/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* uC/OS-II

\* The Real-Time Kernel

\* TASK MANAGEMENT

\*

\* (c) Copyright 1992-2009, Micrium, Weston, FL

\* All Rights Reserved

\*

\* File : OS\_TASK.C

\* By : Jean J. Labrosse

\* Version : V2.91

\*

\* LICENSING TERMS:

\* ---------------

\* uC/OS-II is provided in source form for FREE evaluation, for educational use or for peaceful research.

\* If you plan on using uC/OS-II in a commercial product you need to contact Micri祄 to properly license

\* its use in your product. We provide ALL the source code for your convenience and to help you experience

\* uC/OS-II. The fact that the source is provided does NOT mean that you can use it without paying a

\* licensing fee.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

#ifndef OS\_MASTER\_FILE

#include <ucos\_ii.h>

#endif

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CHANGE PRIORITY OF A TASK

\*

\* Description: This function allows you to change the priority of a task dynamically. Note that the new

\* priority MUST be available.

\*

\* Arguments : oldp is the old priority

\*

\* newp is the new priority

\*

\* Returns : OS\_ERR\_NONE is the call was successful

\* OS\_ERR\_PRIO\_INVALID if the priority you specify is higher that the maximum allowed

\* (i.e. >= OS\_LOWEST\_PRIO)

\* OS\_ERR\_PRIO\_EXIST if the new priority already exist.

\* OS\_ERR\_PRIO there is no task with the specified OLD priority (i.e. the OLD task does

\* not exist.

\* OS\_ERR\_TASK\_NOT\_EXIST if the task is assigned to a Mutex PIP.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

#if OS\_TASK\_CHANGE\_PRIO\_EN > 0u

INT8U OSTaskChangePrio (INT8U oldprio,

INT8U newprio)

{

#if (OS\_EVENT\_EN)

OS\_EVENT \*pevent;

#if (OS\_EVENT\_MULTI\_EN > 0u)

OS\_EVENT \*\*pevents;

#endif

#endif

OS\_TCB \*ptcb;

INT8U y\_new;

INT8U x\_new;

INT8U y\_old;

OS\_PRIO bity\_new;

OS\_PRIO bitx\_new;

OS\_PRIO bity\_old;

OS\_PRIO bitx\_old;

#if OS\_CRITICAL\_METHOD == 3u

OS\_CPU\_SR cpu\_sr = 0u; /\* Storage for CPU status register \*/

#endif

#if OS\_ARG\_CHK\_EN > 0u

if (oldprio >= OS\_LOWEST\_PRIO) {

if (oldprio != OS\_PRIO\_SELF) {

return (OS\_ERR\_PRIO\_INVALID);

}

}

if (newprio >= OS\_LOWEST\_PRIO) {

return (OS\_ERR\_PRIO\_INVALID);

}

#endif

OS\_ENTER\_CRITICAL();

if (OSTCBPrioTbl[newprio] != (OS\_TCB \*)0) { /\* New priority must not already exist \*/

OS\_EXIT\_CRITICAL();

return (OS\_ERR\_PRIO\_EXIST);

}

if (oldprio == OS\_PRIO\_SELF) { /\* See if changing self \*/

oldprio = OSTCBCur->OSTCBPrio; /\* Yes, get priority \*/

}

ptcb = OSTCBPrioTbl[oldprio];

if (ptcb == (OS\_TCB \*)0) { /\* Does task to change exist? \*/

OS\_EXIT\_CRITICAL(); /\* No, can't change its priority! \*/

return (OS\_ERR\_PRIO);

}

if (ptcb == OS\_TCB\_RESERVED) { /\* Is task assigned to Mutex \*/

OS\_EXIT\_CRITICAL(); /\* No, can't change its priority! \*/

return (OS\_ERR\_TASK\_NOT\_EXIST);

}

#if OS\_LOWEST\_PRIO <= 63u

y\_new = (INT8U)(newprio >> 3u); /\* Yes, compute new TCB fields \*/

x\_new = (INT8U)(newprio & 0x07u);

#else

y\_new = (INT8U)((INT8U)(newprio >> 4u) & 0x0Fu);

x\_new = (INT8U)(newprio & 0x0Fu);

#endif

bity\_new = (OS\_PRIO)(1uL << y\_new);

bitx\_new = (OS\_PRIO)(1uL << x\_new);

OSTCBPrioTbl[oldprio] = (OS\_TCB \*)0; /\* Remove TCB from old priority \*/

OSTCBPrioTbl[newprio] = ptcb; /\* Place pointer to TCB @ new priority \*/

y\_old = ptcb->OSTCBY;

bity\_old = ptcb->OSTCBBitY;

bitx\_old = ptcb->OSTCBBitX;

if ((OSRdyTbl[y\_old] & bitx\_old) != 0u) { /\* If task is ready make it not \*/

OSRdyTbl[y\_old] &= (OS\_PRIO)~bitx\_old;

if (OSRdyTbl[y\_old] == 0u) {

OSRdyGrp &= (OS\_PRIO)~bity\_old;

}

OSRdyGrp |= bity\_new; /\* Make new priority ready to run \*/

OSRdyTbl[y\_new] |= bitx\_new;

}

#if (OS\_EVENT\_EN)

pevent = ptcb->OSTCBEventPtr;

if (pevent != (OS\_EVENT \*)0) {

pevent->OSEventTbl[y\_old] &= (OS\_PRIO)~bitx\_old; /\* Remove old task prio from wait list \*/

if (pevent->OSEventTbl[y\_old] == 0u) {

pevent->OSEventGrp &= (OS\_PRIO)~bity\_old;

}

pevent->OSEventGrp |= bity\_new; /\* Add new task prio to wait list \*/

pevent->OSEventTbl[y\_new] |= bitx\_new;

}

#if (OS\_EVENT\_MULTI\_EN > 0u)

if (ptcb->OSTCBEventMultiPtr != (OS\_EVENT \*\*)0) {

pevents = ptcb->OSTCBEventMultiPtr;

pevent = \*pevents;

while (pevent != (OS\_EVENT \*)0) {

pevent->OSEventTbl[y\_old] &= (OS\_PRIO)~bitx\_old; /\* Remove old task prio from wait lists \*/

if (pevent->OSEventTbl[y\_old] == 0u) {

pevent->OSEventGrp &= (OS\_PRIO)~bity\_old;

}

pevent->OSEventGrp |= bity\_new; /\* Add new task prio to wait lists \*/

pevent->OSEventTbl[y\_new] |= bitx\_new;

pevents++;

pevent = \*pevents;

}

}

#endif

#endif

ptcb->OSTCBPrio = newprio; /\* Set new task priority \*/

ptcb->OSTCBY = y\_new;

ptcb->OSTCBX = x\_new;

ptcb->OSTCBBitY = bity\_new;

ptcb->OSTCBBitX = bitx\_new;

OS\_EXIT\_CRITICAL();

if (OSRunning == OS\_TRUE) {

OS\_Sched(); /\* Find new highest priority task \*/

}

return (OS\_ERR\_NONE);

}

#endif

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CREATE A TASK

\*

\* Description: This function is used to have uC/OS-II manage the execution of a task. Tasks can either

\* be created prior to the start of multitasking or by a running task. A task cannot be

\* created by an ISR.

\*

\* Arguments : task is a pointer to the task's code

\*

\* p\_arg is a pointer to an optional data area which can be used to pass parameters to

\* the task when the task first executes. Where the task is concerned it thinks

\* it was invoked and passed the argument 'p\_arg' as follows:

\*

\* void Task (void \*p\_arg)

\* {

\* for (;;) {

\* Task code;

\* }

\* }

\*

\* ptos is a pointer to the task's top of stack. If the configuration constant

\* OS\_STK\_GROWTH is set to 1, the stack is assumed to grow downward (i.e. from high

\* memory to low memory). 'pstk' will thus point to the highest (valid) memory

\* location of the stack. If OS\_STK\_GROWTH is set to 0, 'pstk' will point to the

\* lowest memory location of the stack and the stack will grow with increasing

\* memory locations.

\*

\* prio is the task's priority. A unique priority MUST be assigned to each task and the

\* lower the number, the higher the priority.

\*

\* Returns : OS\_ERR\_NONE if the function was successful.

\* OS\_PRIO\_EXIT if the task priority already exist

\* (each task MUST have a unique priority).

\* OS\_ERR\_PRIO\_INVALID if the priority you specify is higher that the maximum allowed

\* (i.e. >= OS\_LOWEST\_PRIO)

\* OS\_ERR\_TASK\_CREATE\_ISR if you tried to create a task from an ISR.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

#if OS\_TASK\_CREATE\_EN > 0u

INT8U OSTaskCreate (void (\*task)(void \*p\_arg),

void \*p\_arg,

OS\_STK \*ptos,

INT8U prio)

{

OS\_STK \*psp;

INT8U err;

#if OS\_CRITICAL\_METHOD == 3u /\* Allocate storage for CPU status register \*/

OS\_CPU\_SR cpu\_sr = 0u;

#endif

#ifdef OS\_SAFETY\_CRITICAL\_IEC61508

if (OSSafetyCriticalStartFlag == OS\_TRUE) {

OS\_SAFETY\_CRITICAL\_EXCEPTION();

}

#endif

#if OS\_ARG\_CHK\_EN > 0u

if (prio > OS\_LOWEST\_PRIO) { /\* Make sure priority is within allowable range \*/

return (OS\_ERR\_PRIO\_INVALID);

}

#endif

OS\_ENTER\_CRITICAL();

if (OSIntNesting > 0u) { /\* Make sure we don't create the task from within an ISR \*/

OS\_EXIT\_CRITICAL();

return (OS\_ERR\_TASK\_CREATE\_ISR);

}

if (OSTCBPrioTbl[prio] == (OS\_TCB \*)0) { /\* Make sure task doesn't already exist at this priority \*/

OSTCBPrioTbl[prio] = OS\_TCB\_RESERVED;/\* Reserve the priority to prevent others from doing ... \*/

/\* ... the same thing until task is created. \*/

OS\_EXIT\_CRITICAL();

psp = OSTaskStkInit(task, p\_arg, ptos, 0u); /\* Initialize the task's stack \*/

err = OS\_TCBInit(prio, psp, (OS\_STK \*)0, 0u, 0u, (void \*)0, 0u);

if (err == OS\_ERR\_NONE) {

if (OSRunning == OS\_TRUE) { /\* Find highest priority task if multitasking has started \*/

OS\_Sched();

}

} else {

OS\_ENTER\_CRITICAL();

OSTCBPrioTbl[prio] = (OS\_TCB \*)0;/\* Make this priority available to others \*/

OS\_EXIT\_CRITICAL();

}

return (err);

}

OS\_EXIT\_CRITICAL();

return (OS\_ERR\_PRIO\_EXIST);

}

#endif

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CREATE A TASK (Extended Version)

\*

\* Description: This function is used to have uC/OS-II manage the execution of a task. Tasks can either

\* be created prior to the start of multitasking or by a running task. A task cannot be

\* created by an ISR. This function is similar to OSTaskCreate() except that it allows

\* additional information about a task to be specified.

\*

\* Arguments : task is a pointer to the task's code

\*

\* p\_arg is a pointer to an optional data area which can be used to pass parameters to

\* the task when the task first executes. Where the task is concerned it thinks

\* it was invoked and passed the argument 'p\_arg' as follows:

\*

\* void Task (void \*p\_arg)

\* {

\* for (;;) {

\* Task code;

\* }

\* }

\*

\* ptos is a pointer to the task's top of stack. If the configuration constant

\* OS\_STK\_GROWTH is set to 1, the stack is assumed to grow downward (i.e. from high

\* memory to low memory). 'ptos' will thus point to the highest (valid) memory

\* location of the stack. If OS\_STK\_GROWTH is set to 0, 'ptos' will point to the

\* lowest memory location of the stack and the stack will grow with increasing

\* memory locations. 'ptos' MUST point to a valid 'free' data item.

\*

\* prio is the task's priority. A unique priority MUST be assigned to each task and the

\* lower the number, the higher the priority.

\*

\* id is the task's ID (0..65535)

\*

\* pbos is a pointer to the task's bottom of stack. If the configuration constant

\* OS\_STK\_GROWTH is set to 1, the stack is assumed to grow downward (i.e. from high

\* memory to low memory). 'pbos' will thus point to the LOWEST (valid) memory

\* location of the stack. If OS\_STK\_GROWTH is set to 0, 'pbos' will point to the

\* HIGHEST memory location of the stack and the stack will grow with increasing

\* memory locations. 'pbos' MUST point to a valid 'free' data item.

\*

\* stk\_size is the size of the stack in number of elements. If OS\_STK is set to INT8U,

\* 'stk\_size' corresponds to the number of bytes available. If OS\_STK is set to

\* INT16U, 'stk\_size' contains the number of 16-bit entries available. Finally, if

\* OS\_STK is set to INT32U, 'stk\_size' contains the number of 32-bit entries

\* available on the stack.

\*

\* pext is a pointer to a user supplied memory location which is used as a TCB extension.

\* For example, this user memory can hold the contents of floating-point registers

\* during a context switch, the time each task takes to execute, the number of times

\* the task has been switched-in, etc.

\*

\* opt contains additional information (or options) about the behavior of the task. The

\* LOWER 8-bits are reserved by uC/OS-II while the upper 8 bits can be application

\* specific. See OS\_TASK\_OPT\_??? in uCOS-II.H. Current choices are:

\*

\* OS\_TASK\_OPT\_STK\_CHK Stack checking to be allowed for the task

\* OS\_TASK\_OPT\_STK\_CLR Clear the stack when the task is created

\* OS\_TASK\_OPT\_SAVE\_FP If the CPU has floating-point registers, save them

\* during a context switch.

\*

\* Returns : OS\_ERR\_NONE if the function was successful.

\* OS\_PRIO\_EXIT if the task priority already exist

\* (each task MUST have a unique priority).

\* OS\_ERR\_PRIO\_INVALID if the priority you specify is higher that the maximum allowed

\* (i.e. > OS\_LOWEST\_PRIO)

\* OS\_ERR\_TASK\_CREATE\_ISR if you tried to create a task from an ISR.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

#if OS\_TASK\_CREATE\_EXT\_EN > 0u

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)

{

OS\_STK \*psp;

INT8U err;

#if OS\_CRITICAL\_METHOD == 3u /\* Allocate storage for CPU status register \*/

OS\_CPU\_SR cpu\_sr = 0u;

#endif

#ifdef OS\_SAFETY\_CRITICAL\_IEC61508

if (OSSafetyCriticalStartFlag == OS\_TRUE) {

OS\_SAFETY\_CRITICAL\_EXCEPTION();

}

#endif

#if OS\_ARG\_CHK\_EN > 0u

if (prio > OS\_LOWEST\_PRIO) { /\* Make sure priority is within allowable range \*/

return (OS\_ERR\_PRIO\_INVALID);

}

#endif

OS\_ENTER\_CRITICAL();

if (OSIntNesting > 0u) { /\* Make sure we don't create the task from within an ISR \*/

OS\_EXIT\_CRITICAL();

return (OS\_ERR\_TASK\_CREATE\_ISR);

}

if (OSTCBPrioTbl[prio] == (OS\_TCB \*)0) { /\* Make sure task doesn't already exist at this priority \*/

OSTCBPrioTbl[prio] = OS\_TCB\_RESERVED;/\* Reserve the priority to prevent others from doing ... \*/

/\* ... the same thing until task is created. \*/

OS\_EXIT\_CRITICAL();

#if (OS\_TASK\_STAT\_STK\_CHK\_EN > 0u)

OS\_TaskStkClr(pbos, stk\_size, opt); /\* Clear the task stack (if needed) \*/

#endif

psp = OSTaskStkInit(task, p\_arg, ptos, opt); /\* Initialize the task's stack \*/

err = OS\_TCBInit(prio, psp, pbos, id, stk\_size, pext, opt);

if (err == OS\_ERR\_NONE) {

if (OSRunning == OS\_TRUE) { /\* Find HPT if multitasking has started \*/

OS\_Sched();

}

} else {

OS\_ENTER\_CRITICAL();

OSTCBPrioTbl[prio] = (OS\_TCB \*)0; /\* Make this priority avail. to others \*/

OS\_EXIT\_CRITICAL();

}

return (err);

}

OS\_EXIT\_CRITICAL();

return (OS\_ERR\_PRIO\_EXIST);

}

#endif

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* DELETE A TASK

\*

\* Description: This function allows you to delete a task. The calling task can delete itself by

\* its own priority number. The deleted task is returned to the dormant state and can be

\* re-activated by creating the deleted task again.

\*

\* Arguments : prio is the priority of the task to delete. Note that you can explicitely delete

\* the current task without knowing its priority level by setting 'prio' to

\* OS\_PRIO\_SELF.

\*

\* Returns : OS\_ERR\_NONE if the call is successful

\* OS\_ERR\_TASK\_DEL\_IDLE if you attempted to delete uC/OS-II's idle task

\* OS\_ERR\_PRIO\_INVALID if the priority you specify is higher that the maximum allowed

\* (i.e. >= OS\_LOWEST\_PRIO) or, you have not specified OS\_PRIO\_SELF.

\* OS\_ERR\_TASK\_DEL if the task is assigned to a Mutex PIP.

\* OS\_ERR\_TASK\_NOT\_EXIST if the task you want to delete does not exist.

\* OS\_ERR\_TASK\_DEL\_ISR if you tried to delete a task from an ISR

\*

\* Notes : 1) To reduce interrupt latency, OSTaskDel() 'disables' the task:

\* a) by making it not ready

\* b) by removing it from any wait lists

\* c) by preventing OSTimeTick() from making the task ready to run.

\* The task can then be 'unlinked' from the miscellaneous structures in uC/OS-II.

\* 2) The function OS\_Dummy() is called after OS\_EXIT\_CRITICAL() because, on most processors,

\* the next instruction following the enable interrupt instruction is ignored.

\* 3) An ISR cannot delete a task.

\* 4) The lock nesting counter is incremented because, for a brief instant, if the current

\* task is being deleted, the current task would not be able to be rescheduled because it

\* is removed from the ready list. Incrementing the nesting counter prevents another task

\* from being schedule. This means that an ISR would return to the current task which is

\* being deleted. The rest of the deletion would thus be able to be completed.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

#if OS\_TASK\_DEL\_EN > 0u

INT8U OSTaskDel (INT8U prio)

{

#if (OS\_FLAG\_EN > 0u) && (OS\_MAX\_FLAGS > 0u)

OS\_FLAG\_NODE \*pnode;

#endif

OS\_TCB \*ptcb;

#if OS\_CRITICAL\_METHOD == 3u /\* Allocate storage for CPU status register \*/

OS\_CPU\_SR cpu\_sr = 0u;

#endif

if (OSIntNesting > 0u) { /\* See if trying to delete from ISR \*/

return (OS\_ERR\_TASK\_DEL\_ISR);

}

if (prio == OS\_TASK\_IDLE\_PRIO) { /\* Not allowed to delete idle task \*/

return (OS\_ERR\_TASK\_DEL\_IDLE);

}

#if OS\_ARG\_CHK\_EN > 0u

if (prio >= OS\_LOWEST\_PRIO) { /\* Task priority valid ? \*/

if (prio != OS\_PRIO\_SELF) {

return (OS\_ERR\_PRIO\_INVALID);

}

}

#endif

OS\_ENTER\_CRITICAL();

if (prio == OS\_PRIO\_SELF) { /\* See if requesting to delete self \*/

prio = OSTCBCur->OSTCBPrio; /\* Set priority to delete to current \*/

}

ptcb = OSTCBPrioTbl[prio];

if (ptcb == (OS\_TCB \*)0) { /\* Task to delete must exist \*/

OS\_EXIT\_CRITICAL();

return (OS\_ERR\_TASK\_NOT\_EXIST);

}

if (ptcb == OS\_TCB\_RESERVED) { /\* Must not be assigned to Mutex \*/

OS\_EXIT\_CRITICAL();

return (OS\_ERR\_TASK\_DEL);

}

OSRdyTbl[ptcb->OSTCBY] &= (OS\_PRIO)~ptcb->OSTCBBitX;

if (OSRdyTbl[ptcb->OSTCBY] == 0u) { /\* Make task not ready \*/

OSRdyGrp &= (OS\_PRIO)~ptcb->OSTCBBitY;

}

#if (OS\_EVENT\_EN)

if (ptcb->OSTCBEventPtr != (OS\_EVENT \*)0) {

OS\_EventTaskRemove(ptcb, ptcb->OSTCBEventPtr); /\* Remove this task from any event wait list \*/

}

#if (OS\_EVENT\_MULTI\_EN > 0u)

if (ptcb->OSTCBEventMultiPtr != (OS\_EVENT \*\*)0) { /\* Remove this task from any events' wait lists\*/

OS\_EventTaskRemoveMulti(ptcb, ptcb->OSTCBEventMultiPtr);

}

#endif

#endif

#if (OS\_FLAG\_EN > 0u) && (OS\_MAX\_FLAGS > 0u)

pnode = ptcb->OSTCBFlagNode;

if (pnode != (OS\_FLAG\_NODE \*)0) { /\* If task is waiting on event flag \*/

OS\_FlagUnlink(pnode); /\* Remove from wait list \*/

}

#endif

ptcb->OSTCBDly = 0u; /\* Prevent OSTimeTick() from updating \*/

ptcb->OSTCBStat = OS\_STAT\_RDY; /\* Prevent task from being resumed \*/

ptcb->OSTCBStatPend = OS\_STAT\_PEND\_OK;

if (OSLockNesting < 255u) { /\* Make sure we don't context switch \*/

OSLockNesting++;

}

OS\_EXIT\_CRITICAL(); /\* Enabling INT. ignores next instruc. \*/

OS\_Dummy(); /\* ... Dummy ensures that INTs will be \*/

OS\_ENTER\_CRITICAL(); /\* ... disabled HERE! \*/

if (OSLockNesting > 0u) { /\* Remove context switch lock \*/

OSLockNesting--;

}

OSTaskDelHook(ptcb); /\* Call user defined hook \*/

OSTaskCtr--; /\* One less task being managed \*/

OSTCBPrioTbl[prio] = (OS\_TCB \*)0; /\* Clear old priority entry \*/

if (ptcb->OSTCBPrev == (OS\_TCB \*)0) { /\* Remove from TCB chain \*/

ptcb->OSTCBNext->OSTCBPrev = (OS\_TCB \*)0;

OSTCBList = ptcb->OSTCBNext;

} else {

ptcb->OSTCBPrev->OSTCBNext = ptcb->OSTCBNext;

ptcb->OSTCBNext->OSTCBPrev = ptcb->OSTCBPrev;

}

ptcb->OSTCBNext = OSTCBFreeList; /\* Return TCB to free TCB list \*/

OSTCBFreeList = ptcb;

#if OS\_TASK\_NAME\_EN > 0u

ptcb->OSTCBTaskName = (INT8U \*)(void \*)"?";

#endif

OS\_EXIT\_CRITICAL();

if (OSRunning == OS\_TRUE) {

OS\_Sched(); /\* Find new highest priority task \*/

}

return (OS\_ERR\_NONE);

}

#endif

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* REQUEST THAT A TASK DELETE ITSELF

\*

\* Description: This function is used to:

\* a) notify a task to delete itself.

\* b) to see if a task requested that the current task delete itself.

\* This function is a little tricky to understand. Basically, you have a task that needs

\* to be deleted however, this task has resources that it has allocated (memory buffers,

\* semaphores, mailboxes, queues etc.). The task cannot be deleted otherwise these

\* resources would not be freed. The requesting task calls OSTaskDelReq() to indicate that

\* the task needs to be deleted. Deleting of the task is however, deferred to the task to

\* be deleted. For example, suppose that task #10 needs to be deleted. The requesting task

\* example, task #5, would call OSTaskDelReq(10). When task #10 gets to execute, it calls

\* this function by specifying OS\_PRIO\_SELF and monitors the returned value. If the return

\* value is OS\_ERR\_TASK\_DEL\_REQ, another task requested a task delete. Task #10 would look like

\* this:

\*

\* void Task(void \*p\_arg)

\* {

\* .

\* .

\* while (1) {

\* OSTimeDly(1);

\* if (OSTaskDelReq(OS\_PRIO\_SELF) == OS\_ERR\_TASK\_DEL\_REQ) {

\* Release any owned resources;

\* De-allocate any dynamic memory;

\* OSTaskDel(OS\_PRIO\_SELF);

\* }

\* }

\* }

\*

\* Arguments : prio is the priority of the task to request the delete from

\*

\* Returns : OS\_ERR\_NONE if the task exist and the request has been registered

\* OS\_ERR\_TASK\_NOT\_EXIST if the task has been deleted. This allows the caller to know whether

\* the request has been executed.

\* OS\_ERR\_TASK\_DEL if the task is assigned to a Mutex.

\* OS\_ERR\_TASK\_DEL\_IDLE if you requested to delete uC/OS-II's idle task

\* OS\_ERR\_PRIO\_INVALID if the priority you specify is higher that the maximum allowed

\* (i.e. >= OS\_LOWEST\_PRIO) or, you have not specified OS\_PRIO\_SELF.

\* OS\_ERR\_TASK\_DEL\_REQ if a task (possibly another task) requested that the running task be

\* deleted.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

#if OS\_TASK\_DEL\_EN > 0u

INT8U OSTaskDelReq (INT8U prio)

{

INT8U stat;

OS\_TCB \*ptcb;

#if OS\_CRITICAL\_METHOD == 3u /\* Allocate storage for CPU status register \*/

OS\_CPU\_SR cpu\_sr = 0u;

#endif

if (prio == OS\_TASK\_IDLE\_PRIO) { /\* Not allowed to delete idle task \*/

return (OS\_ERR\_TASK\_DEL\_IDLE);

}

#if OS\_ARG\_CHK\_EN > 0u

if (prio >= OS\_LOWEST\_PRIO) { /\* Task priority valid ? \*/

if (prio != OS\_PRIO\_SELF) {

return (OS\_ERR\_PRIO\_INVALID);

}

}

#endif

if (prio == OS\_PRIO\_SELF) { /\* See if a task is requesting to ... \*/

OS\_ENTER\_CRITICAL(); /\* ... this task to delete itself \*/

stat = OSTCBCur->OSTCBDelReq; /\* Return request status to caller \*/

OS\_EXIT\_CRITICAL();

return (stat);

}

OS\_ENTER\_CRITICAL();

ptcb = OSTCBPrioTbl[prio];

if (ptcb == (OS\_TCB \*)0) { /\* Task to delete must exist \*/

OS\_EXIT\_CRITICAL();

return (OS\_ERR\_TASK\_NOT\_EXIST); /\* Task must already be deleted \*/

}

if (ptcb == OS\_TCB\_RESERVED) { /\* Must NOT be assigned to a Mutex \*/

OS\_EXIT\_CRITICAL();

return (OS\_ERR\_TASK\_DEL);

}

ptcb->OSTCBDelReq = OS\_ERR\_TASK\_DEL\_REQ; /\* Set flag indicating task to be DEL. \*/

OS\_EXIT\_CRITICAL();

return (OS\_ERR\_NONE);

}

#endif

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* GET THE NAME OF A TASK

\*

\* Description: This function is called to obtain the name of a task.

\*

\* Arguments : prio is the priority of the task that you want to obtain the name from.

\*

\* pname is a pointer to a pointer to an ASCII string that will receive the name of the task.

\*

\* perr is a pointer to an error code that can contain one of the following values:

\*

\* OS\_ERR\_NONE if the requested task is resumed

\* OS\_ERR\_TASK\_NOT\_EXIST if the task has not been created or is assigned to a Mutex

\* OS\_ERR\_PRIO\_INVALID if you specified an invalid priority:

\* A higher value than the idle task or not OS\_PRIO\_SELF.

\* OS\_ERR\_PNAME\_NULL You passed a NULL pointer for 'pname'

\* OS\_ERR\_NAME\_GET\_ISR You called this function from an ISR

\*

\*

\* Returns : The length of the string or 0 if the task does not exist.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

#if OS\_TASK\_NAME\_EN > 0u

INT8U OSTaskNameGet (INT8U prio,

INT8U \*\*pname,

INT8U \*perr)

{

OS\_TCB \*ptcb;

INT8U len;

#if OS\_CRITICAL\_METHOD == 3u /\* Allocate storage for CPU status register \*/

OS\_CPU\_SR cpu\_sr = 0u;

#endif

#ifdef OS\_SAFETY\_CRITICAL

if (perr == (INT8U \*)0) {

OS\_SAFETY\_CRITICAL\_EXCEPTION();

}

#endif

#if OS\_ARG\_CHK\_EN > 0u

if (prio > OS\_LOWEST\_PRIO) { /\* Task priority valid ? \*/

if (prio != OS\_PRIO\_SELF) {

\*perr = OS\_ERR\_PRIO\_INVALID; /\* No \*/

return (0u);

}

}

if (pname == (INT8U \*\*)0) { /\* Is 'pname' a NULL pointer? \*/

\*perr = OS\_ERR\_PNAME\_NULL; /\* Yes \*/

return (0u);

}

#endif

if (OSIntNesting > 0u) { /\* See if trying to call from an ISR \*/

\*perr = OS\_ERR\_NAME\_GET\_ISR;

return (0u);

}

OS\_ENTER\_CRITICAL();

if (prio == OS\_PRIO\_SELF) { /\* See if caller desires it's own name \*/

prio = OSTCBCur->OSTCBPrio;

}

ptcb = OSTCBPrioTbl[prio];

if (ptcb == (OS\_TCB \*)0) { /\* Does task exist? \*/

OS\_EXIT\_CRITICAL(); /\* No \*/

\*perr = OS\_ERR\_TASK\_NOT\_EXIST;

return (0u);

}

if (ptcb == OS\_TCB\_RESERVED) { /\* Task assigned to a Mutex? \*/

OS\_EXIT\_CRITICAL(); /\* Yes \*/

\*perr = OS\_ERR\_TASK\_NOT\_EXIST;

return (0u);

}

\*pname = ptcb->OSTCBTaskName;

len = OS\_StrLen(\*pname);

OS\_EXIT\_CRITICAL();

\*perr = OS\_ERR\_NONE;

return (len);

}

#endif

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* ASSIGN A NAME TO A TASK

\*

\* Description: This function is used to set the name of a task.

\*

\* Arguments : prio is the priority of the task that you want the assign a name to.

\*

\* pname is a pointer to an ASCII string that contains the name of the task.

\*

\* perr is a pointer to an error code that can contain one of the following values:

\*

\* OS\_ERR\_NONE if the requested task is resumed

\* OS\_ERR\_TASK\_NOT\_EXIST if the task has not been created or is assigned to a Mutex

\* OS\_ERR\_PNAME\_NULL You passed a NULL pointer for 'pname'

\* OS\_ERR\_PRIO\_INVALID if you specified an invalid priority:

\* A higher value than the idle task or not OS\_PRIO\_SELF.

\* OS\_ERR\_NAME\_SET\_ISR if you called this function from an ISR

\*

\* Returns : None

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

#if OS\_TASK\_NAME\_EN > 0u

void OSTaskNameSet (INT8U prio,

INT8U \*pname,

INT8U \*perr)

{

OS\_TCB \*ptcb;

#if OS\_CRITICAL\_METHOD == 3u /\* Allocate storage for CPU status register \*/

OS\_CPU\_SR cpu\_sr = 0u;

#endif

#ifdef OS\_SAFETY\_CRITICAL

if (perr == (INT8U \*)0) {

OS\_SAFETY\_CRITICAL\_EXCEPTION();

}

#endif

#if OS\_ARG\_CHK\_EN > 0u

if (prio > OS\_LOWEST\_PRIO) { /\* Task priority valid ? \*/

if (prio != OS\_PRIO\_SELF) {

\*perr = OS\_ERR\_PRIO\_INVALID; /\* No \*/

return;

}

}

if (pname == (INT8U \*)0) { /\* Is 'pname' a NULL pointer? \*/

\*perr = OS\_ERR\_PNAME\_NULL; /\* Yes \*/

return;

}

#endif

if (OSIntNesting > 0u) { /\* See if trying to call from an ISR \*/

\*perr = OS\_ERR\_NAME\_SET\_ISR;

return;

}

OS\_ENTER\_CRITICAL();

if (prio == OS\_PRIO\_SELF) { /\* See if caller desires to set it's own name \*/

prio = OSTCBCur->OSTCBPrio;

}

ptcb = OSTCBPrioTbl[prio];

if (ptcb == (OS\_TCB \*)0) { /\* Does task exist? \*/

OS\_EXIT\_CRITICAL(); /\* No \*/

\*perr = OS\_ERR\_TASK\_NOT\_EXIST;

return;

}

if (ptcb == OS\_TCB\_RESERVED) { /\* Task assigned to a Mutex? \*/

OS\_EXIT\_CRITICAL(); /\* Yes \*/

\*perr = OS\_ERR\_TASK\_NOT\_EXIST;

return;

}

ptcb->OSTCBTaskName = pname;

OS\_EXIT\_CRITICAL();

\*perr = OS\_ERR\_NONE;

}

#endif

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* RESUME A SUSPENDED TASK

\*

\* Description: This function is called to resume a previously suspended task. This is the only call that

\* will remove an explicit task suspension.

\*

\* Arguments : prio is the priority of the task to resume.

\*

\* Returns : OS\_ERR\_NONE if the requested task is resumed

\* OS\_ERR\_PRIO\_INVALID if the priority you specify is higher that the maximum allowed

\* (i.e. >= OS\_LOWEST\_PRIO)

\* OS\_ERR\_TASK\_RESUME\_PRIO if the task to resume does not exist

\* OS\_ERR\_TASK\_NOT\_EXIST if the task is assigned to a Mutex PIP

\* OS\_ERR\_TASK\_NOT\_SUSPENDED if the task to resume has not been suspended

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

#if OS\_TASK\_SUSPEND\_EN > 0u

INT8U OSTaskResume (INT8U prio)

{

OS\_TCB \*ptcb;

#if OS\_CRITICAL\_METHOD == 3u /\* Storage for CPU status register \*/

OS\_CPU\_SR cpu\_sr = 0u;

#endif

#if OS\_ARG\_CHK\_EN > 0u

if (prio >= OS\_LOWEST\_PRIO) { /\* Make sure task priority is valid \*/

return (OS\_ERR\_PRIO\_INVALID);

}

#endif

OS\_ENTER\_CRITICAL();

ptcb = OSTCBPrioTbl[prio];

if (ptcb == (OS\_TCB \*)0) { /\* Task to suspend must exist \*/

OS\_EXIT\_CRITICAL();

return (OS\_ERR\_TASK\_RESUME\_PRIO);

}

if (ptcb == OS\_TCB\_RESERVED) { /\* See if assigned to Mutex \*/

OS\_EXIT\_CRITICAL();

return (OS\_ERR\_TASK\_NOT\_EXIST);

}

if ((ptcb->OSTCBStat & OS\_STAT\_SUSPEND) != OS\_STAT\_RDY) { /\* Task must be suspended \*/

ptcb->OSTCBStat &= (INT8U)~(INT8U)OS\_STAT\_SUSPEND; /\* Remove suspension \*/

if (ptcb->OSTCBStat == OS\_STAT\_RDY) { /\* See if task is now ready \*/

if (ptcb->OSTCBDly == 0u) {

OSRdyGrp |= ptcb->OSTCBBitY; /\* Yes, Make task ready to run \*/

OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;

OS\_EXIT\_CRITICAL();

if (OSRunning == OS\_TRUE) {

OS\_Sched(); /\* Find new highest priority task \*/

}

} else {

OS\_EXIT\_CRITICAL();

}

} else { /\* Must be pending on event \*/

OS\_EXIT\_CRITICAL();

}

return (OS\_ERR\_NONE);

}

OS\_EXIT\_CRITICAL();

return (OS\_ERR\_TASK\_NOT\_SUSPENDED);

}

#endif

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* STACK CHECKING

\*

\* Description: This function is called to check the amount of free memory left on the specified task's

\* stack.

\*

\* Arguments : prio is the task priority

\*

\* p\_stk\_data is a pointer to a data structure of type OS\_STK\_DATA.

\*

\* Returns : OS\_ERR\_NONE upon success

\* OS\_ERR\_PRIO\_INVALID if the priority you specify is higher that the maximum allowed

\* (i.e. > OS\_LOWEST\_PRIO) or, you have not specified OS\_PRIO\_SELF.

\* OS\_ERR\_TASK\_NOT\_EXIST if the desired task has not been created or is assigned to a Mutex PIP

\* OS\_ERR\_TASK\_OPT if you did NOT specified OS\_TASK\_OPT\_STK\_CHK when the task was created

\* OS\_ERR\_PDATA\_NULL if 'p\_stk\_data' is a NULL pointer

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

#if (OS\_TASK\_STAT\_STK\_CHK\_EN > 0u) && (OS\_TASK\_CREATE\_EXT\_EN > 0u)

INT8U OSTaskStkChk (INT8U prio,

OS\_STK\_DATA \*p\_stk\_data)

{

OS\_TCB \*ptcb;

OS\_STK \*pchk;

INT32U nfree;

INT32U size;

#if OS\_CRITICAL\_METHOD == 3u /\* Allocate storage for CPU status register \*/

OS\_CPU\_SR cpu\_sr = 0u;

#endif

#if OS\_ARG\_CHK\_EN > 0u

if (prio > OS\_LOWEST\_PRIO) { /\* Make sure task priority is valid \*/

if (prio != OS\_PRIO\_SELF) {

return (OS\_ERR\_PRIO\_INVALID);

}

}

if (p\_stk\_data == (OS\_STK\_DATA \*)0) { /\* Validate 'p\_stk\_data' \*/

return (OS\_ERR\_PDATA\_NULL);

}

#endif

p\_stk\_data->OSFree = 0u; /\* Assume failure, set to 0 size \*/

p\_stk\_data->OSUsed = 0u;

OS\_ENTER\_CRITICAL();

if (prio == OS\_PRIO\_SELF) { /\* See if check for SELF \*/

prio = OSTCBCur->OSTCBPrio;

}

ptcb = OSTCBPrioTbl[prio];

if (ptcb == (OS\_TCB \*)0) { /\* Make sure task exist \*/

OS\_EXIT\_CRITICAL();

return (OS\_ERR\_TASK\_NOT\_EXIST);

}

if (ptcb == OS\_TCB\_RESERVED) {

OS\_EXIT\_CRITICAL();

return (OS\_ERR\_TASK\_NOT\_EXIST);

}

if ((ptcb->OSTCBOpt & OS\_TASK\_OPT\_STK\_CHK) == 0u) { /\* Make sure stack checking option is set \*/

OS\_EXIT\_CRITICAL();

return (OS\_ERR\_TASK\_OPT);

}

nfree = 0u;

size = ptcb->OSTCBStkSize;

pchk = ptcb->OSTCBStkBottom;

OS\_EXIT\_CRITICAL();

#if OS\_STK\_GROWTH == 1u

while (\*pchk++ == (OS\_STK)0) { /\* Compute the number of zero entries on the stk \*/

nfree++;

}

#else

while (\*pchk-- == (OS\_STK)0) {

nfree++;

}

#endif

p\_stk\_data->OSFree = nfree \* sizeof(OS\_STK); /\* Compute number of free bytes on the stack \*/

p\_stk\_data->OSUsed = (size - nfree) \* sizeof(OS\_STK); /\* Compute number of bytes used on the stack \*/

return (OS\_ERR\_NONE);

}

#endif

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* SUSPEND A TASK

\*

\* Description: This function is called to suspend a task. The task can be the calling task if the

\* priority passed to OSTaskSuspend() is the priority of the calling task or OS\_PRIO\_SELF.

\*

\* Arguments : prio is the priority of the task to suspend. If you specify OS\_PRIO\_SELF, the

\* calling task will suspend itself and rescheduling will occur.

\*

\* Returns : OS\_ERR\_NONE if the requested task is suspended

\* OS\_ERR\_TASK\_SUSPEND\_IDLE if you attempted to suspend the idle task which is not allowed.

\* OS\_ERR\_PRIO\_INVALID if the priority you specify is higher that the maximum allowed

\* (i.e. >= OS\_LOWEST\_PRIO) or, you have not specified OS\_PRIO\_SELF.

\* OS\_ERR\_TASK\_SUSPEND\_PRIO if the task to suspend does not exist

\* OS\_ERR\_TASK\_NOT\_EXITS if the task is assigned to a Mutex PIP

\*

\* Note : You should use this function with great care. If you suspend a task that is waiting for

\* an event (i.e. a message, a semaphore, a queue ...) you will prevent this task from

\* running when the event arrives.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

#if OS\_TASK\_SUSPEND\_EN > 0u

INT8U OSTaskSuspend (INT8U prio)

{

BOOLEAN self;

OS\_TCB \*ptcb;

INT8U y;

#if OS\_CRITICAL\_METHOD == 3u /\* Allocate storage for CPU status register \*/

OS\_CPU\_SR cpu\_sr = 0u;

#endif

#if OS\_ARG\_CHK\_EN > 0u

if (prio == OS\_TASK\_IDLE\_PRIO) { /\* Not allowed to suspend idle task \*/

return (OS\_ERR\_TASK\_SUSPEND\_IDLE);

}

if (prio >= OS\_LOWEST\_PRIO) { /\* Task priority valid ? \*/

if (prio != OS\_PRIO\_SELF) {

return (OS\_ERR\_PRIO\_INVALID);

}

}

#endif

OS\_ENTER\_CRITICAL();

if (prio == OS\_PRIO\_SELF) { /\* See if suspend SELF \*/

prio = OSTCBCur->OSTCBPrio;

self = OS\_TRUE;

} else if (prio == OSTCBCur->OSTCBPrio) { /\* See if suspending self \*/

self = OS\_TRUE;

} else {

self = OS\_FALSE; /\* No suspending another task \*/

}

ptcb = OSTCBPrioTbl[prio];

if (ptcb == (OS\_TCB \*)0) { /\* Task to suspend must exist \*/

OS\_EXIT\_CRITICAL();

return (OS\_ERR\_TASK\_SUSPEND\_PRIO);

}

if (ptcb == OS\_TCB\_RESERVED) { /\* See if assigned to Mutex \*/

OS\_EXIT\_CRITICAL();

return (OS\_ERR\_TASK\_NOT\_EXIST);

}

y = ptcb->OSTCBY;

OSRdyTbl[y] &= (OS\_PRIO)~ptcb->OSTCBBitX; /\* Make task not ready \*/

if (OSRdyTbl[y] == 0u) {

OSRdyGrp &= (OS\_PRIO)~ptcb->OSTCBBitY;

}

ptcb->OSTCBStat |= OS\_STAT\_SUSPEND; /\* Status of task is 'SUSPENDED' \*/

OS\_EXIT\_CRITICAL();

if (self == OS\_TRUE) { /\* Context switch only if SELF \*/

OS\_Sched(); /\* Find new highest priority task \*/

}

return (OS\_ERR\_NONE);

}

#endif

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* QUERY A TASK

\*

\* Description: This function is called to obtain a copy of the desired task's TCB.

\*

\* Arguments : prio is the priority of the task to obtain information from.

\*

\* p\_task\_data is a pointer to where the desired task's OS\_TCB will be stored.

\*

\* Returns : OS\_ERR\_NONE if the requested task is suspended

\* OS\_ERR\_PRIO\_INVALID if the priority you specify is higher that the maximum allowed

\* (i.e. > OS\_LOWEST\_PRIO) or, you have not specified OS\_PRIO\_SELF.

\* OS\_ERR\_PRIO if the desired task has not been created

\* OS\_ERR\_TASK\_NOT\_EXIST if the task is assigned to a Mutex PIP

\* OS\_ERR\_PDATA\_NULL if 'p\_task\_data' is a NULL pointer

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

#if OS\_TASK\_QUERY\_EN > 0u

INT8U OSTaskQuery (INT8U prio,

OS\_TCB \*p\_task\_data)

{

OS\_TCB \*ptcb;

#if OS\_CRITICAL\_METHOD == 3u /\* Allocate storage for CPU status register \*/

OS\_CPU\_SR cpu\_sr = 0u;

#endif

#if OS\_ARG\_CHK\_EN > 0u

if (prio > OS\_LOWEST\_PRIO) { /\* Task priority valid ? \*/

if (prio != OS\_PRIO\_SELF) {

return (OS\_ERR\_PRIO\_INVALID);

}

}

if (p\_task\_data == (OS\_TCB \*)0) { /\* Validate 'p\_task\_data' \*/

return (OS\_ERR\_PDATA\_NULL);

}

#endif

OS\_ENTER\_CRITICAL();

if (prio == OS\_PRIO\_SELF) { /\* See if suspend SELF \*/

prio = OSTCBCur->OSTCBPrio;

}

ptcb = OSTCBPrioTbl[prio];

if (ptcb == (OS\_TCB \*)0) { /\* Task to query must exist \*/

OS\_EXIT\_CRITICAL();

return (OS\_ERR\_PRIO);

}

if (ptcb == OS\_TCB\_RESERVED) { /\* Task to query must not be assigned to a Mutex \*/

OS\_EXIT\_CRITICAL();

return (OS\_ERR\_TASK\_NOT\_EXIST);

}

/\* Copy TCB into user storage area \*/

OS\_MemCopy((INT8U \*)p\_task\_data, (INT8U \*)ptcb, sizeof(OS\_TCB));

OS\_EXIT\_CRITICAL();

return (OS\_ERR\_NONE);

}

#endif

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* GET THE CURRENT VALUE OF A TASK REGISTER

\*

\* Description: This function is called to obtain the current value of a task register. Task registers

\* are application specific and can be used to store task specific values such as 'error

\* numbers' (i.e. errno), statistics, etc. Each task register can hold a 32-bit value.

\*

\* Arguments : prio is the priority of the task you want to get the task register from. If you

\* specify OS\_PRIO\_SELF then the task register of the current task will be obtained.

\*

\* id is the 'id' of the desired task register. Note that the 'id' must be less

\* than OS\_TASK\_REG\_TBL\_SIZE

\*

\* perr is a pointer to a variable that will hold an error code related to this call.

\*

\* OS\_ERR\_NONE if the call was successful

\* OS\_ERR\_PRIO\_INVALID if you specified an invalid priority

\* OS\_ERR\_ID\_INVALID if the 'id' is not between 0 and OS\_TASK\_REG\_TBL\_SIZE-1

\*

\* Returns : The current value of the task's register or 0 if an error is detected.

\*

\* Note(s) : The maximum number of task variables is 254

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

#if OS\_TASK\_REG\_TBL\_SIZE > 0u

INT32U OSTaskRegGet (INT8U prio,

INT8U id,

INT8U \*perr)

{

#if OS\_CRITICAL\_METHOD == 3u /\* Allocate storage for CPU status register \*/

OS\_CPU\_SR cpu\_sr = 0u;

#endif

INT32U value;

OS\_TCB \*ptcb;

#if OS\_ARG\_CHK\_EN > 0u

if (prio >= OS\_LOWEST\_PRIO) {

if (prio != OS\_PRIO\_SELF) {

\*perr = OS\_ERR\_PRIO\_INVALID;

return (0u);

}

}

if (id >= OS\_TASK\_REG\_TBL\_SIZE) {

\*perr = OS\_ERR\_ID\_INVALID;

return (0u);

}

#endif

OS\_ENTER\_CRITICAL();

if (prio == OS\_PRIO\_SELF) { /\* See if need to get register from current task \*/

ptcb = OSTCBCur;

} else {

ptcb = OSTCBPrioTbl[prio];

}

value = ptcb->OSTCBRegTbl[id];

OS\_EXIT\_CRITICAL();

\*perr = OS\_ERR\_NONE;

return (value);

}

#endif

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* SET THE CURRENT VALUE OF A TASK VARIABLE

\*

\* Description: This function is called to change the current value of a task register. Task registers

\* are application specific and can be used to store task specific values such as 'error

\* numbers' (i.e. errno), statistics, etc. Each task register can hold a 32-bit value.

\*

\* Arguments : prio is the priority of the task you want to set the task register for. If you

\* specify OS\_PRIO\_SELF then the task register of the current task will be obtained.

\*

\* id is the 'id' of the desired task register. Note that the 'id' must be less

\* than OS\_TASK\_REG\_TBL\_SIZE

\*

\* value is the desired value for the task register.

\*

\* perr is a pointer to a variable that will hold an error code related to this call.

\*

\* OS\_ERR\_NONE if the call was successful

\* OS\_ERR\_PRIO\_INVALID if you specified an invalid priority

\* OS\_ERR\_ID\_INVALID if the 'id' is not between 0 and OS\_TASK\_REG\_TBL\_SIZE-1

\*

\* Returns : The current value of the task's variable or 0 if an error is detected.

\*

\* Note(s) : The maximum number of task variables is 254

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

#if OS\_TASK\_REG\_TBL\_SIZE > 0u

void OSTaskRegSet (INT8U prio,

INT8U id,

INT32U value,

INT8U \*perr)

{

#if OS\_CRITICAL\_METHOD == 3u /\* Allocate storage for CPU status register \*/

OS\_CPU\_SR cpu\_sr = 0u;

#endif

OS\_TCB \*ptcb;

#if OS\_ARG\_CHK\_EN > 0u

if (prio >= OS\_LOWEST\_PRIO) {

if (prio != OS\_PRIO\_SELF) {

\*perr = OS\_ERR\_PRIO\_INVALID;

return;

}

}

if (id >= OS\_TASK\_REG\_TBL\_SIZE) {

\*perr = OS\_ERR\_ID\_INVALID;

return;

}

#endif

OS\_ENTER\_CRITICAL();

if (prio == OS\_PRIO\_SELF) { /\* See if need to get register from current task \*/

ptcb = OSTCBCur;

} else {

ptcb = OSTCBPrioTbl[prio];

}

ptcb->OSTCBRegTbl[id] = value;

OS\_EXIT\_CRITICAL();

\*perr = OS\_ERR\_NONE;

}

#endif

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CATCH ACCIDENTAL TASK RETURN

\*

\* Description: This function is called if a task accidentally returns without deleting itself. In other

\* words, a task should either be an infinite loop or delete itself if it's done.

\* Arguments : none

\* Returns : none

\* Note(s) : This function is INTERNAL to uC/OS-II and your application should not call it.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void OS\_TaskReturn (void)

{

OSTaskReturnHook(OSTCBCur); /\* Call hook to let user decide on what to do \*/

#if OS\_TASK\_DEL\_EN > 0u

(void)OSTaskDel(OS\_PRIO\_SELF); /\* Delete task if it accidentally returns! \*/

#else

for (;;) {

OSTimeDly(OS\_TICKS\_PER\_SEC);

}

#endif

}

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLEAR TASK STACK

\*

\* Description: This function is used to clear the stack of a task (i.e. write all zeros)

\*

\* Arguments : pbos is a pointer to the task's bottom of stack. If the configuration constant

\* OS\_STK\_GROWTH is set to 1, the stack is assumed to grow downward (i.e. from high

\* memory to low memory). 'pbos' will thus point to the lowest (valid) memory

\* location of the stack. If OS\_STK\_GROWTH is set to 0, 'pbos' will point to the

\* highest memory location of the stack and the stack will grow with increasing

\* memory locations. 'pbos' MUST point to a valid 'free' data item.

\* size is the number of 'stack elements' to clear.

\* opt contains additional information (or options) about the behavior of the task. The

\* LOWER 8-bits are reserved by uC/OS-II while the upper 8 bits can be application

\* specific. See OS\_TASK\_OPT\_??? in uCOS-II.H.

\* Returns : none

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

#if (OS\_TASK\_STAT\_STK\_CHK\_EN > 0u) && (OS\_TASK\_CREATE\_EXT\_EN > 0u)

void OS\_TaskStkClr (OS\_STK \*pbos,

INT32U size,

INT16U opt)

{

if ((opt & OS\_TASK\_OPT\_STK\_CHK) != 0x0000u) { /\* See if stack checking has been enabled \*/

if ((opt & OS\_TASK\_OPT\_STK\_CLR) != 0x0000u) { /\* See if stack needs to be cleared \*/

#if OS\_STK\_GROWTH == 1u

while (size > 0u) { /\* Stack grows from HIGH to LOW memory \*/

size--;

\*pbos++ = (OS\_STK)0; /\* Clear from bottom of stack and up! \*/

}

#else

while (size > 0u) { /\* Stack grows from LOW to HIGH memory \*/

size--;

\*pbos-- = (OS\_STK)0; /\* Clear from bottom of stack and down \*/

}

#endif

}

}

}

#endif