stack growth. If the processor port does not use this field then you can set this value to 0.

(8) µC/OS-III also needs to know the size of the stack for the task. This allows µC/OS-III to perform stack checking at run time. This argument represents the number of CPU_STK elements, not the number of bytes.

(9) µC/OS-III allows tasks or ISRs to send messages directly to a task. This argument specifies how many such messages can be received by this task.

(10) This argument specifies how much time (in number of ticks) this task will run on the CPU before µC/OS-III will force the CPU away from this task and run the next task at the same priority (if there are more than one task at the same priority that is ready-to-run).

(11) µC/OS-III allows the user to “extend” the capabilities of the TCB by allowing passing a pointer to some memory location that could contain additional information about the task. For example, there may be a CPU that supports floating-point math and the user would likely need to save the floating-point registers during a context switch. This pointer could point to the storage area for these registers.
(12) When creating a task, options must be specified. Specifically, such options as, whether the stack of the task will be cleared (i.e., filled with 0x00) when the task is created (OS_OPT_TASK_STK_CLR), whether µC/OS-III will be allowed to check for stack usage (OS_OPT_TASK_STK_CHK), whether the CPU supports floating-point math, and whether the task will make use of the floating-point registers and therefore need to save and restore them during a context switch (OS_OPT_TASK_SAVE_FP). The options are additive.

(13) Most of µC/OS-III’s services return an error code indicating the outcome of the call. The error code is always returned as a pointer to a variable of type OS_ERR. The user must allocate storage for this variable prior to calling OSTaskCreate().

(14) It is highly recommended that the user examine the error code whenever calling a µC/OS-III function. If the call is successful, the error code will always be OS_ERR_NONE. If the call is not successful, the returned code will indicate the reason for the failure (see p_err and OS_ERR_??? in os.h).