Copyright ©1995 by NeXT Computer, Inc. All Rights Reserved.

10

Kernel Support Functions

This chapter gives detailed descriptions of the C functions provided by the NeXT Mach kernel for loadable kernel servers. Also included are some macros that behave like functions. For this chapter, the functions and macros are divided into two groups: general functions and network functions.

Network functions are those that are specifically for network modules. All the other functions are under the ªGeneral Functionsº section.

Note: All functions described in this chapter work only in the kernel. The few that have user-level equivalents are noted.

Within each section, functions are subgrouped with other functions that perform related tasks. These subgroups are described in alphabetical order by the name of the first function listed in the subgroup. Functions within subgroups are also listed alphabetically, with a pointer to the subgroup's description.

For convenience, these functions are summarized in Appendix C, ªSummary of Kernel Support Functions.º The summary lists functions by the same subgroups used in this chapter and combines several related subgroups under a heading such as ªTime Functionsº or ªMemory Functions.º For each function, the appendix shows the calling sequence.

General Functions

ASSERT()

SUMMARY Panic if an assumption isn't true

SYNOPSIS **#import <kernserv/prototypes.h>**

void **ASSERT(**int *expression***)**

ARGUMENTS *expression*: A C expression that's 0 when the assumption isn't true.

DESCRIPTION **ASSERT()** is a macro that works only if you specify the DEBUG C preprocessor macro when you compile your server. If *expression* is 0, **ASSERT()** calls **panic()** after printing the line and file that the assertion failed in.

EXAMPLE In your makefile:

```
CFLAGS  = ... -DDEBUG
```

In your server:

```
ASSERT(ptr != NULL);
```

SEE ALSO **panic()**, **kern_serv_panic()**

assert_wait()

SUMMARY Arrange for a thread to sleep on an event

SYNOPSIS **#import <kernserv/prototypes.h>**

void **assert_wait(**int *event*, boolean_t *interruptible***)**

ARGUMENTS *event*: An integer that identifies the event. Typically, this is the address of a structure. If *event* is zero, then **thread_wakeup()** won't work on the thread; only **clear_wait()** and **thread_set_timeout()** will be able to wake it up.

interruptible: Used by **clear_wait()**. If *interruptible* is false and the *interrupt_only* argument to a later call to **clear_wait()** is true, then this thread won't be waked up by that call to **clear_wait()**.

DESCRIPTION Use this function before calling **thread_block()**. This function sets up the event that the thread wants to wait for, but the thread doesn't start sleeping until it executes **thread_block()**.

EXAMPLE

```
extern  hz;

assert_wait(0, FALSE);
thread_set_timeout(hz*2);
thread_block();
```

SEE ALSO **clear_wait()**, **thread_block()**, **thread_set_timeout()**, **thread_wakeup()**, **biowait()**

bcopy()

SUMMARY Copy data into a buffer

SYNOPSIS **#import <kernserv/prototypes.h>**

void **bcopy(**void **from*, void **to*, int *length***)**

ARGUMENTS *from*: Start of buffer to be copied from.

to: Start of buffer to be copied to.

length: Number of bytes to copy.

DESCRIPTION Like the C library **bcopy()** function, this function copies bytes from one buffer to another buffer in the same virtual space. This function can't be used to copy data between user space and kernel space. The caller of this function must have already checked the access rights to this memory and wired it down.

Important: Use **bytecopy()** instead of **bcopy()** if you're copying to or from hardware device space that's only 8 or 16 bits wide. (**bcopy()** often uses 32-bit accesses for efficiency, but the 68040 processor doesn't allow 32-bit accesses to 8-bit or 16-bit hardware.)

SEE ALSO **bytecopy()**, **strcpy()**, **copyin()**, **copyout()**

biodone()

SUMMARY Wake up the function doing a **biowait()** on a buffer

SYNOPSIS **#import <kernserv/prototypes.h>**

void **biodone(**struct buf ***bp***)**

ARGUMENTS *bp*: The address of a **buf** structure. This structure is defined in the header file **sys/buf.h**.

DESCRIPTION This function marks the buffer as done and wakes up any threads waiting for it. If the B_DONE flag is already set, **biodone()** panics. Otherwise, if B_CALL is set, **biodone()** clears it and calls the function pointed to by **bp***->b_iodone**. Next, if B_ASYNC is set, **biodone()** releases the buffer pointed to by *bp*; if B_ASYNC isn't set, **biodone()** clears the B_WANTED flag and wakes up all threads that had called **biowait()** on *bp*.

EXAMPLE

```
one_thread(void)
{
    struct buf  mybuf;
    . . .
    biowait (&mybuf);
    . . .
}

    other_thread(struct buf *bp)
    {
        . . .
        biodone(bp)
        . . .
    }
```

SEE ALSO **biowait()**

biowait()

SUMMARY Wait until a function calls **biodone()** on a buffer

SYNOPSIS **#import <kernserv/prototypes.h>**

void **biowait(**struct buf ***bp***)**

ARGUMENTS *bp*: The address of a **buf** structure. This structure is defined in the header file **sys/buf.h**.

DESCRIPTION If the B_DONE flag in the buffer pointed to by *bp* is already set, this function won't sleep. Otherwise, this function sleeps until another thread calls **biodone()** on *bp*.

EXAMPLE

```
one_thread(void)
{
    struct buf  mybuf;
    . . .
    biowait (&mybuf);
    . . .
}

other_thread(struct buf *bp)
{
```

```
    . . .
    biodone(bp)
   . . .
}
```

SEE ALSO **biodone()**, **assert_wait()**

bytecopy()

SUMMARY Copy bytes into a buffer

SYNOPSIS void **bytecopy(**void **from*, void **to*, int *length***)**

ARGUMENTS *from*: Start of buffer to be copied from.

to: Start of buffer to be copied to.

length: Number of bytes to copy.

DESCRIPTION This function is like **bcopy()**, except that it uses only 8-bit instructions to copy data. The **bytecopy()** function, like **bcopy()** and the C library **bcopy()** function, copies bytes from one buffer to another buffer in the same virtual space. The **bytecopy()** function can't be used to copy data between user space and kernel space. The caller of this function must have already checked the access rights to this memory and wired it down.

Note: This function is less efficient than **bcopy()**, so you should use **bcopy()** unless you're copying to or from hardware device space that's only 8 or 16 bits wide.

SEE ALSO **bcopy()**, **strcpy()**, **copyin()**, **copyout()**

bzero()

SUMMARY Zero out a region of memory

SYNOPSIS **#import <kernserv/prototypes.h>**

void **bzero(**void **address*, int *length***)**

ARGUMENTS *address*: The address of the first byte of the region of memory.

length: The number of bytes to write zeros to.

DESCRIPTION This acts the same as the **bzero()** C library function.

SEE ALSO **bzero(3)** UNIX manual page

clear_wait()

SUMMARY Stop a thread from waiting for an event

SYNOPSIS **#import <kernserv/prototypes.h>**
#import <kernserv/sched_prim.h>

void **clear_wait(**thread_t *thread*, int *result*, boolean_t *interrupt_only***)**

ARGUMENTS *thread*: The thread to wake up.

result: The wakeup result the thread should see.

interrupt_only: If true, don't wake up the thread unless **assert_wait()** was called with *interruptible* set to true.

DESCRIPTION Use this function to wake up a thread that's waiting for an event (as the result of **assert_wait()** and **thread_block()**), whether or not the event has happened. If *interrupt_only* is false or if **assert_wait()** was called with *interruptible* set to false, then the thread is guaranteed to wake up. The thread will receive *result* when it calls **thread_wait_result()**.

EXAMPLE
```
void        new_thread(void);
extern      hz;
char        data;
thread_t    thread1;
. . .
{
    . . .
    thread1 = (thread_t)current_thread();
    kernel_thread(current_task(), new_thread);
    assert_wait(&data, FALSE);
    thread_block();
    printf("Wait result:  %d\n", thread_wait_result());
}

            void new_thread()
            {
                . . .
                clear_wait(thread1, THREAD_AWAKENED, FALSE);
                . . .
            } /* new_thread */
```

SEE ALSO **assert_wait()**, **thread_block()**, **thread_wait_result()**, **thread_wakeup()**, **us_untimeout()**

clock_attributes()

SUMMARY Get information about a clock

SYNOPSIS **#import <kernserv/clock_timer.h>**

chrono_attributes_t **clock_attributes(**clock_types_t *which_clock***)**

ARGUMENTS *which_clock*: Either **Calendar** or **System**.

DESCRIPTION This function lets you get information about the system and calendar clocks, such as what their accuracy is and what their maximum value is.

EXAMPLE
```
char *s;
chrono_attributes_t    attr;

if (which_clock == System)
   s = SYSTEMSTRING;
else
   s = CALENDARSTRING;
attr = clock_attributes(which_clock);
printf("%s clock has a max value of %d:%d ns and an accuracy of %d ns\n",
   s,
   *((int*) &attr->max_value), *((int *) &attr->max_value + 1),
   *((int *) &attr->resolution + 1));
```

Note that the kernel version of **printf()** does not interpret **unsigned** integers or **long long** integers. This means that printing an **ns_time_t** value results in numbers that are difficult to interpret and sometimes negative, as shown in the following printout. If you need to print time values, you can perform one of the conversions shown in the example for **clock_value()**, later in this chapter.

System clock has a max value of -1:-1 ns and an accuracy of 1000 ns.

SEE ALSO **clock_value()**, **ns_time_to_timeval()**, **set_clock()**

clock_value()

SUMMARY Get the current time

SYNOPSIS **#import <kernserv/clock_timer.h>**

ns_time_t **clock_value(**clock_types_t *which_clock***)**

ARGUMENTS *which_clock*: Either **Calendar** or **System**.

DESCRIPTION This function returns the value of either the system clock (which starts over when the machine is booted) or the calendar clock (which continues to keep time between reboots), depending on the value of *which_clock*.

EXAMPLE
```
unsigned int    ms_time;
struct timeval  tv_time;
ns_time_t       now;

now = clock_value(System);
printf("Time since boot:  %d:%d ns == ",
    *((int*) &now), *((int *) &now + 1));

/* Since the value printed above is fairly useless, convert it */
ms_time = now / 1000000ULL;    /* convert to millisecs */
printf("%u ms == ", ms_time);
ns_time_to_timeval(now, &tv_time);
printf("%u seconds and %d microseconds.\n", tv_time.tv_sec,
    tv_time.tv_usec);
```

A typical printout:

```
Time since boot:  8942:-283072128 ns == 38409609 ms == 38409 seconds and
609456 microseconds.
```

SEE ALSO **clock_attributes()**, **set_clock()**

copyin()

SUMMARY Copy bytes from user to kernel space

SYNOPSIS **#import <kernserv/prototypes.h>**

int **copyin(**void ***from***, void ****to***, int *length***)**

ARGUMENTS *from*: The start of the region in user space.

to: The start of the region in kernel space.

length: The number of bytes to copy from user to kernel space.

DESCRIPTION This function works only in UNIX-style servers. It returns 0 if successful, -1 otherwise.

SEE ALSO **bcopy()**, **copyout()**

copyout()

SUMMARY Copy bytes from kernel to user space

SYNOPSIS **#import <kernserv/prototypes.h>**

int **copyout(**void **from*, void **to*, int *length***)**

ARGUMENTS *from*: The start of the region in kernel space.

to: The start of the region in user space.

length: The number of bytes to copy from kernel to user space.

DESCRIPTION The same as **copyin()**, except the direction of the copy is reversed.

SEE ALSO **bcopy()**, **copyin()**

curipl()

SUMMARY Get the current interrupt level

SYNOPSIS int **curipl(**void**)**

DESCRIPTION This function returns the CPU interrupt level, which is a number between 0 and 7.

EXAMPLE
```
#define panic(s) (curipl() == 0 ? \
kern_serv_panic((kern_serv_bootstrap_port(&instance), s) \
: printf("Can't panic: %s\n", s))
```

SEE ALSO **spl*n*()**, **splx()**

current_task()

SUMMARY Get the current task

SYNOPSIS **#import <kernserv/prototypes.h>**

task_t **current_task(**void**)**

DESCRIPTION This macro returns the task structure for the current task. Use **current_task()** whenever you need to refer to the task in which your loadable kernel server executes. Don't use **current_task()** to refer to memory unless you specifically want the task's native memory map, and not the kernel map that your server uses.

EXAMPLE
```
kernel_thread(current_task(), new_thread);
```

SEE ALSO **kernel_thread()**

DELAY()

SUMMARY Busy-wait for a certain number of microseconds

SYNOPSIS **#import <machine/machparam.h>**

void **DELAY(**unsigned int *usecs***)**

ARGUMENTS *usecs*: The number of microseconds to delay.

DESCRIPTION This macro makes the processor loop for the number of microseconds specified in the argument. Interrupts are not disabled by this function, so surround **DELAY()** with **spl*n*()** and **splx()** if interrupts need to be disabled. Because the microsecond resolution clock is used to count the spin interval, the delay is independent of CPU instruction clock speed.

This macro doesn't sleep, so it's safe to use in interrupt handlers. It's often used to wait for the hardware.

EXAMPLE
```
/* set the hardware register for at least 100 microseconds */
hardware_register = 1;
DELAY(100);
hardware_register = 0;
```

SEE ALSO **us_timeout()**, **us_abstimeout()**, **us_untimeout()**, **microtime()**, **microboot()**, **spl*n*()**, **splx()**

install_polled_intr()

SUMMARY Install an interrupt handler for a polled device

SYNOPSIS **#import <kernserv/prototypes.h>**

int **install_polled_intr(**int *which*, int (**my_intr*)(void)**)**

ARGUMENTS *which:* Specifies the device and interrupt level. For devices attached through the NeXTbus interface, this should be the constant I_BUS.

my_intr: The function in your server that handles this interrupt.

DESCRIPTION This function installs an interrupt handler; you can later remove this interrupt handler by calling **uninstall_polled_intr()**.

This function returns 0 if the call is successful, or -1 if the interrupt level specified by *which* isn't capable of interrupt polling.

EXAMPLE
```
device_interrupt(void) {
    if (interrupt_is_for_us) {
        /* -process interrupt- */
        return (1);  /* say interrupt was for us */
    }
    else
        return (0);  /* it must be for someone else */
}

        device_initialize(void) {
            install_polled_intr(I_BUS, device_interrupt);
            . . .
        }
```

SEE ALSO **uninstall_polled_intr()**

kalloc()

SUMMARY Allocate wired-down kernel memory

SYNOPSIS **#import <kernserv/prototypes.h>**

void ***kalloc(**int *size***)**

ARGUMENTS *size*: The size in bytes to be allocated.

DESCRIPTION This function is guaranteed to return wired-down memory of the requested size. The returned memory might not contain all zeros. You can't call **kalloc()** from an interrupt handler because it might sleep.

Memory returned isn't guaranteed to be aligned in any way unless size is a multiple of the page size (in which case the memory is page-aligned). If you need to ensure alignment, you should allocate twice what you need and align the address you start with to the boundary you want. Memory isn't guaranteed to be contained on the same physical page unless you allocate in multiples of the page size and keep track of the page location of addresses you use. The page size is dynamic; there's currently no way to get its value from inside the kernel. However, on 680x0-based machines, 8192 is guaranteed to be an integer multiple of the page size in bytes.

EXAMPLE
```
my_data_t *arg;

arg = (my_data_t *)kalloc(sizeof (my_data_t));
. . .
kfree(arg, sizeof (my_data_t));
```

SEE ALSO **kfree()**, **kget()**

kern_serv_bootstrap_port()

SUMMARY Get the port used to initialize your server

SYNOPSIS **#import <kernserv/kern_server_types.h>**

port_t **kern_serv_bootstrap_port(**kern_server_t **ksp***)*

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

DESCRIPTION This function returns the port that the kernel uses to initialize (or ªbootstrapº) your server when loading it. Normally, the only reason to use this port is as an argument to **kern_serv_panic()**.

EXAMPLE
```
bootstrap_port=kern_serv_bootstrap_port(&instance);
kern_serv_panic(bootstrap_port, "Couldn't send message");
```

SEE ALSO **kern_serv_panic()**, **kern_serv_local_port()**, **kern_serv_notify_port()**, **kern_serv_port_set()**

kern_serv_callout()

SUMMARY Run a function in the loadable kernel server's main thread

SYNOPSIS **#import <kernserv/kern_server_types.h>**

kern_return_t **kern_serv_callout(**kern_server_t *_ksp_, void (*_func_)(void *), void *_arg_**)**

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

func: The function to be called.

arg: The argument to be passed to *func*.

DESCRIPTION This function provides a way for interrupt handlers to call functions in the same loadable kernel server that may sleep or deal with a user context. The function *func* is called with argument *arg* at some point in the future.

EXAMPLE
```
void mydriver_func(mydriver_data_t data)
{
    . . .
}

kern_serv_callout ((kern_server_t *)&instance, mydriver_func, (void *)arg);
```

RETURN KERN_SUCCESS: The callout was scheduled successfully.

KERN_RESOURCE_SHORTAGE: The callout couldn't be scheduled.

kern_serv_kernel_task_port()

SUMMARY Get the kernel's task port

SYNOPSIS **#import <kernserv/kern_server_types.h>**

port_t **kern_serv_kernel_task_port(**void**)**

DESCRIPTION This function returns the kernel's task port. You need to specify this port when copying out-of-line data to or from a message, as shown in the following example.

EXAMPLE
```
log_data_t tmp, local_data;
local_data = (log_data_t)kalloc(8192);

printf("Calling vm_write\n");
r = vm_write((vm_task_t)kernel_task, (vm_address_t)local_data,
    (pointer_t)log_data, 8192);
if (r != KERN_SUCCESS)
    printf("Call to vm_write failed \n");
else {
    tmp = (log_data_t)kalloc(length+1);
    (void)strncpy(tmp, local_data, length);
    printf("Contents of data are: %s\n", tmp);
}
```

kern_serv_local_port()

SUMMARY Determine on which port the kernel just received a message

SYNOPSIS **#import <kernserv/kern_server_types.h>**

port_t **kern_serv_local_port(**kern_server_t **ksp***)*

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

DESCRIPTION This function returns the port on which the kernel just received a message in your server's behalf. The only time this function is useful is when your server was just loaded as the result of a message to one of its ports.

EXAMPLE
```
port=kern_serv_local_port(&instance);
if (port==debug_port)
    debug=TRUE;
```

SEE ALSO **kern_serv_notify_port()**, **kern_serv_port_set()**

kern_serv_log()

SUMMARY Put a message in the loadable kernel server's error log

SYNOPSIS **#import <kernserv/kern_server_types.h>**

void **kern_serv_log(**kern_server_t **ksp*, int *log_level*, char **format*, *arg1*, ..., *arg5***)**

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

log_level: A number indicating the urgency of this log entry. Higher numbers indicate greater urgency, but the particular range of numbers used in a loadable kernel server is up to the writer of that server.

format: A string containing formatting information. See **printf()**.

arg1, ..., *arg5*: Arguments to be printed. (If you don't specify all five arguments, the compiler will display a warning, but the call will still succeed.) See **printf()**.

DESCRIPTION This function puts a message in the error log. The message can be retrieved by a user process that calls **kern_loader_get_log()**, or by the command **kl_log**.

EXAMPLE
```
kern_serv_log(&instance, 5, "Reset value of timeout to %d\n", time, 0, 0, 0, 0);
```

SEE ALSO **log()**, **printf()**

kern_serv_notify()

SUMMARY Ask to receive notification messages about a certain port

SYNOPSIS **#import <kernserv/kern_server_types.h>**

kern_return_t **kern_serv_notify(**kern_server_t **ksp*, port_t *reply_port*, port_t *request_port***)**

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

reply_port: The port that should receive the notification messages. This should normally be the value returned by **kern_serv_notify_port()**.

request_port: The port you want to be notified about.

DESCRIPTION This function requests that notification messages about *request_port* be sent to *reply_port*. The types of notification messages are defined in the header file **mach/notify.h**.

EXAMPLE
```
notify_port=kern_serv_notify_port(&instance);
kern_serv_notify(&instance, notify_port, bootstrap_port);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_FAILURE: The same *reply_port-request_port* pair has already been entered.

SEE ALSO **kern_serv_notify_port()**

kern_serv_notify_port()

SUMMARY Get the notify port of this server

SYNOPSIS **#import <kernserv/kern_server_types.h>**

port_t **kern_serv_notify_port(**kern_server_t **ksp***)*

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

DESCRIPTION This function returns this server's notify port, which can be used in calls to **kern_serv_notify()**.

EXAMPLE
```
notify_port=kern_serv_notify_port(&instance);
kern_serv_notify(&instance, notify_port, bootstrap_port);
```

SEE ALSO **kern_serv_notify()**

kern_serv_panic()

SUMMARY Unload this server without panicking the system

SYNOPSIS **#import <kernserv/kern_server_reply.h>**

kern_return_t **kern_serv_panic(**port_t *bootstrap_port*, panic_msg_t *message***)**

ARGUMENTS *bootstrap_port*: This server's bootstrap port, which is returned by **kern_serv_bootstrap_port()**.

message: A string to be added to the panic message that's logged.

DESCRIPTION This function unloads the server after logging a message in the kernel-server loader's log. The message is logged at the priority LOG_WARNING and contains the name of the server that called this function, followed by *message*.

This function should not be called when the CPU interrupt level is greater than 0.

This function can return, so your server should avoid doing further work after calling it. After **kern_serv_panic()** is called, the kernel attempts to call the server's unload functions.

EXAMPLE
```
kern_serv_panic(bootstrap_port,
"my_server_main: received bad return from msg_receive");
```

RETURN KERN_SUCCESS: The server will be unloaded.

SEE ALSO **ASSERT()**, **panic()**, **kern_serv_bootstrap_port()**

kern_serv_port_gone()

SUMMARY Notify the kernel that a port will be deleted

SYNOPSIS **#import <kernserv/kern_server_types.h>**

void **kern_serv_port_gone(**kern_server_t **ksp*, port_name_t *port***)**

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

port: The port that will be deleted.

DESCRIPTION Use this function to make sure that the kernel won't send any more messages to a certain port.

EXAMPLE
```
/*
 * Deallocate transmit port.
 */
kern_serv_port_gone(&instance, my_dev->xmit_port);
(void)port_deallocate((task_t)task_self(), my_dev->xmit_port);
my_dev->xmit_port = PORT_NULL;
```

SEE ALSO **kern_serv_port_proc()**, **kern_serv_port_serv()**

kern_serv_port_proc()

SUMMARY Set which function is a port's handler

SYNOPSIS **#import <kernserv/kern_server_types.h>**

kern_return_t **kern_serv_port_proc(**kern_server_t **ksp*, port_all_t *port*, port_map_proc_t *function*, int *arg***)**

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

port: The port that the function should be associated with.

function: The function that handles messages sent to *port*.

arg: An integer to be passed in the call to *function* whenever *port* receives a message.

DESCRIPTION Use this function to register a message-receiving function in a handler-style (not server-style) loadable kernel server. This function provides the functionality of the HMAP load command to your server.

EXAMPLE
```
/* Create the port. */
r = port_allocate((task_t)task_self(), &port_name);
if (r != KERN_SUCCESS)
```

```
    kern_serv_panic(&instance, "couldn't allocate a port");
else printf("Created port %d\n", port_name);

/* Specify which function is its handler. */
r = kern_serv_port_proc(&instance, port_name,
    (port_map_proc_t)myhandler, 0);
if (r != KERN_SUCCESS) {
    kern_serv_panic("port_allocate failed (%d)\n", r);
    exit(1);
}

            /* . . . */

            kern_serv_port_gone(&instance, port_name);
            port_deallocate((task_t)task_self(), port_name);
            port_name = PORT_NULL;
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_RESOURCE_SHORTAGE: No more port-to-function mappings are available for your loadable kernel server.

KERN_NOT_RECEIVER: You don't have receive rights for *port*.

KERN_INVALID_ARGUMENT: *port* isn't a valid port.

SEE ALSO **kern_serv_port_gone()**, **kern_serv_port_serv()**

kern_serv_port_serv()

SUMMARY Set which function is a port's message server

SYNOPSIS **#import <kernserv/kern_server_types.h>**

kern_return_t **kern_serv_port_serv(**kern_server_t ******ksp*, port_all_t *port*, port_map_proc_t *function*, int *arg***)**

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

port: The port that the function should be associated with.

function: The function that handles messages sent to *port*.

arg: An integer to be passed in the call to *function* whenever *port* receives a message.

DESCRIPTION This function is just like **kern_serv_port_proc()** except that it registers a function with a server-style, as opposed to a handler-style, interface. This function performs the same function as the SMAP load command.

EXAMPLE
```
/* Create the port. */
r = port_allocate((task_t)task_self(), &port_name);
if (r != KERN_SUCCESS)
    kern_serv_panic(&instance, "couldn't allocate a port");
else printf("Created port %d\n", port_name);

/* Specify which function is its server. */
r = kern_serv_port_serv(&instance, port_name, (port_map_proc_t)myserv, 0);
if (r != KERN_SUCCESS) {
    kern_serv_panic("port_allocate failed (%d)\n", r);
    exit(1);
}
```

```
/* . . . */

kern_serv_port_gone(&instance, port_name);
port_deallocate((task_t)task_self(), port_name);
port_name = PORT_NULL;
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_RESOURCE_SHORTAGE: No more port-to-function mappings are available for your loadable kernel server.

KERN_NOT_RECEIVER: You don't have receive rights for *port*.

KERN_INVALID_ARGUMENT: *port* isn't a valid port.

SEE ALSO **kern_serv_port_gone()**, **kern_serv_port_proc()**

kern_serv_port_set()

SUMMARY Get the port set

SYNOPSIS **#import <kernserv/kern_server_types.h>**

port_set_name_t **kern_serv_port_set(**kern_server_t **ksp***)*

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

DESCRIPTION This function returns the name of the port set on which messages to the loadable kernel server arrive. The kernel listens to this port set on behalf of your server. Usually, this function is used after you've temporarily removed a port from the port set, and you need the name of the port set as an argument to **port_set_add()** so you can put the port back into the port set.

EXAMPLE

```
/* Don't accept any more requests until we get rid of the old ones. */
port_set_remove((task_t)task_self(), dev->xmit_port);

. . . /* Get rid of some old requests. */

/* Re-enable listening on the port. */
port_set_add((task_t)task_self, kern_serv_port_set(&instance),
    dev->xmit_port);
```

SEE ALSO **kern_serv_port_gone()**, **kern_serv_port_proc()**, **kern_serv_port_serv()**

kern_serv_unwire_range()

SUMMARY Unwire the specified range of memory in the kernel map

SYNOPSIS **#import <kernserv/kern_server_types.h>**

kern_return_t **kern_serv_unwire_range(**kern_server_t **ksp*, vm_address_t *address*, vm_size_t *size***)**

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

address: A virtual address in the kernel map.

size: The size in bytes to be wired down.

DESCRIPTION This function makes a region of kernel memory subject to swapping. Usually, you call it when you're preparing to deallocate the memory with **vm_deallocate()**.

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: The range of memory wasn't wired down.

SEE ALSO **kalloc()**, **kfree()**, **kget()**, **kern_serv_wire_range()**

kern_serv_wire_range()

SUMMARY Wire down the specified range of memory in the kernel map

SYNOPSIS **#import <kernserv/kern_server_types.h>**

kern_return_t **kern_serv_wire_range(**kern_server_t **ksp*, vm_address_t *address*, vm_size_t *size***)**

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

address: A virtual address in the kernel map.

size: The size in bytes to be wired down.

DESCRIPTION This function wires down a range of kernel memory. Usually you call it after you've copied out-of-line data into the kernel map.

RETURN KERN_SUCCESS: The call succeeded.

SEE ALSO **kalloc()**, **kget()**, **kern_serv_unwire_range()**

kernel_thread()

SUMMARY Start a new kernel thread in the specified task

SYNOPSIS **#import <kernserv/prototypes.h>**

thread_t **kernel_thread(**task_t *task*, void (**start*)(void)**)**

ARGUMENTS *task*: For loadable kernel servers, this must be **current_task()**.

start: The first function to be called by the new thread.

DESCRIPTION This function can sleep, so don't call it from an interrupt handler. The new thread uses the kernel address map, but the loadable kernel server's task.

EXAMPLE

```
void new_thread(void);

kernel_thread(current_task(), new_thread);

void new_thread(void)
{
    /* Do something, then (if necessary) shut down */
    thread_terminate(current_thread());
```

```
    thread_halt_self();
} /* new_thread */
```

SEE ALSO **current_task()**

kfree()

SUMMARY Free memory that was allocated using **kalloc()** or **kget()**

SYNOPSIS **#import <kernserv/prototypes.h>**

void **kfree(**void ***address**, int *size***)**

ARGUMENTS *address*: The memory to be freed. This must be exactly the same address as was returned by **kalloc()** or **kget()**.

size: The size in bytes to be freed. This must be the same size as was specified in the call to **kalloc()** or **kget()**.

DESCRIPTION The memory freed will be available for subsequent **kalloc()** and **kget()** calls only if the size they specify is the same as *size*.

EXAMPLE

```
my_data_t *arg;

arg = (my_data_t *)kalloc(sizeof (my_data_t));
. . .
kfree(arg, sizeof (my_data_t));
```

SEE ALSO **kalloc()**, **kget()**

kget()

SUMMARY Try to quickly allocate wired-down kernel memory

SYNOPSIS **#import <kernserv/prototypes.h>**

void ***kget(**int *size***)**

ARGUMENTS *size*: The size in bytes to be allocated. This size, rounded up to the nearest power of 2, must be less than the page size (default 8192 bytes), or the kernel will panic.

DESCRIPTION Use this function in interrupt handlers to try to get kernel memory. If no memory of the appropriate size can be allocated without blocking, **kget()** returns 0. Otherwise, it returns the address of the chunk of memory.

EXAMPLE

```
my_data_t *arg;

arg = (my_data_t *)kget(sizeof (my_data_t));
if (arg != 0)
{ . . .
    kfree(arg, sizeof (my_data_t));
}
```

SEE ALSO **kalloc()**, **kfree()**

lock_alloc(), lock_free()

SUMMARY Create or destroy a lock

SYNOPSIS **#import <kernserv/prototypes.h>**

lock_t **lock_alloc(**void**)**
void **lock_free(**lock_t *lock***)**

ARGUMENTS *lock*: The lock to be freed.

DESCRIPTION The function **lock_alloc()** returns a pointer to a new lock. Before you use the lock, you should initialize it by calling **lock_init()**.

The function **lock_free()** frees the lock structure pointed to by *lock*.

See **lock_done()** for information on using locks.

SEE ALSO **lock_done()**, **lock_init()**, **lock_read()**, **lock_write()**, **simple_lock_alloc()**, **simple_lock_free()**

lock_done()

SUMMARY Release a read or write lock

SYNOPSIS **#import <kernserv/prototypes.h>**

void **lock_done(**lock_t *lock***)**

ARGUMENTS *lock*: A pointer to the lock that the reader or writer wants to release.

DESCRIPTION The **lock_*xxx*()** functions provide reader/writer synchronization. Any number of readers can read, as long as no one has a lock for writing. A writer can get a lock only if no reader or writer locks exist. Once a writer tries to get a lock, no more readers can get the lock, and the writer gets the lock as soon as the last reader releases its lock. The writer sleeps or busy-waits until it can get a lock; you determine which it does when you initialize the lock.

Use the **lock_done()** function to relinquish a read or write lock.

EXAMPLE
```
lock_write(lock1);
/* write to the protected data */
lock_done(lock1);
```

SEE ALSO **lock_alloc()**, **lock_free()**, **lock_init()**, **lock_read()**, **lock_write()**, **simple_lock_unlock()**

lock_init()

SUMMARY Initialize a lock

SYNOPSIS **#import <kernserv/prototypes.h>**

void **lock_init(**lock_t *lock*, boolean_t *can_sleep***)**

ARGUMENTS *lock*: A pointer to the lock that the reader or writer wants to initialize.

can_sleep: If true, threads waiting to acquire a lock can sleep. If false, threads will busy-wait while trying to acquire a lock. This should usually be true.

DESCRIPTION Use this function to initialize a lock when you first create it. See **lock_done()** for a description of how locking works. Use **lock_alloc()** to create the lock.

EXAMPLE
```
lock_t lock1 = lock_alloc();
lock_init(lock1, TRUE);

/*. . .*/

lock_free(lock1);
```

SEE ALSO **lock_alloc()**, **lock_done()**, **lock_free()**, **lock_read()**, **lock_write()**, **simple_lock_init()**

lock_read()

SUMMARY Get a lock for reading

SYNOPSIS **#import <kernserv/prototypes.h>**

void **lock_read(**lock_t *lock***)**

ARGUMENTS *lock*: A pointer to the lock that the reader wants to get.

DESCRIPTION Use this function to get a lock for reading some data. If a writer holds or is waiting for a lock, you won't get the lock until the writer is done. Otherwise, you'll get the lock, even if other readers have it locked.

EXAMPLE
```
lock_t lock1 = lock_alloc();
lock_init(lock1, TRUE);
.
.
.
lock_read(lock1);
if (DONE_READING)
    lock_done(lock1);
```

SEE ALSO **lock_alloc()**, **lock_done()**, **lock_free()**, **lock_init()**, **lock_write()**, **simple_lock**

lock_write()

SUMMARY Get a lock for writing

SYNOPSIS **#import <kernserv/prototypes.h>**

void **lock_write(**lock_t *lock***)**

ARGUMENTS *lock*: A pointer to the lock that the writer wants to get.

DESCRIPTION Use this function to get a lock for writing some data. If another writer has or is waiting for a lock, you won't get the lock until the writer is done. If any readers have locks, you won't get the lock until every reader releases its lock.

EXAMPLE
```
lock_t lock1;

lock1 = lock_alloc();
lock_init(lock1, TRUE);
.
```

```
.
.
lock_write(lock1);
if (DONE_WRITING)
    lock_done(lock1);
```

SEE ALSO **lock_alloc()**, **lock_done()**, **lock_free()**, **lock_init()**, **lock_read()**, **simple_lock**

log()

SUMMARY Write a message in the system log buffer

SYNOPSIS **#import <sys/syslog.h>**
#import <kernserv/prototypes.h>

int **log(**int *level*, char **format*, *arg*, ...**)**

ARGUMENTS *level*: The priority of the information. These priorities are defined in the header file **sys/syslog.h**.

format: A string containing formatting information. See **printf()**.

arg, ...: Arguments to be printed. See **printf()**.

DESCRIPTION Prints the time of day and who sent the message (for loadable kernel servers, it's usually sent by Mach). This function doesn't sleep, so it can be called by interrupt functions. If no process is currently reading the system log, **log()** also writes to the console. This function always returns zero.

EXAMPLE `log(LOG_INFO, "My driver: device %s attached\n", device_type);`

SEE ALSO **kern_serv_log()**; **printf()**; UNIX manual pages for **syslog()** and **syslogd**

map_addr()

SUMMARY Convert a physical address to a virtual address

SYNOPSIS **#import <kernserv/prototypes.h>**

caddr_t **map_addr(**caddr_t *address*, int *size***)**

ARGUMENTS *address*: The physical address.

size: The number of bytes to map.

DESCRIPTION This function returns a virtual address that corresponds to *address*. At least *size* bytes of hardware addresses are mapped into virtual memory. (Currently, **map_addr()** maps in multiples of the page size, nominally 8192 bytes.)

If you aren't sure whether a hardware address is implemented, you should use **map_addr()** to get a virtual address for it, and then call **probe_rb()** on the virtual address.

EXAMPLE `volatile unsigned int *my_reg;`

```
my_reg = (unsigned int *)map_addr(REG_ADDRESS, 4);
if (probe_rb (my_reg))
    *my_reg |= A_FLAG;
```

```
else
    printf("Hardware at physical address 0x%x caused bus error\n",
        REG_ADDRESS);
```

SEE ALSO **probe_rb()**

ns_abstimeout(), ns_timeout()

SUMMARY Schedule the execution of a function at a specific time in the future

SYNOPSIS **#import <sys/callout.h>**
#import <kernserv/ns_timer.h>

void **ns_abstimeout(**func *function*, vm_address_t *arg*, ns_time_t *deadline*, int *priority***)**
void **ns_timeout(**func *function*, vm_address_t *arg*, ns_time_t *time*, int *priority***)**

ARGUMENTS *function*: The function to call.

arg: The argument to pass to *function*.

time: The number of nanoseconds from the time **ns_timeout()** is called to the time *function* should be called.

deadline: The time, in nanoseconds since system boot time, when *function* is to be called.

priority: The priority at which to execute *function*. This should almost always be the value CALLOUT_PRI_SOFTINT0 (defined in the header file **sys/callout.h**). Other values might not be supported by future releases.

DESCRIPTION The function **ns_timeout()** schedules the function to be executed at a time relative to the current time; **ns_abstimeout()** schedules at a time relative to when the system booted. Although these functions allow nanosecond resolution to be specified, the time is rounded up to the system clock tick interval (one microsecond on 68030-based and 68040-based NeXT computers). Your driver should not rely on *function* running at exactly the specified time, since it takes an unpredictable amount of time to interrupt the current task and start the execution of *function*.

The *priority* argument specifies how the function will be executed. The value CALLOUT_PRI_SOFTINT0 means that the function will be run from a software interrupt rather than at the interrupt level of the system clock. This prevents the function from delaying interrupts at or below the system clock level.

The function is executed only once per call to **ns_timeout()** or **ns_abstimeout()**. Use **ns_untimeout()** to unschedule the execution of the function before it has been run.

EXAMPLE
```
#define ONE_SECOND 1000000000ULL
/* . . . */
/* Schedule initial execution in one second. */
ns_timeout(every_second, (void *)0, ONE_SECOND, CALLOUT_PRI_SOFTINT0);
/* . . . */

void every_second (void *arg)
{
    /* Do something. */

    /* Reschedule execution for one second from now. */
    ns_timeout(every_second, (void *)0, ONE_SECOND, CALLOUT_PRI_SOFTINT0);
}

/* mydriver_signoff:  Called when mydriver is unloaded. */
void mydriver_signoff(void)
{
```

```
    if (ns_untimeout(every_second, (void *)0))
        printf("Unscheduled every_second.\n");
    else
        printf("every_second wasn't found.\n");

    printf("My driver unloaded\n\n");
}
```

SEE ALSO **ns_untimeout()**, **DELAY()**, **clock_value()**, **timeval_to_ns_time()**

ns_time_to_timeval(), timeval_to_ns_time()

SUMMARY Convert between **timeval** and **ns_time_t** time formats

SYNOPSIS **#import <kernserv/ns_timer.h>**

void **ns_time_to_timeval(**ns_time_t *ns*, struct timeval **tv***)*
ns_time_t **timeval_to_ns_time(**struct timeval **tv***)*

ARGUMENTS *ns*: The time in nanoseconds to be converted.

tv: The equivalent time in **struct timeval** format.

DESCRIPTION The **timeval_to_ns_time()** function converts a value from **struct timeval** format (seconds and microseconds) to **ns_time_t** (nanoseconds).

The **ns_time_to_timeval()** function does the opposite conversionÐfrom nanoseconds to seconds and microseconds. This conversion might be useful when printing a value returned by **clock_value()**, as shown in the following example.

EXAMPLE

```
unsigned int    ms_time;
struct timeval  tv_time;
ns_time_t       now;

    now = clock_value(System);
    printf("Time since boot:  %d:%d ns == ",
        *((int*) &now), *((int *) &now + 1));

    /* Since the value printed above is fairly useless, convert it */
    ms_time = now / 1000000ULL;    /* convert to millisecs */
    printf("%u ms == ", ms_time);
    ns_time_to_timeval(now, &tv_time);
    printf("%u seconds and %d microseconds.\n", tv_time.tv_sec,
        tv_time.tv_usec);
```

ns_untimeout()

SUMMARY Unschedule a timeout

SYNOPSIS **#import <kernserv/ns_timer.h>**

boolean_t **ns_untimeout(**func *function*, vm_address_t *arg***)**

ARGUMENTS *function*: The function that was to be called.

arg: The argument that was to be passed to *function*.

DESCRIPTION This function is used to unschedule a call to a function previously arranged by **ns_timeout()** or **ns_abstimeout()**. Only one instance of the *function-arg* pair is removed, so it may

be necessary to call **ns_untimeout()** multiple times. The function has no effect if the *function-arg* pair isn't found or if the function is already being executed. The **ns_untimeout()** function returns true if the timeout was found and unscheduled; otherwise it returns false.

EXAMPLE

```
#define ONE_SECOND 1000000000ULL
/* . . . */
/* Schedule execution in five seconds. */
ns_timeout(call_me, (void *)0, 5ULL*ONE_SECOND, CALLOUT_PRI_SOFTINT0);
/* . . . */

void call_me (void *arg)
{
    /* do something */
}

/* mydriver_signoff:  Called when mydriver is unloaded. */
void mydriver_signoff(void)
{
    if (ns_untimeout(call_me, (void *)0))
        printf("Unscheduled call_me.\n");
    else
        printf("call_me already executed.\n");

    printf("My driver unloaded\n\n");
}
```

SEE ALSO **ns_timeout()**, **ns_abstimeout()**

panic()

SUMMARY Hang the system and bring up the Panic window

SYNOPSIS **#import <kernserv/prototypes.h>**

void **panic(**char **string***string***)**

ARGUMENTS *string*: The message to be printed to the console, message log, and **/usr/adm/messages**.

DESCRIPTION Calling **panic()** brings up the Panic window (similar to the NMI mini-monitor window) and either hangs or reboots the system, depending on whether you booted with the **-p** option. See Chapter 9, ªBuilding, Loading, and Debugging Loadable Kernel Servers,º for information on the Panic window.

Instead of using **panic()**, you should use **kern_serv_panic()** when possible, since it doesn't cause the whole system to panic. However, **kern_serv_panic()** can't be called when the interrupt level is greater than 0.

EXAMPLE

```
if (curipl() == 0)
    kern_serv_panic(bootstrap_port, "Couldn't get resource");
else
    panic("mydriver:  Couldn't get resource");
```

SEE ALSO **ASSERT()**, **kern_serv_panic()**

printf()

SUMMARY Display a message on the console

SYNOPSIS **#import <kernserv/prototypes.h>**

int **printf(**char ******format*, *arg*, ...**)**

ARGUMENTS *format*: The format string. It's just like the C library **printf()** function's format string, except that the only conversions available are **%s**, **%c**, **%x**, **%d**, and **%o**. (**%X**, **%D**, **%u**, and **%O** are recognized but are treated like **%x**, **%d**, **%d**, and **%o**, respectively.)

arg, ...: Optional arguments, to be formatted according to the *format* string.

DESCRIPTION This function is a scaled-down version of the C library **printf()** function. Output goes not only to the console, but also to the message buffer and to **/usr/adm/messages**. Since **printf()** disables interrupts while printing messages, all system activities are suspended while it writes to the console.

Although **printf()** is safe to call in interrupt handlers, its output isn't guaranteed to print on the console. The message buffer, however, should be up-to-date. You can read the message buffer using the **msg** command in the NMI mini-monitor.

printf() always returns zero.

SEE ALSO **sprintf()**, **kern_serv_log()**, **log()**

probe_rb()

SUMMARY Check whether an address exists

SYNOPSIS **#import <kernserv/prototypes.h>**

int **probe_rb(**void ******address***)**

ARGUMENTS *address*: A virtual address that refers to a physical address.

DESCRIPTION This function returns 1 if *address* refers to a valid hardware address, 0 otherwise.

EXAMPLE

```
volatile unsigned int *my_reg;

my_reg = (unsigned int *)map_addr(REG_ADDRESS, 4);
if (probe_rb (my_reg))
    *my_reg |= A_FLAG;
else
    printf("Hardware at physical address 0x%x caused bus error\n",
        REG_ADDRESS);
```

SEE ALSO **map_addr()**

selthreadcache(), selthreadclear(), selwakeup()

SUMMARY Help for handling the **select()** system call

SYNOPSIS **#import <kernserv/prototypes.h>**

int **selthreadcache(**void ****waiterPtr***)**
void **selthreadclear(**void ****waiterPtr***)**
int **selwakeup(**void ******waiter*, int *collided***)**

ARGUMENTS *waiterPtr*: A pointer to a handle for the thread that's waiting for device activity. This handle should be initialized to 0 before calling **selthreadcache()** for the first time. After the waited-for device activity occurs and the thread handle is no longer needed, the handle should be cleared by a call to **selthreadclear()**.

waiter: A handle for a thread that's waiting for the device; this value is obtained by calling **selthreadcache()**.

collided: Should be 0 if only one thread is waiting for the device; otherwise, 1.

DESCRIPTION These functions let a UNIX-style driver handle the **select()** system call. Chapter 6 has more information and examples on how to use these functions.

The **selthreadcache()** function returns in *waiterPtr* a handle for the thread that's waiting for device activity. This function returns 0 if no other thread is waiting for the device activity; otherwise, it returns a nonzero value.

The **selthreadclear()** function clears the handle pointed to by *waiterPtr*.

If *collided* is 0, **selwakeup()** wakes up the thread represented by *waiter*. If *collided* is nonzero, **selwakeup()** wakes up all parties that are sleeping as the result of a **select()** system call. The **selwakeup()** function returns no meaningful value.

set_clock()

SUMMARY Sets the current time of the calendar clock

SYNOPSIS **#import <kernserv/clock_timer.h>**

void **set_clock(**clock_types_t *which_clock*, ns_time_t *ns***)**

ARGUMENTS *which_clock*: Must be **Calendar**.

ns: The time in nanoseconds (since midnight, January 1, 1970, Greenwich Mean Time) to set the clock to.

DESCRIPTION This function lets you set the current time of the calendar clock. Because the time is normally set by the Network Time Server or by the Preferences application, you don't usually need to call **set_clock()** directly.

EXAMPLE
```
unsigned int    ms_time;
ns_time_t       now;

now = clock_value(Calendar);
ms_time = now / 1000000ULL;    /* convert to millisecs */
printf("The current calendar clock time is %d ms.\n", ms_time);

set_clock(Calendar, now + timeToAdd);

now = clock_value(Calendar);
ms_time = now / 1000000ULL;    /* convert to millisecs */
printf("The new calendar clock time is %d ms.\n", ms_time);
```

SEE ALSO **clock_attributes()**, **clock_value()**, **timeval_to_ns_time()**, **settimeofday(2)** UNIX manual page

simple_lock()

SUMMARY Get a simple lock

SYNOPSIS **#import <kernserv/prototypes.h>**

void **simple_lock(**simple_lock_t *lock***)**

ARGUMENTS *lock*: A pointer to the simple lock.

DESCRIPTION Simple locks are simple spin-loops that implement exclusive locks. They're designed to be used when you plan to hold the lock for only a short time and/or when you can't sleep.

If someone else already has the lock, this function will busy-wait until it gets the lock.

Note: Simple locks are most useful on multiprocessor systems.

EXAMPLE

```
simple_lock_t   slock;

slock = simple_lock_alloc();
simple_lock_init(slock);
/* . . . */

        /* Set a lock before manipulating a data structure */
        simple_lock(slock);
        mydriver->data1 = VALUE;
        simple_unlock(slock);

        /* . . . */
        simple_lock_free(lock1);
```

SEE ALSO **simple_lock_alloc()**, **simple_lock_free()**, **simple_lock_init()**, **simple_unlock()**, **lock_read()**, **lock_write()**

simple_lock_alloc(), simple_lock_free()

SUMMARY Allocate or free a simple lock

SYNOPSIS **#import <kernserv/prototypes.h>**

simple_lock_t **simple_lock_alloc(**void**)**
void **simple_lock_free(**simple_lock_t *lock***)**

ARGUMENTS *lock*: The simple lock to be freed.

DESCRIPTION The **simple_lock_alloc()** function returns a pointer to a new simple lock. Before you use the simple lock, you should initialize it by calling **simple_lock_init()**.

The **simple_lock_free()** function frees the structure pointed to by *lock*.

See **simple_lock()** for information on how to use simple locks.

EXAMPLE

```
simple_lock_t   slock;

slock = simple_lock_alloc();
simple_lock_init(slock);

/* . . . */
simple_lock_free(lock1);
```

SEE ALSO **simple_lock()**, **simple_lock_init()**, **simple_unlock()**, **lock_alloc()**, **lock_free()**

simple_lock_init()

SUMMARY Initialize a simple lock

SYNOPSIS **#import <kernserv/prototypes.h>**

void **simple_lock_init(**simple_lock_t *lock***)**

ARGUMENTS *lock*: A pointer to the simple lock to be initialized.

DESCRIPTION Use this function to initialize a new simple lock. You should use **simple_lock_alloc()** to create the lock.

EXAMPLE

```
simple_lock_t   slock;

slock = simple_lock_alloc();
simple_lock_init(slock);

/* . . . */
simple_lock_free(lock1);
```

SEE ALSO **simple_lock()**, **simple_lock_alloc()**, **simple_lock_free()**, **simple_unlock()**, **lock_init()**

simple_unlock()

SUMMARY Release a simple lock

SYNOPSIS **#import <kernserv/prototypes.h>**

void **simple_unlock(**simple_lock_t *lock***)**

ARGUMENTS *lock*: A pointer to the simple lock to be released.

EXAMPLE

```
simple_lock_t   slock;

slock = simple_lock_alloc();
simple_lock_init(slock);
/* . . . */

    /* Set a lock before manipulating a data structure */
    simple_lock(slock);
    mydriver->data1 = VALUE;
    simple_unlock(slock);

    /* . . . */
    simple_lock_free(lock1);
```

SEE ALSO **simple_lock()**, **simple_lock_alloc()**, **simple_lock_free()**, **simple_lock_init()**, **lock_done()**

spl*n*()

SUMMARY Set the CPU interrupt level to *n*

SYNOPSIS **#import <kernserv/***architecture***/spl.h>**

int **spl0(**void**)**, **spl1(**void**)**, **spl2(**void**)**, **spl3(**void**)**, **spl4(**void**)**, **spl5(**void**)**, **spl6(**void**)**, **spl7(**void**)**

DESCRIPTION The **spl*n*()** macros set the hardware interrupt level of the CPU to level *n*. This means that devices whose hardware interrupt level is greater than *n* will be serviced immediately on an interrupt. Devices with interrupt levels equal to or less than *n* will not be serviced until the CPU interrupt level drops below the device interrupt level. The **spl0()** macro sets the CPU interrupt level to the lowest level, enabling all interrupts.

The **spl*n*()** macros return an integer suitable for use with **splx()** to reset the CPU interrupt level.

The following table shows the interrupts that occur at each hardware interrupt level. Because the NMI and power fail interrupts are always serviced, **spl6()** has the same effect as **spl7()** on NeXT computers.

Interrupt Level	Interrupts at This Level
7	NMI (non-maskable interrupt) key sequence Power-fail interrupt (non-maskable)
6	System-clock timeout interrupt All DMA-completion interrupts except video out
5	RS-422 (serial) device interrupt NeXTbus interrupts
4	DSP-device interrupt
3	Disk-device interrupt SCSI-device interrupt Laser-printer device interrupt Ethernet transmit/receive device interrupts (not DMA) Sound-out underrun or sound-in overrun Video-out DMA completion interrupt Monitor-control interrupt Keyboard or mouse event Power-on switch Network-device interrupts
2	Network-related software interrupts Software interrupt 1
1	Software clock interrupts (timeouts) Software interrupt 0

EXAMPLE
```
#define spl_NB()  spl5() /* NeXTbus interrupt level */
int s;

s = spl_NB();             /* Lock out all NeXTbus interrupts. */
/* Do something that requires that we not be interrupted. */
splx(s);                  /* Return to the previous interrupt level */
```

SEE ALSO **curipl()**, **splx()**

splx()

SUMMARY Reset the CPU interrupt level

SYNOPSIS **#import <kernserv/***architecture***/spl.h>**

void **splx(**int *priority***)**

ARGUMENTS *priority*: The value returned from the previous call to **spl*n*()**.

DESCRIPTION This macro returns the hardware priority interrupt level to the level that it was before issuing the last **spl*n*()** command. You must set *priority* to the value returned from the previous call to **spl*n*()**; setting it to anything else doesn't work.

EXAMPLE
```
#define spl_NB()  spl5()  /* NeXTbus interrupt level */
int  s;

s = spl_NB();   /* Lock out all NeXTbus interrupts. */
/* Do something that requires that we not be interrupted. */
splx(s);        /* Return to the previous interrupt level */
```

SEE ALSO **curipl()**, **spl*n*()**

sprintf()

SUMMARY Put characters into a string

SYNOPSIS **#import <kernserv/prototypes.h>**

int **sprintf(**char **string*, char **format*, *arg*, ...**)**

ARGUMENTS *string*: The string that you want to put the characters in.

format: The format string. It's just like the C library **printf()** function's format string, except that the only conversions available are **%s**, **%c**, **%x**, **%d**, and **%o**. (**%X**, **%D**, **%u**, and **%O** are recognized but are treated like **%x**, **%d**, **%d**, and **%o**, respectively.)

arg, ...: Optional arguments, to be formatted according to the *format* string.

DESCRIPTION This works like the C library function **sprintf()**, except that it handles only the formats allowed by the kernel **printf()** function.

SEE ALSO **printf()**, **strcat()**, **strcpy()**

strcat()

SUMMARY Concatenate two strings

SYNOPSIS **#import <kernserv/prototypes.h>**

char ***strcat(**char **string1*, char **string2***)**

ARGUMENTS *string1*: The string to add the second string to. It must have enough space for *string2* plus a null character.

string2: The string to copy to the end of *string1*.

DESCRIPTION This acts the same as the **strcat()** C library function. It returns a pointer to *string1*.

SEE ALSO **sprintf()**, **strcpy()**, **strlen()**

strcmp(), strncmp()

SUMMARY Compare two strings

SYNOPSIS **#import <kernserv/prototypes.h>**

int **strcmp(**char **string1*, char **string2***)**
int **strncmp(**char **string1*, char **string2*, unsigned long *length***)**

ARGUMENTS *string1*: The string to be compared to *string2*.

string2: The string being compared against.

length: The number of characters to compare.

DESCRIPTION These functions act the same as the **strcmp()** and **strncmp()** C library functions. They return an integer greater than, equal to, or less than 0, depending on whether *string1* is lexicographically greater than, equal to, or less than *string2*.

SEE ALSO **strlen()**

strcpy(), strncpy()

SUMMARY Copy one string to another

SYNOPSIS **#import <kernserv/prototypes.h>**

char ***strcpy(**char **to*, char **from***)**
char ***strncpy(**char **to*, char **from*, unsigned long *length***)**

ARGUMENTS *to*: The string to copy *from* to. For **strcpy()**, it must have enough space to hold all of *from*, including the null character. For **strncpy()**, it must be able to hold *length* + 1 characters.

from: The string to copy to *to*.

length: The number of characters to copy.

DESCRIPTION These functions act the same as the **strcpy()** and **strncpy()** C library functions. They return a pointer to *to*.

SEE ALSO **sprintf()**, **strcat()**, **strlen()**

strlen()

SUMMARY Get the length of a string

SYNOPSIS **#import <kernserv/prototypes.h>**

int **strlen(**char **string***)**

ARGUMENTS *string*: The string you want the length of.

DESCRIPTION This acts the same as the **strlen()** C library function. It returns the number of non-null characters in *string*.

SEE ALSO **strcmp()**

suser()

SUMMARY Check whether the user is the superuser

SYNOPSIS **#import <kernserv/prototypes.h>**

int **suser(**void**)**

DESCRIPTION This function is valid only for UNIX-style servers because message-based servers don't have access to user process information. If the user is the superuser, this returns 1 and sets a flag bit indicating that the process has used superuser privileges. Otherwise, it returns 0 and sets **u.u_error** to EPERM.

thread_block()

SUMMARY Put the current thread to sleep

SYNOPSIS **#import <kernserv/prototypes.h>**

void **thread_block(**void**)**

DESCRIPTION This function blocks the current thread from execution. You must call **assert_wait()** before calling **thread_block()**. This thread can be waked up by a timeout (set using **thread_set_timeout()**), by a call to **clear_wait()**, or by a call to **thread_wakeup()**.

EXAMPLE
```
extern  hz;
. . .
splx(s);
assert_wait(0, FALSE);
thread_set_timeout(hz/2);
thread_block();
```

SEE ALSO **assert_wait()**, **clear_wait()**, **thread_set_timeout()**, **thread_sleep()**, **thread_wakeup()**

thread_halt_self()

SUMMARY Stop the current thread

SYNOPSIS **#import <kernserv/prototypes.h>**

void **thread_halt_self(**void**)**

DESCRIPTION This makes the current thread stop running. You must first call **thread_terminate()** on the current thread.

EXAMPLE
```
thread_terminate(current_thread());
thread_halt_self();
```

SEE ALSO **thread_terminate()**

thread_set_timeout()

SUMMARY Set a timer before calling **thread_block()**

SYNOPSIS **#import <kernserv/prototypes.h>**

void **thread_set_timeout(**int *ticks***)**

ARGUMENTS *ticks*: The number of ticks to wait for. To wait for *n* seconds, this value should be *n* multiplied by the external variable **hz**.

DESCRIPTION This function sets a timer for the current thread. If you use it, you must call it between **assert_wait()** and **thread_block()**. Use the external variable **hz** (ticks per second) to convert from seconds into ticks. The thread will be waked up in *ticks*/**hz** seconds with a value of THREAD_TIMED_OUT as its wait result (obtained by calling **thread_wait_result()**).

EXAMPLE
```
splx(s);
assert_wait(0, FALSE);
thread_set_timeout(hz*2); /* set the timer to 2 seconds */
thread_block();
```

SEE ALSO **thread_block()**, **thread_wait_result()**, **us_timeout()**, **us_abs_timeout()**

thread_sleep()

SUMMARY Sleep until the specified event occurs

SYNOPSIS **#import <kernserv/prototypes.h>**

void **thread_sleep(**int *event*, simple_lock_t *lock*, boolean_t *interruptible***)**

ARGUMENTS *event*: The event to wait for. This should be a unique integer, such as the address of a buffer. If event is zero, then **thread_wakeup()** won't work on the thread; only **clear_wait()** and **thread_set_timeout()** will be able to wake it up.

lock: The simple lock to unlock before calling **thread_block()**.

interruptible: Used by **clear_wait()**. If *interruptible* is false and the *interrupt_only* argument to a later call to **clear_wait()** is true, then this thread won't be waked up by that call to **clear_wait()**.

DESCRIPTION This is a convenient way to sleep without manually calling **assert_wait()**. This function causes the current thread to wait until the specified event occurs. The specified lock is unlocked before releasing the CPU.

This function is equivalent to:

```
assert_wait(event, interruptible);  /* assert event */
simple_unlock(lock);                /* release the lock */
thread_block();                     /* block ourselves */
```

EXAMPLE
```
extern void     thread_wakeup();
struct timeval  tv = {1, 0};

s = splmine();
simple_lock(data.slock);
if (SOME_CONDITION) {
    /* wait */
    us_timeout(thread_wakeup, (int)&data, &tv, CALLOUT_PRI_SOFTINT0);
    thread_sleep((int)&data, data.slock, TRUE);
}
simple_unlock(data.slock);
splx(s);
```

SEE ALSO **assert_wait()**, **simple_unlock()**, **thread_block()**, **thread_wakeup()**

thread_wait_result()

SUMMARY Get the wait result of the current thread

SYNOPSIS **#import <kernserv/prototypes.h>**
#import <kernserv/sched_prim.h>

int **thread_wait_result(**void**)**

DESCRIPTION A thread that wakes up for any reason has a result in its thread structure; **thread_wait_result()** returns this result. Possible return values are defined in the header file **kernserv/sched_prim.h** as THREAD_AWAKENED, THREAD_TIMED_OUT, THREAD_INTERRUPTED, THREAD_SHOULD_TERMINATE, and THREAD_RESTART.

EXAMPLE
```
assert_wait(&data, FALSE);
thread_block();
printf("Wait result:  %d\n", thread_wait_result());
```

SEE ALSO **thread_set_timeout()**, **thread_wakeup()**

thread_wakeup()

SUMMARY Wake up all threads that are waiting for the specified event

SYNOPSIS void **thread_wakeup(**int *event***)**

ARGUMENTS *event*: The event that was specified in the matching **assert_wait()** or **thread_sleep()** call. This should not be zero, since this function can't wake up threads that were put to sleep with an event of zero.

DESCRIPTION The threads that this macro wakes up have THREAD_AWAKENED in their wait result (obtainable by calling **thread_wait_result()**).

Warning: This function must be called at an interrupt level of IPLSCHED or below. You can use **curipl()** to determine what the interrupt level is.

EXAMPLE
```
extern void     thread_wakeup();
struct timeval  tv = {1, 0};      /* 1-second timeout */

s = splmine();
simple_lock(data.slock);
if (SOME_CONDITION) {
    /* wait */
    us_timeout(thread_wakeup, (int)&data, &tv, CALLOUT_PRI_SOFTINT0);
    thread_sleep((int)&data, data.slock, TRUE);
}
simple_unlock(data.slock);
splx(s);
```

In an interrupt handler for a NeXTbus driver:

```
if ( (sp->flags & SERVER_THREAD_PAUSED) != 0 )
{
    kern_serv_callout( &instance, thread_wakeup,
        (void *)&sp->server_thread );
}
```

SEE ALSO **assert_wait()**, **thread_sleep()**

uninstall_polled_intr()

SUMMARY Remove an interrupt handler for a polled device

SYNOPSIS **#import <kernserv/prototypes.h>**

int **uninstall_polled_intr(**int *which*, int (**my_intr*)()**)**

ARGUMENTS *which:* Specifies the device and interrupt level. For devices attached through the NeXTbus interface, this should be the constant I_BUS, which is defined in the header file **architecture/m68k/intr.h**.

my_intr: The function in your server that handles this interrupt.

DESCRIPTION This function removes *my_intr* from the list of functions that are called when an interrupt occurs at interrupt level *which*.

This function returns 0 if the call is successful. It returns -1 if the interrupt level specified by *which* isn't capable of interrupt polling, or if *my_intr* isn't found.

EXAMPLE `device_cleanup()`

```
{
    /* . . . */
    uninstall_polled_intr(I_BUS, device_interrupt);
    /* . . . */
}
```

SEE ALSO **install_polled_intr()**