/\*

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

\* uC/OS-II

\* The Real-Time Kernel

\* EVENT FLAG MANAGEMENT

\*

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

\* All Rights Reserved

\*

\* File : OS\_FLAG.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

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

/\*

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

\* LOCAL PROTOTYPES

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

\*/

static void OS\_FlagBlock(OS\_FLAG\_GRP \*pgrp, OS\_FLAG\_NODE \*pnode, OS\_FLAGS flags, INT8U wait\_type, INT32U timeout);

static BOOLEAN OS\_FlagTaskRdy(OS\_FLAG\_NODE \*pnode, OS\_FLAGS flags\_rdy);

/\*

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

\* CHECK THE STATUS OF FLAGS IN AN EVENT FLAG GROUP

\*

\* Description: This function is called to check the status of a combination of bits to be set or cleared

\* in an event flag group. Your application can check for ANY bit to be set/cleared or ALL

\* bits to be set/cleared.

\*

\* This call does not block if the desired flags are not present.

\*

\* Arguments : pgrp is a pointer to the desired event flag group.

\*

\* flags Is a bit pattern indicating which bit(s) (i.e. flags) you wish to check.

\* The bits you want are specified by setting the corresponding bits in

\* 'flags'. e.g. if your application wants to wait for bits 0 and 1 then

\* 'flags' would contain 0x03.

\*

\* wait\_type specifies whether you want ALL bits to be set/cleared or ANY of the bits

\* to be set/cleared.

\* You can specify the following argument:

\*

\* OS\_FLAG\_WAIT\_CLR\_ALL You will check ALL bits in 'flags' to be clear (0)

\* OS\_FLAG\_WAIT\_CLR\_ANY You will check ANY bit in 'flags' to be clear (0)

\* OS\_FLAG\_WAIT\_SET\_ALL You will check ALL bits in 'flags' to be set (1)

\* OS\_FLAG\_WAIT\_SET\_ANY You will check ANY bit in 'flags' to be set (1)

\*

\* NOTE: Add OS\_FLAG\_CONSUME if you want the event flag to be 'consumed' by

\* the call. Example, to wait for any flag in a group AND then clear

\* the flags that are present, set 'wait\_type' to:

\*

\* OS\_FLAG\_WAIT\_SET\_ANY + OS\_FLAG\_CONSUME

\*

\* perr is a pointer to an error code and can be:

\* OS\_ERR\_NONE No error

\* OS\_ERR\_EVENT\_TYPE You are not pointing to an event flag group

\* OS\_ERR\_FLAG\_WAIT\_TYPE You didn't specify a proper 'wait\_type' argument.

\* OS\_ERR\_FLAG\_INVALID\_PGRP You passed a NULL pointer instead of the event flag

\* group handle.

\* OS\_ERR\_FLAG\_NOT\_RDY The desired flags you are waiting for are not

\* available.

\*

\* Returns : The flags in the event flag group that made the task ready or, 0 if a timeout or an error

\* occurred.

\*

\* Called from: Task or ISR

\*

\* Note(s) : 1) IMPORTANT, the behavior of this function has changed from PREVIOUS versions. The

\* function NOW returns the flags that were ready INSTEAD of the current state of the

\* event flags.

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

\*/

#if OS\_FLAG\_ACCEPT\_EN > 0u

OS\_FLAGS OSFlagAccept (OS\_FLAG\_GRP \*pgrp,

OS\_FLAGS flags,

INT8U wait\_type,

INT8U \*perr)

{

OS\_FLAGS flags\_rdy;

INT8U result;

BOOLEAN consume;

#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 (pgrp == (OS\_FLAG\_GRP \*)0) { /\* Validate 'pgrp' \*/

\*perr = OS\_ERR\_FLAG\_INVALID\_PGRP;

return ((OS\_FLAGS)0);

}

#endif

if (pgrp->OSFlagType != OS\_EVENT\_TYPE\_FLAG) { /\* Validate event block type \*/

\*perr = OS\_ERR\_EVENT\_TYPE;

return ((OS\_FLAGS)0);

}

result = (INT8U)(wait\_type & OS\_FLAG\_CONSUME);

if (result != (INT8U)0) { /\* See if we need to consume the flags \*/

wait\_type &= ~OS\_FLAG\_CONSUME;

consume = OS\_TRUE;

} else {

consume = OS\_FALSE;

}

\*perr = OS\_ERR\_NONE; /\* Assume NO error until proven otherwise. \*/

OS\_ENTER\_CRITICAL();

switch (wait\_type) {

case OS\_FLAG\_WAIT\_SET\_ALL: /\* See if all required flags are set \*/

flags\_rdy = (OS\_FLAGS)(pgrp->OSFlagFlags & flags); /\* Extract only the bits we want \*/

if (flags\_rdy == flags) { /\* Must match ALL the bits that we want \*/

if (consume == OS\_TRUE) { /\* See if we need to consume the flags \*/

pgrp->OSFlagFlags &= (OS\_FLAGS)~flags\_rdy; /\* Clear ONLY the flags we wanted \*/

}

} else {

\*perr = OS\_ERR\_FLAG\_NOT\_RDY;

}

OS\_EXIT\_CRITICAL();

break;

case OS\_FLAG\_WAIT\_SET\_ANY:

flags\_rdy = (OS\_FLAGS)(pgrp->OSFlagFlags & flags); /\* Extract only the bits we want \*/

if (flags\_rdy != (OS\_FLAGS)0) { /\* See if any flag set \*/

if (consume == OS\_TRUE) { /\* See if we need to consume the flags \*/

pgrp->OSFlagFlags &= (OS\_FLAGS)~flags\_rdy; /\* Clear ONLY the flags we got \*/

}

} else {

\*perr = OS\_ERR\_FLAG\_NOT\_RDY;

}

OS\_EXIT\_CRITICAL();

break;

#if OS\_FLAG\_WAIT\_CLR\_EN > 0u

case OS\_FLAG\_WAIT\_CLR\_ALL: /\* See if all required flags are cleared \*/

flags\_rdy = (OS\_FLAGS)~pgrp->OSFlagFlags & flags; /\* Extract only the bits we want \*/

if (flags\_rdy == flags) { /\* Must match ALL the bits that we want \*/

if (consume == OS\_TRUE) { /\* See if we need to consume the flags \*/

pgrp->OSFlagFlags |= flags\_rdy; /\* Set ONLY the flags that we wanted \*/

}

} else {

\*perr = OS\_ERR\_FLAG\_NOT\_RDY;

}

OS\_EXIT\_CRITICAL();

break;

case OS\_FLAG\_WAIT\_CLR\_ANY:

flags\_rdy = (OS\_FLAGS)~pgrp->OSFlagFlags & flags; /\* Extract only the bits we want \*/

if (flags\_rdy != (OS\_FLAGS)0) { /\* See if any flag cleared \*/

if (consume == OS\_TRUE) { /\* See if we need to consume the flags \*/

pgrp->OSFlagFlags |= flags\_rdy; /\* Set ONLY the flags that we got \*/

}

} else {

\*perr = OS\_ERR\_FLAG\_NOT\_RDY;

}

OS\_EXIT\_CRITICAL();

break;

#endif

default:

OS\_EXIT\_CRITICAL();

flags\_rdy = (OS\_FLAGS)0;

\*perr = OS\_ERR\_FLAG\_WAIT\_TYPE;

break;

}

return (flags\_rdy);

}

#endif

/\*

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

\* CREATE AN EVENT FLAG

\*

\* Description: This function is called to create an event flag group.

\*

\* Arguments : flags Contains the initial value to store in the event flag group.

\*

\* perr is a pointer to an error code which will be returned to your application:

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

\* OS\_ERR\_CREATE\_ISR if you attempted to create an Event Flag from an

\* ISR.

\* OS\_ERR\_FLAG\_GRP\_DEPLETED if there are no more event flag groups

\*

\* Returns : A pointer to an event flag group or a NULL pointer if no more groups are available.

\*

\* Called from: Task ONLY

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

\*/

OS\_FLAG\_GRP \*OSFlagCreate (OS\_FLAGS flags,

INT8U \*perr)

{

OS\_FLAG\_GRP \*pgrp;

#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

#ifdef OS\_SAFETY\_CRITICAL\_IEC61508

if (OSSafetyCriticalStartFlag == OS\_TRUE) {

OS\_SAFETY\_CRITICAL\_EXCEPTION();

}

#endif

if (OSIntNesting > 0u) { /\* See if called from ISR ... \*/

\*perr = OS\_ERR\_CREATE\_ISR; /\* ... can't CREATE from an ISR \*/

return ((OS\_FLAG\_GRP \*)0);

}

OS\_ENTER\_CRITICAL();

pgrp = OSFlagFreeList; /\* Get next free event flag \*/

if (pgrp != (OS\_FLAG\_GRP \*)0) { /\* See if we have event flag groups available \*/

/\* Adjust free list \*/

OSFlagFreeList = (OS\_FLAG\_GRP \*)OSFlagFreeList->OSFlagWaitList;

pgrp->OSFlagType = OS\_EVENT\_TYPE\_FLAG; /\* Set to event flag group type \*/

pgrp->OSFlagFlags = flags; /\* Set to desired initial value \*/

pgrp->OSFlagWaitList = (void \*)0; /\* Clear list of tasks waiting on flags \*/

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

pgrp->OSFlagName = (INT8U \*)(void \*)"?";

#endif

OS\_EXIT\_CRITICAL();

\*perr = OS\_ERR\_NONE;

} else {

OS\_EXIT\_CRITICAL();

\*perr = OS\_ERR\_FLAG\_GRP\_DEPLETED;

}

return (pgrp); /\* Return pointer to event flag group \*/

}

/\*

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

\* DELETE AN EVENT FLAG GROUP

\*

\* Description: This function deletes an event flag group and readies all tasks pending on the event flag

\* group.

\*

\* Arguments : pgrp is a pointer to the desired event flag group.

\*

\* opt determines delete options as follows:

\* opt == OS\_DEL\_NO\_PEND Deletes the event flag group ONLY if no task pending

\* opt == OS\_DEL\_ALWAYS Deletes the event flag group even if tasks are

\* waiting. In this case, all the tasks pending will be

\* readied.

\*

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

\* OS\_ERR\_NONE The call was successful and the event flag group was

\* deleted

\* OS\_ERR\_DEL\_ISR If you attempted to delete the event flag group from

\* an ISR

\* OS\_ERR\_FLAG\_INVALID\_PGRP If 'pgrp' is a NULL pointer.

\* OS\_ERR\_EVENT\_TYPE If you didn't pass a pointer to an event flag group

\* OS\_ERR\_INVALID\_OPT An invalid option was specified

\* OS\_ERR\_TASK\_WAITING One or more tasks were waiting on the event flag

\* group.

\*

\* Returns : pgrp upon error

\* (OS\_EVENT \*)0 if the event flag group was successfully deleted.

\*

\* Note(s) : 1) This function must be used with care. Tasks that would normally expect the presence of

\* the event flag group MUST check the return code of OSFlagAccept() and OSFlagPend().

\* 2) This call can potentially disable interrupts for a long time. The interrupt disable

\* time is directly proportional to the number of tasks waiting on the event flag group.

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

\*/

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

OS\_FLAG\_GRP \*OSFlagDel (OS\_FLAG\_GRP \*pgrp,

INT8U opt,

INT8U \*perr)

{

BOOLEAN tasks\_waiting;

OS\_FLAG\_NODE \*pnode;

OS\_FLAG\_GRP \*pgrp\_return;

#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 (pgrp == (OS\_FLAG\_GRP \*)0) { /\* Validate 'pgrp' \*/

\*perr = OS\_ERR\_FLAG\_INVALID\_PGRP;

return (pgrp);

}

#endif

if (OSIntNesting > 0u) { /\* See if called from ISR ... \*/

\*perr = OS\_ERR\_DEL\_ISR; /\* ... can't DELETE from an ISR \*/

return (pgrp);

}

if (pgrp->OSFlagType != OS\_EVENT\_TYPE\_FLAG) { /\* Validate event group type \*/

\*perr = OS\_ERR\_EVENT\_TYPE;

return (pgrp);

}

OS\_ENTER\_CRITICAL();

if (pgrp->OSFlagWaitList != (void \*)0) { /\* See if any tasks waiting on event flags \*/

tasks\_waiting = OS\_TRUE; /\* Yes \*/

} else {

tasks\_waiting = OS\_FALSE; /\* No \*/

}

switch (opt) {

case OS\_DEL\_NO\_PEND: /\* Delete group if no task waiting \*/

if (tasks\_waiting == OS\_FALSE) {

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

pgrp->OSFlagName = (INT8U \*)(void \*)"?";

#endif

pgrp->OSFlagType = OS\_EVENT\_TYPE\_UNUSED;

pgrp->OSFlagWaitList = (void \*)OSFlagFreeList; /\* Return group to free list \*/

pgrp->OSFlagFlags = (OS\_FLAGS)0;

OSFlagFreeList = pgrp;

OS\_EXIT\_CRITICAL();

\*perr = OS\_ERR\_NONE;

pgrp\_return = (OS\_FLAG\_GRP \*)0; /\* Event Flag Group has been deleted \*/

} else {

OS\_EXIT\_CRITICAL();

\*perr = OS\_ERR\_TASK\_WAITING;

pgrp\_return = pgrp;

}

break;

case OS\_DEL\_ALWAYS: /\* Always delete the event flag group \*/

pnode = (OS\_FLAG\_NODE \*)pgrp->OSFlagWaitList;

while (pnode != (OS\_FLAG\_NODE \*)0) { /\* Ready ALL tasks waiting for flags \*/

(void)OS\_FlagTaskRdy(pnode, (OS\_FLAGS)0);

pnode = (OS\_FLAG\_NODE \*)pnode->OSFlagNodeNext;

}

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

pgrp->OSFlagName = (INT8U \*)(void \*)"?";

#endif

pgrp->OSFlagType = OS\_EVENT\_TYPE\_UNUSED;

pgrp->OSFlagWaitList = (void \*)OSFlagFreeList;/\* Return group to free list \*/

pgrp->OSFlagFlags = (OS\_FLAGS)0;

OSFlagFreeList = pgrp;

OS\_EXIT\_CRITICAL();

if (tasks\_waiting == OS\_TRUE) { /\* Reschedule only if task(s) were waiting \*/

OS\_Sched(); /\* Find highest priority task ready to run \*/

}

\*perr = OS\_ERR\_NONE;

pgrp\_return = (OS\_FLAG\_GRP \*)0; /\* Event Flag Group has been deleted \*/

break;

default:

OS\_EXIT\_CRITICAL();

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

pgrp\_return = pgrp;

break;

}

return (pgrp\_return);

}

#endif

/\*

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

\* GET THE NAME OF AN EVENT FLAG GROUP

\*

\* Description: This function is used to obtain the name assigned to an event flag group

\*

\* Arguments : pgrp is a pointer to the event flag group.

\*

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

\* group.

\*

\* 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\_EVENT\_TYPE if 'pevent' is not pointing to an event flag group

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

\* OS\_ERR\_FLAG\_INVALID\_PGRP if you passed a NULL pointer for 'pgrp'

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

\*

\* Returns : The length of the string or 0 if the 'pgrp' is a NULL pointer.

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

\*/

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

INT8U OSFlagNameGet (OS\_FLAG\_GRP \*pgrp,

INT8U \*\*pname,

INT8U \*perr)

{

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 (pgrp == (OS\_FLAG\_GRP \*)0) { /\* Is 'pgrp' a NULL pointer? \*/

\*perr = OS\_ERR\_FLAG\_INVALID\_PGRP;

return (0u);

}

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

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

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 (pgrp->OSFlagType != OS\_EVENT\_TYPE\_FLAG) {

OS\_EXIT\_CRITICAL();

\*perr = OS\_ERR\_EVENT\_TYPE;

return (0u);

}

\*pname = pgrp->OSFlagName;

len = OS\_StrLen(\*pname);

OS\_EXIT\_CRITICAL();

\*perr = OS\_ERR\_NONE;

return (len);

}

#endif

/\*

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

\* ASSIGN A NAME TO AN EVENT FLAG GROUP

\*

\* Description: This function assigns a name to an event flag group.

\*

\* Arguments : pgrp is a pointer to the event flag group.

\*

\* pname is a pointer to an ASCII string that will be used as the name of the event flag

\* group.

\*

\* 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\_EVENT\_TYPE if 'pevent' is not pointing to an event flag group

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

\* OS\_ERR\_FLAG\_INVALID\_PGRP if you passed a NULL pointer for 'pgrp'

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

\*

\* Returns : None

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

\*/

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

void OSFlagNameSet (OS\_FLAG\_GRP \*pgrp,

INT8U \*pname,

INT8U \*perr)

{

#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 (pgrp == (OS\_FLAG\_GRP \*)0) { /\* Is 'pgrp' a NULL pointer? \*/

\*perr = OS\_ERR\_FLAG\_INVALID\_PGRP;

return;

}

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

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

return;

}

#endif

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

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

return;

}

OS\_ENTER\_CRITICAL();

if (pgrp->OSFlagType != OS\_EVENT\_TYPE\_FLAG) {

OS\_EXIT\_CRITICAL();

\*perr = OS\_ERR\_EVENT\_TYPE;

return;

}

pgrp->OSFlagName = pname;

OS\_EXIT\_CRITICAL();

\*perr = OS\_ERR\_NONE;

return;

}

#endif

/\*

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

\* WAIT ON AN EVENT FLAG GROUP

\*

\* Description: This function is called to wait for a combination of bits to be set in an event flag

\* group. Your application can wait for ANY bit to be set or ALL bits to be set.

\*

\* Arguments : pgrp is a pointer to the desired event flag group.

\*

\* flags Is a bit pattern indicating which bit(s) (i.e. flags) you wish to wait for.

\* The bits you want are specified by setting the corresponding bits in

\* 'flags'. e.g. if your application wants to wait for bits 0 and 1 then

\* 'flags' would contain 0x03.

\*

\* wait\_type specifies whether you want ALL bits to be set or ANY of the bits to be set.

\* You can specify the following argument:

\*

\* OS\_FLAG\_WAIT\_CLR\_ALL You will wait for ALL bits in 'mask' to be clear (0)

\* OS\_FLAG\_WAIT\_SET\_ALL You will wait for ALL bits in 'mask' to be set (1)

\* OS\_FLAG\_WAIT\_CLR\_ANY You will wait for ANY bit in 'mask' to be clear (0)

\* OS\_FLAG\_WAIT\_SET\_ANY You will wait for ANY bit in 'mask' to be set (1)

\*

\* NOTE: Add OS\_FLAG\_CONSUME if you want the event flag to be 'consumed' by

\* the call. Example, to wait for any flag in a group AND then clear

\* the flags that are present, set 'wait\_type' to:

\*

\* OS\_FLAG\_WAIT\_SET\_ANY + OS\_FLAG\_CONSUME

\*

\* timeout is an optional timeout (in clock ticks) that your task will wait for the

\* desired bit combination. If you specify 0, however, your task will wait

\* forever at the specified event flag group or, until a message arrives.

\*

\* perr is a pointer to an error code and can be:

\* OS\_ERR\_NONE The desired bits have been set within the specified

\* 'timeout'.

\* OS\_ERR\_PEND\_ISR If you tried to PEND from an ISR

\* OS\_ERR\_FLAG\_INVALID\_PGRP If 'pgrp' is a NULL pointer.

\* OS\_ERR\_EVENT\_TYPE You are not pointing to an event flag group

\* OS\_ERR\_TIMEOUT The bit(s) have not been set in the specified

\* 'timeout'.

\* OS\_ERR\_PEND\_ABORT The wait on the flag was aborted.

\* OS\_ERR\_FLAG\_WAIT\_TYPE You didn't specify a proper 'wait\_type' argument.

\*

\* Returns : The flags in the event flag group that made the task ready or, 0 if a timeout or an error

\* occurred.

\*

\* Called from: Task ONLY

\*

\* Note(s) : 1) IMPORTANT, the behavior of this function has changed from PREVIOUS versions. The

\* function NOW returns the flags that were ready INSTEAD of the current state of the

\* event flags.

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

\*/

OS\_FLAGS OSFlagPend (OS\_FLAG\_GRP \*pgrp,

OS\_FLAGS flags,

INT8U wait\_type,

INT32U timeout,

INT8U \*perr)

{

OS\_FLAG\_NODE node;

OS\_FLAGS flags\_rdy;

INT8U result;

INT8U pend\_stat;

BOOLEAN consume;

#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 (pgrp == (OS\_FLAG\_GRP \*)0) { /\* Validate 'pgrp' \*/

\*perr = OS\_ERR\_FLAG\_INVALID\_PGRP;

return ((OS\_FLAGS)0);

}

#endif

if (OSIntNesting > 0u) { /\* See if called from ISR ... \*/

\*perr = OS\_ERR\_PEND\_ISR; /\* ... can't PEND from an ISR \*/

return ((OS\_FLAGS)0);

}

if (OSLockNesting > 0u) { /\* See if called with scheduler locked ... \*/

\*perr = OS\_ERR\_PEND\_LOCKED; /\* ... can't PEND when locked \*/

return ((OS\_FLAGS)0);

}

if (pgrp->OSFlagType != OS\_EVENT\_TYPE\_FLAG) { /\* Validate event block type \*/

\*perr = OS\_ERR\_EVENT\_TYPE;

return ((OS\_FLAGS)0);

}

result = (INT8U)(wait\_type & OS\_FLAG\_CONSUME);

if (result != (INT8U)0) { /\* See if we need to consume the flags \*/

wait\_type &= (INT8U)~(INT8U)OS\_FLAG\_CONSUME;

consume = OS\_TRUE;

} else {

consume = OS\_FALSE;

}

OS\_ENTER\_CRITICAL();

switch (wait\_type) {

case OS\_FLAG\_WAIT\_SET\_ALL: /\* See if all required flags are set \*/

flags\_rdy = (OS\_FLAGS)(pgrp->OSFlagFlags & flags); /\* Extract only the bits we want \*/

if (flags\_rdy == flags) { /\* Must match ALL the bits that we want \*/

if (consume == OS\_TRUE) { /\* See if we need to consume the flags \*/

pgrp->OSFlagFlags &= (OS\_FLAGS)~flags\_rdy; /\* Clear ONLY the flags we wanted \*/

}

OSTCBCur->OSTCBFlagsRdy = flags\_rdy; /\* Save flags that were ready \*/

OS\_EXIT\_CRITICAL(); /\* Yes, condition met, return to caller \*/

\*perr = OS\_ERR\_NONE;

return (flags\_rdy);

} else { /\* Block task until events occur or timeout \*/

OS\_FlagBlock(pgrp, &node, flags, wait\_type, timeout);

OS\_EXIT\_CRITICAL();

}

break;

case OS\_FLAG\_WAIT\_SET\_ANY:

flags\_rdy = (OS\_FLAGS)(pgrp->OSFlagFlags & flags); /\* Extract only the bits we want \*/

if (flags\_rdy != (OS\_FLAGS)0) { /\* See if any flag set \*/

if (consume == OS\_TRUE) { /\* See if we need to consume the flags \*/

pgrp->OSFlagFlags &= (OS\_FLAGS)~flags\_rdy; /\* Clear ONLY the flags that we got \*/

}

OSTCBCur->OSTCBFlagsRdy = flags\_rdy; /\* Save flags that were ready \*/

OS\_EXIT\_CRITICAL(); /\* Yes, condition met, return to caller \*/

\*perr = OS\_ERR\_NONE;

return (flags\_rdy);

} else { /\* Block task until events occur or timeout \*/

OS\_FlagBlock(pgrp, &node, flags, wait\_type, timeout);

OS\_EXIT\_CRITICAL();

}

break;

#if OS\_FLAG\_WAIT\_CLR\_EN > 0u

case OS\_FLAG\_WAIT\_CLR\_ALL: /\* See if all required flags are cleared \*/

flags\_rdy = (OS\_FLAGS)~pgrp->OSFlagFlags & flags; /\* Extract only the bits we want \*/

if (flags\_rdy == flags) { /\* Must match ALL the bits that we want \*/

if (consume == OS\_TRUE) { /\* See if we need to consume the flags \*/

pgrp->OSFlagFlags |= flags\_rdy; /\* Set ONLY the flags that we wanted \*/

}

OSTCBCur->OSTCBFlagsRdy = flags\_rdy; /\* Save flags that were ready \*/

OS\_EXIT\_CRITICAL(); /\* Yes, condition met, return to caller \*/

\*perr = OS\_ERR\_NONE;

return (flags\_rdy);

} else { /\* Block task until events occur or timeout \*/

OS\_FlagBlock(pgrp, &node, flags, wait\_type, timeout);

OS\_EXIT\_CRITICAL();

}

break;

case OS\_FLAG\_WAIT\_CLR\_ANY:

flags\_rdy = (OS\_FLAGS)~pgrp->OSFlagFlags & flags; /\* Extract only the bits we want \*/

if (flags\_rdy != (OS\_FLAGS)0) { /\* See if any flag cleared \*/

if (consume == OS\_TRUE) { /\* See if we need to consume the flags \*/

pgrp->OSFlagFlags |= flags\_rdy; /\* Set ONLY the flags that we got \*/

}

OSTCBCur->OSTCBFlagsRdy = flags\_rdy; /\* Save flags that were ready \*/

OS\_EXIT\_CRITICAL(); /\* Yes, condition met, return to caller \*/

\*perr = OS\_ERR\_NONE;

return (flags\_rdy);

} else { /\* Block task until events occur or timeout \*/

OS\_FlagBlock(pgrp, &node, flags, wait\_type, timeout);

OS\_EXIT\_CRITICAL();

}

break;

#endif

default:

OS\_EXIT\_CRITICAL();

flags\_rdy = (OS\_FLAGS)0;

\*perr = OS\_ERR\_FLAG\_WAIT\_TYPE;

return (flags\_rdy);

}

OS\_Sched(); /\* Find next HPT ready to run \*/

OS\_ENTER\_CRITICAL();

if (OSTCBCur->OSTCBStatPend != OS\_STAT\_PEND\_OK) { /\* Have we timed-out or aborted? \*/

pend\_stat = OSTCBCur->OSTCBStatPend;

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

OS\_FlagUnlink(&node);

OSTCBCur->OSTCBStat = OS\_STAT\_RDY; /\* Yes, make task ready-to-run \*/

OS\_EXIT\_CRITICAL();

flags\_rdy = (OS\_FLAGS)0;

switch (pend\_stat) {

case OS\_STAT\_PEND\_ABORT:

\*perr = OS\_ERR\_PEND\_ABORT; /\* Indicate that we aborted waiting \*/

break;

case OS\_STAT\_PEND\_TO:

default:

\*perr = OS\_ERR\_TIMEOUT; /\* Indicate that we timed-out waiting \*/

break;

}

return (flags\_rdy);

}

flags\_rdy = OSTCBCur->OSTCBFlagsRdy;

if (consume == OS\_TRUE) { /\* See if we need to consume the flags \*/

switch (wait\_type) {

case OS\_FLAG\_WAIT\_SET\_ALL:

case OS\_FLAG\_WAIT\_SET\_ANY: /\* Clear ONLY the flags we got \*/

pgrp->OSFlagFlags &= (OS\_FLAGS)~flags\_rdy;

break;

#if OS\_FLAG\_WAIT\_CLR\_EN > 0u

case OS\_FLAG\_WAIT\_CLR\_ALL:

case OS\_FLAG\_WAIT\_CLR\_ANY: /\* Set ONLY the flags we got \*/

pgrp->OSFlagFlags |= flags\_rdy;

break;

#endif

default:

OS\_EXIT\_CRITICAL();

\*perr = OS\_ERR\_FLAG\_WAIT\_TYPE;

return ((OS\_FLAGS)0);

}

}

OS\_EXIT\_CRITICAL();

\*perr = OS\_ERR\_NONE; /\* Event(s) must have occurred \*/

return (flags\_rdy);

}

/\*

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

\* GET FLAGS WHO CAUSED TASK TO BECOME READY

\*

\* Description: This function is called to obtain the flags that caused the task to become ready to run.

\* In other words, this function allows you to tell "Who done it!".

\*

\* Arguments : None

\*

\* Returns : The flags that caused the task to be ready.

\*

\* Called from: Task ONLY

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

\*/

OS\_FLAGS OSFlagPendGetFlagsRdy (void)

{

OS\_FLAGS flags;

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

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

#endif

OS\_ENTER\_CRITICAL();

flags = OSTCBCur->OSTCBFlagsRdy;

OS\_EXIT\_CRITICAL();

return (flags);

}

/\*

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

\* POST EVENT FLAG BIT(S)

\*

\* Description: This function is called to set or clear some bits in an event flag group. The bits to

\* set or clear are specified by a 'bit mask'.

\*

\* Arguments : pgrp is a pointer to the desired event flag group.

\*

\* flags If 'opt' (see below) is OS\_FLAG\_SET, each bit that is set in 'flags' will

\* set the corresponding bit in the event flag group. e.g. to set bits 0, 4

\* and 5 you would set 'flags' to:

\*

\* 0x31 (note, bit 0 is least significant bit)

\*

\* If 'opt' (see below) is OS\_FLAG\_CLR, each bit that is set in 'flags' will

\* CLEAR the corresponding bit in the event flag group. e.g. to clear bits 0,

\* 4 and 5 you would specify 'flags' as:

\*

\* 0x31 (note, bit 0 is least significant bit)

\*

\* opt indicates whether the flags will be:

\* set (OS\_FLAG\_SET) or

\* cleared (OS\_FLAG\_CLR)

\*

\* perr is a pointer to an error code and can be:

\* OS\_ERR\_NONE The call was successfull

\* OS\_ERR\_FLAG\_INVALID\_PGRP You passed a NULL pointer

\* OS\_ERR\_EVENT\_TYPE You are not pointing to an event flag group

\* OS\_ERR\_FLAG\_INVALID\_OPT You specified an invalid option

\*

\* Returns : the new value of the event flags bits that are still set.

\*

\* Called From: Task or ISR

\*

\* WARNING(s) : 1) The execution time of this function depends on the number of tasks waiting on the event

\* flag group.

\* 2) The amount of time interrupts are DISABLED depends on the number of tasks waiting on

\* the event flag group.

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

\*/

OS\_FLAGS OSFlagPost (OS\_FLAG\_GRP \*pgrp,

OS\_FLAGS flags,

INT8U opt,

INT8U \*perr)

{

OS\_FLAG\_NODE \*pnode;

BOOLEAN sched;

OS\_FLAGS flags\_cur;

OS\_FLAGS flags\_rdy;

BOOLEAN rdy;

#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 (pgrp == (OS\_FLAG\_GRP \*)0) { /\* Validate 'pgrp' \*/

\*perr = OS\_ERR\_FLAG\_INVALID\_PGRP;

return ((OS\_FLAGS)0);

}

#endif

if (pgrp->OSFlagType != OS\_EVENT\_TYPE\_FLAG) { /\* Make sure we are pointing to an event flag grp \*/

\*perr = OS\_ERR\_EVENT\_TYPE;

return ((OS\_FLAGS)0);

}

OS\_ENTER\_CRITICAL();

switch (opt) {

case OS\_FLAG\_CLR:

pgrp->OSFlagFlags &= (OS\_FLAGS)~flags; /\* Clear the flags specified in the group \*/

break;

case OS\_FLAG\_SET:

pgrp->OSFlagFlags |= flags; /\* Set the flags specified in the group \*/

break;

default:

OS\_EXIT\_CRITICAL(); /\* INVALID option \*/

\*perr = OS\_ERR\_FLAG\_INVALID\_OPT;

return ((OS\_FLAGS)0);

}

sched = OS\_FALSE; /\* Indicate that we don't need rescheduling \*/

pnode = (OS\_FLAG\_NODE \*)pgrp->OSFlagWaitList;

while (pnode != (OS\_FLAG\_NODE \*)0) { /\* Go through all tasks waiting on event flag(s) \*/

switch (pnode->OSFlagNodeWaitType) {

case OS\_FLAG\_WAIT\_SET\_ALL: /\* See if all req. flags are set for current node \*/

flags\_rdy = (OS\_FLAGS)(pgrp->OSFlagFlags & pnode->OSFlagNodeFlags);

if (flags\_rdy == pnode->OSFlagNodeFlags) {

rdy = OS\_FlagTaskRdy(pnode, flags\_rdy); /\* Make task RTR, event(s) Rx'd \*/

if (rdy == OS\_TRUE) {

sched = OS\_TRUE; /\* When done we will reschedule \*/

}

}

break;

case OS\_FLAG\_WAIT\_SET\_ANY: /\* See if any flag set \*/

flags\_rdy = (OS\_FLAGS)(pgrp->OSFlagFlags & pnode->OSFlagNodeFlags);

if (flags\_rdy != (OS\_FLAGS)0) {

rdy = OS\_FlagTaskRdy(pnode, flags\_rdy); /\* Make task RTR, event(s) Rx'd \*/

if (rdy == OS\_TRUE) {

sched = OS\_TRUE; /\* When done we will reschedule \*/

}

}

break;

#if OS\_FLAG\_WAIT\_CLR\_EN > 0u

case OS\_FLAG\_WAIT\_CLR\_ALL: /\* See if all req. flags are set for current node \*/

flags\_rdy = (OS\_FLAGS)~pgrp->OSFlagFlags & pnode->OSFlagNodeFlags;

if (flags\_rdy == pnode->OSFlagNodeFlags) {

rdy = OS\_FlagTaskRdy(pnode, flags\_rdy); /\* Make task RTR, event(s) Rx'd \*/

if (rdy == OS\_TRUE) {

sched = OS\_TRUE; /\* When done we will reschedule \*/

}

}

break;

case OS\_FLAG\_WAIT\_CLR\_ANY: /\* See if any flag set \*/

flags\_rdy = (OS\_FLAGS)~pgrp->OSFlagFlags & pnode->OSFlagNodeFlags;

if (flags\_rdy != (OS\_FLAGS)0) {

rdy = OS\_FlagTaskRdy(pnode, flags\_rdy); /\* Make task RTR, event(s) Rx'd \*/

if (rdy == OS\_TRUE) {

sched = OS\_TRUE; /\* When done we will reschedule \*/

}

}

break;

#endif

default:

OS\_EXIT\_CRITICAL();

\*perr = OS\_ERR\_FLAG\_WAIT\_TYPE;

return ((OS\_FLAGS)0);

}

pnode = (OS\_FLAG\_NODE \*)pnode->OSFlagNodeNext; /\* Point to next task waiting for event flag(s) \*/

}

OS\_EXIT\_CRITICAL();

if (sched == OS\_TRUE) {

OS\_Sched();

}

OS\_ENTER\_CRITICAL();

flags\_cur = pgrp->OSFlagFlags;

OS\_EXIT\_CRITICAL();

\*perr = OS\_ERR\_NONE;

return (flags\_cur);

}

/\*

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

\* QUERY EVENT FLAG

\*

\* Description: This function is used to check the value of the event flag group.

\*

\* Arguments : pgrp is a pointer to the desired event flag group.

\*

\* perr is a pointer to an error code returned to the called:

\* OS\_ERR\_NONE The call was successfull

\* OS\_ERR\_FLAG\_INVALID\_PGRP You passed a NULL pointer

\* OS\_ERR\_EVENT\_TYPE You are not pointing to an event flag group

\*

\* Returns : The current value of the event flag group.

\*

\* Called From: Task or ISR

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

\*/

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

OS\_FLAGS OSFlagQuery (OS\_FLAG\_GRP \*pgrp,

INT8U \*perr)

{

OS\_FLAGS flags;

#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 (pgrp == (OS\_FLAG\_GRP \*)0) { /\* Validate 'pgrp' \*/

\*perr = OS\_ERR\_FLAG\_INVALID\_PGRP;

return ((OS\_FLAGS)0);

}

#endif

if (pgrp->OSFlagType != OS\_EVENT\_TYPE\_FLAG) { /\* Validate event block type \*/

\*perr = OS\_ERR\_EVENT\_TYPE;

return ((OS\_FLAGS)0);

}

OS\_ENTER\_CRITICAL();

flags = pgrp->OSFlagFlags;

OS\_EXIT\_CRITICAL();

\*perr = OS\_ERR\_NONE;

return (flags); /\* Return the current value of the event flags \*/

}

#endif

/\*

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

\* SUSPEND TASK UNTIL EVENT FLAG(s) RECEIVED OR TIMEOUT OCCURS

\*

\* Description: This function is internal to uC/OS-II and is used to put a task to sleep until the desired

\* event flag bit(s) are set.

\*

\* Arguments : pgrp is a pointer to the desired event flag group.

\*

\* pnode is a pointer to a structure which contains data about the task waiting for

\* event flag bit(s) to be set.

\*

\* flags Is a bit pattern indicating which bit(s) (i.e. flags) you wish to check.

\* The bits you want are specified by setting the corresponding bits in

\* 'flags'. e.g. if your application wants to wait for bits 0 and 1 then

\* 'flags' would contain 0x03.

\*

\* wait\_type specifies whether you want ALL bits to be set/cleared or ANY of the bits

\* to be set/cleared.

\* You can specify the following argument:

\*

\* OS\_FLAG\_WAIT\_CLR\_ALL You will check ALL bits in 'mask' to be clear (0)

\* OS\_FLAG\_WAIT\_CLR\_ANY You will check ANY bit in 'mask' to be clear (0)

\* OS\_FLAG\_WAIT\_SET\_ALL You will check ALL bits in 'mask' to be set (1)

\* OS\_FLAG\_WAIT\_SET\_ANY You will check ANY bit in 'mask' to be set (1)

\*

\* timeout is the desired amount of time that the task will wait for the event flag

\* bit(s) to be set.

\*

\* Returns : none

\*

\* Called by : OSFlagPend() OS\_FLAG.C

\*

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

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

\*/

static void OS\_FlagBlock (OS\_FLAG\_GRP \*pgrp,

OS\_FLAG\_NODE \*pnode,

OS\_FLAGS flags,

INT8U wait\_type,

INT32U timeout)

{

OS\_FLAG\_NODE \*pnode\_next;

INT8U y;

OSTCBCur->OSTCBStat |= OS\_STAT\_FLAG;

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

OSTCBCur->OSTCBDly = timeout; /\* Store timeout in task's TCB \*/

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

OSTCBCur->OSTCBFlagNode = pnode; /\* TCB to link to node \*/

#endif

pnode->OSFlagNodeFlags = flags; /\* Save the flags that we need to wait for \*/

pnode->OSFlagNodeWaitType = wait\_type; /\* Save the type of wait we are doing \*/

pnode->OSFlagNodeTCB = (void \*)OSTCBCur; /\* Link to task's TCB \*/

pnode->OSFlagNodeNext = pgrp->OSFlagWaitList; /\* Add node at beginning of event flag wait list \*/

pnode->OSFlagNodePrev = (void \*)0;

pnode->OSFlagNodeFlagGrp = (void \*)pgrp; /\* Link to Event Flag Group \*/

pnode\_next = (OS\_FLAG\_NODE \*)pgrp->OSFlagWaitList;

if (pnode\_next != (void \*)0) { /\* Is this the first NODE to insert? \*/

pnode\_next->OSFlagNodePrev = pnode; /\* No, link in doubly linked list \*/

}

pgrp->OSFlagWaitList = (void \*)pnode;

y = OSTCBCur->OSTCBY; /\* Suspend current task until flag(s) received \*/

OSRdyTbl[y] &= (OS\_PRIO)~OSTCBCur->OSTCBBitX;

if (OSRdyTbl[y] == 0x00u) {

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

}

}

/\*

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

\* INITIALIZE THE EVENT FLAG MODULE

\*

\* Description: This function is called by uC/OS-II to initialize the event flag module. Your application

\* MUST NOT call this function. In other words, this function is internal to uC/OS-II.

\*

\* Arguments : none

\*

\* Returns : none

\*

\* WARNING : You MUST NOT call this function from your code. This is an INTERNAL function to uC/OS-II.

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

\*/

void OS\_FlagInit (void)

{

#if OS\_MAX\_FLAGS == 1u

OSFlagFreeList = (OS\_FLAG\_GRP \*)&OSFlagTbl[0]; /\* Only ONE event flag group! \*/

OSFlagFreeList->OSFlagType = OS\_EVENT\_TYPE\_UNUSED;

OSFlagFreeList->OSFlagWaitList = (void \*)0;

OSFlagFreeList->OSFlagFlags = (OS\_FLAGS)0;

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

OSFlagFreeList->OSFlagName = (INT8U \*)"?";

#endif

#endif

#if OS\_MAX\_FLAGS >= 2u

INT16U ix;

INT16U ix\_next;

OS\_FLAG\_GRP \*pgrp1;

OS\_FLAG\_GRP \*pgrp2;

OS\_MemClr((INT8U \*)&OSFlagTbl[0], sizeof(OSFlagTbl)); /\* Clear the flag group table \*/

for (ix = 0u; ix < (OS\_MAX\_FLAGS - 1u); ix++) { /\* Init. list of free EVENT FLAGS \*/

ix\_next = ix + 1u;

pgrp1 = &OSFlagTbl[ix];

pgrp2 = &OSFlagTbl[ix\_next];

pgrp1->OSFlagType = OS\_EVENT\_TYPE\_UNUSED;

pgrp1->OSFlagWaitList = (void \*)pgrp2;

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

pgrp1->OSFlagName = (INT8U \*)(void \*)"?"; /\* Unknown name \*/

#endif

}

pgrp1 = &OSFlagTbl[ix];

pgrp1->OSFlagType = OS\_EVENT\_TYPE\_UNUSED;

pgrp1->OSFlagWaitList = (void \*)0;

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

pgrp1->OSFlagName = (INT8U \*)(void \*)"?"; /\* Unknown name \*/

#endif

OSFlagFreeList = &OSFlagTbl[0];

#endif

}

/\*

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

\* MAKE TASK READY-TO-RUN, EVENT(s) OCCURRED

\*

\* Description: This function is internal to uC/OS-II and is used to make a task ready-to-run because the

\* desired event flag bits have been set.

\*

\* Arguments : pnode is a pointer to a structure which contains data about the task waiting for

\* event flag bit(s) to be set.

\*

\* flags\_rdy contains the bit pattern of the event flags that cause the task to become

\* ready-to-run.

\*

\* Returns : OS\_TRUE If the task has been placed in the ready list and thus needs scheduling

\* OS\_FALSE The task is still not ready to run and thus scheduling is not necessary

\*

\* Called by : OSFlagsPost() OS\_FLAG.C

\*

\* Note(s) : 1) This function assumes that interrupts are disabled.

\* 2) This function is INTERNAL to uC/OS-II and your application should not call it.

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

\*/

static BOOLEAN OS\_FlagTaskRdy (OS\_FLAG\_NODE \*pnode,

OS\_FLAGS flags\_rdy)

{

OS\_TCB \*ptcb;

BOOLEAN sched;

ptcb = (OS\_TCB \*)pnode->OSFlagNodeTCB; /\* Point to TCB of waiting task \*/

ptcb->OSTCBDly = 0u;

ptcb->OSTCBFlagsRdy = flags\_rdy;

ptcb->OSTCBStat &= (INT8U)~(INT8U)OS\_STAT\_FLAG;

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

if (ptcb->OSTCBStat == OS\_STAT\_RDY) { /\* Task now ready? \*/

OSRdyGrp |= ptcb->OSTCBBitY; /\* Put task into ready list \*/

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

sched = OS\_TRUE;

} else {

sched = OS\_FALSE;

}

OS\_FlagUnlink(pnode);

return (sched);

}

/\*

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

\* UNLINK EVENT FLAG NODE FROM WAITING LIST

\*

\* Description: This function is internal to uC/OS-II and is used to unlink an event flag node from a

\* list of tasks waiting for the event flag.

\*

\* Arguments : pnode is a pointer to a structure which contains data about the task waiting for

\* event flag bit(s) to be set.

\*

\* Returns : none

\*

\* Called by : OS\_FlagTaskRdy() OS\_FLAG.C

\* OSFlagPend() OS\_FLAG.C

\* OSTaskDel() OS\_TASK.C

\*

\* Note(s) : 1) This function assumes that interrupts are disabled.

\* 2) This function is INTERNAL to uC/OS-II and your application should not call it.

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

\*/

void OS\_FlagUnlink (OS\_FLAG\_NODE \*pnode)

{

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

OS\_TCB \*ptcb;

#endif

OS\_FLAG\_GRP \*pgrp;

OS\_FLAG\_NODE \*pnode\_prev;

OS\_FLAG\_NODE \*pnode\_next;

pnode\_prev = (OS\_FLAG\_NODE \*)pnode->OSFlagNodePrev;

pnode\_next = (OS\_FLAG\_NODE \*)pnode->OSFlagNodeNext;

if (pnode\_prev == (OS\_FLAG\_NODE \*)0) { /\* Is it first node in wait list? \*/

pgrp = (OS\_FLAG\_GRP \*)pnode->OSFlagNodeFlagGrp;

pgrp->OSFlagWaitList = (void \*)pnode\_next; /\* Update list for new 1st node \*/

if (pnode\_next != (OS\_FLAG\_NODE \*)0) {

pnode\_next->OSFlagNodePrev = (OS\_FLAG\_NODE \*)0; /\* Link new 1st node PREV to NULL \*/

}

} else { /\* No, A node somewhere in the list \*/

pnode\_prev->OSFlagNodeNext = pnode\_next; /\* Link around the node to unlink \*/

if (pnode\_next != (OS\_FLAG\_NODE \*)0) { /\* Was this the LAST node? \*/

pnode\_next->OSFlagNodePrev = pnode\_prev; /\* No, Link around current node \*/

}

}

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

ptcb = (OS\_TCB \*)pnode->OSFlagNodeTCB;

ptcb->OSTCBFlagNode = (OS\_FLAG\_NODE \*)0;

#endif

}

#endif