PlayStation®2 IOP Library Reference
Release 2.4.3

Kernel Libraries

Publication date: January 2002

Sony Computer Entertainment Inc.
1-1, Akasaka 7-chome, Minato-ku
Tokyo 107-0052, Japan

Sony Computer Entertainment America
919 E. Hillsdale Blvd.
Foster City, CA 94404, U.S.A.

Sony Computer Entertainment Europe
30 Golden Square
London W1F 9LD, U.K.

Summary Table of Contents

About This Manual

This is the Runtime Library Release 2.4.3 version of the *PlayStation®2 IOP Library Reference - Kernel Libraries* manual.

The purpose of this manual is to define all available PlayStation®2 IOP kernel library structures and functions. The companion *PlayStation®2 IOP Library Overview - Kernel Libraries* describes the structure and purpose of the library.

Changes Since Last Release

Chapter 2: IOP Kernel Library

- The description of ReleaseIntrHandler() has been removed.
- In the "Notes" section of alarmhandler(), a description on the maximum clock speed represented with a 32-bit unsigned integer of the return value has been added.

Related Documentation

Library specifications for the EE can be found in the *PlayStation®2 EE Library Reference* manuals and the *PlayStation®2 EE Library Overview* manuals.

Note: the Developer Support Web site posts current developments regarding the Libraries and also provides notice of future documentation releases and upgrades.

Typographic Conventions

Certain Typographic Conventions are used throughout this manual to clarify the meaning of the text:

Convention	Meaning
`courier`	Indicates literal program code.
italic	Indicates names of arguments and structure members (in structure/function definitions only).
medium bold	Indicates data types and structure/function names (in structure/function definitions only).
blue	Indicates a hyperlink.

Developer Support

Sony Computer Entertainment America (SCEA)

SCEA developer support is available to licensees in North America only. You may obtain developer support or additional copies of this documentation by contacting the following addresses:

Order Information	Developer Support
In North America:	*In North America:*
Attn: Developer Tools Coordinator Sony Computer Entertainment America 919 East Hillsdale Blvd. Foster City, CA 94404, U.S.A. Tel: (650) 655-8000	E-mail: PS2_Support@playstation.sony.com Web: http://www.devnet.scea.com/ Developer Support Hotline: (650) 655-5566 (Call Monday through Friday, 8 a.m. to 5 p.m., PST/PDT)

Sony Computer Entertainment Europe (SCEE)

SCEE developer support is available to licensees in Europe only. You may obtain developer support or additional copies of this documentation by contacting the following addresses:

Order Information	Developer Support
In Europe:	*In Europe:*
Attn: Production Coordinator Sony Computer Entertainment Europe 30 Golden Square London W1F 9LD, U.K. Tel: +44 (0) 20 7859-5000	E-mail: ps2_support@scee.net Web: https://www.ps2-pro.com/ Developer Support Hotline: +44 (0) 20 7859-5777 (Call Monday through Friday, 9 a.m. to 6 p.m., GMT)

Chapter 1: Standard C Functions
Table of Contents

Built-in Basic C Functions

atob

Convert decimal string to numeric value

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <string.h>	
char *atob(	
char **s*,	String to be converted
int **i***);**	Pointer to int-type variable for storing conversion result

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function interprets the string given by s as a decimal number, converts it to a numeric value, and stores the conversion result in the int-type variable pointed to by i. A pointer to the remaining unconverted string is returned as the return value.

Return value

string	Pointer to remaining unconverted string

atoi

Convert decimal string to int-type numeric value (macro)

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <stdlib.h>

int atoi(

const char **s*);	String to be converted

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function interprets the string given by s as a decimal number and converts it to a numeric value. This is a macro.

Return value

int-type numeric value	Conversion result

atol

Convert decimal string to long-type numeric value (macro)

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <stdlib.h>	
long atol(	
const char **s***);**	String to be converted

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function interprets the string given by s as a decimal number and converts it to a numeric value. This is a macro.

Return value

long-type numeric value	Conversion result

bcmp

Compare memory

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <memory.h>

int bcmp(

const void **s1,*	Address of data to be compared
const void **s2,*	Address of data to be compared
size_t *n***);**	Number of bytes to be compared

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function compares consecutive elements of the two unsigned char-type arrays of size n given by s1 and s2 until a different element is found. This function is equivalent to memcmp().

Return value

<0	Smaller element was found in s1
=0	All elements were equal
>0	Larger element was found in s1

bcopy

Copy memory

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <memory.h>

void bcopy(

const void **src,*	Copy source
void **dest,*	Copy destination
size_t *n***);**	Number of bytes to be copied

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function copies the unsigned char-type array of size n given by src to the array given by dest. Accurate copying of the src data to dest is guaranteed when the arrays overlap. This function is equivalent to calling memmove(dest,src,n) except for the return value.

If it is certain that the arrays are non-overlapping, the memcpy() function should be used for better performance.

Return value

None

bzero

Zero clear memory

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <memory.h>	
void bzero(	
void **s,*	Memory address
size_t *n***);**	Number of bytes

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function assigns 0 to all elements of the unsigned char-type array of size n given by s.

Return value

None

index

Search for character within string

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <string.h>	
char *index(	
const char **s,*	String to be searched
int *c***);**	Search character

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function searches for a character equal to c within the string given by s and returns a pointer to the first one that was found. If the character cannot be found, this function returns NULL.

Return value

=NULL	Character was not found
!=NULL	Pointer to character that was found

isalnum

Test for alphanumeric character (macro)

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <ctype.h>	
int isalnum(	
char *c***);**	Character to be tested

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function tests whether c is an alphabetic character or digit. This function works correctly only for ascii characters.

Return value

Test result (true or false)

isalpha

Test for alphabetic character (macro)

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <ctype.h>	
int isalpha(	
char *c***);**	Character to be tested

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function tests whether c is an alphabetic character. This function works correctly only for ascii characters.

Return value

Test result (true or false)

isascii

Test for ascii character (macro)

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <ctype.h>	
int isascii(	
char *c***);**	Character to be tested

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function tests whether c is an ascii character having a code value less than 0x80.

Return value

Test result (true or false)

iscntrl

Test for control character (macro)

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <ctype.h>	
int iscntrl(	
char *c***);**	Character to be tested

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function tests whether c is a delete character or general control character. This function works correctly only for ascii characters.

Return value

Test result (true or false)

isdigit

Test for digit (macro)

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <ctype.h>

int isdigit(

char *c***);** Character to be tested

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function tests whether c is a digit in the range 0 to 9. This function works correctly only for ascii characters.

Return value

Test result (true or false)

isgraph

Test for visible graphic character (macro)

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <ctype.h>	
int isgraph(	
char *c***);**	Character to be tested

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function tests whether c is a visible graphic character. This function works correctly only for ascii characters.

Return value

Test result (true or false)

islower

Test for lowercase letter (macro)

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <ctype.h>	
int islower(	
char *c***);**	Character to be tested

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function tests whether c is a lowercase letter. This function works correctly only for ascii characters.

Return value

Test result (true or false)

isprint

Test for printing character (macro)

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <ctype.h>	
int isprint(	
char *c***);**	Character to be tested

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function tests whether c is a printing character. This function works correctly only for ascii characters.

Return value

Test result (true or false)

ispunct

Test for punctuation character (macro)

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <ctype.h>

int ispunct(

char *c***);**	Character to be tested

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function tests whether c is a punctuation character (excluding control characters and alphanumeric characters). This function works correctly only for ascii characters.

Return value

Test result (true or false)

isspace

Test for space character (macro)

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <ctype.h>

int isspace(

char *c***);**	Character to be tested

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function tests whether c is a SPACE, TAB, RETURN, NEWLINE, FORMFEED, or vertical tab. This function works correctly only for ascii characters.

Return value

Test result (true or false)

isupper

Test for uppercase letter (macro)

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <ctype.h>

int isupper(

char *c***);**	Character to be tested

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function tests whether c is an uppercase letter. This function works correctly only for ascii characters.

Return value

Test result (true or false)

isxdigit

Test for hexadecimal digit

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <ctype.h>

int isxdigit(

char *c***);**	Character to be tested

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function tests whether c is a hexadecimal digit in the ranges 0 to 9, A to F, or a to f. This function works correctly only for ascii characters.

Return value

Test result (true or false)

longjmp

Non-local jump

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <setjmp.h>	
void longjmp(	
jmp_buf *env,*	Jump destination context
int *value***);**	Return value after jump

Calling conditions

Can be called from a thread

Multithread safe

Description

This function causes a second return to setjmp having the context that was saved in env. Then, setjmp will return value.

Return value

None

memchr

Search for data within memory

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <memory.h>	
void *memchr(	
const void **s,*	Array to be searched
int *c,*	Search data
size_t *n***);**	Number of bytes of array to be searched

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function searches for an element equal to c within the unsigned char-type array of size n given by s. If such an element is found, the function returns a pointer to that element. If no such element is found, the function returns NULL.

Return value

=NULL	Character was not found
!=NULL	Pointer to character that was found

memcmp

Compare memory

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <memory.h>	
int memcmp(	
const void **s1,*	Address of data to be compared
const void **s2,*	Address of data to be compared
size_t *n***);**	Number of bytes to be compared

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function compares consecutive elements of the two unsigned char-type arrays of size n given by s1 and s2 until a different element is found.

Return value

<0	Smaller element was found in s1
=0	All elements were equal
>0	Larger element was found in s1

memcpy

Copy memory

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <memory.h>

void *memcpy(

void **dest,*	Copy destination
const void **src,*	Copy source
size_t *n***);**	Number of bytes to be copied

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function copies the unsigned char-type array of size n given by src to the array given by dest. The copy operation is not guaranteed when the arrays overlap. When all arguments are multiples of 4, copying can be executed much faster by calling wmemcopy().

Return value

Value of dest

memmove

Move data in memory

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <memory.h>	
void *memmove(	
void **dest,*	Copy destination
const void **src,*	Copy source
size_t *n***);**	Number of bytes to be copied

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function copies the unsigned char-type array of size n given by src to the array given by dest. Accurate copying of the src data to dest is guaranteed when the arrays overlap.

Return value

Value of dest

memset

Set memory value

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <memory.h>	
void *memset(	
void **s,*	Memory address
int *c,*	Configuration value
size_t *n***);**	Number of bytes

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function assigns c to all elements of the unsigned char-type array of size n given by s.

Return value

Value of s

rindex

Search for character within string

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <string.h>	
char *rindex(	
const char **s,*	String to be searched
int *c***);**	Search character

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function searches for a character equal to c within the string given by s and returns a pointer to the last one that was found. If the character cannot be found, this function returns NULL.

Return value

=NULL	Character was not found
!=NULL	Pointer to character that was found

setjmp

Set non-local jump point

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <setjmp.h>

int setjmp(

jmp_buf *env***);**	Jump destination context

Calling conditions

Can be called from a thread

Multithread safe

Description

This function saves the current context in env and returns 0.

Return value

0 or value of longjmp

sprintf

Convert data output format

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <stdio.h>	
int sprintf(	
char **buf,*	Character array where conversion result is stored
const char **format,*	Conversion format
...);	

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function converts the argument data to a string according to the format indicated by format and stores it in buf.

For the formats that are supported by the format argument, see the description of printf().

Return value

Number of converted characters

strcat

Concatenate strings

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <string.h>	
char *strcat(	
char **dest,*	String that is concatenated to
const char **src***);**	String to be concatenated

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function copies the string indicated by src so that it is concatenated to the end of the string indicated by dest.

Return value

Value of dest

strchr

Search for character within string

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <string.h>

char *strchr(

const char **s,*	String to be searched
int *c***);**	Search data

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function searches for a character equal to c within the string given by s. If such a character is found, this function returns a pointer to the first character that was found. If no such character is found, this function returns NULL.

Return value

=NULL	Character was not found
!=NULL	Pointer to character that was found

strcmp

Compare strings

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <string.h>

int strcmp(

const char **s1,*	String to be compared
const char **s2***);**	String to be compared

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function compares the strings given by s1 and s2 from the beginning of the strings until a different character is found.

Return value

<0	Smaller element was found in s1
=0	All elements were equal
>0	Larger element was found in s1

strcpy

Copy string

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <string.h>

char *strcpy(

char **dest,*	Copy destination
const char **src***);**	Copy source

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function copies the string indicated by src to dest. The copy operation is not guaranteed when src and dest overlap.

Return value

Value of dest

strcspn

Search for set of characters from string

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <string.h>	
int strcspn(	
const char **s1,*	String to be searched
const char **s2***);**	Character set

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function searches for the first character within string s1 that is equal to any of the elements of the string s2 and returns the index of that character within s1. If no such character is found, this function returns the length of string s1.

Return value

Index of character that was found

strlen

Find length of string

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <string.h>

size_t strlen(

const char **s***);**	String to be examined

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function finds the length of the string indicated by s.

Return value

Length of string

strncat

Concatenate strings (with length restriction)

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <string.h>	
char *strncat(	
char **dest,*	String that is concatenated to
const char **src,*	String to be concatenated
size_t *n*);	Maximum number of bytes to be concatenated

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function copies the string indicated by src so that it is concatenated to the end of the string indicated by dest until at most n characters have been copied, not including the terminating NULL character. At the end, dest will be terminated by a NULL character.

Return value

Value of dest

strncmp

Compare strings (with length restriction)

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <string.h>	
int strncmp(	
const char **s1,*	String to be compared
const char **s2,*	String to be compared
size_t *n***);**	Maximum number of characters to be compared

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function compares the strings given by s1 and s2 from the beginning of the strings until either a different character is found or n characters were compared.

Return value

<0	Smaller element was found in s1
=0	All elements were equal
>0	Larger element was found in s1

strncpy

Copy string (with length specification)

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <string.h>	
char *strncpy(	
char **dest,*	Copy destination
const char **src,*	Copy source
size_t *n***);**	Number of characters to be copied

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function copies the first n characters of the string indicated by src to dest. If src does not have n characters, NULL characters are copied for the remaining portion. Note that the dest string will not necessarily be terminated with a NULL character. The copy operation is not guaranteed when src and dest overlap.

Return value

Value of dest

strpbrk

Search for set of characters from string

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <string.h>

char *strpbrk(

const char **s1,*	String to be searched
const char **s2***);**	Character set

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function searches for the first character contained in string s1 that is equal to any of the elements of the string s2 and returns a pointer to that character. If no such character is found, this function returns NULL.

Return value

!=NULL	Pointer to character that was found
=NULL	Character was not found

strrchr

Search for character within string

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <string.h>	
char *strrchr(	
const char **s,*	String to be searched
int *c***);**	Search data

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function searches for a character equal to c within the string given by s. If such a character is found, this function returns a pointer to the last character that was found. If no such character is found, this function returns NULL.

Return value

=NULL	Character was not found
!=NULL	Pointer to character that was found

strspn

Search for character from string that is not within set of characters

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <string.h>	
int strspn(	
const char **s1,*	String to be searched
const char **s2***);**	Character set

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function searches for a character within string s1 that is not equal to any of the elements of the string s2 and returns the index of that character within s1. If no such character is found, this function returns the length of string s1.

Return value

Index of character that was found

strstr

Indicate position of substring

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <string.h>

char *strstr(

const char **s1,*	String to be searched
const char **s2*);	Search string

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function searches for a portion of string s1 that matches string s2, and if such a substring is found, this function returns a pointer to the first character of that substring.

Return value

=NULL	Substring was not found
!=NULL	Pointer to first character of substring that was found

strtok

Divide string into tokens

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <string.h>

char *strtok(

char **s,*	String to be divided
const char **delim*);	String where division delimiters are stored

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Not multithread safe

Description

This function divides the string indicated by s using the delim string in which delimiters are stored.

When this function is first called with a string assigned, the address of the string to be divided is recorded in a static variable within strtok(). By specifying NULL for s the second and subsequent times this function is called, a divided string can be obtained piece by piece. If delimiters appear consecutively within the string to be divided, those other than the first delimiter are ignored. Therefore, the division result contains no string of length 0. The delimiter parts of the original string to be divided are overwritten with NULL characters.

It is clear from the description given above that this function is not multithread safe.

Return value

Divided string

strtol

Convert string to long-type numeric value

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <stdlib.h>

long strtol(

const char **s,*	String to be converted
char ***endp,*	Pointer to variable for returning uninterpreted part of string
int *base***);**	Value of base for conversion (when base is 0, the base is automatically recognized; when this is 1 to 36, it indicates the specified base)

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function converts the string specified by s to a long-type numeric value as a function of base. If endp is not NULL, a pointer to the character where the string interpretation ended will be stored in endp.

Return value

Conversion result

strtoul

Convert string to long-type numeric value

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <stdlib.h>

unsigned long strtoul(

const char **s,*	String to be converted
char ***endp,*	Pointer to variable for returning uninterpreted part of string
int *base*);	Value of base for conversion (when base is 0, the base is automatically recognized; when this is 1 to 36, it indicates the specified base)

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function converts the string specified by s to an unsigned long-type numeric value as a function of base. If endp is not NULL, a pointer to the character where the string interpretation ended will be stored in endp.

Return value

Conversion result

toascii

Convert to ascii character (macro)

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <ctype.h>

int toascii(

char *c***);**	Character to be converted

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function uses an appropriate value to mask c so that c becomes an ASCII character having a code value from 0 to 0x7f. However, this function will not map from a non-ASCII coded character set to ASCII.

Return value

Conversion result

tolower

Convert to lowercase

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <ctype.h>	
char tolower(	
char *ch***);**	Character to be converted

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function converts ch to the equivalent lowercase character. This function works correctly only for ascii characters.

Return value

Conversion result

toupper

Convert to uppercase

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <ctype.h>	
char toupper(	
char *ch***);**	Character to be converted

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function converts ch to the equivalent uppercase character. This function works correctly only for ascii characters.

Return value

Conversion result

vsprintf

Convert data output format

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <stdarg.h>

#include <stdio.h>

int vsprintf(

char **buf,*	Character array where conversion result is stored
const char **format,*	Conversion format
va_list *ap***);**	Conversion argument data list

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function converts the argument data list ap to a string according to the format indicated by format and stores it in buf. For the formats that are supported by the format argument, see the description of printf().

Return value

Number of converted characters

wmemcopy

Copy memory in words

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <memory.h>

void *wmemcopy(

u_long **dest,*	Copy destination
const u_long **src,*	Copy source
u_long *bytes***);**	Number of bytes to be copied (must be a multiple of 4)

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function copies the unsigned long-type array given by src, which has a size in bytes equal to the value of the bytes argument, to the array given by dest. It has been tuned so that the array can be copied very fast by taking into account the cache line of the IOP.

Return value

Value of dest

wmemset

Set value in memory in words

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <memory.h>	
void *wmemset(	
u_long **dest,*	Memory address
u_long *c,*	Configuration value
u_long *bytes***);**	Number of bytes (must be a multiple of 4)

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function assigns c to all elements of the unsigned long-type array given by dest, which has a size in bytes equal to the value of the bytes argument.

Return value

Value of dest

Basic Character Input/Output Functions

fdgetc

Read one character from file

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <stdio.h>

int fdgetc(

int *fd***);**	File descriptor obtained when file was opened with open()

Calling conditions

Can be called from a thread

Multithread safe (must be called in interrupt-enabled state)

Description

This function reads one character from the file indicated by the file descriptor fd.

Note: Character reading returns the raw data that is returned by the device driver. No end-of-line character conversion is performed. Also, no echo back is performed.

Return value

Character that was read or EOF

fdgets

Read one line from file

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <stdio.h>

char *fdgets(

char **buf,*	Read buffer
int *fd***);**	File descriptor obtained when file was opened with open()

Calling conditions

Can be called from a thread

Multithread safe (must be called in interrupt-enabled state)

Description

This function reads one line from the file indicated by the file descriptor fd and stores it in buf.

When the file indicated by fd is a TTY-type character device, fdgets() itself performs simple editing functions such as echo back or character deletion by a backspace. The character string read does not include the end-of-line character.

Return value

Value of buf

fdprintf

File output with output format conversion

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <stdio.h>

int fdprintf(

int *fd,*	File descriptor obtained when file was opened with open()
const char **format,*	Output format
...);	

Calling conditions

Can be called from a thread

Multithread safe (must be called in interrupt-enabled state)

Description

This function converts the argument data to a character string according to the format indicated by the format argument and outputs it to the file indicated by the file descriptor fd. For the formats that are supported by the format argument, see the description of printf().

Return value

Number of characters that were output

fdputc

Write one character to file

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <stdio.h>	
int fdputc(	
int *c*,	Character to be output
int *fd***);**	File descriptor obtained when file was opened with open()

Calling conditions

Can be called from a thread

Multithread safe (must be called in interrupt-enabled state)

Description

This function writes one character to the file indicated by the file descriptor fd.

Note: Character writing passes raw data to the device driver. No end-of-line character conversion is performed.

Return value

Character that was output

fdputs

Write character string to file

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <stdio.h>	
int fdputs(	
const char **s,*	Character string to be output
int *fd***);**	File descriptor obtained when file was opened with open()

Calling conditions

Can be called from a thread

Multithread safe (must be called in interrupt-enabled state)

Description

This function writes a character string to the file indicated by the file descriptor fd.

Note: Character writing passes raw data to the device driver. No end-of-line character conversion is performed.

Return value

0

getchar

Read one character from standard input

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <stdio.h>

int getchar();

Calling conditions

Can be called from a thread

Multithread safe (must be called in interrupt-enabled state)

Description

This function reads one character from standard input (file descriptor 0).

Note: Character reading returns raw data that is returned by the device driver. No end-of-line character conversion is performed. Also, no echo back is performed.

Return value

Character that was read or EOF

gets

Read one line from standard input

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <stdio.h>	
char *gets(	
char **buf***);**	Read buffer

Calling conditions

Can be called from a thread

Multithread safe (must be called in interrupt-enabled state)

Description

This function reads one line from standard input (file descriptor 0) and stores it in buf. It differs from getchar() in that gets() itself performs simple editing functions such as echo back or character deletion by a backspace. The character string read does not include the end-of-line character.

Return value

Value of buf

printf

Output to standard output with output format conversion

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <stdio.h>

int printf(

const char **format,*	Output format
...);	

Calling conditions

Can be called from a thread

Multithread safe (must be called in interrupt-enabled state)

Description

This function converts the argument data to a character string according to the format indicated by the format argument and outputs it to standard output (file descriptor 1). The formats that are supported by the format argument are as follows.

Flags -, +, #, blank

Field width Decimal number, *

Precision specification h, l, c,

Conversion type D, d, i, O, o, p, u, x, X, s, c, n, %

Return value

Number of characters that were output

putchar

Output one character to standard output

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <stdio.h>	
int putchar(	
int *c*);	Character to be output

Calling conditions

Can be called from a thread

Multithread safe (must be called in interrupt-enabled state)

Description

This function writes one character to standard output (file descriptor 1).

Note: Character writing passes raw data to the device driver. No end-of-line character conversion is performed.

Return value

Character that was output

puts

Output character string to standard output

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <stdio.h>

int puts(

const char **s***);** Character string to be output

Calling conditions

Can be called from a thread

Multithread safe (must be called in interrupt-enabled state)

Description

This function writes a character string to standard output (file descriptor 1).

Note: Character writing passes raw data to the device driver. No end-of-line character conversion is performed.

Return value

0

vfdprintf

File output with output format conversion

Library	*Introduced*	*Documentation last modified*
c	2.4	October 11, 2001

Syntax

#include <stdio.h>

int vfdprintf(

int *fd,*	File descriptor obtained when file was opened with open()
const char **format,*	Output format
va_list *ap***);**	Conversion argument data list

Calling conditions

Can be called from a thread

Multithread safe (must be called in interrupt-enabled state)

Description

This function converts the argument data list ap to a character string according to the format indicated by the format argument and outputs it to the file indicated by the file descriptor fd. For the formats that are supported by the format argument, see the description of printf().

Return value

Number of characters that were output

Chapter 2: IOP Kernel Library
Table of Contents

System Memory Management Functions

AllocLoadMemory

Allocate memory area dedicated for module loading

Library	*Introduced*	*Documentation last modified*
ikrnl	2.3.4	August 31, 2001

Syntax

#include <kernel.h>

void * AllocLoadMemory(

int *type,*	Memory allocation policy specified as either SMEM_Low, SMEM_High, or SMEM_Addr.
unsigned long *size,*	Allocation memory size in bytes.
void **addr***);**	Address when type==SMEM_Addr.

Calling conditions

Can be called from a thread

Not multithread safe

Description

This function allocates the number of bytes of memory specified by the *size* argument, where the allocated memory will be used only for module loading. The Load ModuleAddress(), LoadModuleBufferAddress(), and LoadModuleWithOption() functions, which will be described later, can be used to place multiple modules in the memory area that was allocated by AllocLoadMemory().

When *type* is SMEM_Low, this function will search for an empty area to allocate beginning with the lowest memory address.

When *type* is SMEM_High, this function will search for an empty area to allocate beginning with the highest memory address.

When *type* is SMEM_Addr, this function will allocate an area beginning with the address specified by addr.

Return value

Non-NULL	Starting address of allocated memory
NULL	Allocation failed

AllocSysMemory

Allocate memory area

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	August 31, 2001

Syntax

#include <kernel.h>

void * AllocSysMemory(

int *type,*	Memory allocation policy specified as either SMEM_Low, SMEM_High, or SMEM_Addr.
unsigned long *size,*	Allocation memory size in bytes.
void **addr*);	Address when type==SMEM_Addr. A multiple of 256 should be specified.

Calling conditions

Can be called from a thread

Not multithread safe

Description

This function allocates the number of bytes of memory specified by the *size* argument rounded up to a multiple of 256. The allocation address that is returned when allocation succeeds will always be a multiple of 256.

When *type* is SMEM_Low, this function will search for an empty area to allocate beginning with the lowest memory address.

When *type* is SMEM_High, this function will search for an empty area to allocate beginning with the highest memory address.

When *type* is SMEM_Addr, this function will allocate an area beginning with the address specified by addr.

Return value

Non-NULL	Starting address of allocated memory
NULL	Allocation failed

FreeLoadMemory

Free memory area

Library	*Introduced*	*Documentation last modified*
ikrnl	2.3.4	August 31, 2001

Syntax

#include <kernel.h>	
int FreeLoadMemory(	
void **area***);**	Starting address of memory area to be freed

Calling conditions

Can be called from a thread

Not multithread safe

Description

This function frees memory that was allocated by AllocLoadMemory().

Return value

KE_OK	Normal termination
KE_ERROR	Specified area was not allocated
KE_MEMINUSE	Module remains in memory area

FreeSysMemory

Free memory

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int FreeSysMemory(

void **area);*	Starting address of memory area to be freed

Calling conditions

Can be called from a thread

Not multithread safe

Description

Frees the memory specified by *area*.

Return value

KE_OK	Normal termination
KE_ERROR	Specified area had not been allocated

QueryBlockSize

Query size of a memory block

Library	*Introduced*	*Documentation last modified*
ikrnl	2.2	March 26, 2001

Syntax

#include <kernel.h>

unsigned long QueryBlockSize(

void ****addr*);**	Arbitrary address

Calling conditions

Can be called from a thread

Not multithread safe

Description

Checks to see which memory block is associated with the address specified by the *addr* argument, and returns the size of the memory block.

The most significant bit of the return value indicates the state of the memory block. If the bit is 1, the memory block is in an unallocated state. If the bit is 0, then the memory block is in an allocated state.

The system memory manager manages memory in units of memory blocks. Memory areas are allocated and unallocated using AllocSysMemory().

Return value

Not KE_ERROR	The most significant bit is the memory state, other bits are the address.
KE_ERROR	The address is invalid, and is outside of the process.

QueryBlockTopAddress

Query memory block

Library	*Introduced*	*Documentation last modified*
ikrnl	2.2	March 26, 2001

Syntax

#include <kernel.h>

void *QueryBlockTopAddress(

void ****addr*);**	Arbitrary address

Calling conditions

Can be called from a thread

Not multithread safe

Description

Checks to see which memory block is associated with the address specified by the *addr* argument, and returns the starting address of the memory block.

The most significant bit of the return value indicates the state of the memory block. If the bit is 1, the memory block is in an unallocated state. If the bit is 0, then the memory block is in an allocated state.

The system memory manager manages memory in units of memory blocks. Memory areas are allocated and unallocated using AllocSysMemory().

Return value

Not KE_ERROR	The most significant bit is the memory state, other bits are the address.
KE_ERROR	The address is invalid, and is outside of the process.

QueryMaxFreeMemSize

Obtain maximum memory size that can be allocated

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

```
#include  <kernel.h>
unsigned long QueryMaxFreeMemSize();
```

Calling conditions

Can be called from a thread

Not multithread safe

Description

Obtains the size of the largest block among the memory blocks that can be allocated.

Return value

Positive (>=0)	Number of bytes in maximum memory block that can be allocated

QueryMemSize

Obtain total memory size

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

unsigned long QueryMemSize();

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

Obtains the size of memory that is being managed by the system memory manager.

Return value

Positive (>0)	Total number of bytes of memory being managed

QueryTotalFreeMemSize

Obtain total memory size that can be allocated

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	October 11, 2001

Syntax

#include <kernel.h>

unsigned long QueryTotalFreeMemSize();

Calling conditions

Can be called from a thread

Not multithread safe

Description

Obtains the total memory size that can be allocated.

Notes

The following example displays the free capacity and maximum free block size of the memory.

```
/* compile
    iop-elf-gcc iopmem.c -o iopmem.irx
*/

#include <kernel.h>
#include <stdio.h>

int start()
{
    int freesize, maxblock;

    maxblock = QueryMaxFreeMemSize();
    freesize = QueryTotalFreeMemSize();
    printf("IOP system memory  0x%x(%d) byte free, Max free block size 
0x%x\n",
           freesize, freesize, maxblock);
    return NO_RESIDENT_END;
}
```

Return value

Positive (>=0)	Total number of bytes of memory that can be allocated

Module Management Functions

GetModuleIdList

Get list of loaded program modules

Library	*Introduced*	*Documentation last modified*
ikrnl	2.3	July 2, 2001

Syntax

#include <kernel.h>

int GetModuleIdList(

int **readbuf,*	Pointer to an integer array that will store the module list.
int *readbufsize,*	Size of readbuf (number of entries that can be stored)
int **modulecount***);**	Pointer to variable that wil get the total number of modules. If NULL is specified, the total number of modules will not be obtained.

Calling conditions

Can be called from a thread

Multithread safe

Description

Gets a list of modules in memory.

Return value

Positive (>=0): Number of entries read into the buffer

GetModuleIdListByName

Get list of loaded program modules

Library	*Introduced*	*Documentation last modified*
ikrnl	2.3.4	August 31, 2001

Syntax

#include <kernel.h>

int GetModuleIdListByName (

const char **modulename,*	Module name.
int **readbuf,*	Pointer to beginning of integer array for storing module list.
int *readbufsize,*	Size of readbuf (number of entries that can be stored).
int **modulecount*)**;**	Pointer to variable for getting total number of modules. If NULL is specified, the total number of modules will not be obtained.

Calling conditions

Can be called from a thread

Multithread safe

Description

This function gets a list of module IDs for the module name specified by the argument modulename among the modules in memory.

Return value

Positive value (>=0)	Number of entries that were read into the buffer

LoadModule

Load program module from file

Library	*Introduced*	*Documentation last modified*
ikrnl	2.1	July 2, 2001

Syntax

#include <kernel.h>

int LoadModule

const char **filename);*	Name of file where program module is stored.

Calling conditions

Can be called from a thread

Multithread safe (must be called in an interrupt-enabled state)

Description

This function loads a program module from a file.

After the program module has been loaded, it must be started by calling StartModule().

Return value

Positive (>=0)	ID number of loaded module
KE_ILLEGAL_CONTEXT	Called from exception handler / interrupt handler
KE_ILLEGAL_OBJECT	Object file format is invalid
KE_LINKERR	Resident library required by loaded module does not exist
KE_NOFILE	Specified file was not found
KE_FILEERR	Error occurred when reading file
KE_NO_MEMORY	Insufficient memory

LoadModuleAddress

Load program module from file at specified address

Library	*Introduced*	*Documentation last modified*
ikrnl	2.3.4	August 31, 2001

Syntax

#include <kernel.h>

int LoadModuleAddress (

const char **filename,*	Name of file where program module is stored.
void **addr,*	Load starting address or address of allocated memory area.
int *offset***);**	Specifies 0, 1, or an offset from the beginning of the memory area as a multiple of 16.

Calling conditions

Can be called from a thread

Multithread safe (must be called in interrupt-enabled state)

Description

This function loads a program module at the specified address from a file. After the program module has been loaded, it must be started by calling StartModule().

There are three ways to use this function, according to the combination of addr and offset.

- addr == NULL

  In this case, LoadModuleAddress() behaves exactly like LoadModule().

- addr != NULL && offset == 0

  First, a memory area with a size needed to load the module is allocated using AllocSystemMemory(), then the program module is loaded. When a module that was loaded in this way is unloaded, the memory is freed with FreeSysMemory() in the same way as for a module that was loaded using LoadModule() / LoadStartModule().

- addr != NULL && offset != 0

  This combination means that the module is loaded in the memory area dedicated for module loading, which was allocated with AllocLoadMemory(). When a module that was loaded in this way is unloaded, the range that had been occupied by the module within the memory area dedicated for module loading will become unused, but the memory area dedicated for module loading itself will not be freed.

  By specifying an appropriate offset, an application program can intentionally control the placement of a module within the memory area dedicated for module loading.

  addr specifies the starting address of the memory area dedicated for module loading, and either of the following is specified for offset.

  When offset is 1, the module is loaded, following the module that was loaded last, in memory allocated by AllocLoadMemory().

  When offset is a multiple of 16 (greater than or equal to 32), the module is loaded at the offset location from the beginning of the memory allocated by AllocLoadMemory(). The module cannot be loaded so that it overlaps a previously loaded module.

Return value	
Positive (>=0)	ID number of loaded module
KE_ILLEGAL_CONTEXT	Called from exception handler / interrupt handler
KE_ILLEGAL_OBJECT	Object file format is invalid
KE_LINKERR	Resident library required by loaded module does not exist
KE_NOFILE	Specified file not found
KE_FILEERR	Error occurred while reading file
KE_NO_MEMORY	Insufficient memory
KE_ILLEGAL_OFFSET	Illegal offset argument value

LoadModuleBuffer

Load program module from memory

Library	*Introduced*	*Documentation last modified*
ikrnl	2.1	July 2, 2001

Syntax

#include <kernel.h>

int LoadModuleBuffer

const u_int **modbuf);*	Memory address where object data is stored

Calling conditions

Can be called from a thread

Multithread safe (must be called in an interrupt-enabled state)

Description

This function loads a program module from object data that was placed in memory.

After the program module has been loaded, it must be started by calling StartModule().

Return value

Positive (>=0)	ID number of loaded module
KE_ILLEGAL_CONTEXT	Called from exception handler / interrupt handler
KE_ILLEGAL_OBJECT	Object file format is invalid
KE_LINKERR	Resident library required by loaded module does not exist
KE_NO_MEMORY	Insufficient memory

LoadModuleBufferAddress

Load program module from memory according at specified address

Library	*Introduced*	*Documentation last modified*
ikrnl	2.3.4	August 31, 2001

Syntax

#include <kernel.h>

int LoadModuleBufferAddress (

const u_int **modbuf,*	Memory address where object data is stored.
void **addr,*	Load starting address or address of allocated memory area.
int *offset***);**	Specifies 0, 1, or an offset from the beginning of the memory area as a multiple of 16.

Calling conditions

Can be called from a thread

Multithread safe (must be called in interrupt-enabled state)

Description

This function loads a program module at the specified address from object data that was placed in memory.

The methods of specifying *addr* and *offset* are the same as those described for LoadModuleAddress().

After the program module is loaded, it must be started by calling StartModule().

Return value

Positive (>=0)	ID number of loaded module
KE_ILLEGAL_CONTEXT	Called from exception handler / interrupt handler
KE_ILLEGAL_OBJECT	Object file format is invalid
KE_LINKERR	Resident library required by loaded module does not exist
KE_MEMINUSE	Specified address already being used
KE_NO_MEMORY	Insufficient memory
KE_ILLEGAL_OFFSET	Illegal offset argument value

LoadModuleWithOption

LoadModule with option function

Library	*Introduced*	*Documentation last modified*
ikrnl	2.3.4	August 31, 2001

Syntax

#include <kernel.h>

int LoadModuleBufferAddress (

const char **filename*

Name of file where program module is stored.

const LMWOoption **option***);**

Pointer to LMWOoption structure that specifies behavior when module is loaded. The LMWOoption structure has the following members.

```
char      position;
char      access;
void      *distaddr;
int       distoffset;
LDfilefunc      *functable;
void      *funcopt;
```

The LMWOoption structure also has several reserved fields. 0 must be entered in the reserved fields in anticipation of future extensions. Therefore, execute memset(&option, 0, sizeof(LMWOoption)) before setting values in each of the members.

The contents of the various members of *option* are as follows.

position

Specifies one of the following indicating the module placement policy. This is similar to the type argument of AllocSystemMemory().

LMWO_POS_Low

Places the module at the lowest possible address (Same as normal LoadModule*())

LMWO_POS_High

Places the module at the highest possible address

LMWO_POS_Addr

Places the module according to the specifications of distaddr and distoffset, which are described below.

distaddr

Same as addr of LoadModuleAddress()

distoffset

Same as offset of LoadModuleAddress()

position
Specifies one of the following indicating the module placement policy. This is similar to the type argument of AllocSystemMemory().

LMWO_POS_Low
Places the module at the lowest possible address (Same as normal LoadModule*())

LMWO_POS_High
Places the module at the highest possible address

LMWO_POS_Addr
Places the module according to the specifications of distaddr and distoffset, which are described below.

access
Specifies one of the following indicating the object file access method.

LMWO_ACCESS_Noseek
Temporarily allocates a buffer for reading in the entire file, then reads the entire file in a single read operation. (Same as normal LoadModule*())

LMWO_ACCESS_Seekfew
Temporarily allocates a buffer for reading the file by individual ELF format sections, then reads the file in several read operations.

LMWO_ACCESS_Seekmany
Reads the file bit-by-bit in individual words without specifically allocating a buffer for reading the file.

functable
Pointer to file access function table.

If functable is set to NULL, the module loader will use the normal open(), close(), read(), and lseek() functions for file access.

funcopt
The loader does not touch the contents of this member. It can be used for additional arguments to functions that are registered in functable.

Calling conditions

Can be called from a thread

Multithread safe (must be called in interrupt-enabled state)

Description

This function is a combination of LoadModule() and LoadModuleAddress(). The behavior when the module is loaded can be specified by the *option* argument. These behaviors consist of:

- Policy for placing the module in memory.
- File access policy when the module is read.
- Replacement of the file access function when the module is read.

For information about the policy for placing a module in memory and the file access policy when a module is read, see the description of the arguments.

The file access function when the module is read can be replaced by passing the following function table in the option->functable argument.

If a function table has not been provided, NULL should be specified for the option->functable argument.

```
typedef struct _ldfilefunc {
     int (*beforeOpen)(void *opt, const char *filename, int flag);
     int (*afterOpen)(void *opt, int fd);
     int (*close)(void *opt, int fd);
     int (*setBufSize)(void *opt, int fd, size_t nbyte);
     int (*beforeRead)(void *opt, int fd, size_t nbyte);
     int (*read)(void *opt, int fd, void *buf, size_t nbyte);
     int (*lseek)(void *opt, int fd, long offset, int whence);
     int (*getfsize)(void *opt, int fd);
} LDfilefunc;
```

The specifications of functions registered in the function table shown above are as follows.

Note that when a function that is registered in the function table is called, the gp register value will be used by the file loader, therefore with respect to accessing global variables/data, restrictions exist that are similar to those for entry functions of resident libraries.

LDfilefunc.beforeOpen()

This function is used to notify the application immediately before the file loader opens a file. KE_OK should be returned for the return value of this function.

LDfilefunc.afterOpen()

This function is used to notify the application immediately after the file loader opened a file. The return value of the open() function is passed to the fd argument. KE_OK should be returned for the return value of this function.

LDfilefunc.close()

This function is called when the file loader closes a file. The close() function should be called within this function and the return value of the close() function should be set as the return value of this function.

```
int myclose(void *opt, int fd)
{
    /* Application-dependent processing */
    return close(fd);
}
```

LDfilefunc.setBufSize()

This function informs the application of the desired size of buffer to be prepared before the file loader randomly accesses a file. KE_OK or KE_NO_MEMORY should be returned for the return value of this function. When KE_NO_MEMORY is returned, loading is considered to have failed. This function is called only when option->access is LMWO_ACCESS_Seekmany.

LDfilefunc.beforeOpen()
This function is used to notify the application immediately before the file loader opens a file. KE_OK should be returned for the return value of this function.

LDfilefunc.beforeRead()
The file loader will divide the contiguous area within the file into small pieces and call read() multiple times. This function informs the application of the size of the contiguous area before the file loader begins this operation. KE_OK or KE_FILEERR should be returned for the return value of this function. When KE_FILEERR is returned, loading is considered to have failed. This function is called only when option->access is LMWO_ACCESS_Seekmany.

LDfilefunc.read()
This function is used by the file loader to read data from the file. Operation equivalent to that of the standard read() function is expected.

LDfilefunc.lseek()
This function is used by the file loader to perform a seek on the file. Operation equivalent to that of the standard lseek() function is expected.

LDfilefunc.getfsize()
This function is used by the file loader to check the file size. Normally, it is implemented as follows.

```
int mygetfsize(void *opt, int fd)
{
    int size;
    size = lseek( fd, 0, SEEK_END );
    if( size >= 0 ) lseek( fd, 0, SEEK_SET );
    return size;
}
```

Return value

Positive (>=0)	ID number of loaded module
KE_ILLEGAL_CONTEXT	Called from exception handler or interrupt handler
KE_ILLEGAL_OBJECT	Object file format is invalid
KE_LINKERR	Resident library required by loaded module does not exist
KE_NOFILE	Specified file not found
KE_FILEERR	Error occurred while reading file
KE_MEMINUSE	Specified address already being used
KE_NO_MEMORY	Insufficient memory
KE_ILLEGAL_OFFSET	Illegal offset argument value

LoadStartModule

Load and start program module from file

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	July 2, 2001

Syntax

#include <kernel.h>

int LoadStartModule(

const char **filename,*	The name of the file in which the program module is stored. This is also used as the character string that is passed to argv[0] of the program module.
int *args,*	Number of valid data in the character array specified by argp (including the terminating null characters of each character string).
const char **argp,*	Character array consisting of consecutively stored and null-terminated argument character strings that is passed to the program module.
int **result);*	Pointer to a variable that stores the value returned by the module initialization routine. RESIDENT_END (0): module is resident in memory (resident module) NO_RESIDENT_END (1): module is removed from memory. (non-resident module) REMOVABLE_RESIDENT_END (2): module is resident in memory (unloadable resident module)

Calling conditions

Can be called from a thread

Multithread safe (must be called in an interrupt-enabled state)

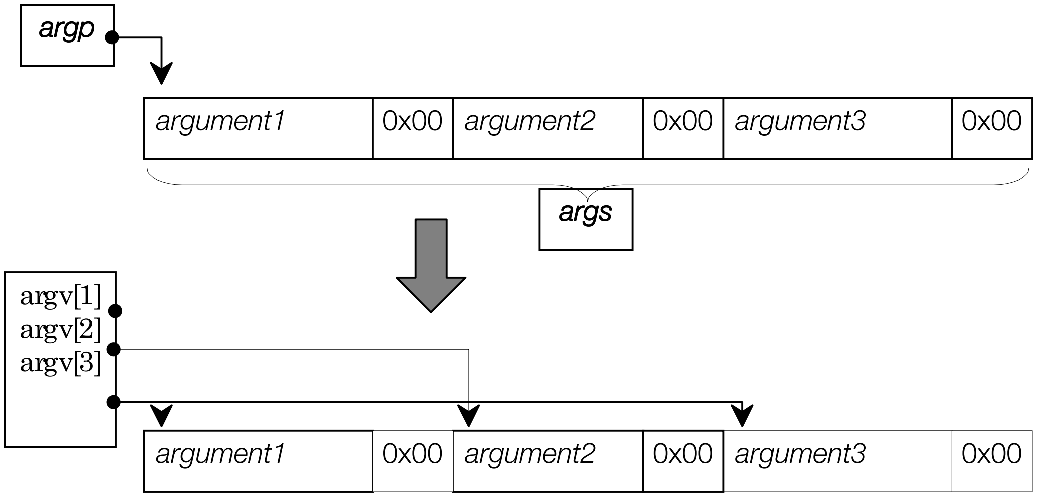
Description

Loads and starts the program module from a file.

Null terminates and consecutively stores argument character strings passed to the entry routine of the program module in the array pointed to by argument *argp*. After the system loader copies the concatenated character string indicated by the filename character string, *args* and *argp* to the stack, the number of character strings included within the range indicated by *args* and the starting pointer of each character string are determined, and are passed to the entry routine of the program module as argc, argv as shown below.

- argc = Number of argument character strings + 1
- argv[0] = Starting address of copy of *filename*
- argv[1] ... argv[argc-1] = Starting address of each character string which has delimited a copy of the contatenated character string indicated by *argp* and *args* with a null character.

Figure 2-1



Return Value

Positive number (>=0)	ID number of loaded module
KE_ILLEGAL_CONTEXT	Called from exception handler / interrupt handler
KE_ILLEGAL_OBJECT	Object file format is invalid
KE_LINKERR	Resident library required by loaded module does not exist
KE_NOFILE	Specified file cannot be found
KE_FILEERR	Error occurred when reading file
KE_NO_MEMORY	Insufficient memory

ReferModuleStatus

Get information about loaded program modules

Library	*Introduced*	*Documentation last modified*
ikrnl	2.3	July 2, 2001

Syntax

#include <kernel.h>

int ReferModuleStatus(

int *modid,*	ID number of module for which information will be obtained
ModuleStatus **status***);**	Specifies a pointer to a structure variable that will receive the module information. The following members are provided.

```
int  id;             /*Module identification ID number*/
char  name[56]; /*Copy of the first 55
                   characters of the module name.*/
u_short  version;  /*Module version*/
u_long   entry_addr;
u_long   gp_value;
u_long   text_addr;
u_long   text_size;
u_long   data_size;
u_long   bss_size;
```

Calling conditions

Can be called from a thread

Multithread safe

Description

Gets detailed information about the module in memory.

Return value

KE_OK	Normal
KE_UNKNOWN_MODULE	Did not find specified module.

RegisterLibraryEntries

Register resident library entry table

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	July 2, 2001

Syntax

#include <kernel.h>	
int RegisterLibraryEntries(	
libhead **lib);*	Pointer to entry table created by the Ioplibgen utility

Calling conditions

Can be called from a thread

Not multithread safe

Description

Registers the resident library entry table in the system.

The resident program module can register any number of entry tables.

Return value

KE_OK	Normal
KE_ILLEGAL_LIBRARY	Specified library header is illegal
KE_LIBRARY_FOUND	Library already registered

ReleaseLibraryEntries

Delete entry table registration

Library	*Introduced*	*Documentation last modified*
ikrnl	2.3	July 2, 2001

Syntax

#include <kernel.h>	
int ReleaseLibraryEntries(	
libhead **lib***);**	Pointer to entry table created by the Ioplibgen utility.

Calling conditions

Can be called from a thread

Description

Deletes the registration of a resident library entry table.

Deletion cannot be performed if there are modules using the resident library. The modules using the library must be deleted first.

Return value

KE_OK	Normal
KE_LIBRARY_NOTFOUND	Library not registered
KE_LIBRARY_INUSE	Library being used

SearchModuleByAddress

Find loaded modules by address

Library	*Introduced*	*Documentation last modified*
ikrnl	2.3	July 2, 2001

Syntax

#include <kernel.h>	
int SearchModuleByAddress(	
const void **addr***);**	Memory address which belongs to a module such as the address of a function within the module.

Calling conditions

Can be called from a thread

Multithread safe

Description

Searches for a loaded module that contains a specified address, then returns its module ID.

Notes

If another resident library entry is called from this module, a temporary jump will be made to the other module through an entry label in the jump table inside this module. Consequently, the module ID of this module is obtained when the entry of the resident library is made an argument of SearchModuleByAddress().

Return value

Positive (>=0)	Module ID of located module.
KE_UNKNOWN_MODULE	Specified module not found.

SearchModuleByName

Find loaded modules by name

Library	*Introduced*	*Documentation last modified*
ikrnl	2.3	July 2, 2001

Syntax

#include <kernel.h>

int SearchModuleByName(	
const char **_*modulename_);**	Module name

Calling conditions

Can be called from a thread

Multithread safe

Description

Searches for a loaded module with a specified module name, then returns its module ID.

If more than one module with the same name is loaded, the ID of the module that was loaded last will be returned.

Return value

Positive (>=0)	Module ID of located module.
KE_UNKNOWN_MODULE	Specified module not found.

SelfStopModule

Stop this program module

Library	*Introduced*	*Documentation last modified*
ikrnl	2.3	July 2, 2001

Syntax

#include <kernel.h>

int SelfStopModule(

int *args,*	Number of valid data of the character string array pointed to by argp (including the terminating null characters of each character string).
const char **argp,*	Specifies a character array consisting of consecutively stored and null-terminated argument character strings. See also the description of arguments in LoadStartModule().
int **result***);**	Pointer to a variable that stores the value returned by the end process routine of the module.

Calling conditions

Can be called from a thread

Multithread safe (must be called in interrupt-enabled state)

Description

This program module is stopped.

The program module must be in an unloadable resident state.

Return value

Positive (>=0)	ID number of this module.
KE_ILLEGAL_CONTEXT	Called from exception handler/interrupt handler.
KE_NOT_REMOVABLE	Cannot delete specified module.
KE_NOT_STARTED	Specified module did not start.
KE_ALREADY_STOPPED	Specified module already stopped.
KE_CAN_NOT_STOP	Could not stop module.

See also

LoadStartModule()

SelfUnloadModule

Unload this program module

Library	*Introduced*	*Documentation last modified*
ikrnl	2.3	July 2, 2001

Syntax

#include <kernel.h>

void SelfUnloadModule(void);

Calling conditions

Can be called from a thread

Multithread safe (must be called in interrupt-enabled state)

Description

Unloads this program module. The program module must be stopped by SelfStopModule(). Normally, the memory area where the program module is located is also freed although there may be cases where this is not the case (See the description of LoadModuleAddress() for details on whether or not memory is freed.)

There is no return from this service call. If an error occurs, the error will be displayed and SelfUnloadModule() will enter an infinite loop executing SleepThread().

Return value

None

See also

LoadModuleAddress(), LoadModuleBufferAddress()

SetRebootTimeLibraryHandlingMode

Set timing for resident library termination entry

Library	*Introduced*	*Documentation last modified*
ikrnl	2.4	October 11, 2001

Syntax

#include <kernel.h>

int SetRebootTimeLibraryHandlingMode(

libhead **lib*,	Pointer to entry table generated by Ioplibgen utility
int *mode***)**	Specify one of the following to indicate the termination entry call timing. RTLH_MODE_di Call termination entry after disabling interrupts. (default) RTLH_MODE_ei Call termination entry before disabling interrupts. RTLH_MODE_ei_di Call termination entry once before and once after disabling interrupts.

Calling conditions

Can be called from a thread

Description

This function sets the timing for calling the termination entry of a registered resident library during a Reboot.

Return value

KE_OK	Normal termination
KE_LIBRARY_NOTFOUND	The library is not registered

StartModule

Start up a program module that was previously loaded but has not yet been started

Library	*Introduced*	*Documentation last modified*
ikrnl	2.1	August 31, 2001

Syntax

#include <kernel.h>

int StartModule(

int *modid,*	Module ID
const char **filename,*	Name of the file where the program module is stored. Since this is only used to pass a string to argv[0] of the program module, the file is not accessed and a dummy filename can be used.
int *args,*	Number of valid data in character array specified by *argp* (including the terminating null characters of each character string).
const char **argp,*	Character array where argument strings are stored. Argument strings are stored as consecutive null-terminated strings. See also the description of *argp* in LoadStartModule().
int **result);*	Pointer to variable where return value from module initialization routine is stored.

Calling conditions

Can be called from a thread

Multithread safe (must be called in an interrupt-enabled state)

Description

Starts up a program module that was loaded with LoadModule().

Return value

Positive (>=0)	ID number of started module.
KE_ILLEGAL_CONTEXT	Called from exception handler/interrupt handler.
KE_UNKNOWN_MODULE	Specified module was not found
KE_ALREADY_STARTED	Specified module was already started

See also

LoadModule(), LoadModuleAddress(). LoadModuleBuffer(), LoadModuleBufferAddress

StopModule

Stop a program module that has been previously loaded and started

Library	*Introduced*	*Documentation last modified*
ikrnl	2.3	August 31, 2001

Syntax

#include <kernel.h>

int StopModule(

int *modid,*	Module ID.
int *args,*	Number of valid data of the character string array pointed to by *argp*. The size should include all null characters terminating each character string.
const char **argp,*	Character array consisting of the stored argument character strings. The argument character string consecutively stores multiple character strings that are null-terminated. See also the description of *argp* in LoadStartModule().
int **result*);	Pointer to a variable that stores the value that the end process routine of the module returns.

Calling conditions

Can be called from a thread

Multithread safe (must be called in interrupt-enabled state)

Description

Stops a program module started by LoadStartModule()/StartModule(). The program module must be in an deletable resident state.

Return value

Positive (>=0)	Module ID of the stopped module.
KE_ILLEGAL_CONTEXT	Called from exception handler/interrupt handler.
KE_UNKNOWN_MODULE	Could not find specified module.
KE_NOT_REMOVABLE	Cannot delete specified module.
KE_NOT_STARTED	Specified module is not started.
KE_ALREADY_STOPPED	Specified module already stopped.
KE_ALREADY_STOPPING	Specified module in stop processing.
KE_CAN_NOT_STOP	Could not stop module.

See also

LoadStartModule()

UnloadModule

Unload program module

Library	*Introduced*	*Documentation last modified*
ikrnl	2.3	July 2, 2001

Syntax

#include <kernel.h>	
int UnloadModule(	
int *modid***);**	Module ID

Calling conditions

Can be called from a thread

Multithread safe (must be called in interrupt-enabled state)

Description

Unloads a program module. The target module must not be running or already stopped.

Normally, the memory area where the program module is located is also freed although there may be cases where this is not the case (See the description of LoadModuleAddress() for details related to the freeing of memory.)

Return value

Positive (>=0)	Module ID of the unloaded module.
KE_ILLEGAL_CONTEXT	Called from exception handler/interrupt handler.
KE_UNKNOWN_MODULE	Could not find specified module.
KE_NOT_STOPPED	Specified module is not stopped.
KE_NOT_REMOVABLE	Cannot delete specified module.

See also

LoadModuleAddress()

Thread Management Functions

ChangeThreadPriority / iChangeThreadPriority

Change thread priority

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int ChangeThreadPriority(

int *thid,*	Thread ID of the thread for which the priority is to be changed. The calling thread can be specified by TH_SELF(=0).
int *priority);*	Specifies the new priority (after the change). A number from LOWEST_PRIORITY(=126) to HIGHEST_PRIORITY(=1) can be specified. A smaller number indicates a higher priority. Normal usage is in the range USER_LOWEST_PRIORITY(=123) to USER_HIGHEST_PRIORITY(=9). The current priority of the calling thread can be specified by specifying TPRI_RUN(=0).

int iChangeThreadPriority(

int *thid,*	Thread ID of the thread for which the priority is to be changed. The calling thread can be specified by TH_SELF(=0).
int *priority);*	Specifies the new priority (after the change). A number from LOWEST_PRIORITY(=126) to HIGHEST_PRIORITY(=1) can be specified. A smaller number indicates a higher priority. Normal usage is in the range USER_LOWEST_PRIORITY(=123) to USER_HIGHEST_PRIORITY(=9). The current priority of the calling thread can be specified by specifying TPRI_RUN(=0).

Calling conditions

ChangeThread Priority	Can be called from a thread Multithread safe
iChangeThreadPriority	Can be called from an interrupt handler

Description

Changes the priority of the thread specified by *thid* to *priority*.

The new priority to which the priority will be changed by this service call is effective until the thread is terminated, as long as it is not changed again. If the thread is in DORMANT state, the priority of the thread when it was terminated will be discarded, and the priority when the thread is restarted will be the startup priority (initPriority) that was specified when the thread was created.

If the specified thread had been enqueued in the ready queue or another queue, the queue order may change as a result of this service call.

If ChangeThreadPriority() is executed for a thread within the ready queue (including threads in RUN state) or for a thread within the priority queue, the specified thread will be moved to the end of the queue for that priority. Even if the thread priority doesn't change as a result of calling ChangeThreadPriority(), the thread will be moved to the end of the queue for that priority. Consequently, execution rights can be relinquished for the calling thread by issuing ChangeThreadPriority() with the same priority as the current priority.

Return value

KE_OK	Normal termination
KE_UNKNOWN_THID	Specified thread does not exist
KE_ILLEGAL_PRIORITY	Invalid priority specification
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_DORMANT	Specified thread was in DORMANT state

CheckThreadStack

Get remaining size of thread's stack

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int CheckThreadStack();

Calling conditions

Can be called from a thread

Multithread safe

Description

Calculates the amount of remaining free space on the local thread's stack.

This is a support function for determining the stack size required by the thread. If the result has clearly caused a stack overflow, a warning will be displayed and the system will then stop.

Return value

Remaining size of thread's stack

CreateThread

Create thread

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int CreateThread(

struct ThreadParam **param);*

Pointer to a structure that holds setup information for the thread to be created.

The structure has the following members.

```
int       attr;
void      *entry;
int       initPriority;
int       stackSize;
u_int     option;
```

The contents of each member are shown below.

- *attr*

  Specifies the thread description language as TH_ASM or TH_C. TH_COP1, TH_COP2, or TH_COP3 can also be specified to indicate that the corresponding coprocessor can be accessed from a new thread. TH_COP1, TH_COP2, and TH_COP3 can be combined with a logical OR.

- *entry*

  Specifies the entry address of the thread. The thread's entry point function can have one argument. The argument is assigned by StartThread(), which is described later.

- *initPriority*

  Specifies the thread's startup (StartThread()) priority. Any number from LOWEST_PRIORITY(=126) to HIGHEST_PRIORITY(=1) can be specified. A smaller number indicates a higher priority. Normal usage is in the range USER_LOWEST_PRIORITY(=123) to USER_HIGHEST_PRIORITY(=9).

- *stackSize*

  Specifies the thread's required stack size in bytes. Since a 150-byte stack is used to save registers when an external interrupt occurs, allow for this amount of margin when specifying the stack size. If the specified stack size is less than or equal to 300 bytes, an error will occur.

- *option*

  Specifies additional information related to the thread. This value can be obtained using ReferThreadStatus() and is independent of the multithread manager. It can be used for arguments passed to the starting thread. The difference between option and the argument arg of StartThread(), which is described later, is that option is maintained even if the thread is in DORMANT state. To pass information that cannot fit in a u_long, reserve a separate memory area and pass its address in option.

Calling conditions

Can be called from a thread

Multithread safe

Description

Creates a thread.

Allocates a thread management area for the thread to be created, specifies its initial settings, and reserves stack area.

Information about the thread to be created is specified in *param*, and the thread's ID is returned as the return value.

The created thread will be placed in DORMANT state.

Return value

Positive (>0)	Thread ID
KE_NO_MEMORY	Insufficient memory
KE_ILLEGAL_ATTR	Invalid attr specification
KE_ILLEGAL_STACK_SIZE	Invalid stack size specification
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_ILLEGAL_PRIORITY	Invalid priority specification
KE_ILLEGAL_ENTRY	Invalid entry address of thread

DeleteThread

Delete thread

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int DeleteThread(

int *thid);*	ID of the thread to be deleted.

Calling conditions

Can be called from a thread

Multithread safe

Description

Deletes the thread specified by *thid*.

When the specified thread is deleted, the stack area and thread management area are freed.

The specified thread must be in DORMANT state.

Since the executing thread cannot be in DORMANT state, it cannot be set as the thread to delete(a KE_NOT_DORMANT error will occur).

To delete the executing thread, use ExitDeleteThread(). (Currently ExitDeleteThread() is not implemented yet.)

Return value

KE_OK	Normal termination
KE_UNKNOWN_THID	Specified thread does not exist
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_NOT_DORMANT	Specified thread was not in DORMANT state

ExitThread

Exit calling thread

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int ExitThread();

Calling conditions

Can be called from a thread

Multithread safe

Description

Causes normal termination of the calling thread by placing it in DORMANT state.

ExitThread() is a service call that does not return to the caller.

Resources (such as memory or semaphores) that were acquired by the thread to be exited will not be automatically released.

If the exited thread is restarted by StartThread(), information contained in the thread management area, such as the thread priority, will be reset. Information at the time that the thread was exited is not inherited.

Return value

KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler

GetThreadId

Get thread ID of calling thread

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int GetThreadId();

Calling conditions

Can be called from a thread

Multithread safe

Description

Gets the ID of the calling thread.

Return value

Positive (>0)	Thread ID
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler

ReferThreadStatus / iReferThreadStatus

Get thread state

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int ReferThreadStatus(

int *thid,*

Thread ID for which the state is to be obtained.
The calling thread can be specified by TH_SELF(=0).

struct ThreadInfo **info);*

Specifies a pointer to a structure for receiving the thread state.

The structure has the following members.

u_int *attr;*
u_int *option;*
int *status;*
void **entry;*
void **stack;*
int *stackSize;*
int *initPriority;*
int *currentPriority;*
int *waitType;*
int *waitId;*
int *wakeupCount;*

The contents of each member are shown below.

int iReferThreadStatus(

int *thid,*

Thread ID for which the state is to be obtained.
The calling thread can be specified by TH_SELF(=0).

struct ThreadInfo **info);*

Specifies a pointer to a structure for receiving the thread state.

The structure has the following members.

u_int *attr;*
u_int *option;*
int *status;*
void **entry;*
void **stack;*
int *stackSize;*
int *initPriority;*
int *currentPriority;*
int *waitType;*
int *waitId;*
int *wakeupCount;*

The contents of each member are shown below.

- *attr*

  Thread attribute set by CreateThread()

- *option*

  Additional information set by CreateThread()

- *status*

  The thread state is represented by a combination of the following bits.

0x01	THS_RUN	RUN state
0x02	THS_READY	READY state
0x04	THS_WAIT	WAIT state
0x08	THS_SUSPEND	SUSPEND state
0x0c	THS_WAITSUSPEND	WAIT-SUSPEND state
0x10	THS_DORMANT	DORMANT state

- *entry*

  Entry address set by CreateThread()

- *stack*

  Starting address of stack area reserved by the kernel when CreateThread() was executed

- *stackSize*

  Stack size set by CreateThread()

- *initPriority*

  Thread startup (StartThread()) priority set by CreateThread().

- *currentPriority*

  Current priority

- *waittype*

  Indicates the type of WAIT state when the thread is in a WAIT state.

TSW_SLEEP	WAIT state due to SleepThread()
TSW_DELAY	WAIT state due to DelayThread()
TSW_SEMA	Semaphore WAIT state
TSW_EVENTFLAG	Event flag WAIT state
TSW_MBX	Message box WAIT state
TSW_VPL	Variable-length memory pool acquisition WAIT state
TSW_FPL	Fixed-length memory block acquisition WAIT state

- *waitId*

  ID of wait target of above waitType (such as event flag ID)

- *wakeupCount*

  Unprocessed WakeupThread() count

Calling conditions

ReferThreadStatus	Can be called from a thread Multithread safe
iReferThreadStatus	Can be called from an interrupt handler

Description

Obtains state of the specified thread. This service call is provided mainly for debugging, and is normally not used.

Return value

KE_OK	Normal termination
KE_UNKNOWN_THID	Specified thread does not exist
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler

ReleaseWaitThread / iReleaseWaitThread

Forcibly cancel WAIT state of another thread

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int ReleaseWaitThread(

int *thid);*	Thread ID of the thread for which WAIT state is to be forcibly canceled.

int iReleaseWaitThread(

int *thid);*	Thread ID of the thread for which WAIT state is to be forcibly canceled.

Calling conditions

ReleaseWaitThread	Can be called from a thread
	Multithread safe
iReleaseWaitThread	Can be called from an interrupt handler

Description

When the thread specified by *thid* is in WAIT state, this function forcibly cancels the WAIT state.

The thread for which WAIT state was canceled is returned from the service call that placed it in WAIT state (such as SleepThread(), WaitEventFlag(), or WaitSEma()), and error code KE_RELEASE_WAIT is returned.

ReleaseWaitThread() does not perform WAIT state cancellation request queuing. That is, if the specified thread is in WAIT state, that WAIT state will be canceled. However, if the specified thread is not in WAIT state, error code KE_NOT_WAIT will be returned to the caller.

ReleaseWaitThread() does not cancel SUSPEND state.

If ReleaseWaitThread() is issued for a thread in a dual wait state (WAIT-SUSPEND), the specified thread will be placed in SUSPEND state.

Return value

KE_OK	Normal termination
KE_UNKNOWN_THID	Specified thread does not exist
KE_NOT_WAIT	Specified thread was not in WAIT state
KE_ILLEGAL_THID	Specified thread was calling thread
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler

RotateThreadReadyQueue / iRotateThreadReadyQueue

Rotate thread ready queue

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int RotateThreadReadyQueue(

int *priority);*	The priority for which queue rotation is to be performed.

int iRotateThreadReadyQueue(

int *priority);*	The priority for which queue rotation is to be performed.

Calling conditions

RotateThreadReadyQueue	Can be called from a thread
	Multithread safe
iRotateThreadReadyQueue	Can be called from an interrupt handler

Description

Rotates the section of the ready queue corresponding to the specified *priority*.

The thread enqueued at the beginning of the section of the ready queue corresponding to the specified priority is moved to the end of the ready queue for that priority, and execution is switched to another thread of the same priority. An application program can implement round-robin scheduling by issuing this service call at fixed intervals.

When RotateReadyQueue() is issued from a thread context, the section of the ready queue at the same priority as that of the calling thread can be rotated by specifying TPRI_RUN(=0) as the priority.

If TPRI_RUN or the priority of the calling thread is specified as the priority, the calling thread will be rotated to the end of that section of the ready queue. In other words, RotateReadyQueue() can be issued to relinquish a thread's execution rights. The term "ready queue" in this description also includes threads in RUN state. If no thread exists in the ready queue at the specified priority, no processing will be performed and no error will occur.

iRotateReadyQueue(TPRI_RUN) can also be issued from a thread-independent context such as a timer handler. In this case, the section of the ready queue that contains threads that are executing, or the section of the ready queue that contains the highest priority threads within the ready queue, will be rotated. Normally, these two ready queue sections are the same. However, they may not be the same if thread dispatching is delayed. In this case, the section of the ready queue that contains the highest priority threads will be rotated.

Return value

KE_OK	Normal termination
KE_ILLEGAL_PRIORITY	Invalid priority specification
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler

StartThread

Start thread

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int StartThread(

int *thid,*	ID of the thread to be started.
u_long *arg);*	Arguments of the thread's entry function.

Calling conditions

Can be called from a thread

Multithread safe

Description

Starts execution of the thread specified by *thid* and places it in READY state.

Arguments can be passed to the thread using arg.The priority of the specified thread will be the value of initPriority specified when the thread was created.

No start request queuing is performed for this service call. That is, if the specified thread is not in DORMANT state, this service call is ignored, and a KE_NOT_DORMANT error is returned to the issuing thread.

Return value

KE_OK	Normal termination
KE_UNKNOWN_THID	Specified thread does not exist
KE_ILLEGAL_THID	TH_SELF cannot be specified
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_NOT_DORMANT	Specified thread was not in DORMANT state

StartThreadArgs

Start thread

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int StartThreadArgs(

int *thid,*	ID of the thread to be started.
int *args,*	Number of bytes in the argument block of the thread's entry function.
void **argp);*	Pointer to the argument block of the thread's entry function.

Calling conditions

Can be called from a thread

Multithread safe

Description

Starts execution of the thread specified by *thid* and places it in READY state.

The argument block specified by *args* and *argp* is copied onto the thread's stack, *args* is passed directly as the first argument of the thread's entry function, and the address of the argument block that was copied onto the stack is passed as the second argument of the entry function.

The priority of the specified thread will be the initPriority value that was specified when the thread was created. No start request queuing is performed for this service call. That is, if the specified thread is not in DORMANT state, this service call is ignored, and a KE_NOT_DORMANT error is returned to the issuing thread.

Return value

KE_OK	Normal termination
KE_UNKNOWN_THID	Specified thread does not exist
KE_ILLEGAL_THID	TH_SELF cannot be specified
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_NOT_DORMANT	Specified thread was not in DORMANT state

TerminateThread / iTerminateThread

Forcibly terminate another thread

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	July 2, 2001

Syntax

#include <kernel.h>	
int TerminateThread(	
int *thid);*	Thread ID of thread to be forcibly terminated
int iTerminateThread(	
int *thid);*	Thread ID of thread to be forcibly terminated

Calling conditions

TerminateThread	Can be called from a thread
	Multithread safe
iTerminateThread	Can be called from an interrupt handler

Description

Forcibly terminates the thread specified by *thid* and places it in DORMANT state.

If the specified thread was in WAIT state (including SUSPEND state), the wait will be canceled and the thread will be placed in DORMANT state. Also, if the thread had been enqueued in a queue (such as a semaphore wait), it will be deleted from that queue. The calling thread cannot be specified by *thid*. If it is, an error will occur.

Resources (such as memory or semaphores) that were acquired by the thread to be terminated will not be automatically released. If the terminated thread is restarted by StartThread(), information contained in the thread management area, such as the thread priority, will be reset. Information at the time that the thread was terminated is not inherited.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_UNKNOWN_THID	Specified thread does not exist
KE_ILLEGAL_THID	Specified thread was calling thread
KE_ DORMANT	Specified thread was in DORMANT state

Direct Thread Synchronization Functions

CancelWakeupThread / iCancelWakeupThread

Cancel thread wakeup request

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int CancelWakeupThread(

int *thid);*	ID of the thread for which the wakeup requests are to be canceled. The calling thread can be specified by TH_SELF(=0).

int iCancelWakeupThread(

int *thid);*	ID of the thread for which the wakeup requests are to be canceled. The calling thread can be specified by TH_SELF(=0).

Calling conditions

CancelWakeupThread	Can be called from a thread
	Multithread safe
iCancelWakeupThread	Can be called from an interrupt handler

Description

Reads the wakeup request count of the thread specified by *thid* and cancels all wakeup requests.

Return value

Positive (>=0)	Wakeup request count
KE_UNKNOWN_THID	Specified thread does not exist
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler

ResumeThread / iResumeThread

Restart thread that is in SUSPEND state Not implemented

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>	
int ResumeThread(	
int *thid);*	Thread ID of the thread for which SUSPEND state is to be canceled.
int iResumeThread(	
int *thid);*	Thread ID of the thread for which SUSPEND state is to be canceled.

Calling conditions

ResumeThread	Can be called from a thread
	Multithread safe
iResumeThread	Can be called from an interrupt handler

Description

Cancels SUSPEND state of the thread specified by *thid*.

Return value

KE_OK	Normal termination
KE_UNKNOWN_THID	Specified thread does not exist
KE_NOT_SUSPEND	Specified thread was not in SUSPEND state
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler

SleepThread

Switch calling thread to wakeup-wait state

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int SleepThread();

Calling conditions

Can be called from a thread

Multithread safe (must be called in an interrupt-enabled state)

Description

Places the calling thread into wakeup-wait state (WAIT state).

A thread that was placed in wakeup-wait state will return from WAIT state by WakeupThread() or ReleaseWaitThread().

If WakeupThread() was already issued by another thread when SleepThread() is issued, the wakeup request count will only be decremented, and control will return from SleepThread() without the thread being placed into WAIT state. For more information, see the description of WakeupThread().

Notes

SleepThread() should not be called from an interrupt-inhibited area.

Although this should be considered an error, the following actions are currently performed: a warning is printed, interrupt-inhibited state is temporarily canceled, a switch is made to another thread, and interrupt-inhibited area is restored when the calling thread is again returned to RUN state.

Return value

KE_OK	Normal termination
KE_RELEASE_WAIT	State canceled due to ReleaseWait
KE_CAN_NOT_WAIT	Attempted to enter thread in wait state during dispatch disabled state
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler

SuspendThread / iSuspendThread

Switch another thread to SUSPEND state Not implemented

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int SuspendThread(

int *thid);*	Thread ID of the thread to be switched to SUSPEND state. The calling thread cannot be specified.

int iSuspendThread(

int *thid);*	Thread ID of the thread to be switched to SUSPEND state. The calling thread cannot be specified.

Calling conditions

SuspendThread	Can be called from a thread
	Multithread safe
iSuspendThread	Can be called from an interrupt handler

Description

Places the thread specified by *thid* in SUSPEND state and suspends thread execution.

SUSPEND state is canceled by ResumeThread().

If the thread to be placed in SUSPEND state was already in WAIT state, it will enter a WAIT-SUSPEND state, which is a combination of WAIT state and SUSPEND state. If this thread's condition for cancelling the wait is subsequently satisfied, it will then be placed in SUSPEND state.However, if ResumeThread() is issued for a thread that is in a WAIT-SUSPEND state, it will be returned to the same WAIT state in which it had previously been.

SUSPEND state is a state in which execution has been suspended due to a service call that was issued by another thread. Therefore, the calling thread cannot be specified in this service call.

If SuspendThread() is issued multiple times for a given thread, an error will occur for the second and subsequent SuspendThread().

Return value

KE_OK	Normal termination
KE_UNKNOWN_THID	Specified thread does not exist
KE_ILLEGAL_THID	Specified thread was the calling thread
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler

WakeupThread / iWakeupThread

Wake up another thread

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>	
int WakeupThread(	
int *thid);*	ID of the thread to be awakened.
int iWakeupThread(	
int *thid);*	ID of the thread to be awakened.

Calling conditions

WakeupThread	Can be called from a thread
	Multithread safe (must be called in an interrupt-enabled state)
iWakeupThread	Can be called from an interrupt handler

Description

Cancels the wakeup-wait state of the thread specified by *thid*.

If the specified thread is not in WAIT state, that is, if SleepThread() has not been executed, the wakeup request count will be incremented.

Even if the specified thread issues SleepThread(), it will not be placed in WAIT state until it is issued the number of times equal to the wakeup request count.

Notes

Currently, when WakeupThread() is called from an interrupt-inhibited area, if the awakened thread has a higher priority than the calling thread, the following actions will be performed: interrupt-inhibited state is canceled, a switch is made to the awakened thread, and interrupt-inhibited state is restored when the calling thread is again returned to RUN state. These actions are expected to change somewhat so that thread switching will be delayed until interrupt-inhibited state is canceled. Since the behavior will change, for now this function should not be called from an interrupt-inhibited area.

Return value

KE_OK	Normal termination
KE_UNKNOWN_THID	Specified thread does not exist
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler

Exclusive Control Functions Using Semaphores

CreateSema

Generate semaphore

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int CreateSema(

struct SemaParam **param);*	Pointer to a structure that contains configuration information for the semaphore to be created.
	This structure has the following members.
	u_int *attr;*
	int *initCount;*
	int *maxCount;*
	u_int *option;*
	The contents of each member are described below.

- *attr*

  The semaphore's attribute. Either of the following can be specified.

SA_THFIFO	Enqueue waiting threads using FIFO.
SA_THPRI	Enqueue waiting threads according to the thread priority.

- *initCount*

  Semaphore initial value.

- *maxCount*

  Semaphore maximum value.

- *option*

  Additional information related to the semaphore. This value can be obtained using ReferSemaStatus().

  The multithread manager ignores this value.

Calling conditions

Can be called from a thread

Multithread safe

Description

Creates a semaphore. The semaphore ID is returned as the return value.

Return value

Positive (>0)	Semaphore ID
KE_NO_MEMORY	Insufficient memory
KE_ILLEGAL_ATTR	Invalid *attr* specification
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler

DeleteSema

Delete semaphore

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int DeleteSema(

int *semid);*	Semaphore ID of the semaphore to be deleted.

Calling conditions

Can be called from a thread

Multithread safe

Description

Deletes the semaphore indicated by *semid*.

An error (KE_WAIT_DELETE) is returned for a thread that had been entered in the semaphore queue.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_UNKNOWN_SEMID	Specified semaphore does not exist

ReferSemaStatus / iReferSemaStatus

Obtain semaphore state

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int ReferSemaStatus(

int *semid,*	Semaphore ID of semaphore for which state is to be obtained

struct SemaInfo **info);*	Pointer to a structure variable for receiving the semaphore state.
	This argument has the following members.
	u_int *attr;*
	u_int *option;*
	int *initCount;*
	int *currentCount;*
	int *maxCount;*
	int *numWaitThreads;*
	The contents of each member are described below.

int iReferSemaStatus(

int *semid,*	Semaphore ID of semaphore for which state is to be obtained

struct SemaInfo **info);*	Pointer to a structure variable for receiving the semaphore state.
	This argument has the following members.
	u_int *attr;*
	u_int *option;*
	int *initCount;*
	int *currentCount;*
	int *maxCount;*
	int *numWaitThreads;*
	The contents of each member are described below.

- *attr*

  Semaphore attribute that was set by CreateSema()

- *option*

  Additional information that was set by CreateSema()

- *initCount*

  Semaphore initial value that was set by CreateSema()

- *currentCount*

  Semaphore current value

- *maxCount*

  Semaphore maximum value that was set by CreateSema()

- *numWaitThreads*

  Number of threads waiting for the semaphore

Calling conditions

ReferSemaStatus	Can be called from a thread
	Multithread safe
iReferSemaStatus	Can be called from an interrupt handler

Description

Obtains the semaphore state.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_UNKNOWN_SEMID	Specified semaphore does not exist

SignalSema / iSignalSema

Return semaphore resource

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int SignalSema(

int *semid);*	Semaphore ID of semaphore for which resource is to be returned

int iSignalSema(

int *semid);*	Semaphore ID of semaphore for which resource is to be returned

Calling conditions

SignalSema	Can be called from a thread
	Multithread safe (must be called in an interrupt-enabled state)
iSignalSema	Can be called from an interrupt handler

Description

Performs operations for returning one resource to the semaphore indicated by *semid*.

Specifically, if there is a thread that is already waiting for the specified semaphore, the thread at the start of the queue is switched to READY state. In this case, the count value of that semaphore is unchanged. On the other hand, if no thread is waiting for the specified semaphore, the count value of that semaphore is incremented by 1. However, if the counter has already reached the maximum value, an error (KE_SEMA_OVF) will occur and the count value will not be changed.

Notes

Currently, when SignalSema() is called from an interrupt-inhibited area, if the thread for which WAIT state was canceled has a higher priority than the calling thread, the following actions will be performed: interrupt-inhibited state is canceled, a switch is made to that thread, and interrupt-inhibited state is restored when the calling thread is again returned to RUN state.

These actions are expected to change somewhat so that thread switching will be delayed until interrupt-inhibited state is canceled. Since the behavior will change, for now this function should not be called from an interrupt-inhibited area.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_UNKNOWN_SEMID	Specified semaphore does not exist
KE_SEMA_OVF	Semaphore counter reached maximum value and cannot be updated

WaitSema / PollSema

Acquire semaphore resource

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int WaitSema(

int *semid);*	Semaphore ID of semaphore for which resource is to be acquired

int PollSema(

int *semid);*	Semaphore ID of semaphore for which resource is to be acquired

Calling conditions

WaitSema	Can be called from a thread
	Multithread safe (must be called in an interrupt-enabled state)
PollSema	Can be called from a thread
	Multithread safe

Description

Performs operations for acquiring one resource from the semaphore indicated by *semid*.

Specifically, if the count value of the specified semaphore is greater than or equal to 1, the count value is decremented by 1. In this case, the thread that issued this service call does not enter WAIT state, and execution continues. On the other hand, if the count value of the specified semaphore is 0, the thread that issued this service call enters WAIT state, and it is enqueued in that semaphore queue.

The PollSema service call is equivalent to WaitSema except that the function for entering the WAIT state has been removed. It differs from WaitSema in that when the count value of the specified semaphore is 0, it returns the error KE_SEMA_ZERO.

Notes

Do not call WaitSema() from an interrupt-inhibited area. Although this should be considered an error, the following actions will be performed: a warning is printed, interrupt-inhibited state is temporarily canceled, a switch is made to another thread, and interrupt-inhibited state is restored when the calling thread is again returned to RUN state.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_UNKNOWN_SEMID	Specified semaphore does not exist
KE_SEMA_ZERO	Semaphore resource cannot be acquired
KE_RELEASE_WAIT	WAIT state was forcibly canceled
KE_CAN_NOT_WAIT	Attempted to enter WAIT state from dispatch-disabled state
KE_WAIT_DELETE	WAIT-target object was deleted

Synchronization Functions Using an Event Flag

ClearEventFlag / iClearEventFlag

Clear event flag

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int ClearEventFlag(

int *evfid,*	ID of the event flag to be cleared.
u_long *bitpattern);*	Clears bits in the event flag for which the corresponding bits in bitpattern are zero. In other words, the logical AND of the event flag and bitpattern will be set as the new value of the event flag.

int iClearEventFlag(

int *evfid,*	ID of the event flag to be cleared.
u_long *bitpattern);*	Clears bits in the event flag for which the corresponding bits in bitpattern are zero. In other words, the logical AND of the event flag and bitpattern will be set as the new value of the event flag.

Calling conditions

ClearEventFlag	Can be called from a thread Multithread safe
iClearEventFlag	Can be called from an interrupt handler

Description

Clears bits of the event flag indicated by *evfid*.

The WAIT state of an event-waiting thread will not be canceled due to this service call.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_UNKNOWN_EVFID	Specified event flag does not exist

CreateEventFlag

Create event flag

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int CreateEventFlag(

struct EventFlagParam **param);*	Specifies a pointer to a structure that holds setup information for the event flag to be created.
	The structure has the following members.
	int *attr;*
	int *initPattern;*
	u_int *option;*
	The contents of each member are described below.

- *attr*

  Specifies the event flag attribute. Specify either of the following values:

EA_SINGLE	Multiple thread waits are not permitted
EA_MULTI	Multiple thread waits are permitted

- *initPattern*

  Event flag initial value

- *option*

  Additional information related to the event flag. This value can be referenced by ReferEventFlagStatus() and is independent of the multithread manager.

Calling conditions

Can be called from a thread

Multithread safe

Description

Creates an event flag and sets its initial value.

The ID of the created event flag is returned as the return value.

Return value

Positive (>0)	Event flag ID
KE_NO_MEMORY	Insufficient memory
KE_ILLEGAL_ATTR	Invalid *attr* specification
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler

DeleteEventFlag

Delete event flag

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int DeleteEventFlag(

int *evfid);*	ID of the event flag to be deleted.

Calling conditions

Can be called from a thread

Multithread safe

Description

Deletes the event flag specified by *evfid.*

An error (KE_WAIT_DELETE) will be returned for a thread that is waiting for a condition to be satisfied on the specified event flag.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_UNKNOWN_EVFID	Specified event flag does not exist

ReferEventFlagStatus / iReferEventFlagStatus

Get event flag state

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int ReferEventFlagStatus(

int *evfid,*	ID of the event flag whose state is to be obtained

struct EventFlagInfo **info);*	Pointer to a structure variable for receiving the event flag state.
	This structure has the following members.
	u_int *attr;*
	u_int *option;*
	u_int *initPattern;*
	u_int *currentPattern;*
	int *numWaitThreads;*
	The contents of each member are described below.

int iReferEventFlagStatus(

int *evfid,*	ID of the event flag whose state is to be obtained

struct EventFlagInfo **info);*	Pointer to a structure variable for receiving the event flag state.
	This structure has the following members.
	u_int *attr;*
	u_int *option;*
	u_int *initPattern;*
	u_int *currentPattern;*
	int *numWaitThreads;*
	The contents of each member are described below.

- *attr*

  Event flag attribute that was set by CreateEventFlag()

- *option*

  Additional information that was set by CreateEventFlag()

- *initPattern*

  Initial value of event flag

- *currentPattern*

  Current value of event flag

- *numWaitThreads*

  Number of threads waiting for event flag

Calling conditions

ReferEventFlagStatus	Can be called from a thread
	Multithread safe
iReferEventFlagStatus	Can be called from an interrupt handler

Description

Obtains the state of the event flag.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_UNKNOWN_EVFID	Specified event flag does not exist

SetEventFlag / iSetEventFlag

Set event flag

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>	
int SetEventFlag(	
int *evfid,*	ID of the event flag to be set.
u_long *bitpattern);*	Sets bits indicating the new value of the event flag. That is, the logical OR of the event flag and bitpattern will be set as the new value of the event flag.
int iSetEventFlag(	
int *evfid,*	ID of the event flag to be set.
u_long *bitpattern);*	Sets bits indicating the new value of the event flag. That is, the logical OR of the event flag and bitpattern will be set as the new value of the event flag.

Calling conditions

SetEventFlag	Can be called from a thread Multithread safe (must be called in an interrupt-enabled state)
iSetEventFlag	Can be called from an interrupt handler

Description

Sets bits of the event flag indicated by *evfid*.

WAIT state will be canceled for a thread in WAIT state for which the wait condition was satisfied with the new value of the event flag.

Notes

Currently, when SetEventFlag() is called from an interrupt-inhibited area, if the thread for which WAIT state was canceled has a higher priority than the calling thread, the following actions are performed: interrupt-inhibited state is canceled, a switch is made to that thread, and interrupt-inhibited state is restored when the calling thread is again returned to RUN state.

These actions are expected to change somewhat so that thread switching will be delayed until interrupt-inhibited state is canceled. Since the behavior will change, for now this function should not be called from an interrupt-inhibited area.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_UNKNOWN_EVFID	Specified event flag does not exist

WaitEventFlag / PollEventFlag

Wait for event flag

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int WaitEventFlag(

int *evfid,*	ID of the target event flag.
u_long *bitpattern,*	Bit pattern to be compared with the event flag.
int *waitmode,*	Specifies the wait mode. Specify either of the following values. EW_AND AND wait EW_OR OR wait The following can also be logically ORed if desired. EW_CLEAR Clear after wait condition is satisfied
u_long **resultpat);*	Pointer to a variable that receives the event flag value when the wait is canceled

int PollEventFlag(

int *evfid,*	ID of the target event flag.
u_long *bitpattern,*	Bit pattern to be compared with the event flag.
int *waitmode,*	Specifies the wait mode. Specify either of the following values. EW_AND AND wait EW_OR OR wait The following can also be logically ORed if desired. EW_CLEAR Clear after wait condition is satisfied
u_long **resultpat);*	Pointer to a variable that receives the event flag value when the wait is canceled

Calling conditions

WaitEventFlag	Can be called from a thread
	Multithread safe (must be called in an interrupt-enabled state)
PollEventFlag	Can be called from a thread
	Multithread safe

Description

WaitEventFlag() is a service call that waits for the event flag indicated by *evfid* to be set according to the condition for cancelling the wait indicated by *waitmode*. If the event flag indicated by *evfid* already satisfies the condition for cancelling the wait indicated by *waitmode*, the issuing thread continues executing without entering WAIT state.

If EW_AND is specified for *waitmode*, the service call waits until all bits indicated by *bitpattern* become 1. If EW_OR is specified for *waitmode*, the service call waits until any of the bits indicated by *bitpattern* becomes 1. If EW_CLEAR was also specified for *waitmode*, all bits of the event flag are cleared to 0 when the wait is canceled for this thread.

The value of the event flag immediately after the condition for cancelling the wait was satisfied (the value before the flag is cleared when EW_CLEAR is specified) is returned in *resultpat*.

PollEventFlag() is like WaitEventFlag() except that it returns control immediately to the caller and does not enter WAIT state. PollEventFlag() will return the error code KE_EVF_COND if the condition for cancelling the wait was not satisfied. If EW_CLEAR is specified for waitmode, it is ignored by PollEventFlag(). If a thread is waiting on an event flag that has the EA_SINGLE attribute set, another thread cannot execute WaitEventFlag() or PollEventFlag() for that event flag. In this case, control returns immediately to the thread that executed WaitEventFlag() or PollEventFlag() last, and an error is returned.

When the event flag has the EA_MULTI attribute set, a thread queue is created if more than one thread enters WAIT state. In this case, the WAIT state may be canceled for all of the threads with a single call to SetEventFlag().

The order of entries in the thread queue will be such that the thread that entered WAIT state first will be at the head of the queue, and subsequent threads will be placed behind it in the order that they entered WAIT state.

If the queue contains a thread with the EW_CLEAR attribute set, the event flag will be cleared when the condition for cancelling the WAIT state is met, and the WAIT state is canceled. Threads that are behind the thread which has the EW_CLEAR attribute set will see the event flag after it is cleared, so their WAIT states will not be canceled.

Notes

Do not call WaitEventFlag() from an interrupt-inhibited area. Although this should be considered an error, the following actions will be performed: a warning is printed, interrupt-inhibited state is temporarily canceled, a switch is made to another thread, and interrupt-inhibited state is restored when the calling thread is again returned to RUN state.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_UNKNOWN_EVFID	Specified event flag does not exist
KE_CAN_NOT_WAIT	Attempted to enter thread in wait state during dispatch disabled state
KE_WAIT_DELETE	WAIT-target object was deleted
KE_RELEASE_WAIT	WAIT state was forcibly canceled

Communication Functions Using a Message Box

CreateMbx

Generate message box

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int CreateMbx(

struct MbxParam **param);*

Pointer to a structure variable holding configuration information about the message box to be generated.

This structure has the following members.

```
int      attr;
u_int    option;
```

The contents of each member is as follows.

- *attr*

  Specify the message box attribute. Either of the following can be specified.

MBA_THFIFO	Enqueue waiting threads using FIFO.
MBA_THPRI	Enqueue waiting threads according to the thread priority.
MBA_MSFIFO	Enqueue messages using FIFO.
MBA_MSPRI	Enqueue messages according to message priority.

- *option*

  Additional information related to the message box. This value can be obtained using ReferMbxStatus().

  The multithread manager ignores this value.

Calling conditions

Can be called from a thread

Multithread safe

Description

Creates a message box. The ID of the message box that was created is returned as the return value.

Return value

Positive (>0)	Message box ID
KE_NO_MEMORY	Insufficient memory
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_ILLEGAL_ATTR	Invalid *attr* specification

DeleteMbx

Delete message box

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int DeleteMbx(

int *mbxid);*	Message box ID of the message box to be deleted.

Calling conditions

Can be called from a thread

Multithread safe

Description

Deletes the message box indicated by *mbxid*.

If there was a thread waiting for a message in the specified message box, this service call will terminate normally, and an error (KE_WAIT_DELETE) will be returned for a thread that was in WAIT state.

Also, even if a message is remaining in the specified message box, no error will occur, the message box will be deleted, and the message that was in the message box will be left as is.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_UNKNOWN_MBXID	Specified message box does not exist

ReceiveMbx / PollMbx

Receive from message box

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>	
int ReceiveMbx(	
struct MsgPacket ***recvmsg,*	Pointer to a variable for receiving the starting address of the receive message packet.
int *mbxid);*	Receiving message box.
int PollMbx(	
struct MsgPacket ***recvmsg,*	Pointer to a variable for receiving the starting address of the receive message packet.
int *mbxid);*	Receiving message box.

Calling conditions

ReceiveMbx	Can be called from a thread
	Multithread safe (must be called in an interrupt-enabled state)
PollMbx	Can be called from a thread
	Multithread safe

Description

ReceiveMbx receives a message from the specified message box indicated by *mbxid*.

If a message had not yet been sent to the specified message box (the message box is empty), the thread that issued this service call enters WAIT state and is enqueued in the message arrival queue of the message box. On the other hand, if messages have already been entered in the specified message box, the first message is extracted, stored in the recvmsg return parameter, and returned.

The PollMbx service call is equivalent to ReceiveMbx except that the function for entering the queue has been removed. It differs from ReceiveMbx in that if a message had not yet been sent to the specified message box, it terminates with an error (KE_MBOX_NOMSG).

Note: Do not call ReceiveMbx() from an interrupt-inhibited area. Although this should be considered an error, the following actions will be performed: a warning is printed, interrupt-inhibited state is temporarily canceled, a switch is made to another thread, and interrupt-inhibited state is restored when the calling thread is again returned to RUN state.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_UNKNOWN_MBXID	Specified message box does not exist
KE_RELEASE_WAIT	WAIT state was forcibly canceled
KE_CAN_NOT_WAIT	Attempted to enter WAIT state from dispatch-disabled state
KE_WAIT_DELETE	WAIT-target object was deleted

ReferMbxStatus / iReferMbxStatus

Reference message box state

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int ReferMbxStatus(

int *mbxid,*	Message box ID of message box for which state is to be obtained

struct MbxInfo **info);*	Pointer to a structure variable for receiving the message box state.
	This structure has the following members.
	u_int *attr;*
	u_int *option;*
	int *numWaitThreads;*
	int *numMessage;*
	struct MsgPacket **topPacket;*
	The contents of each member are described below.

int iReferMbxStatus(

int *mbxid,*	Message box ID of message box for which state is to be obtained

struct MbxInfo **info);*	Pointer to a structure variable for receiving the message box state.
	This structure has the following members.
	u_int *attr;*
	u_int *option;*
	int *numWaitThreads;*
	int *numMessage;*
	struct MsgPacket **topPacket;*
	The contents of each member are described below.

- *attr*

  Message box attribute that was set by CreateMbx()

- *option*

  Additional information that was set by CreateMbx()

- *numWaitThreads*

  Number of threads waiting for messages

- *numMessage*

  Number of receive messages remaining in the message box

- *topPacket*

  Starting receive message

Calling conditions

ReferMbxStatus	Can be called from a thread
	Multithread safe
iReferMbxStatus	Can be called from an interrupt handler

Description

Obtains the message box state.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_UNKNOWN_MBXID	Specified message box does not exist

SendMbx / iSendMbx

Send to message box

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>	
int SendMbx(	
int *mbxid,*	Destination message box.
struct MsgPacket **sendmsg);*	Starting address of the message packet to be sent.
int iSendMbx(	
int *mbxid,*	Destination message box.
struct MsgPacket **sendmsg);*	Starting address of the message packet to be sent.

Calling conditions

SendMbx	Can be called from a thread
	Multithread safe (must be called in an interrupt-enabled state)
iSendMbx	Can be called from an interrupt handler

Description

Sends the message packet pointed to by *sendmsg* to the specified message box specified by *mbxid*. The message packet contents are not copied. Only the start address (sendmsg value) is passed on receipt.

If a thread is already waiting for a message in the specified message box, the WAIT state of the thread at the start of the queue will be canceled, the value of *sendmsg* specified in SendMbx will be sent to that thread, and this value will become the *recvmsg* return parameter of ReceiveMbx, which is described later.

On the other hand, if no thread is waiting for a message in the specified message box, the message that was sent is entered in the message queue within the message box. In either case, the thread that issued SendMbx will not be in WAIT state.

A message packet consists of a system-defined message header immediately followed by a message body in which the application program stores data.

The application program can set msgPriority in the message header as necessary. The application program need not manipulate any other part of the message header.

The multithread manager is not at all concerned with the message body. Decisions such as what size to make the message body or how that size is to be exchanged between threads (implicitly defining the size or placing information indicating the size in the body) are left up to the application program.

The management of memory in which message packets are stored is also left up to the application program. The sending thread of the application program stores and sends message packet data by allocating memory using the memory pool management function provided by the multithread manager or by allocating memory from an array variable that was declared within the program.

The receiving thread processes received message packet data, then returns memory using the memory pool management function or returns memory to the array variable.

Managing memory in a consistent fashion between the sending and receiving threads is the responsibility of the application program.

Notes

Currently, when SendMbx() is called from an interrupt-inhibited area, if the thread for which WAIT state was canceled has a higher priority than the calling thread, the following actions will be performed: interrupt-inhibited state is canceled, a switch is made to that thread, and interrupt-inhibited state is restored when the calling thread is again returned to RUN state.

These actions are expected to change somewhat so that thread switching will be delayed until interrupt-inhibited state is canceled. Since the behavior will change, for now this function should not be called from an interrupt-inhibited area.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_UNKNOWN_MBXID	Specified message box does not exist

Interrupt Management Functions

CpuDisableIntr

Disable interrupts and dispatching

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int CpuDisableIntr();

Calling conditions

Can be called from a thread

Multithread safe

Description

Disables all interrupts and thread dispatching.

The thread that issued CpuDisableIntr() will no longer be preempted by another thread until interrupts and dispatching are subsequently enabled by issuing CpuEnableIntr() or CpuResumeIntr().

The disabling of interrupts disables CPU interrupts (or hard interrupts corresponding to them). Disabling is performed independently of the enabling or disabling of interrupts for each interrupt cause (that is, the interrupt controller's interrupt mask register for individual causes is not changed).

If CpuDisablIntr() is issued again when interrupts and dispatching are already disabled, a KE_CPUDI error is returned.

Return value

KE_OK	Normal termination
KE_CPUDI	Interrupt was already disabled

CpuEnableIntr

Enable interrupts and dispatching

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int CpuEnableIntr();

Calling conditions

Can be called from a thread

Multithread safe

Description

Enables interrupts and dispatching.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler

CpuResumeIntr

Return interrupt and dispatching state

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int CpuResumeIntr(

int *oldstat);*	Passes the previous state that had been acquired by CpuSuspendIntr.

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

Returns the interrupt and dispatching state. This function is used together with CpuSuspendIntr.

Return value

KE_OK	Normal termination

CpuSuspendIntr

Save interrupts and dispatching state and disable interrupts and dispatching

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int CpuSuspendIntr(

int **oldstat);*	Pointer to a variable for returning the previous state.

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

Disables interrupts and thread dispatching in a similar manner as CpuDisableIntr. CpuSuspendIntr differs from CpuDisableIntr in that it saves the state in effect immediately before the disabling operation in the variable pointed to by oldstat.

If CpuSuspendIntr is issued again when interrupts and dispatching have already been disabled, a KE_CPUDI error is returned. However, even in this case, the appropriate value is set in **oldstat*.

Return value

KE_OK	Normal termination
KE_CPUDI	Interrupts were already disabled

Memory Pool Management Functions

AllocateFpl / pAllocateFPL / ipAllocateFpl

Allocate fixed length memory block

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

void * AllocateFpl(

int *fplid);*	Memory pool ID of the fixed length memory pool from which the memory block is to be allocated.

void * pAllocateFpl(

int *fplid);*	Memory pool ID of the fixed length memory pool from which the memory block is to be allocated.

void * ipAllocateFpl(

int *fplid);*	Memory pool ID of the fixed length memory pool from which the memory block is to be allocated.

Calling conditions

AllocateFpl	Can be called from a thread
	Multithread safe (must be called in an interrupt-enabled state)
pAllocateFpl	Can be called from a thread
	Multithread safe
ipAllocateFpl	Can be called from an interrupt handler

Description

Allocates one memory block from the fixed length memory pool indicated by *fplid.* The size of the allocated memory block will be the block size that was specified when the fixed length memory pool was created. The contents of the allocated memory block are undefined.

If the AllocateFpl service call cannot allocate the memory block from the specified memory pool, the thread that called AllocateFpl() enters a WAIT state (memory allocation wait state) and waits until the memory can be allocated.

The pAllocateFpl service call is equivalent to the AllocateFpl service call except that the function for entering WAIT state has been removed. It differs from AllocateFpl in that if the memory cannot be allocated, an error (KE_NO_MEMORY) is returned.

Notes

Do not call AllocateFpl() from an interrupt-inhibited area.

Although this should be considered an error, the following actions will be performed: a warning is printed, interrupt-inhibited state is temporarily canceled, a switch is made to another thread, and interrupt-inhibited state is restored when the calling thread is again returned to RUN state.

Return value

Positive (>0)	Address of allocated memory block
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_NO_MEMORY	Insufficient memory
KE_UNKNOWN_FPLID	Specified fixed length memory pool does not exist
KE_RELEASE_WAIT	WAIT state was forcibly canceled
KE_CAN_NOT_WAIT	Attempted to enter WAIT state from dispatch-disabled state
KE_WAIT_DELETE	WAIT-target object was deleted

AllocateVpl / iAllocateVpl / ipAllocateVpl

Allocate variable length memory block

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

void * AllocateVpl(

int *vplid,*	Memory pool ID of the variable length memory pool from which the memory block is to be allocated.
int *size);*	Memory block size in bytes.

void * pAllocateVpl(

int *vplid,*	Memory pool ID of the variable length memory pool from which the memory block is to be allocated.
int *size);*	Memory block size in bytes.

void * ipAllocateVpl(

int *vplid,*	Memory pool ID of the variable length memory pool from which the memory block is to be allocated.
int *size);*	Memory block size in bytes.

Calling conditions

AllocateVpl	Can be called from a thread Multithread safe (must be called in an interrupt-enabled state)
pAllocateVpl	Can be called from a thread Multithread safe
ipAllocateVpl	Can be called from an interrupt handler

Description

Allocates a memory block having a size of size bytes from the variable length memory pool indicated by *vplid*. The contents of the allocated memory block are undefined.

If the AllocateVpl service call cannot allocate the memory block from the specified memory pool, the thread that called AllocateVpl() enters a WAIT state (memory acquisition wait state) and waits until the memory can be allocated.

The pAllocateVpl service call is equivalent to the AllocateVpl service call except that the function for entering WAIT state has been removed. It differs from AllocateVpl in that if memory cannot be allocated, an error (KE_NO_MEMORY) is returned.

Notes

Do not call AllocateVpl() from an interrupt-inhibited area.

Although this should be considered an error, the following actions will be performed: a warning is printed, interrupt-inhibited state is temporarily canceled, a switch is made to another thread, and interrupt-inhibited state is restored when the calling thread is again returned to RUN state.

Return value

Positive (>0)	Address of allocated memory block
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_NO_MEMORY	Insufficient memory
KE_UNKNOWN_VPLID	Specified variable length memory pool does not exist
KE_RELEASE_WAIT	WAIT state was forcibly canceled
KE_CAN_NOT_WAIT	Attempted to enter WAIT state from dispatch-disabled state
KE_WAIT_DELETE	WAIT-target object was deleted

CreateFpl

Create a fixed length memory pool

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int CreateFpl(

struct FplParam **param);*

Pointer to a structure that has configuration information for the fixed length memory pool to be created.

This argument has the following members.

u_int	*attr;*
u_int	*option;*
int	*blockSize;*
int	*numBlocks;*

The contents of each member are as follows.

- *attr*

  Fixed length memory pool attribute. Either of the following can be specified.

FA_THFIFO	Enqueue waiting threads using FIFO.
FA_THPRI	Enqueue waiting threads according to the thread priority.

  Optionally, the following can also be specified by logically ORing:

  FA_MEMBTM

  Allocate the memory pool in the direction from the bottom of memory (high addresses). If not specified, the memory pool will be allocated in the direction from the top of memory (low addresses).

- *option*

  Additional information related to the fixed length memory pool. This value can be obtained using ReferFplStatus(). The multithread manager ignores this value.

- *blockSize*

  Memory block size that can be allocated from the fixed length memory pool.

- *numBlocks*

  Number of memory blocks that can be allocated from the fixed length memory pool.

Calling conditions

Can be called from a thread

Multithread safe

Description

Creates a memory pool from which fixed length memory blocks can be allocated.

A fixed length memory pool differs from a variable length memory pool in that it takes less time to allocate memory because only a fixed size memory block need be allocated at one time.

Return value

Positive (>0)	Fixed length memory pool ID
KE_NO_MEMORY	Insufficient memory
KE_ILLEGAL_ATTR	Invalid *attr* specification
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_ILLEGAL_MEMSIZE	Invalid memory size specification

CreateVpl

Create a variable length memory pool

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int CreateVpl(

struct VplParam **param);*

Pointer to a structure that has configuration information for the variable length memory pool to be created.

This structure has the following members.

```
u_int    attr;
u_int    option;
int      size;
```

The contents of each member is as follows.

- *attr*

  Variable length memory pool attribute. Either of the following can be specified.

VA_THFIFO	Enqueue waiting threads using FIFO.
VA_THPRI	Enqueue waiting threads according to the thread priority.

  Optionally, the following can also be specified by logically ORing:

  VA_MEMBTM Allocate the memory pool in the direction from the bottom of memory (high addresses). If not specified, the memory pool will be allocated in the direction from the top of memory (low addresses).

- *option*

  Additional information related to the variable length memory pool. This value can be obtained using ReferVplStatus(). The multithread manager ignores this value.

- *size*

  Size of the entire variable length memory pool in bytes.

Calling conditions

Can be called from a thread

Multithread safe

Description

Creates a memory pool from which a variable length memory block can be allocated.

Although a memory block of any size that does not exceed the memory pool size can be allocated from the variable length memory pool, processing will take more time than for the fixed length memory pool, which is described later.

Return value

Positive (>0)	Variable length memory pool ID
KE_NO_MEMORY	Insufficient memory
KE_ILLEGAL_ATTR	Invalid attr specification
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_ILLEGAL_MEMSIZE	Invalid memory size specification

DeleteFpl

Delete fixed length memory pool

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int DeleteFpl(

int *fplid);*	Memory pool ID of the fixed length memory pool to be deleted.

Calling conditions

Can be called from a thread

Multithread safe

Description

Deletes the fixed length memory pool indicated by *fplid*.

No error will occur if there exists a memory block that has not been freed among the memory blocks that were allocated from this memory pool. However, the operation of the system is not guaranteed if a memory block that was not freed is used after the memory pool has been deleted. Application programs must not use unfreed memory blocks after a memory pool is deleted.

An error is returned for a thread that had been registered in the queue waiting to allocate memory.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_UNKNOWN_FPLID	Specified fixed length memory pool does not exist

DeleteVpl

Delete variable length memory pool

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int DeleteVpl(

int *vplid);*	Memory pool ID of the variable length memory pool to be deleted.

Calling conditions

Can be called from a thread

Multithread safe

Description

Deletes the variable length memory pool indicated by *vplid*.

No error will occur if there exists a memory block that has not been freed among the memory blocks that were allocated from this memory pool. However, the operation of the system is not guaranteed if a memory block that was not freed is used after the memory pool has been deleted. Application programs must not use unfreed memory blocks after a memory pool is deleted.

An error (KE_WAIT_DELETE) is returned for a thread that had been registered in the queue waiting to allocate memory.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_UNKNOWN_VPLID	Specified variable length memory pool does not exist

FreeFpl

Free fixed length memory block

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int FreeFpl(

int *fplid,*	Memory pool ID of the fixed length memory pool to which the memory block is to be freed.
void **block);*	Address of the memory block to be freed.

Calling conditions

Can be called from a thread

Multithread safe (must be called in an interrupt-enabled state)

Description

Frees the memory block indicated by block to memory pool *fplid*.

If this operation makes a memory block available in the memory pool, another thread that had been in WAIT state waiting to allocate memory may proceed and its WAIT state will be canceled. If the memory block had not been allocated from the specified memory pool, a KE_ILLEGAL_MEMBLOCK error will occur.

Notes

Currently, when FreeFpl() is called from an interrupt-inhibited area, if the thread for which WAIT state was canceled has a higher priority than the calling thread, the following actions are performed: interrupt-inhibited state is canceled, a switch is made to that thread, and interrupt-inhibited state is restored when the calling thread is again returned to RUN state.

These actions are expected to change somewhat so that thread switching will be delayed until interrupt-inhibited state is canceled. Since the behavior will change, for now this function should not be called from an interrupt-inhibited area.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_ILLEGAL_MEMBLOCK	Memory block to be freed does not belong to memory pool
KE_UNKNOWN_FPLID	Specified fixed length memory pool does not exist

FreeVpl

Free variable length memory block

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int FreeVpl(

int *vplid,*	Memory pool ID of the variable length memory pool to which the memory block is to be freed.
void **block);*	Address of the memory block to be freed.

Calling conditions

Can be called from a thread

Multithread safe (must be called in an interrupt-enabled state)

Description

Frees the memory block indicated by *block* to memory pool *vplid*.

If this operation makes a memory block available in the memory pool, another thread that had been in WAIT state waiting to allocate memory may proceed and its WAIT state will be canceled. If the memory block had not been allocated from the specified memory pool, a KE_ILLEGAL_MEMBLOCK error will occur.

Notes

Currently, when FreeVpl() is called from an interrupt-inhibited area, if the thread for which WAIT state was canceled has a higher priority than the calling thread, the following actions are performed: interrupt-inhibited state is canceled, a switch is made to that thread, and interrupt-inhibited state is restored when the calling thread is again returned to RUN state.

These actions are expected to change somewhat so that thread switching will be delayed until interrupt-inhibited state is canceled. Since the behavior will change, for now this function should not be called from an interrupt-inhibited area.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_ILLEGAL_MEMBLOCK	Memory block to be freed does not belong to memory pool
KE_UNKNOWN_VPLID	Specified variable length memory pool does not exist

ReferFplStatus / iReferFplStatus

Obtain fixed length memory pool state

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int ReferFplStatus(

int *fplid,*	Memory pool ID of fixed length memory pool for which state is to be obtained

struct FplInfo **info);*	Pointer to a structure for receiving the memory pool state.
	This structure has the following members.
	u_int *attr;*
	u_int *option;*
	int *blockSize;*
	int *numBlocks;*
	int *freeBlocks;*
	int *numWaitThreads;*
	The contents of each member are described below.

int iReferFplStatus(

int *fplid,*	Memory pool ID of fixed length memory pool for which state is to be obtained

struct FplInfo **info);*	Pointer to a structure for receiving the memory pool state.
	This structure has the following members.
	u_int *attr;*
	u_int *option;*
	int *blockSize;*
	int *numBlocks;*
	int *freeBlocks;*
	int *numWaitThreads;*
	The contents of each member are described below.

- *attr*

  Fixed length memory pool attribute that was set by CreateFpl()

- *option*

  Additional information that was set by CreateFpl()

- *blockSize*

  Memory block size (in bytes) that was set by CreateFpl()

- *numBlocks*

  Number of memory blocks that was set by CreateFpl()

- *freeBlocks*

  Number of unused memory blocks within the memory pool

- *numWaitThreads*

  Number of threads waiting to allocate memory

Calling conditions

ReferFplStatus	Can be called from a thread
	Multithread safe
iReferFplStatus	Can be called from an interrupt handler

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_UNKNOWN_FPLID	Specified fixed length memory pool does not exist

ReferVplStatus / iReferVplStatus

Obtain variable length memory pool state

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int ReferVplStatus(

int *vplid,*	Memory pool ID of variable length memory pool for which state is to be obtained

struct VplInfo **info* **);**	Pointer to a structure variable for receiving the memory pool state.
	This structure has the following members.
	u_int *attr;*
	u_int *option;*
	int *size;*
	int *freeSize;*
	int *numWaitThreads;*
	The contents of each member are described below.

int iReferVplStatus(

int *vplid,*	Memory pool ID of variable length memory pool for which state is to be obtained

struct VplInfo **info* **);**	Pointer to a structure variable for receiving the memory pool state.
	This structure has the following members.
	u_int *attr;*
	u_int *option;*
	int *size;*
	int *freeSize;*
	int *numWaitThreads;*
	The contents of each member are described below.

- *attr*

  Variable length memory pool attribute that was set by CreateVpl()

- *option*

  Additional information that was set by CreateVpl()

- *size*

  Maximum number of bytes that can be allocated from the memory pool. This is the value obtained by subtracting the memory pool management area size from the memory pool size that was specified in CreateVpl.

- *freeSize*

  Number of unused bytes of memory in the memory pool

- *numWaitThreads*

  Number of threads waiting to allocate memory

Calling conditions

ReferVplStatus	Can be called from a thread
	Multithread safe
iReferVplStatus	Can be called from an interrupt handler

Description

Obtains the variable memory pool state.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_UNKNOWN_VPLID	Specified variable length memory pool does not exist

Time/Software Timer Management Functions

alarmhandler

Alarm handler prototype

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	January 4, 2002

Syntax

#include <kernel.h>

u_int alarmhandler(

void **common);*	The common argument that was specified in SetAlarm() is passed.

Description

You can specify that the alarm handler is to be called again according to its return value.

When the value returned by the alarm handler is 0, that alarm handler will be deleted.

When the value returned by the alarm handler is greater than or equal to 1, the scheduled time for the next call will be determined by adding to the scheduled time of the current call of the handler. However, if the value is less than 100 microseconds, it will be rounded up to approximately 100 microseconds.

As explained in the description of SetAlarm(), the handler is not necessarily called precisely at the specified time, but may be delayed. However, since the next handler calling time is calculated based on the scheduled calling time, not the time that the handler was actually called, the call delays are not accumulated.

Notes

An alarm handler is a type of interrupt handler. Therefore, the use of system service calls is restricted. See "Service Calls Issued from a Thread-independent Part" in the section entitled "System States Under the Control of the Multithread Manager."

The maximum clock speed expressed with the return value 32 bit unsigned integer is equivalent to 116.5084444 seconds.

Return value

0	The alarm handler will be deleted.
1 or more	The alarm handler will be called again after the number of clock ticks indicated by the return value.

CancelAlarm / iCancelAlarm

Cancel alarm handler

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>	
int CancelAlarm(	
u_int *(*handler)***(void*),**	Alarm handler entry point
void **common);*	Pointer to memory to be shared between alarm handler and general routines
int iCancelAlarm(	
u_int *(*handler)***(void*),**	Alarm handler entry point
void **common);*	Pointer to memory to be shared between alarm handler and general routines

Calling conditions

CancelAlarm	Can be called from a thread
	Multithread safe
iCancelAlarm	Can be called from an interrupt handler

Description

Cancels the alarm handler that was set by SetAlarm(), without waiting for the interval to elapse.

Return value

KE_OK	Normal termination
KE_NOTFOUND_HANDLER	Handler not registered

DelayThread

Delay thread

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int DelayThread(

unsigned int *usec);*	Specifies the time in microseconds to suspend the thread. (up to 4294.97 seconds)

Calling conditions

Can be called from a thread

Multithread safe (must be called in an interrupt-enabled state)

Description

Temporarily suspends execution of the calling thread and places it in an interval-expiration-wait state.

Since an interval-expiration-wait state is one type of WAIT state, it can be canceled by the ReleaseWaitThread() service call.

Although the suspend time can be specified in microseconds, if the value is less than 100 microseconds, it will be rounded up to 100 microseconds.

When several threads are within 200 microseconds of exiting a DelayThread() WAIT state, they may all collectively return at a time determined by the thread which has been waiting the longest.

Notes

Do not call DelayThread() from an interrupt-inhibited area.

Although this should be considered an error, the following actions will be performed: a warning is printed, interrupt-inhibited state is temporarily canceled, a switch is made to another thread, and interrupt-inhibited state is restored when the calling thread is again returned to RUN state.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_RELEASE_WAIT	State canceled due to ReleaseWait.
KE_CAN_NOT_WAIT	Attempted to enter WAIT state from dispatch-disabled state

GetSystemTime

Get system time

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

```
#include  <kernel.h>
int GetSystemTime(
 struct SysClock *clock);
```

hi

Stores high-order 32 bits of clock tick count.

low

Stores low-order 32 bits of clock tick count.

Calling conditions

Can be called from a thread

Multithread safe

Description

Gets the elapsed time in terms of clock ticks since system operation started.

Return value

KE_OK	Normal termination

SetAlarm / iSetAlarm

Set alarm handler

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int SetAlarm(

struct SysClock **clock,*	hi High-order 32 bits of elapsed time until handler is started low Low-order 32 bits of elapsed time until handler is started
u_int *(*handler)***(void*),**	Alarm handler entry point
void **common);*	Pointer to memory to be shared by alarm handler and general routines

int iSetAlarm(

struct SysClock **clock,*	hi High-order 32 bits of elapsed time until handler is started low Low-order 32 bits of elapsed time until handler is started
u_int *(*handler)***(void*),**	Alarm handler entry point
void **common);*	Pointer to memory to be shared by alarm handler and general routines

Calling conditions

SetAlarm	Can be called from a thread Multithread safe
iSetAlarm	Can be called from an interrupt handler

Description

Sets the alarm handler that is to be called after the specified interval has elapsed. An alarm handler, which is similar to an interrupt handler, has one argument and is called as a thread-independent context.

Although the elapsed time interval is specified as number of system clock ticks, if a value less than 100 microseconds is specified, it will be rounded up to approximately 100 microseconds.

When several threads are within 200 microseconds of exiting a DelayThread() WAIT state or of having their alarm elapsed time intervals expire, they may all collectively return at a time determined by the thread which has been waiting the longest.

The return value of the alarm handler, called after the specified interval has elapsed, determines whether the alarm is to be canceled or whether alarm operation is to continue.The alarm can also be canceled without waiting for the specified interval to elapse by using the CancelAlarm() service call described later.

The multithread manager distinguishes between alarm handlers using the handler's address and its argument (common). Therefore, setting an alarm handler in which both the handler address and common argument are equal will be considered as the same alarm, and an error will occur.

Return value

KE_OK	Normal termination
KE_NO_MEMORY	Insufficient memory
KE_FOUND_HANDLER	Handler was already registered

SysClock2USec

Convert system clock value to actual time

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

void SysClock2USec(

struct SysClock **clock,*	hi High-order 32 bits of system clock value to be converted low Low-order 32 bits of system clock value to be converted
int **sec,*	Pointer to variable for storing second units of converted result
int **usec);*	Pointer to variable for storing microsecond units of converted result

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This is a utility function that converts the system clock value to microseconds.

Return value

None

USec2SysClock

Convert microseconds to system clock value

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>	
void USec2SysClock(	
unsigned int *usec,*	Specifies the value to be converted, in microseconds
struct SysClock **clock);*	hi Stores the high-order 32 bits of the converted result low Stores he low-order 32 bits of the converted result

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This is a utility function that converts microseconds to system clock ticks.

Notes

Since microseconds are represented as 32-bit unsigned integers, this value may be up to 4294.97 seconds.

Return value

None

Hardware Timer Management Functions

AllocHardTimer

Get hardware timer

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int AllocHardTimer(

int *source,*	Specifies either TC_SYSCLOCK, TC_PIXEL, or TC_HLINE to indicate the source to be counted.
int *size,*	Specifies either 32 or 16 to indicate the timer's counter size (number of bits).
int *prescale);*	Specifies either 1, 8, 16, or 256 to indicate the prescale to be used.

Calling conditions

Can be called from a thread

Multithread safe

Description

This function obtains a hardware timer.

Specify required functions for the timer in the arguments.

When TC_PIXEL or TC_HLINE are specified for the source argument, the specified source and the system clock can both be counted.

Return value

Positive (>0)	Timer ID
KE_NO_TIMER	Hardware time could not be obtained
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler

FreeHardTimer

Return hardware timer

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int FreeHardTimer(

int *timid);*	ID of timer to be returned

Calling conditions

Can be called from a thread

Multithread safe

Description

Returns the hardware timer that was obtained by AllocHardTimer().

Return value

KE_OK	Normal termination
KE_ILLEGAL_TIMERID	Invalid hardware timer ID
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler

GetTimerCounter / iGetTimerCounter

Read hardware timer counter register

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>	
u_long GetTimerCounter(	
int *timid);*	Timer ID
u_long iGetTimerCounter(	
int *timid);*	Timer ID

Calling conditions

GetTimerCounter	Can be called from a thread
	Multithread safe
iGetTimerCounter	Can be called from an interrupt handler

Description

Reads the current value of the hardware timer's counter register.

Return value

Current value of counter register

overflowhandler

Overflow handler prototype

Library	*Introduced*	*Documentation last modified*
ikrnl	2.1	March 26, 2001

Syntax

#include <kernel.h>

u_int overflowhandler(

void **common);*	Passes the common argument specified in SetOverflowHandler().

Description

When the hardware timer counter register overflows, the overflow handler is called.

When the value returned by the overflow handler is zero, the timer is set to "not in use" state after which the overflow handler can no longer be called. When the value returned by the overflow handler is non-zero, the handler will be called again the next time an overflow occurs.

Notes

The overflow handler is a type of interrupt handler.

Therefore, the use of system service calls is restricted. Refer to "Service Calls Issued from Thread-independent Sections" in the section entitled "System States Under the Control of the Multithread Manager."

Return value

0	Hardware timer is set to "not in use" state.
>=1	Counting continues.

SetOverflowHandler

Set overflow handler

Library	*Introduced*	*Documentation last modified*
ikrnl	2.1	March 26, 2001

Syntax

#include <kernel.h>	
int SetOverflowHandler(	
int *timid,*	Timer ID
u_int *(*handler)***(void*),**	Specify the overflow handler that is called when the count register overflows. If NULL is specified, the handler will be cancelled.
void **common);*	Pointer to memory common between time-up handler and general routines.

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function sets the overflow handler of the hardware timer counter register. The hardware timer counter register begins counting up from zero after SetupHardTimer() and StartHardTimer(), which are described later, are executed.

If the counter register overflows after counting is started, an interrupt occurs, the counter register is returned to zero, and counting continues. The overflow handler is called via this interrupt.

Return value

KE_OK	Normal termination
KE_ILLEGAL_TIMERID	Invalid hardware timer ID
KE_TIMER_BUSY	Hardware timer is in use

SetTimerHandler

Set time-up handler

Library	*Introduced*	*Documentation last modified*
ikrnl	2.1	March 26, 2001

Syntax

#include <kernel.h>

int SetTimerHandler(

int *timid,*	Timer ID
u_long *comparevalue,*	Count comparison value.
u_int *(*timeuphandler)***(void*),**	Specify the time-up handler that is called when count matches comparison value. If NULL is specified, the handler will be cancelled.
void **common);*	Pointer to memory common between time-up handler and general routines.

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function sets the comparison value and time-up handler of the hardware timer counter register.

The hardware timer counter register begins counting up from zero after SetupHardTimer() and StartHardTimer(), which are described later, are executed.

If the counter register matches the comparison value that was set here after counting is started, an interrupt occurs, the counter register is returned to zero, and counting continues. The time-up handler is called via this interrupt.

Return value

KE_OK	Normal termination
KE_TIMER_BUSY	Hardware timer is in use
KE_ILLEGAL_TIMERID	Invalid hardware timer ID

SetupHardTimer

Set hardware timer operating mode

Library	*Introduced*	*Documentation last modified*
ikrnl	2.1	March 26, 2001

Syntax

#include <kernel.h>

int SetupHardTimer(

int *timid,*	Timer ID
int *source,*	Specify TC_SYSCLOCK, TC_PIXEL, or TC_HLINE to indicate the source that is to be actually counted. The source that can be specified here is either the source that was specified when the timer was allocated by AllocHardTimer() or TC_SYSCLOCK.
int *mode,*	Specify any of the following. TM_NO_GATE TM_GATE_ON_Count TM_GATE_ON_ClearStart TM_GATE_ON_Clear_OFF_Start TM_GATE_ON_Start
int *prescale);*	Specify 1, 8, 16, or 256 to indicate the prescale to be used. This is valid only when TC_SYSCLOCK is specified for source.

Calling conditions

Can be called from a thread

Multithread safe

Description

This function sets up the hardware timer using the mode that was specified by the argument and enables the timer to be started.

The hardware timer counter register begins counting up from zero after StartHardTimer(), which is described later, is executed.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Function was called from exception handler or interrupt handler
KE_ILLEGAL_TIMERID	Hardware timer ID was invalid
KE_TIMER_BUSY	Hardware timer is in use
KE_ILLEGAL_SOURCE	source specification was invalid
KE_ILLEGAL_MODE	mode specification was invalid
KE_ILLEGAL_PRESCALE	prescale specification was invalid

StartHardTimer

Start hardware timer counting

Library	*Introduced*	*Documentation last modified*
ikrnl	2.1	March 26, 2001

Syntax

#include <kernel.h>

int StartHardTimer(

int *timid);*	Timer ID

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function starts hardware timer counting according to the settings that were specified by SetupHardTimer() and sets the timer to "in use" state.

At the same time, if handlers were set up by SetTimerHandler() and SetOverflowHandler(), this function also enables timer interrupts and allows the handlers to be called.

Return value

KE_OK	Normal termination
KE_ILLEGAL_TIMERID	Hardware timer ID was invalid
KE_TIMER_BUSY	Hardware timer is in use
KE_TIMER_NOT_SETUP	Could not be started up because SetupHardTimer() has not been called yet

StopHardTimer

Stop hardware timer counting

Library	*Introduced*	*Documentation last modified*
ikrnl	2.1	March 26, 2001

Syntax

```
#include  <kernel.h>
int StopHardTimer(
 int timid);                                   Timer ID
```

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This function stops hardware timer counting and sets the state to "not in use."

At the same time, it also prohibits timer interrupts so that the time-up handler and overflow handler cannot be called.

Return value

KE_OK	Normal termination
KE_ILLEGAL_TIMERID	Hardware timer ID was invalid
KE_TIMER_NOT_INUSE	Hardware timer was not in use

timeuphandler

Time-up handler prototype

Library	*Introduced*	*Documentation last modified*
ikrnl	2.1	March 26, 2001

Syntax

#include <kernel.h>

u_int timeuphandler(

void **common);*	Passes the common argument specified in SetTimerHandler().

Description

When the hardware timer counter register matches the comparison value that was set by SetTimerHandler(), the time-up handler is called.

When the value returned by the time-up handler is zero, the timer is set to "not in use" state after which the time-up handler can no longer be called.

When the value returned by the time-up handler is greater than or equal to one, a new count comparison value is set and the time-up handler is called again the next time the counter register matches the comparison value.

Note that the counter register will have already started a new count from zero when the time-up handler is called. Therefore, if the value of the counter register is more than the value returned by the time-up handler at the time it returns, the counter will overflow, return to zero, begin re-counting and will subsequently match the comparison value.

Notes

The time-up handler is a type of interrupt handler. Therefore, the use of system service calls is restricted. Refer to "Service Calls Issued from Thread-independent Sections" in the section entitled "System States Under the Control of the Multithread Manager."

Return value

0	Hardware timer is set to "not in use" state.
>=1	Specifies new comparison value and continues counting.

V-blank Management Functions

RegisterVblankHandler

Register Vblank handler

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int RegisterVblankHandler(

int *edge,*	Specifies the timing of the handler call. Either of the following values may be used: VB_START Call at start of V-blank interval VB_END Call at end of V-blank interval
int *priority,*	Specifies a value from 32 to 223 indicating the calling priority among handlers. A handler having a lower priority is called before one having a higher priority. If there are multiple handlers with the same priority, the handler that was registered last is called first. Priorities 0-31 and 224-255 are reserved and should not be used.
int *(*handler)***(void*),**	Vblank handler entry point
void **common);*	Pointer to memory to be shared by Vblank handler and general routines

Calling conditions

Can be called from a thread

Multithread safe

Description

Registers a Vblank hander that will be called when the V-blank interval starts and ends.

The Vblank handler, which is similar to an interrupt handler, has one argument and is called as a thread-independent part.

Up to four handlers each can be registered for when the V-blank interval starts and ends.

The handlers for implementing the WaitVblankStart(), WaitVblankEnd(), WaitVblank(), and WaitNonVblank() services have also been registered as Vblank handlers, and their priorities have been set to 128.

Notes

If a hardware V-blank interrupt occurs, the system's V-blank interrupt handler will handle it.

The V-blank interrupt handler is a handler for sequentially calling the multiple Vblank handlers that were registered by application programs.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_FOUND_HANDLER	Handler was already registered
KE_NO_MEMORY	Too many registered handlers

ReleaseVblankHandler

Delete Vblank handler

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int ReleaseVblankHandler(

int *edge,*	Specifies handler call timing. Use the same value as the one that was specified when the Vblank handler was registered by RegisterVblankHandler().
int *(*handler)(void*));*	Entry point of the Vblank handler to be deleted. Use the same value as the one that was specified when the Vblank handler was registered by RegisterVblankHandler().

Calling conditions

Can be called from a thread

Multithread safe

Description

Deletes a registered Vblank handler.

Return value

KE_OK	Normal termination
KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_NOTFOUND_HANDLER	Handler not registered

vblankhandler

Prototype Vblank handler

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>	
int vblankhandler(	
void **common);*	The common argument that was specified in RegisterVblankHandler() is passed.

Description

If the value returned by the Vblank handler is NEXT_DISABLE, registration is deleted for that Vblank handler, and the Vblank handler will not be called due to the next Vblank.

Notes

A Vblank handler is a type of interrupt handler.

Therefore, the use of system service calls is restricted. See "Service Calls Issued from a Thread-independent Part" in the section entitled "System States Under the Control of the Multithread Manager."

Return value

NEXT_ENABLE	Also catch next vblank. (=1)
NEXT_DISABLE	Delete this handler. (=0)

WaitNonVblank

Wait until non-V-blank interval occurs

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int WaitNonVblank();

Calling conditions

Can be called from a thread

Multithread safe

Description

The issuing thread enters WAIT state until a non-V-blank interval occurs.

If a non-V-blank interval is already active, control returns without the thread entering WAIT state.

Return value

KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_OK	Normal termination

WaitVblank

Wait until V-blank interval occurs

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>
int WaitVblank();

Calling conditions

Can be called from a thread

Multithread safe

Description

The issuing thread enters WAIT state until the V-blank interval occurs.

If the V-blank interval is already active, control returns without the thread entering WAIT state.

Return value

KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_OK	Normal termination

WaitVblankEnd

Wait until next V-blank end

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int WaitVblankEnd();

Calling conditions

Can be called from a thread

Multithread safe

Description

Causes the calling thread to enter WAIT state until the next V-blank end.

Return value

KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_OK	Normal termination

WaitVblankStart

Wait until next V-blank start

Library	*Introduced*	*Documentation last modified*
ikrnl	1.1	March 26, 2001

Syntax

#include <kernel.h>

int WaitVblankStart();

Calling conditions

Can be called from a thread

Multithread safe

Description

Causes the calling thread to enter WAIT state until the next V-blank start.

Return value

KE_ILLEGAL_CONTEXT	Call was from exception handler or interrupt handler
KE_OK	Normal termination

Cache Operation Functions

FlushDcache

Clear Data Cache

Library	*Introduced*	*Documentation last modified*
ikrnl	1.6	March 26, 2001

Syntax

```
#include  <kernel.h>
void FlushDcache();
```

Calling conditions

Can be called from a thread

Multithread safe

Description

Clears the contents of the CPU data cache.

This function is used to eliminate mismatches between the CPU data cache and main memory when the main memory was overwritten without program intervention, such as by a DMA transfer.

Situations that necessitate the use of this function mainly occur immediately after data has been entered by a device driver. However, since this operation is often performed automatically within the device driver, this function should rarely need to be called from a normal application program. This function should be called when explicitly instructed to do so in the device driver or library documentation.

Return value

None

FlushIcache

Clear Instruction Cache

Library	*Introduced*	*Documentation last modified*
ikrnl	1.6	March 26, 2001

Syntax

#include <kernel.h>

void FlushIcache();

Calling conditions

Can be called from a thread

Multithread safe

Description

Clears the contents of the CPU instruction cache.

This function is provided to eliminate mismatches between the CPU instruction cache and main memory when a program in main memory was overwritten such as when loading a program. However, since this operation is usually performed automatically within the API for loading programs, this function need not be called from a normal application program.

Return value

None

Debugging Functions

Kprintf

Debugging printf

Library	*Introduced*	*Documentation last modified*
ikrnl	2.1	March 26, 2001

Syntax

#include <kernel.h>

void Kprintf(

const char **format,* **...);**	Specify the same format string as for printf.

Calling conditions

Can be called from an interrupt handler

Can be called from a thread

Multithread safe

Description

This is a printf function for debugging. The normal printf() can only be called from a thread, its output is buffered, and control may return immediately. However, Kprintf() can be called from an interrupt routine and control is guaranteed not to return until output has completed.

Notes

Although this function can be called from within an interrupt routine, since it requires time to execute, its use should be restricted only to the required minimum during debugging.

Return value

None