Bull

Technical Reference
Kernel & Subsystems

Volume 1/2

AIX

ORDER REFERENCE
86 A2 85AP 05

Bull

Bull

Technical Reference
Kernel & Subsystems

Volume 1/2

AIX

Software

February 1999

BULL ELECTRONICS ANGERS
CEDOC
34 Rue du Nid de Pie – BP 428
49004 ANGERS CEDEX 01
FRANCE

ORDER REFERENCE
86 A2 85AP 05

Printed in France

Suggestions and criticisms concerning the form, content, and presentation of this book are invited. A form is provided at the end of this book for this purpose.

To order additional copies of this book or other Bull Technical Publications, you are invited to use the Ordering Form also provided at the end of this book.

Trademarks and Acknowledgements

We acknowledge the right of proprietors of trademarks mentioned in this book.

AIX® is a registered trademark of International Business Machines Corporation, and is being used under licence.

UNIX is a registered trademark in the United States of America and other countries licensed exclusively through the Open Group.

Year 2000

The product documented in this manual is Year 2000 Ready.

Table of Contents

About This Book

Kernel and Subsystems Technical Reference, Volumes 1 and 2 provide information about kernel services, device driver operations, file system operations, subroutines, the configuration subsystem, the communications subsystem, the low function terminal (LFT) subsystem, the logical volume subsystem, the M–audio capture and playback adapter subsystem, the printer subsystem, the SCSI subsystem, and the serial DASD subsystem.

These two books are part of the six–volume technical reference set, *AIX Technical Reference*, 86 A2 81AP to 86 A2 91AP, which provides information on system calls, kernel extension calls, and subroutines in the following volumes:

- *Base Operating System and Extensions, Volumes 1 and 2* provide information on system calls, subroutines, functions, macros, and statements associated with AIX base operating system runtime services.
- *Communications, Volumes 1 and 2* provide information on entry points, functions, system calls, subroutines, and operations related to communications services.
- *Kernel and Subsystems, Volumes 1 and 2* provide information on the topics described in the first paragraph.

Kernel Extensions and Device Support Programming Concepts, a companion volume to this book, provides a conceptual introduction to the kernel programming environment and how to extend it.

Who Should Use This Book

Kernel and Subsystems Technical Reference, Volumes 1 and 2 are intended for system programmers wishing to extend the AIX kernel. To use this book effectively, you should be familiar with operating system concepts and kernel programming. To review this background, see *Kernel Extensions and Device Support Programming Concepts*.

How to Use This Book

Overview of Contents

Kernel and Subsystems, Volume 1 contains information needed to write kernel extensions. This includes:

- The kernel services provided in the AIX kernel, in alphabetical order.
- Interface requirements for writing device drivers. Device driver routines and related data structures are discussed here.
- Interface requirements for writing virtual file systems. Descriptions of virtual file system routines are provided.

Highlighting

The following highlighting conventions are used in this book:

Bold	Identifies commands, subroutines, keywords, files, structures, directories, and other items whose names are predefined by the system. Also identifies graphical objects such as buttons, labels, and icons that the user selects.
Italics	Identifies parameters whose actual names or values are to be supplied by the user.
`Monospace`	Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code similar to what you might write as a programmer, messages from the system, or information you should actually type.

ISO 9000

ISO 9000 registered quality systems were used in the development and manufacturing of this product.

AIX 32–Bit Support for the X/Open UNIX95 Specification

Beginning with AIX Version 4.2, the operating system is designed to support the X/Open UNIX95 Specification for portability of UNIX–based operating systems. Many new interfaces, and some current ones, have been added or enhanced to meet this specification. Beginning with Version 4.2, AIX is even more open and portable for applications.

At the same time, compatibility with previous AIX releases is preserved. This is accomplished by the creation of a new environment variable, which can be used to set the system environment on a per–system, per–user, or per–process basis.

To determine the proper way to develop a UNIX95–portable application, you may need to refer to the X/Open UNIX95 Specification, which can be obtained on a CD–ROM by ordering the printed copy of *AIX Commands Reference*, order number 86 A2 38JX to 86 A2 43JX, or by ordering *Go Solo: How to Implement and Go Solo with the Single Unix Specification*, a book which includes the X/Open UNIX95 Specification on a CD–ROM.

AIX 32–Bit and 64–Bit Support for the UNIX98 Specification

Beginning with AIX Version 4.3, the operating system is designed to support the X/Open UNIX98 Specification for portability of UNIX–based operating systems. Many new interfaces, and some current ones, have been added or enhanced to meet this specification. Making AIX Version 4.3 even more open and portable for applications.

At the same time, compatibility with previous AIX releases is preserved. This is accomplished by the creation of a new environment variable, which can be used to set the system environment on a per–system, per–user, or per–process basis.

To determine the proper way to develop a UNIX98–portable application, you may need to refer to the X/Open UNIX98 Specification, which can be obtained on a CD–ROM by ordering the printed copy of *AIX Commands Reference*, order number 86 A2 38JX to 86 A2 43JX, or by ordering *Go Solo: How to Implement and Go Solo with the Single Unix Specification*, a book which includes the X/Open UNIX98 Specification on a CD–ROM.

Related Publications

The following books contain information on device drivers and other kernel extensions:

- *AIX and Related Products Documentation Overview*, Order Number 86 A2 71WE.
- *AIX Kernel Extensions and Device Support Programming Concepts*, Order Number 86 A2 36JX.
- *AIX Communications Programming Concepts*, Order Number 86 A2 35JX.
- *AIX General Programming Concepts : Writing and Debugging Programs*, Order Number 86 A2 34JX.

Ordering Publications

You can order publications from your sales representative or from your point of sale.

If you received a printed copy of *Documentation Overview* with your system, use that book for information on related publications and for instructions on ordering them.

To order additional copies, use the following order numbers:

- *AIX Technical Reference, Volume 5: Kernel and Subsystems*, Order Number 86 A2 85AP.
- *AIX Technical Reference, Volume 6: Kernel and Subsystems*, Order Number 86 A2 86AP.

To order additional copies of the six–volume set, order *AIX Technical Reference*, Order Number 86 A2 81AP to 86 A2 91AP.

Chapter 1. Kernel Services

Alphabetical List of Kernel Services

This list provides the names of all available kernel services. It is divided by the execution environment from which each kernel service can be called:

- Both process and interrupt environments
- Process environment only

"System Calls Available to Kernel Extensions" in *AIX Kernel Extensions and Device Support Programming Concepts* lists the systems calls that can be called by kernel extensions.

Kernel Services Available in Process and Interrupt Environments

add_domain_af	Adds an address family to the Address Family domain switch table.
add_input_type	Adds a new input type to the Network Input table.
add_netisr	Adds a network software interrupt service to the Network Interrupt table.
add_netopt	Adds a network option structure to the list of network options.
bdwrite	Releases the specified buffer after marking it for delayed write.
brelse	Frees the specified buffer.
clrbuf	Sets the memory for the specified buffer structure's buffer to all zeros.
clrjmpx	Removes a saved context by popping the most recently saved jump buffer from the list of saved contexts.
curtime	Reads the current time into a time structure.
d_align	Assists in allocation of DMA buffers.
d_cflush	Flushes the processor and I/O controller (IOCC) data caches when using the long term **DMA_WRITE_ONLY** mapping of Direct Memory Access (DMA) buffers approach to bus device DMA.
d_clear	Frees a DMA channel.
d_complete	Cleans up after a DMA transfer.
d_init	Initializes a DMA channel.
d_mask	Disables a DMA channel.
d_master	Initializes a block–mode DMA transfer for a DMA master.
d_move	Provides consistent access to system memory that is accessed asynchronously by a device and by the processor.
d_roundup	Assists in allocation of DMA buffers.
d_slave	Initializes a block–mode DMA transfer for a DMA slave.
d_unmask	Enables a DMA channel.
del_domain_af	Deletes an address family from the Address Family domain switch table.
del_input_type	Deletes an input type from the Network Input table.
del_netisr	Deletes a network software interrupt service routine from the Network Interrupt table.
del_netopt	Deletes a network option structure from the list of network options.
devdump	Calls a device driver dump–to–device routine.
devstrat	Calls a block device driver's strategy routine.

devswqry	Checks the status of a device switch entry in the device switch table.
DTOM macro	Converts an address anywhere within an **mbuf** structure to the head of that **mbuf** structure.
et_post	Notifies a kernel thread of the occurrence of one or more events.
e_wakeup	Notifies processes waiting on a shared event of the event's occurrence.
errsave and **errlast**	Allows the kernel and kernel extensions to write to the error log.
find_input_type	Finds the given packet type in the Network Input Interface switch table and distributes the input packet according to the table entry for that type.
getc	Retrieves a character from a character list.
getcb	Removes the first buffer from a character list and returns the address of the removed buffer.
getcbp	Retrieves multiple characters from a character buffer and places them at a designated address.
getcf	Retrieves a free character buffer.
getcx	Returns the character at the end of a designated list.
geterror	Determines the completion status of the buffer.
getexcept	Allows kernel exception handlers to retrieve additional exception information.
getpid	Gets the process ID of the current process.
i_disable	Disables all of the interrupt levels at a particular interrupt priority and all interrupt levels at a less–favored interrupt priority.
i_enable	Enables all of the interrupt levels at a particular interrupt priority and all interrupt levels at a more–favored interrupt priority.
i_mask	Disables an interrupt level.
i_reset	Resets the system's hardware interrupt latches.
i_sched	Schedules off–level processing.
i_unmask	Enables an interrupt level.
if_attach	Adds a network interface to the network interface list.
if_detach	Deletes a network interface from the network interface list.
if_down	Marks an interface as down.
if_nostat	Zeroes statistical elements of the interface array in preparation for an attach operation.
ifa_ifwithaddr	Locates an interface based on a complete address.
ifa_ifwithdstaddr	Locates the point–to–point interface with a given destination address.
ifa_ifwithnet	Locates an interface on a specific network.
ifunit	Returns a pointer to the **ifnet** structure of the requested interface.
io_att	Selects, allocates, and maps a region in the current address space for I/O access.
io_det	Unmaps and deallocates the region in the current address space at the given address.
iodone	Performs block I/O completion processing.
IS64U	Determines if the current user–address space is 64–bit or not.
kgethostname	Retrieves the name of the current host.

kgettickd	Retrieves the current status of the systemwide time–of–day timer–adjustment values.
ksettickd	Sets the current status of the systemwide timer–adjustment values.
loifp	Returns the address of the software loopback interface structure.
longjmpx	Allows exception handling by causing execution to resume at the most recently saved context.
looutput	Sends data through a software loopback interface.
m_adj	Adjusts the size of an **mbuf** chain.
m_cat	Appends one **mbuf** chain to the end of another.
m_clattach	Allocates an **mbuf** structure and attaches an external cluster.
m_clget macro	Allocates a page–sized **mbuf** structure cluster.
m_clgetm	Allocates and attaches an external buffer.
m_collapse	Guarantees that an **mbuf** chain contains no more than a given number of **mbuf** structures.
m_copy macro	Creates a copy of all or part of a list of **mbuf** structures.
m_copydata	Copies data from an **mbuf** chain to a specified buffer.
m_copym	Creates a copy of all or part of a list of **mbuf** structures.
m_free	Frees an **mbuf** structure and any associated external storage area.
m_freem	Frees an entire **mbuf** chain.
m_get	Allocates a memory buffer from the **mbuf** pool.
m_getclr	Allocates and zeros a memory buffer from the **mbuf** pool.
m_getclust macro	Allocates an **mbuf** structure from the **mbuf** buffer pool and attaches a page–sized cluster.
m_getclustm	Allocates an **mbuf** structure from the **mbuf** buffer pool and attaches a cluster of the specified size.
m_gethdr	Allocates a header memory buffer from the **mbuf** pool.
M_HASCL macro	Determines if an **mbuf** structure has an attached cluster.
m_pullup	Adjusts an **mbuf** chain so that a given number of bytes is in contiguous memory in the data area of the head **mbuf** structure.
MTOCL macro	Converts a pointer to an **mbuf** structure to a pointer to the head of an attached cluster.
MTOD macro	Converts a pointer to an **mbuf** structure to a pointer to the data stored in that **mbuf** structure.
M_XMEMD macro	Returns the address of an **mbuf** cross–memory descriptor.
net_error	Handles errors for AIX communication network interface drivers.
net_start_done	Starts the done notification handler for AIX communications I/O device handlers.
net_wakeup	Wakes up all sleepers waiting on the specified wait channel.
net_xmit	Transmits data using an AIX communications I/O device handler.
net_xmit_trace	Traces transmit packets. This kernel service was added for those network interfaces that choose not to use the **net_xmit** kernel service to trace transmit packets.
panic	Crashes the system.
pfctlinput	Invokes the **ctlinput** function for each configured protocol.

pffindproto	Returns the address of a protocol switch table entry.
pidsig	Sends a signal to a process.
pgsignal	Sends a signal to a process group.
pio_assist	Provides a standardized programmed I/O exception handling mechanism for all routines performing programmed I/O.
putc	Places a character at the end of a character list.
putcb	Places a character buffer at the end of a character list.
putcbp	Places several characters at the end of a character list.
putcf	Frees a specified buffer.
putcfl	Frees the specified list of buffers.
putcx	Places a character on a character list.
raw_input	Builds a **raw_header** structure for a packet and sends both to the raw protocol handler.
raw_usrreq	Implements user requests for raw protocols.
rtalloc	Allocates a route.
rtfree	Frees the routing table entry.
rtinit	Sets up a routing table entry, typically for a network interface.
rtredirect	Forces a routing table entry with the specified destination to go through the given gateway.
rtrequest	Carries out a request to change the routing table.
schednetisr	Schedules or invokes a network software interrupt service routine.
selnotify	Wakes up processes waiting in a **poll** or **select** subroutine or the **fp_poll** kernel service.
setjmpx	Allows saving the current execution state or context.
setpinit	Sets the parent of the current kernel process to the init process.
tfree	Deallocates a timer request block.
timeout	Schedules a function to be called after a specified interval.
trcgenk	Records a trace event for a generic trace channel.
trcgenkt	Records a trace event, including a time stamp, for a generic trace channel.
tstart	Submits a timer request.
tstop	Cancels a pending timer request.
uexblock	Makes a process non–runnable when called from a user–mode exception handler.
uexclear	Makes a process blocked by the **uexblock** service runnable again.
unpin	Unpins the address range in system (kernel) address space.
unpinu	Unpins the specified address range in user or system memory.
untimeout	Cancels a pending timer request.
vm_att	Maps a specified virtual memory object to a region in the current address space.
vm_det	Unmaps and deallocates the region in the current address space that contains a given address.
xmdetach	Detaches from a user buffer used for cross–memory operations.
xmemdma	Prepares a page for DMA I/O or processes a page after DMA I/O is complete.

xmemin	Performs a cross–memory move by copying data from the specified address space to kernel global memory.
xmemout	Performs a cross–memory move by copying data from kernel global memory to a specified address space.

Kernel Services Available in the Process Environment Only

as_att	Selects, allocates, and maps a region in the specified address space for the specified virtual memory object.
as_att64	Allocates and maps a specified region in the current user address space.
as_det	Unmaps and deallocates a region in the specified address space that was mapped with the **as_att** kernel service.
as_det64	Unmaps and deallocates a region in the current user address space that was mapped with the **as_att64 kernel** service.
as_geth64	Obtains a handle to the virtual memory object for the specified address.
as_getsrval64	Obtains a handle to the virtual memory object for the specified address.
as_puth64	Indicates that no more references will be made to a virtual memory object obtained using the **as_geth64** kernel service.
as_remap64	Remaps an additional 64–bit address to a 32–bit address that can be used by the kernel.
as_seth64	Maps a specified region for the specified virtual memory object.
as_unremap64	Returns the 64–bit original or unremapped address associated with a 32–bit remapped address.
audit_svcbcopy	Appends event information to the current audit event buffer.
audit_svcfinis	Writes an audit record for a kernel service.
audit_svcstart	Initiates an audit record for a system call.
bawrite	Writes the specified buffer's data without waiting for I/O to complete.
bflush	Flushes all write–behind blocks on the specified device from the buffer cache.
binval	Invalidates all of a specified device's data in the buffer cache.
blkflush	Flushes the specified block if it is in the buffer cache.
bread	Reads the specified block's data into a buffer.
breada	Reads in the specified block and then starts I/O on the read–ahead block.
bwrite	Writes the specified buffer's data.
cfgnadd	Registers a notification routine to be called when system–configurable variables are changed.
cfgndel	Removes a notification routine for receiving broadcasts of changes to system configurable variables.
copyin	Copies data between user and kernel memory.
copyin64	Copies data between user and kernel memory.
copyinstr	Copies a character string (including the terminating NULL character) from user to kernel space.
copyinstr64	Copies data between user and kernel memory.
copyout	Copies data between user and kernel memory.

copyout64	Copies data between user and kernel memory.
creatp	Creates a new kernel process.
delay	Suspends the calling process for the specified number of timer ticks.
devswadd	Adds a device entry to the device switch table.
devswdel	Deletes a device driver entry from the device switch table.
dmp_add	Specifies data to be included in a system dump by adding an entry to the master dump table.
dmp_del	Deletes an entry from the master dump table.
dmp_prinit	Initializes the remote dump protocol.
e_sleep	Forces a process to wait for the occurrence of a shared event.
e_sleepl	Forces a process to wait for the occurrence of a shared event.
et_wait	Forces a process to wait for the occurrence of an event.
enque	Sends a request queue element to a device queue.
fp_access	Checks for access permission to an open file.
fp_close	Closes a file.
fp_fstat	Gets the attributes of an open file.
fp_getdevno	Gets the device number and/or channel number for a device.
fp_getf	Retrieves a pointer to a file structure.
fp_hold	Increments the open count for a specified file pointer.
fp_ioctl	Issues a control command to an open device or file.
fp_lseek	Changes the current offset in an open file.
fp_open	Opens a regular file or directory.
fp_opendev	Opens a device special file.
fp_poll	Checks the I/O status of multiple file pointers/descriptors and message queues.
fp_read	Performs a read on an open file with arguments passed.
fp_readv	Performs a read operation on an open file with arguments passed in **iovec** elements.
fp_rwuio	Performs read and write on an open file with arguments passed in a **uio** structure.
fp_select	Provides for cascaded, or redirected, support of the select or poll request.
fp_write	Performs a write operation on an open file with arguments passed.
fp_writev	Performs a write operation on an open file with arguments passed in **iovec** elements.
fubyte	Fetches, or retrieves, a byte of data from user memory.
fubyte64	Retrieves a byte of data from user memory.
fuword	Fetches, or retrieves, a word of data from user memory.
fuword64	Retrieves a word of data from user memory.
getadsp	Obtains a pointer to the current process's address space structure for use with the **as_att** and **as_det** kernel services.
getblk	Assigns a buffer to the specified block.
geteblk	Allocates a free buffer.

getppidx	Gets the parent process ID of the specified process.
getuerror	Allows kernel extensions to retrieve the current value of the **u_error** field.
gfsadd	Adds a file system type to the **gfs** table.
gfsdel	Removes a file system type from the **gfs** table.
i_clear	Removes an interrupt handler from the system.
i_init	Defines an interrupt handler to the system, connects it to an interrupt level, and assigns an interrupt priority to the level.
init_heap	Initializes a new heap to be used with kernel memory management services.
initp	Changes the state of a kernel process from idle to ready.
iostadd	Registers an I/O statistics structure used for updating I/O statistics reported by the **iostat** subroutine.
iostdel	Removes the registration of an I/O statistics structure used for maintaining I/O statistics on a particular device.
iowait	Waits for block I/O completion.
kmod_entrypt	Returns a function pointer to a kernel module's entry point.
kmod_load	Loads an object file into the kernel or queries for an object file already loaded.
kmod_unload	Unloads a kernel object file.
kmsgctl	Provides message queue control operations.
kmsgget	Obtains a message queue identifier.
kmsgrcv	Reads a message from a message queue.
kmsgsnd	Sends a message using a previously defined message queue.
ksettimer	Sets the systemwide time–of–day timer.
lockl	Locks a conventional process lock.
lookupvp	Retrieves the vnode that corresponds to the named path.
m_dereg	Deregisters expected **mbuf** structure usage.
m_reg	Registers expected **mbuf** usage.
net_attach	Opens an AIX communications I/O device handler.
net_detach	Closes an AIX communications I/O device handler.
net_sleep	Sleeps on the specified wait channel.
net_start	Starts network IDs on an AIX communications I/O device handler.
NLuprintf	Submits a request to print an internationalized message to the controlling terminal of a process.
pin	Pins the address range in the system (kernel) space.
pincf	Manages the list of free character buffers.
pincode	Pins the code and data associated with an object file.
pinu	Pins the specified address range in user or system memory.
prochadd	Adds a systemwide process state–change notification routine.
prochdel	Deletes a process state change notification routine.
purblk	Invalidates a specified block's data in the buffer cache.
remap_64	Registers the input remapping of one or more addresses for the duration of a system call for a 64–bit process.

rmmap_create64	Defines an Effective Address [EA] to Real Address [RA] translation region for either 64–bit or 32–bit Effective Addresses.
rmmap_remove64	Destroys an effective address to real address translation region.
setuerror	Allows kernel extensions to set the **u_error** field in the **u** area.
sig_chk	Provides a kernel process the ability to poll for receipt of signals.
sleep	Forces the calling process to wait on a specified channel.
subyte	Stores a byte of data in user memory.
subyte64	Stores a byte of data in user memory.
suser	Determines the privilege state of a process.
suword	Stores a word of data in user memory.
suword64	Stores a word of data in user memory.
talloc	Allocates a timer request block before starting a timer request.
timeoutcf	Allocates or deallocates callout table entries for use with the **timeout** kernel service.
uexadd	Adds a systemwide exception handler for catching user–mode process exceptions.
uexdel	Deletes a previously added systemwide user–mode exception handler.
ufdcreate	Provides a file interface to kernel services.
uiomove	Moves a block of data between kernel space and a space defined by a **uio** structure.
unlockl	Unlocks a conventional process lock.
unpincode	Unpins the code and data associated with an object file.
uprintf	Submits a request to print a message to the controlling terminal of a process.
uphysio	Performs character I/O for a block device using a **uio** structure.
ureadc	Writes a character to a buffer described by a **uio** structure.
uwritec	Retrieves a character from a buffer described by a **uio** structure.
vec_clear	Removes a virtual interrupt handler.
vec_init	Defines a virtual interrupt handler.
vfsrele	Points to a virtual file system structure.
vm_cflush	Flushes the processor's cache for a specified address range.
vm_handle	Constructs a virtual memory handle for mapping a virtual memory object with specified access level.
vm_makep	Makes a page in client storage.
vm_mount	Adds a file system to the paging device table.
vm_move	Moves data between a virtual memory object and a buffer specified in the **uio** structure.
vm_protectp	Sets the page protection key for a page range.
vm_qmodify	Determines whether a mapped file has been changed.
vm_release	Releases virtual memory resources for the specified address range.
vm_releasep	Releases virtual memory resources for the specified page range.
vm_uiomove	Moves data between a virtual memory object and a buffer specified in the uio structure.
vm_umount	Removes a file system from the paging device table.

vm_write	Initiates page–out for a page range in the address space.
vm_writep	Initiates page–out for a page range in a virtual memory object.
vms_create	Creates a virtual memory object of the type and size and limits specified.
vms_delete	Deletes a virtual memory object.
vms_iowait	Waits for the completion of all page–out operations for pages in the virtual memory object.
vn_free	Frees a vnode previously allocated by the **vn_get** kernel service.
vn_get	Allocates a virtual node and inserts it into the list of vnodes for the designated virtual file system.
waitcfree	Checks the availability of a free character buffer.
waitq	Waits for a queue element to be placed on a device queue.
w_clear	Removes a watchdog timer from the list of watchdog timers known to the kernel.
w_init	Registers a watchdog timer with the kernel.
w_start	Starts a watchdog timer.
w_stop	Stops a watchdog timer.
xmalloc	Allocates memory.
xmattach	Attaches to a user buffer for cross–memory operations.
xmattach64	Attaches to a user buffer for cross–memory operations.
xmfree	Frees allocated memory.

add_domain_af Kernel Service

Purpose

Adds an address family to the Address Family domain switch table.

Syntax

#include <sys/types.h>
#include <sys/errno.h>
#include <sys/domain.h>

int add_domain_af (*domain***)**
struct domain ***domain*;**

Parameter

domain	Specifies the domain of the address family.

Description

The **add_domain_af** kernel service adds an address family domain to the Address Family domain switch table.

Execution Environment

The **add_domain_af** kernel service can be called from either the process or interrupt environment.

Return Values

0	Indicates that the address family was successfully added.
EEXIST	Indicates that the address family was already added.
EINVAL	Indicates that the address family number to be added is out of range.

Example

To add an address family to the Address Family domain switch table, invoke the **add_domain_af** kernel service as follows:

```
add_domain_af(&inetdomain);
```

In this example, the family to be added is `inetdomain`.

Implementation Specifics

The **add_domain_af** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **del_domain_af** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

add_input_type Kernel Service

Purpose

Adds a new input type to the Network Input table.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <net/if.h>
#include <net/netisr.h>

int add_input_type (type, service_level, isr, ifq, af)
u_short type;
u_short service_level;
int  (*isr) ();
struct ifqueue *ifq;
u_short af;
```

Parameters

type	Specifies which type of protocol a packet contains. A value of x'FFFF' indicates that this input type is a wildcard type and matches all input packets.
service_level	Determines the processing level at which the protocol input handler is called. If the *service_level* parameter is set to **NET_OFF_LEVEL**, the input handler specified by the *isr* parameter is called directly. Setting the *service_level* parameter to **NET_KPROC** schedules a network dispatcher. This dispatcher calls the subroutine identified by the *isr* parameter.
isr	Identifies the routine that serves as the input handler for an input packet type.
ifq	Specifies an input queue for holding input buffers. If this parameter has a non–null value, an input buffer (**mbuf**) is enqueued. The *ifq* parameter must be specified if the processing level specified by the *service_level* parameter is **NET_KPROC.** Specifying null for this parameter generates a call to the input handler specified by the *isr* parameter, as in the following:
af	Specifies the address family of the calling protocol. The *af* parameter must be specified if the *ifq* parameter is not a null character.

```
(*isr)(CommonPortion,Buffer);
```

In this example, `CommonPortion` points to the network common portion (the **arpcom** structure) of a network interface and `Buffer` is a pointer to a buffer (**mbuf**) containing an input packet.

Description

To enable the reception of packets, an address family calls the **add_input_type** kernel service to register a packet type in the Network Input table. Multiple packet types require multiple calls to*AIX Kernel Extensions and Device Support Programming Concepts* the **add_input_type** kernel service.

Execution Environment

The **add_input_type** kernel service can be called from either the process or interrupt environment.

Return Values

0	Indicates that the type was successfully added.
EEXIST	Indicates that the type was previously added to the Network Input table.
ENOSPC	Indicates that no free slots are left in the table.
EINVAL	Indicates that an error occurred in the input parameters.

Examples

1. To register an Internet packet type (**TYPE_IP**), invoke the **add_input_type** service as follows:

   ```
   add_input_type(TYPE_IP,  NET_KPROC,  ipintr,  &ipintrq,
   AF_INET);
   ```

   This packet is processed through the network `kproc`. The input handler is `ipintr`. The input queue is `ipintrq`.

2. To specify the input handler for ARP packets, invoke the **add_input_type** service as follows:

   ```
   add_input_type(TYPE_ARP,  NET_OFF_LEVEL,  arpinput,  NULL,
   NULL);
   ```

   Packets are not queued and the `arpinput` subroutine is called directly.

Implementation Specifics

The **add_input_type** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **del_input_type** kernel service, **find_input_type** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

add_netisr Kernel Service

Purpose

Adds a network software interrupt service to the Network Interrupt table.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <net/netisr.h>

int add_netisr (soft_intr_level, service_level, isr)
u_short soft_intr_level;
u_short service_level;
int (*isr) ();
```

Parameters

soft_intr_level	Specifies the software interrupt level to add. This parameter must be greater than or equal to 0 and less than **NETISR_MAX**.
service_level	Specifies the processing level of the network software interrupt.
isr	Specifies the interrupt service routine to add.

Description

The **add_netisr** kernel service adds the software–interrupt level specified by the *soft_intr_level* parameter to the Network Software Interrupt table.

The processing level of a network software interrupt is specified by the *service_level* parameter. If the interrupt level specified by the *service_level* parameter equals **NET_KPROC**, a network interrupt scheduler calls the function specified by the *isr* parameter. If you set the *service_level* parameter to **NET_OFF_LEVEL**, the **schednetisr** service calls the interrupt service routine directly.

Execution Environment

The **add_netisr** kernel service can be called from either the process or interrupt environment.

Return Values

0	Indicates that the interrupt service routine was successfully added.
EEXIST	Indicates that the interrupt service routine was previously added to the table.
EINVAL	Indicates that the value specified for the *soft_intr_level* parameter is out of range or at a service level that is not valid.

Implementation Specifics

The **add_netisr** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **del_netisr** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

add_netopt Macro

Purpose

Adds a network option structure to the list of network options.

Syntax

#include <sys/types.h>
#include <sys/errno.h>
#include <net/netopt.h>

add_netopt (*option_name_symbol***,***print_format***)**
*option_name_symbol***;**
char ***print_format*;**

Parameters

option_name_symbol	Specifies the symbol name used to construct the **netopt** structure and default names.
print_format	Specifies the string representing the print format for the network option.

Description

The **add_netopt** macro adds a network option to the linked list of network options. The **no** command can then be used to show or alter the variable's value.

The **add_netopt** macro has no return values.

Execution Environment

The **add_netopt** macro can be called from either the process or interrupt environment.

Implementation Specifics

The **add_netopt** macro is part of Base Operating System (BOS) Runtime.

Related Information

The **no** command.

The **del_netopt** macro.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

as_att Kernel Service

Purpose

Selects, allocates, and maps a region in the specified address space for the specified virtual memory object.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
#include <sys/adspace.h>

caddr_t as_att (adspacep, vmhandle, offset)
adspace_t *adspacep;
vmhandle_t vmhandle;
caddr_t offset;
```

Parameters

adspacep	Points to the address space structure that defines the address space where the region for the virtual memory object is to be allocated. The **getadsp** kernel service can obtain this pointer.
vmhandle	Describes the virtual memory object being made addressable within a region of the specified address space.
offset	Specifies the offset in the virtual memory object and the region being mapped. On this system, the upper 4 bits of this offset are ignored.

Description

The **as_att** kernel service:

- Selects an unallocated region within the address space specified by the *adspacep* parameter.
- Allocates the region.
- Maps the virtual memory object selected by the *vmhandle* parameter with the access permission specified in the handle.
- Constructs the address of the offset specified by the *offset* parameter in the specified address space.

If the specified address space is the current address space, the region becomes immediately addressable. Otherwise, it becomes addressable when the specified address space next becomes the active address space.

Kernel extensions use the **as_att** kernel service to manage virtual memory object addressability within a region of a particular address space. They are also used by base operating system subroutines such as the **shmat** and **shmdt** subroutines.

Subroutines executed by a kernel extension may be executing under a process, with a process address space, or executing under a kernel process, entirely in the current address space. (The **as_att** service never switches to a user–mode address space.) The **getadsp** kernel service should be used to get the correct address–space structure pointer in either case.

The **as_att** kernel service assumes an address space model of fixed–size virtual memory objects and address space regions.

Execution Environment

The **as_att** kernel service can be called from the process environment only.

Return Values

If successful, the **as_att** service returns the address of the offset (specified by the *offset* parameter) within the region in the specified address space where the virtual memory object was made addressable.

If there are no more free regions within the specified address space, the **as_att** service will not allocate a region and returns a null address.

Implementation Specifics

The **as_att** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **as_det** kernel service, **as_geth** kernel service, **as_getsrva**l kernel service, **as_puth** kernel service, **getadsp** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

as_att64 Kernel Service

Purpose

Allocates and maps a specified region in the current user address space.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
#include <sys/adspace.h>

unsigned long long as_att64 (vmhandle, offset)
vmhandle_t vmhandle;
int offset;
```

Parameters

vmhandle	Describes the virtual memory object being made addressable in the address space.
offset	Specifies the offset in the virtual memory object. The upper 4–bits of this offset are ignored.

Description

The **as_att64** kernel service:

- Selects an unallocated region within the current user address space.
- Allocates the region.
- Maps the virtual memory object selected by the vmhandle parameter with the access permission specified in the handle.
- Constructs the address of the offset specified by the offset parameter within the user–address space.

The **as_att64** kernel service assumes an address space model of fixed–size virtual memory objects.

This service will operate correctly for both 32–bit and 64–bit user address spaces. It will also work for kernel processes (*kprocs*).

Note: This service only operates on the current process's address space. It is not allowed to operate on another address space.

Execution Environment

The **as_att64** kernel service can be called from the process environment only.

Return Values

On successful completion, this service returns the base address plus the input offset (offset) into the allocated region.

NULL	An error occurred and ernno indicates the cause:
EINVAL	Address specified is out of range, or
ENOMEM	Could not allocate due to insufficient resources.

Implementation Specifics

The **as_att64** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **as_seth64** kernel service, **as_det64** kernel service, **as_geth64** kernel service, **as_getsrval64** kernel service, **as_puth64** kernel service.

as_det Kernel Service

Purpose

Unmaps and deallocates a region in the specified address space that was mapped with the **as_att** kernel service.

Syntax

```
#include <sys/errno.h>
#include <sys/vmuser.h>
#include <sys/adspace.h>

int as_det (adspacep,eaddr)
adspace_t *adspacep;
caddr_t eaddr;
```

Parameters

adspacep	Points to the address space structure that defines the address space where the region for the virtual memory object is defined. For the current process, the **getadsp** kernel service can obtain this pointer.
eaddr	Specifies the effective address within the region to be deallocated in the specified address space.

Description

The **as_det** kernel service unmaps the virtual memory object from the region containing the specified effective address (specified by the *eaddr* parameter*)* and deallocates the region from the address space specified by the *adspacep* parameter. This region is added to the free list for the specified address space.

The **as_det** kernel service assumes an address space model of fixed–size virtual memory objects and address space regions.

Note: This service should not be used to deallocate a base kernel region, process text, process private or unallocated region: an **EINVAL** return code will result. For this system, the upper 4 bits of the *eaddr* effective address parameter must never be 0, 1, 2, 0xE, or specify an unallocated region.

Execution Environment

The **as_det** kernel service can be called from the process environment only.

Return Values

0	The region was successfully unmapped and deallocated.
EINVAL	An attempt was made to deallocate a region that should not have been deallocated (that is, a base kernel region, process text region, process private region, or unallocated region).

Implementation Specifics

The **as_det** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **as_att** kernel service, **getadsp** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

as_det64 Kernel Service

Purpose

Unmaps and deallocates a region in the current user address space that was mapped with the **as_att64** kernel service.

Syntax

```
#include <sys/errno.h>
#include <sys/adspace.h>

int as_det64 (addr64)
unsigned long long addr64;
```

Parameters

addr64	Specifies an effective address within the region to be deallocated.

Description

The **as_det64** kernel service unmaps the virtual memory object from the region containing the specified effective address (specified by the **addr64** parameter).

The **as_det64** kernel service assumes an address space model of fixed–size virtual memory objects.

This service should not be used to deallocate a base kernel region, process text, process private or an unallocated region. An **EINVAL** return code will result.

This service will operate correctly for both 32–bit and 64–bit user address spaces. It will also work for kernel processes *(kprocs)*.

> **Note:** This service only operates on the current process's address space. It is not allowed to operate on another address space.

Execution Environment

The **as_det64** kernel service can be called from the process environment only.

Return Values

0	The region was successfully unmapped and deallocated.
EINVAL	An attempt was made to deallocate a region that should not have been deallocated (that is, a base kernel region, process text region, process private region, or unallocated region).
EINVAL	Input address out of range.

Implementation Specifics

The **as_det64** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **as_att64** kernel service, **as_seth64** kernel service, **as_geth64** kernel service, **as_getsrval64** kernel service, **as_puth64** kernel service.

as_geth Kernel Service

Purpose

Obtains a handle to the virtual memory object for the specified address given in the specified address space.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
#include <sys/adspace.h>

vmhandle_t as_geth (Adspacep, Addr)
adspace_t *Adspacep;
caddr_t Addr;
```

Parameters

Adspacep	Points to the address space structure to obtain the virtual memory object handle from. The **getadsp** kernel service can obtain this pointer.
Addr	Specifies the virtual memory address that should be used to determine the virtual memory object handle for the specified address space.

Description

The **as_geth** kernel service is used to obtain a handle to the virtual memory object corresponding to a virtual memory address in a particular address space. This handle can then be used with the **as_att** or **vm_att** kernel services to make the object addressable in another address space.

After the last use of the handle and after it is detached from all address spaces, the **as_puth** kernel service must be used to indicate this fact. Failure to call the **as_puth** kernel service may result in resources being permanently unavailable for reuse.

If the handle obtained refers to a virtual memory segment, then that segment is protected from deletion until the **as_puth** kernel service is called.

If for some reason it is known that the virtual memory object cannot be deleted, the **as_getsrval** kernel service may be used. This kernel service does not require that the **as_puth** kernel service be used. This service can also be called from the interrupt environment.

Execution Environment

The **as_geth** kernel service can be called from the process environment only.

Return Values

The **as_geth** kernel service always succeeds and returns the appropriate handle.

Implementation Specifics

The **as_geth** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **getadsp** kernel service, **as_att** kernel service, **vm_att** kernel service, **as_puth** kernel service, and **as_getsrval** kernel service.

as_geth64 Kernel Service

Purpose

Obtains a handle to the virtual memory object for the specified address.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
#include <sys/adspace.h>

vmhandle_t as_geth64 (addr64)
unsigned   long long addr64;
```

Parameter

addr64	Specifies the virtual memory address for which the corresponding handle should be returned.

Description

The **as_geth64** kernel service is used to obtain a handle to the virtual memory object corresponding to the input address (addr64). This handle can then be used with the **as_att64** or **vm_att** kernel service to make the object addressable at a different location.

After the last use of the handle and after it is detached accordingly, the **as_puth64** kernel service must be used to indicate this fact. Failure to call the **as_puth64** service may result in resources being permanently unavailable for re–use.

If the handle returned refers to a virtual memory segment, then that segment is protected from deletion until the **as_puth64** kernel service is called.

If, for some reason, it is known that the virtual memory object cannot be deleted, then the **as_getsrval64** kernel service may be used instead of the **as_geth64** service.

The **as_geth64** kernel service assumes an address space model of fixed–size virtual memory objects.

This service will operate correctly for both 32–bit and 64–bit user address spaces. It will also work for kernel processes (*kprocs*).

> **Note:** This service only operates on the current process's address space. It is not allowed to operate on another address space.

Execution Environment

The **as_geth64** kernel service can be called from the process environment only.

Return Values

On successful completion, this routine returns the appropriate handle.

On error, this routine returns the value INVLSID defined in **sys/seg.h**. This is caused by an address out of range.

Errors include: Input address out of range.

Implementation Specifics

The **as_geth64** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **as_att64** kernel service, **as_seth64** kernel service, **as_det64** kernel service, **as_getsrval64** kernel service, and **as_puth64** kernel service.

as_getsrval Kernel Service

Purpose

Obtains a handle to the virtual memory object for the specified address given in the specified address space.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
#include <sys/adspace.h>

vmhandle_t as_getsrval (Adspacep, Addr)
adspace_t  *Adspacep;
caddr_t    Addr;
```

Parameters

Adspacep	Points to the address space structure to obtain the virtual memory object handle from. The getadsp kernel service can obtain this pointer.
Addr	Specifies the virtual memory address that should be used to determine the virtual memory object handle for the specified address space.

Description

The **as_getsrval** kernel service is used to obtain a handle to the virtual memory object corresponding to a virtual memory address in a particular address space. This handle can then be used with the **as_att** or **vm_att** kernel services to make the object addressable in another address space.

This should only be used when it is known that the virtual memory object cannot be deleted, otherwise the **as_geth** kernel service must be used.

The **as_puth** kernel service must not be called for handles returned by the **as_getsrval** kernel service.

Execution Environment

The **as_getsrval** kernel service can be called from both the interrupt and the process environments.

Return Values

The **as_getsrval** kernel service always succeeds and returns the appropriate handle.

Implementation Specifics

The **as_getsrval** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **getadsp** kernel service, **as_att** kernel service, **vm_att** kernel service, **as_geth** kernel service, and **as_puth** kernel service.

as_getsrval64 Kernel Service

Purpose

Obtains a handle to the virtual memory object for the specified address.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
#include <sys/adspace.h>

vmhandle_t as_getsrval64 (addr64)
unsigned long long  addr64;
```

Parameters

addr64	Specifies the virtual memory address for which the corresponding handle should be returned.

Description

The **as_getsrval64** kernel service is used to obtain a handle to the virtual memory object corresponding to the input address(addr64). This handle can then be used with the **as_att64** or **vm_att** kernel services to make the object addressable at a different location.

This service should only be used when it is known that the virtual memory object cannot be deleted, otherwise the **as_geth64** kernel service must be used.

The **as_puth64** kernel service must not be called for handles returned by the **as_getsrval64** kernel service.

The **as_getsrval64** kernel service assumes an address space model of fixed–size virtual memory objects.

This service will operate correctly for both 32–bit and 64–bit user address spaces. It will also work for kernel processes (*kprocs*).

Note: This service only operates on the current process's address space. It is not allowed to operate on another address space.

Execution Environment

The **as_getsrval64** kernel service can be called from the process environment only when the current user address space is 64–bits. If the current user address space is 32–bits, or is a *kproc*, then **as_getsrval64** may be called from an interrupt environment.

Return Values

On successful completion this routine returns the appropriate handle.

On error, this routine returns the value INVLSID defined in **sys/seg.h**. This is caused by an address out of range.

Errors include: Input address out of range.

Implementation Specifics

The **as_getsrval64** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **as_att64** kernel service, **as_det64** kernel service, **as_geth64** kernel service, and **as_puth64** kernel service, **as_seth64** kernel service.

as_puth Kernel Service

Purpose

Indicates that no more references will be made to a virtual memory object obtained using the **as_geth** kernel service.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
#include <sys/adspace.h>

void as_puth (Adspacep,Vmhandle)
adspace_t *Adspacep;
vmhandle_t Vmhandle;
```

Parameters

Adspacep	Points to the address space structure that the virtual memory object handle was obtained from. This must be the same address space pointer that is given to the **as_geth** kernel service.
Vmhandle	Describes the virtual memory object that will no longer be referenced. This handle must have been returned by the **as_geth** kernel service.

Description

The **as_puth** kernel service is used to indicate that no more references will be made to the virtual memory object returned by a call to the **as_geth** kernel service. The virtual memory object must be detached from all address spaces it may have been attached to using the **as_att** or **vm_att** kernel services.

Failure to call the **as_puth** kernel service may result in resources being permanently unavailable for re–use.

If for some reason it is known that the virtual memory object cannot be deleted, the **as_getsrval** kernel service may be used instead of the **as_geth** kernel service. This kernel service does not require that the **as_puth** kernel service be used. This service can also be called from the interrupt environment.

Execution Environment

The **as_puth** kernel service can be called from the process environment only.

Return Values

The **as_puth** kernel service always succeeds and returns nothing.

Implementation Specifics

The **as_geth** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **getadsp** kernel service, **as_att** kernel service, **vm_att** kernel service, **as_geth** kernel service, and **as_getsrval** kernel service.

as_puth64 Kernel Service

Purpose

Indicates that no more references will be made to a virtual memory object obtained using the **as_geth64** kernel service.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
#include <sys/adspace.h>

int as_puth64 ( addr64, vmhandle )
unsigned long long addr64;
vmhandle_t vmhandle;
```

Parameters

addr64	Specifies the virtual memory address that the virtual memory object handle was obtained from. This must be the same address that was given to the **as_geth64** kernel service previously.
vmhandle	Describes the virtual memory object that will no longer be referenced. This handle must have been returned by the **as_geth64** kernel service.

Description

The **as_puth64** kernel service is used to indicate that no more references will be made to the virtual memory object returned by a call to the **as_geth64** kernel service. The virtual memory object must be detached from the address space already, using either **as_det64** or **vm_det** service.

Failure to call the **as_puth64** kernel service may result in resources being permanently unavailable for re–use.

If, for some reason, it is known that the virtual memory object cannot be deleted, the **as_getsrval64** kernel service may be used instead of the **as_geth64** kernel service. This kernel service does not require that the **as_puth64** kernel service be used.

The **as_puth64** kernel service assumes an address space model of fixed–size virtual memory objects.

This service will operate correctly for both 32–bit and 64–bit user address spaces. It will also work for kernel processes (*kprocs*).

> **Note:** This service only operates on the current process's address space. It is not allowed to operate on another address space.

Execution Environment

The **as_puth64** kernel service can be called from the process environment only.

Return Values

0	Successful completion.
EINVAL	Input address out of range.

Implementation Specifics

The **as_puth64** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **as_att64** kernel service, **as_det64** kernel service, **as_getsrval64** kernel service, **as_geth64** kernel service, and **as_seth64** kernel service.

as_remap64 Kernel Service

Purpose

Remaps an additional 64–bit address to a 32–bit address that can be used by the kernel.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/adspace.h>

int as_remap64 (addr64, len, addr32 )
unsigned long long addr64;
 unsigned int len;
unsigned int*addr32;
```

Parameters

addr64	Specifies the 64–bit effective address of start of range to be remapped.
len	Specifies the number of bytes in the range to be remapped.
addr32	Specifies the new 32–bit remapped address (filled in by **as_remap64**).

Description

The **as_remap64** service will create a 32–bit remapped address from the 64–bit address and return that to the caller. This service may be called when another address needs to be remapped after the **remap_64** service has already been called in the context of the same system call.

A common example when this may be needed is in a device driver ioctl entry point. If the arg parameter is a 64–bit pointer to a structure, it is remapped by the **__remap** library routine and **remap_64** kernel service before the device driver **ioctl** entry point is called. If the structure itself contains 64–bit pointers, however, the **as_remap64** routine may be used by the device driver to remap these additional pointers.

The **as_remap64** kernel service may be called for either a 32–bit or 64–bit process. If called for a 32–bit process and **addr64** is a valid 32–bit address, then this address is simply returned in the **addr32** parameter.

Execution Environment

The **as_remap64** kernel service can be called from the process environment only.

Return Values

0	Successful completion.
EINVAL	The process is 32–bit, and **addr64** is not a valid 32–bit address (or)
	Unable to remap the address range due to insufficient resources.

Implementation Specifics

The **as_remap64** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **as_unremap64** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

as_seth Kernel Service

Purpose

Maps a specified region in the specified address space for the specified virtual memory object.

Syntax

#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
#include <sys/adspace.h>

void as_seth (*adspacep*, *vmhandle*, *addr***)**
adspace_t ******adspacep*;
vmhandle_t *vmhandle*;
caddr_t *addr*;

Parameters

adspacep	Points to the address space structure that defines the address space where the region for the virtual memory object is to be allocated. The **getadsp** kernel service can obtain this pointer.
vmhandle	Describes the virtual memory object being made addressable within a region of the specified address space.
addr	Specifies the virtual memory address which identifies the region of the specified address space to allocate. On this system, the upper 4 bits of this address are used to determine which region to allocate.

Description

The **as_seth** kernel service:

- Allocates the region within the address space specified by the *adspacep* parameter and the *addr* parameter. Any virtual memory object previously mapped in this region of the address space is unmapped.
- Maps the virtual memory object selected by the vmhandle parameter with the access permission specified in the handle.

The **as_seth** kernel service should only be used when it is necessary to map a virtual memory object at a fixed address within an address space. The **as_att** kernel service should be used when it is not absolutely necessary to map the virtual memory object at a fixed address.

Execution Environment

The **as_seth** kernel service can be called from the process environment only.

Return Values

The **as_seth** kernel service always succeeds and returns nothing.

Implementation Specifics

The **as_seth** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **getadsp** kernel service, **as_att** kernel service, **vm_att** kernel service, **as_geth** kernel service, and **as_getsrval** kernel service.

as_seth64 Kernel Service

Purpose

Maps a specified region for the specified virtual memory object.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
#include <sys/adspace.h>

int as_seth64 (addr64,vmhandle)
unsigned long long addr64;
vmhandle_t vmhandle;
```

Parameters

addr64	The region covering this input virtual memory address will be mapped.
vmhandle	Describes the virtual memory object being made addressable within a region of the address space.

Description

The **as_seth64** kernel service maps the region covering the input **addr64** parameter. Any virtual memory object previously mapped within this region is unmapped.

The virtual memory object specified with the **vmhandle** parameter is then mapped with the access permission specified in the handle.

The **as_seth64** kernel service should only be used when it is necessary to map a virtual memory object at a fixed address. The **as_att64** kernel service should be used when it is not absolutely necessary to map the virtual memory object at a fixed address.

The **as_seth64** kernel service assumes an address space model of fixed–size virtual memory objects.

This service will operate correctly for both 32–bit and 64–bit user address spaces. It will also work for kernel processes (*kprocs*).

Note: This service only operates on the current process's address space. It is not allowed to operate on another address space.

Execution Environment

The **as_seth64** kernel service can be called from the process environment only.

Return Values

0	Successful completion.
EINVAL	Input address out of range.

Implementation Specifics

The **as_seth64** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **as_att64** kernel service, **as_det64** kernel service, **as_getsrval64** kernel service, **as_geth64** kernel service, and **as_puth64** kernel service.

as_unremap64 Kernel Service

Purpose

Returns the 64–bit original or unremapped address associated with a 32–bit remapped address.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/adspace.h>

unsigned long long as_unremap (addr32)
caddr_t  addr32;
```

Parameter

addr32	Specifies the 32–bit remapped address to be converted to the corresponding non–remapped 64–bit address.

Description

The **as_unremap64** service will return the original, non–remapped 64–bit address associated with a given 32–bit remapped address. For a 64–bit process, **as_unremap64** will not check the input 32–bit address to see if it has been remapped or not. It is assumed that the input address is remapped. For a 32–bit process, **as_unremap64** simply casts the 32–bit address to 64 bits.

This kernel service must be called in kernel mode.

Execution Environment

The **as_unremap64** kernel service can be called from the process environment only.

Return Values

The 64–bit non–remapped address corresponding to addr32.

Implementation Specifics

The **as_unremap64** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **remap_64** kernel service, **as_remap64** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

attach Device Queue Management Routine

Purpose

Provides a means for performing device–specific processing when the **attchq** kernel service is called.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/deviceq.h>

int attach (dev_parms, path_id)
caddr_t dev_parms;
cba_id path_id;
```

Parameters

dev_parms	Passed to the **creatd** kernel service when the **attach** routine is defined.
path_id	Specifies the path identifier for the queue being attached to.

Description

Each device queue can have an **attach** routine. This routine is optional and must be specified when the **creatd** kernel service defines the device queue. The **attchq** service calls the **attach** routine each time a new path is created to the owning device queue. The processing performed by this routine is dependent on the server function.

The **attach** routine executes under the process under which the **attchq** kernel service is called. The kernel does not serialize the execution of this service with the execution of any other server routines.

Execution Environment

The **attach–device** routine can be called from the process environment only.

Return Values

RC_GOOD	Indicates a successful completion.
RC_NONE	Indicates that resources such as pinned memory are unavailable.
RC_MAX	Indicates that the server already has the maximum number of users that it supports.
Greater than or equal to **RC_DEVICE**	Indicates device–specific errors.

Implementation Specifics

The **attach** routine is part of the Device Queue Management kernel extension.

audit_svcbcopy Kernel Service

Purpose

Appends event information to the current audit event buffer.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int audit_svcbcopy (buf, len)
char *buf;
int len;
```

Parameters

buf	Specifies the information to append to the current audit event record buffer.
len	Specifies the number of bytes in the buffer.

Description

The **audit_svcbcopy** kernel service appends the specified buffer to the event–specific information for the current switched virtual circuit (SVC). System calls should initialize auditing with the **audit_svcstart** kernel service, which creates a record buffer for the named event.

The **audit_svcbcopy** kernel service can then be used to add additional information to that buffer. This information usually consists of system call parameters passed by reference.

If auditing is enabled, the information is written by the **audit_svcfinis** kernel service after the record buffer is complete.

Execution Environment

The **audit_svcbcopy** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
ENOMEM	Indicates that the kernel service is unable to allocate space for the new buffer.

Implementation Specifics

This kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **audit_svcfinis** kernel service, **audit_svcstart** kernel service.

Security Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

audit_svcfinis Kernel Service

Purpose

Writes an audit record for a kernel service.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/audit.h>

int audit_svcfinis ( )
```

Description

The **audit_svcfinis** kernel service completes an audit record begun earlier by the **audit_svcstart** kernel service and writes it to the kernel audit logger. Any space allocated for the record and associated buffers is freed.

If the system call terminates without calling the **audit_svcfinis** service, the switched virtual circuit (SVC) handler exit routine writes the records. This exit routine calls the **audit_svcfinis** kernel service to complete the records.

Execution Environment

The **audit_svcfinis** kernel service can be called from the process environment only.

Return Values

The **audit_svcfinis** kernel service always returns a value of 0.

Implementation Specifics

This kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **audit_svcbcopy** kernel service, **audit_svcstart** kernel service.

Security Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

audit_svcstart Kernel Service

Purpose

Initiates an audit record for a system call.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/audit.h>

int audit_svcstart (eventnam, eventnum, numargs, arg1, arg2 , ...)
char *eventnam;
int *eventnum;
int numargs;
int arg1;
int arg2;
...
```

Parameters

eventnam	Specifies the name of the event. In the current implementation, event names must be less than 17 characters, including the trailing null character. Longer names are truncated.
eventnum	Specifies the number of the event. This is an internal table index meaningful only to the kernel audit logger. The system call should initialize this parameter to 0. The first time the **audit_svcstart** kernel service is called, this parameter is set to the actual table index. The system call should not reset the parameter. The parameter should be declared a static.
numargs	Specifies the number of parameters to be included in the buffer for this record. These parameters are normally zero or more of the system call parameters, although this is not a requirement.
arg1, *arg2*, *...*	Specifies the parameters to be included in the buffer.

Description

The **audit_svcstart** kernel service initiates auditing for a system call event. It dynamically allocates a buffer to contain event information. The arguments to the system call (which should be specified as parameters to this kernel service) are automatically added to the buffer, as is the internal number of the event. You can use the **audit_svcbcopy** service to add additional information that cannot be passed by value.

The system call commits this record with the **audit_svcfinis** kernel service. The system call should call the **audit_svcfinis** kernel service before calling another system call.

Execution Environment

The **audit_svcstart** kernel service can be called from the process environment only.

Return Values

Nonzero	Indicates that auditing is on for this routine.
0	Indicates that auditing is off for this routine.

Example

```
svccrash(int x, int y, int z)
{
        static int eventnum;
        if (audit_svcstart(”crashed”, &eventnum, 2, x, y))
                {
                audit_svcfinis();
                }
        body of svccrash
}
```

The preceding example allocates an audit event record buffer for the `crashed` event and copies the first and second arguments into it. The third argument is unnecessary and not copied.

Implementation Specifics

This kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **audit_svcbcopy** kernel service, **audit_svcfinis** kernel service.

Security Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

bawrite Kernel Service

Purpose

Writes the specified buffer data without waiting for I/O to complete.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/buf.h>

int bawrite (bp)
struct buf *bp;
```

Parameter

bp	Specifies the address of the buffer structure.

Description

The **bawrite** kernel service sets the asynchronous flag in the specified buffer and calls the **bwrite** kernel service to write the buffer.

For a description of how the three buffer–cache write subroutines work, see ”Block I/O Buffer Cache Services: Overview” in *AIX Kernel Extensions and Device Support Programming Concepts*.

Execution Environment

The **bawrite** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
ERRNO	Returns an error number from the **/usr/include/sys/errno.h** file on error.

Implementation Specifics

The **bawrite** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **bwrite** kernel service.

Block I/O Buffer Cache Kernel Services: Overview and I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

bdwrite Kernel Service

Purpose

Releases the specified buffer after marking it for delayed write.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/buf.h>

void bdwrite (bp)
struct buf  *bp;
```

Parameter

bp	Specifies the address of the buffer structure for the buffer to be written.

Description

The **bdwrite** kernel service marks the specified buffer so that the block is written to the device when the buffer is stolen. The **bdwrite** service marks the specified buffer as delayed write and then releases it (that is, puts the buffer on the free list). When this buffer is reassigned or reclaimed, it is written to the device.

The **bdwrite** service has no return values.

For a description of how the three buffer–cache write subroutines work, see ”Block I/O Buffer Cache Kernel Services: Overview” in *AIX Kernel Extensions and Device Support Programming Concepts*.

Execution Environment

The **bdwrite** kernel service can be called from the process environment only.

Implementation Specifics

The **bdwrite** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **brelse** kernel service.

Block I/O Buffer Cache Kernel Services: Overview and I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

bflush Kernel Service

Purpose

Flushes all write–behind blocks on the specified device from the buffer cache.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/buf.h>

void bflush (dev)
dev_t  dev;
```

Parameter

dev	Specifies which device to flush. A value of **NODEVICE** flushes all devices.

Description

The **bflush** kernel service runs the free list of buffers. It notes as busy or writing any dirty buffer whose block is on the specified device. When a value of **NODEVICE** is specified, the **bflush** service flushes all write–behind blocks for all devices. The **bflush** service has no return values.

Execution Environment

The **bflush** kernel service can be called from the process environment only.

Implementation Specifics

The **bflush** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **bwrite** kernel service.

Block I/O Buffer Cache Kernel Services: Overview and I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

bindprocessor Kernel Service

Purpose

Binds or unbinds kernel threads to a processor.

Syntax

```
#include <sys/processor.h>

int bindprocessor (What, Who, Where)
int What;
int Who;
cpu_t Where;
```

Parameters

What — Specifies whether a process or a kernel thread is being bound to a processor. The *What* parameter can take one of the following values:

BINDPROCESS A process is being bound to a processor.

BINDTHREAD A kernel thread is being bound to a processor.

Who — Indicates a process or kernel thread identifier, as appropriate for the *What* parameter, specifying the process or kernel thread which is to be bound to a processor.

Where — If the *Where* parameter is in the range 0–*n* (where *n* is the number of processors in the system), it represents a logical processor identifier to which the process or kernel thread is to be bound. Otherwise, it represents a processor class, from which a processor will be selected. A value of **PROCESSOR_CLASS_ANY** unbinds the specified process or kernel thread, which will then be able to run on any processor.

Description

The **bindprocessor** kernel service binds a single kernel thread, or all kernel threads in a process, to a processor, forcing the bound threads to be scheduled to run on that processor only. It is important to understand that a process itself is not bound, but rather its kernel threads are bound. Once kernel threads are bound, they are always scheduled to run on the chosen processor, unless they are later unbound. When a new thread is created using the **thread_create** kernel service, it has the same bind properties as its creator.

Return Values

On successful completion, the **bindprocessor** kernel service returns 0. Otherwise, a value of –1 is returned and the error code can be checked by calling the **getuerror** kernel service.

Error Codes

The **bindprocessor** kernel service is unsuccessful if one of the following is true:

EINVAL The *What* parameter is invalid, or the *Where* parameter indicates an invalid processor number or a processor class which is not currently available.

ESRCH The specified process or thread does not exist.

EPERM The caller does not have root user authority, and the *Who* parameter specifies either a process, or a thread belonging to a process, having a real or effective user ID different from that of the calling process.

Execution Environment

The **bindprocessor** kernel service can be called from the process environment only.

Implementation Specifics

The **bindprocessor** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **bindprocessor** command.

The **exec** subroutine, **fork** subroutine, **sysconf** subroutine.

binval Kernel Service

Purpose

Makes nonreclaimable all blocks in the buffer cache of a specified device.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/buf.h>

void binval (dev)
dev_t  dev;
```

Parameter

dev	Specifies the device to be purged.

Description

The **binval** kernel service makes nonreclaimable all blocks in the buffer cache of a specified device. Before removing the device from the system, use the **binval** service to remove the blocks.

All of blocks of the device to be removed need to be flushed before you call the **binval** service. Typically, these blocks are flushed after the last close of the device.

Execution Environment

The **binval** kernel service can be called from the process environment only.

Return Values

The **binval** service has no return values.

Implementation Specifics

The **binval** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **bflush** kernel service, **blkflush** kernel service.

Block I/O Buffer Cache Kernel Services: Overview and I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

blkflush Kernel Service

Purpose

Flushes the specified block if it is in the buffer cache.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/buf.h>

int blkflush (dev, blkno)
dev_t  dev;
daddr_t  blkno;
```

Parameters

dev	Specifies the device containing the block to be flushed.
blkno	Specifies the block to be flushed.

Description

The **blkflush** kernel service checks to see if the specified buffer is in the buffer cache. If the buffer is not in the cache, then the **blkflush** service returns a value of 0. If the buffer is in the cache, but is busy, the **blkflush** service calls the **e_sleep** service to wait until the buffer is no longer in use. Upon waking, the **blkflush** service tries again to access the buffer.

If the buffer is in the cache and is not busy, but is dirty, then it is removed from the free list. The buffer is then marked as busy and synchronously written to the device. If the buffer is in the cache and is neither busy nor dirty (that is, the buffer is already clean and therefore does not need to be flushed), the **blkflush** service returns a value of 0.

Execution Environment

The **blkflush** kernel service can be called from the process environment only.

Return Values

1	Indicates that the block was successfully flushed.
0	Indicates that the block was not flushed. The specified buffer is either not in the buffer cache or is in the buffer cache but neither busy nor dirty.

Implementation Specifics

The **blkflush** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **bwrite** kernel service.

Block I/O Buffer Cache Kernel Services: Overview I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

bread Kernel Service

Purpose

Reads the specified block data into a buffer.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/buf.h>

struct buf *bread (dev, blkno)
dev_t  dev;
daddr_t  blkno;
```

Parameters

dev	Specifies the device containing the block to be read.
blkno	Specifies the block to be read.

Description

The **bread** kernel service assigns a buffer to the given block. If the specified block is already in the buffer cache, then the block buffer header is returned. Otherwise, a free buffer is assigned to the specified block and the data is read into the buffer. The **bread** service waits for I/O to complete to return the buffer header.

The buffer is allocated to the caller and marked as busy.

Execution Environment

The **bread** kernel service can be called from the process environment only.

Return Values

The **bread** service returns the address of the selected buffer's header. A nonzero value for **B_ERROR** in the `b_flags` field of the buffer's header (**buf** structure) indicates an error. If this occurs, the caller should release the buffer associated with the block using the **brelse** kernel service.

Implementation Specifics

The **bread** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **getblk** kernel service, **iowait** kernel service.

Block I/O Buffer Cache Kernel Services: Overview in *AIX Kernel Extensions and Device Support Programming Concepts* describes how the buffer cache services manage the block I/O buffer cache mechanism.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

breada Kernel Service

Purpose

Reads in the specified block and then starts I/O on the read–ahead block.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/buf.h>

struct buf *breada (dev, blkno, rablkno)
dev_t   dev;
daddr_t  blkno;
daddr_t  rablkno;
```

Parameters

dev	Specifies the device containing the block to be read.
blkno	Specifies the block to be read.
rablkno	Specifies the read–ahead block to be read.

Description

The **breada** kernel service assigns a buffer to the given block. If the specified block is already in the buffer cache, then the **bread** service is called to:

- Obtain the block.
- Return the buffer header.

Otherwise, the **getblk** service is called to assign a free buffer to the specified block and to read the data into the buffer. The **breada** service waits for I/O to complete and then returns the buffer header.

I/O is also started on the specified read–ahead block if the free list is not empty and the block is not already in the cache. However, the **breada** service does not wait for I/O to complete on this read–ahead block.

”Block I/O Buffer Cache Kernel Services: Overview” in *AIX Kernel Extensions and Device Support Programming Concepts* summarizes how the **getblk**, **bread**, **breada**, and **brelse** services uniquely manage the block I/O buffer cache.

Execution Environment

The **breada** kernel service can be called from the process environment only.

Return Values

The **breada** service returns the address of the selected buffer’s header. A nonzero value for B_ERROR in the `b_flags` field of the buffer header (**buf** structure) indicates an error. If this occurs, the caller should release the buffer associated with the block using the **brelse** kernel service.

Implementation Specifics

The **breada** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **bread** kernel service, **iowait** kernel service.

The **ddstrategy** device driver entry point.

Block I/O Buffer Cache Kernel Services: Overview and I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

brelse Kernel Service

Purpose

Frees the specified buffer.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/buf.h>

void brelse (bp)
struct buf *bp;
```

Parameter

bp	Specifies the address of the **buf** structure to be freed.

Description

The **brelse** kernel service frees the buffer to which the *bp* parameter points.

The **brelse** kernel service awakens any processes waiting for this buffer or for another free buffer. The buffer is then put on the list of available buffers. The buffer is also marked as not busy so that it can either be reclaimed or reallocated.

The **brelse** service has no return values.

Execution Environment

The **brelse** kernel service can be called from either the process or interrupt environment.

Implementation Specifics

The **brelse** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **geteblk** kernel service.

The **buf** structure.

Block I/O Buffer Cache Kernel Services: Overview and I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

bwrite Kernel Service

Purpose

Writes the specified buffer data.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/buf.h>

int bwrite (bp)
struct buf  *bp;
```

Parameter

bp	Specifies the address of the buffer structure for the buffer to be written.

Description

The **bwrite** kernel service writes the specified buffer data. If this is a synchronous request, the **bwrite** service waits for the I/O to complete.

”Block I/O Buffer Cache Kernel Services: Overview” in *AIX Kernel Extensions and Device Support Programming Concepts* describes how the three buffer–cache write routines work.

Execution Environment

The **bwrite** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
ERRNO	Returns an error number from the **/usr/include/sys/errno.h** file on error.

Implementation Specifics

The **bwrite** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **brelse** kernel service, **iowait** kernel service.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

cancel Device Queue Management Routine

Purpose

Provides a means for cleaning up queue element–related resources when a pending queue element is eliminated from the queue.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/deviceq.h>

void cancel (ptr)
struct req_qe *ptr;
```

Parameter

ptr	Specifies the address of the queue element.

Description

The kernel calls the **cancel** routine to clean up resources associated with a queue element. Each device queue can have a **cancel** routine. This routine is optional and must be specified when the device queue is created with the **creatq** service.

The **cancel** routine is called when a pending queue element is eliminated from the queue. This occurs when the path is destroyed or when the **canclq** service is called. The device manager should unpin any data and detach any cross–memory descriptor.

Any operations started as a result of examining the queue with the **peekq** service must be stopped.

The **cancel** routine is also called when a queue is destroyed to get rid of any pending or active queue elements.

Execution Environment

The **cancel–queue–element** routine can be called from the process environment only.

Implementation Specifics

The **cancel** routine is part of the Device Queue Management Kernel extension.

CardServices Kernel Service

Purpose

Provides the programming interface for device drivers and other kernel routines to access PCMCIA Card Services functions.

Syntax

```
#include <sys/pcmciacs.h>
int CardServices (functionID, *objectID, pointer,
                  arglen, argbuf)

int functionID;
int *objectID;
void* pointer;
int arglen;
void* argbuf;
```

Parameters

funtionID	The functions are the items listed under the description showing the syntax and parameters for each.
objectID	Specifies the that each *functionID* may point to a different object, which is specified in the syntax for each function section.
arglen	Specifies the length of the argument listed again in the syntax section for each function.

Description

The **CardServices** system call passes the Card Services function request, as specified by the *functionID* parameter, to the Card Services kernel services software. Each *functionID* has specific values to be used for *objectID*, *pointer*, *arglen* and *argbuf* parameters as specified in the syntax. Details on each function are provided in the *PCMCIA Card Services Specification* available through the Personal Computer Memory Card International Association. The *argbuf* pointer may point to a structure that is defined in the include file /**usr**/**include**/**sys**/**pcmciacs.h**. The following *functionID*'s along with their syntax are supported:

CSAccessConfigurationRegister

This function allows a client to read or write a PC Card Configuration Register.

```
int CardServices(CSAccessConfigurationRegister, null, null,
    sizeof(CSAccCfgRegPkt), (CSAccCfgRegPkt*) buf);
```

CSDeregisterClient

This function removes a client from the list of registered clients maintained by Card Services.

```
int CardServices(CSDeregisterClient, &clientID, null, 0,
    null);
```

CSGetCardServicesInfo

This function returns the number of logical sockets installed and information about Card Services presence, vendor revision number, and release compliance information.

```
int CardServices(CSGetCardServicesInfo, null, null, MaxLen,
    (CSGetCSInfoPkt *) buf);
```

CSGetClientInfo

This function returns information describing a client. This information is expected to be used by browsing utilities.

```
int CardServices(CSGetClientInfo, &clientID, null, MaxLen,
    (CSGetCliInfoPkt *) buf);
```

CSGetConfigurationInfo

This function returns information about the specified socket and PC Card configuration.

```
int CardServices(CSGetConfigurationInfo, &clientID, null,
    sizeof(CSGetCfgInfoPkt), (CSGetCfgInfoPkt *) buf);
```

CSGetEventMask

This function returns the event mask for the client.

```
int CardServices(CSGetEventMask, &clientID, null,
    sizeof(CSGetEvMaskPkt), (CSGetEvMaskPkt *) buf);
```

CSGetEventMask

This function returns the event mask for the client.

```
int CardServices(CSGetEventMask, &clientID, null,
    sizeof(CSGetEvMaskPkt), (CSGetEvMaskPkt *) buf);
```

CSGetFirstClient

This function returns the first ClientHandle of the clients that have registered with Card Services.

```
int CardServices(CSGetFirstClient, &clientID, null,
    sizeof(CSGetClientPkt), (CSGetClientPkt *) buf);
```

CSGetFirstTuple

This function returns the first tuple of the specified type in the CIS for the specified socket. If there are no tuples, the Status argument is set to **CSR_NO_MORE_ITEMS**.

```
int CardServices(CSGetFirstTuple, null, null,
    sizeof(CSGetTuplePkt), (CSGetTuplePkt *) buf);
```

CSGetNextClient

This function returns the ClientHandle for the next registered client. The ClientHandle previously returned by **GetFirstClient** or **GetNextClient** is passed as an argument.

```
int CardServices(CSGetNextClient, &clientID, null,
    sizeof(CSGetClientPkt), (CSGetClientPkt *) buf);
```

CSGetNextTuple

This function returns the next tuple of the specified type in the CIS for the specified socket.

```
int CardServices(CSGetNextTuple, null, null,
    sizeof(CSGetTuplePkt), (CSGetTuplePkt *) buf);
```

CSGetStatus

This function returns the current status of a PC Card and its socket.

```
int CardServices(CSGetStatus, null, null,
    sizeof(CSGetStatPkt), (CSGetStatPkt *) buf);
```

CSGetTupleData

This function returns the content of the last tuple returned by **GetFirstTuple** or **GetNextTuple**.

```
int CardServices(CSGetTupleData, null, null, MaxPktSize,
    (CSGetTplDataPkt *) buf);
```

CSMapLogSocket

This function maps a Card Services logical socket to its Socket Services physical adapter and socket values. The *PhyAdapter* should be the device number for the PCMCIA bus.

```
int CardServices(CSMapLogSocket, null, null,
    sizeof(CSMapSocketPkt), (CSMapSocketPkt *) buf);
```

CSMapLogWindow

This function maps a Card Services WindowHandle passed in the Handle argument to its Socket Services physical adapter and window. The *PhyAdapter* should be the device number for the PCMCIA bus.

```
int CardServices(CSMapLogWindow, &windowID, null,
    sizeof(CSMapWindowPkt), (CSMapWindowPkt *) buf);
```

CSMapMemPage

This function selects the memory area on a PC Card into a page of a window allocated with the **RequestWindow** function.

```
int CardServices(CSMapMemPage, &windowID, null,
    sizeof(CSMapMapMemPagePkt), (CSMapMapMemPagePkt *)
    buf);
```

CSMapPhySocket

This function maps Socket Services physical adapter and socket values to a Card Services logical socket. The *PhyAdapter* should be the device number for the PCMCIA bus.

```
int CardServices(CSMapPhySocket, null, null,
    izeof(CSMapSocketPkt), (CSMapSocketPkt *) buf);
```

CSMapPhyWindow

This function maps Socket Services physical adapter and window values to a Card Services logical WindowHandle. The *PhyAdapter* should be the device number for the PCMCIA bus.

```
int CardServices(CSMapPhyWindow, &windowID, null,
    sizeof(CSMapWindowPkt), (CSMapWindowPkt *) buf);
```

CSModifyConfiguration

This function allows a socket and PC Card configuration to be modified without a pair of **ReleaseConfiguration** and **RequestConfiguration** functions.

```
int CardServices(CSModifyConfiguration, &clientID, null,
    sizeof(CSModCfgPkt), (CSModCfgPkt *) buf);
```

CSModifyWindow

This function modifies the attributes, or access speed of a window previously allocated with the **RequestWindow** function.

```
int CardServices(CSModifyWindow, &windowID, null,
    sizeof(CSModWinPkt), (CSModWinPkt *) buf);
```

CSRegisterClient

This function registers a client with Card Services. ClientData will not be used by Card Services, but callback is always called with ClientData value. If client device driver can be multiplexed by multiple devices, the data area pointed to by ClientData should have data that can identify the device.

```
int CardServices(CSRegisterClient, &clientID,
    clientCallback, sizeof(CSRegCliPkt), (CSRegCliPkt *)
    buf);
```

CSReleaseConfiguration

This function returns a PC Card and socket to a simple memory only to interface the zero configurations. Card Services may remove power from the socket if clients have not indicated their usage of the socket by a **RequestWindow** function. Card Services will not reset the PC Card.

```
int CardServices(CSReleaseConfiguration, &clientID, null,
    sizeof(CSRelCfgPkt), (CSRelCfgPkt *) buf);
```

CSReleaseExclusive

This function releases the exclusive use of a card in a socket for a client.

```
int CardServices(CSReleaseExclusive, &clientID, null,
    sizeof(CSRelExclPkt), (CSRelExclPkt *) buf);
```

CSReleaseIO

This function releases previously requested I/O addresses.

```
int CardServices(CSReleaseIO, &clientID, null,
    sizeof(CSIOPkt), (CSIOPkt *) buf);
```

CSReleaseIRQ

This function releases a previously requested interrupt request line.

```
int CardServices(CSReleaseIRQ, &clientID, null,
    sizeof(CSRelIRQPkt), (CSRelIRQPkt *) buf);
```

CSReleaseSocketMask

This function requests that the client no longer be notified of status changes for this socket.

```
int CardServices(CSReleaseSocketMask, &clientID, null,
    sizeof(CSRelSockMPkt), (CSRelSockMPkt *) buf);
```

CSReleaseWindow

This function releases a block of system memory space which is obtained previously by a corresponding **RequestWindow**.

```
int CardServices(CSReleaseWindow, &windowID, null, 0,
    null);
```

CSRequestConfiguration

This function configures the PC Card and socket.

```
int CardServices(CSRequestConfiguration, &clientID, null,
    sizeof(CSReqCfgPkt), (CSReqCfgPkt *) buf);
```

CSRequestExclusive

This function requests the exclusive use of a PC Card in a socket for a client.

```
int CardServices(CSRequestExclusive, &clientID, null,
    sizeof(CSRelExclPkt), (CSRelExclPkt *) buf);
```

CSRequestIO

This function requests I/O addresses for a socket.

```
int CardServices(CSRequestIO, &clientID, null,
    sizeof(CSIOPkt), (CSIOPkt *) buf);
```

CSRequestIRQ

This function requests an interrupt request line.

```
int CardServices(CSRequestIRQ, &clientID, null,
    sizeof(CSReqIRQPkt), (CSReqIRQPkt *) buf)
```

CSRequestSocketMask

This function requests that the client be notified of status changes for this socket.

```
int CardServices(CSRequestSocketMask, &clientID, null,
    sizeof(CSReqSockMPkt), (CSReqSockMPkt *) buf);
```

CSRequestWindow

This function requests a block of system memory space be assigned to a memory region of a PC Card in a socket. The *objectID* means that the input value should be *clientID* and the output should be *windowID*.

```
int CardServices(CSRequestWindow, &objectID, null,
    sizeof(CSReqWinPkt), (CSReqWinPkt *) buf);
```

CSResetCard

This function resets the PC Card in the specified socket.

```
int CardServices(CSResetCard, &clientID, null,
    sizeof(CSRstCardPkt), (CSRstCardPkt *) buf);
```

CSSetEventMask

This function sets the event mask for the client.

```
int CardServices(CSSetEventMask, &clientID, null,
    sizeof(CSGetEvMaskPkt), (CSGetEvMaskPkt *) buf);
```

CSValidateCIS

This function validates the Card Information Structure on the PC Card in the specified socket.

```
int CardServices(CSValidateCIS, &clientID, null,
    sizeof(CSValCISPkt), (CSValCISPkt *) buf);
```

Event Codes

CSE_PM_RESUME

CSE_PM_SUSPEND

CSE_BATTERY_DEAD

CSE_BATTERY_LOW

CSE_CARD_INSERTION

CSE_CARD_LOCK

CSE_CARD_READY

CSE_CARD_REMOVAL

CSE_CARD_RESET

CSE_CARD_UNLOCK

CSE_EJECTION_COMPLETE

CSE_EJECTION_REQUEST

CSE_ERASE_COMPLETE

CSE_EXCLUSIVE_COMPLETE

CSE_EXCLUSIVE_REQUEST

CSE_INSERTION_COMPLETE

CSE_INSERTION_REQUEST

CSE_REGISTRATION_COMPLETE

CSE_RESET_COMPLETE

CSE_RESET_PHYSICAL

CSE_RESET_REQUEST

CSE_MTD_REQUEST

CSE_CLIENT_INFO

CSE_TIMER_EXPIRED

CSE_SS_UPDATE

CSE_WRITE_PROTECT

Return Values

CSR_SUCCESS Specifies a successful completion.

Error Codes

If the CardServices does not complete successfully, one of the following error codes will be returned:

CSR_BAD_ADAPTER	The specified adapter is invalid.
CSR_BAD_ATTRIBUTE	Value specified for attributes field is invalid.
CSR_BAD_BASE	Specified base system memory address is invalid.
CSR_BAD_EDC	Specified EDC generator is invalid.
CSR_BAD_IRQ	Specified IRQ level is invalid.
CSR_BAD_OFFSET	Specified PC Card memory array offset is invalid.
CSR_BAD_PAGE	Specified page is invalid.
CSR_READ_FAILURE	Unable to complete the read request.
CSR_BAD_SIZE	Specified size is invalid.
CSR_BAD_SOCKET	Specified socket is invalid (logical or physical).
CSR_BAD_TYPE	Window or interface type specified is invalid.
CSR_BAD_VCC	Specified Vcc power level index is invalid
CSR_BAD_VPP	Specified Vpp1 or Vpp2 power level index is invalid.
CSR_BAD_WINDOW	Specified window is invalid.
CSR_WRITE_FAILURE	Unable to complete write request.
CSR_NO_CARD	No PC Card in socket.
CSR_UNSUPPORTED_FUNCTION	Implementation does not support function.
CSR_UNSUPPORTED_MODE	Processor mode is not supported.
CSR_BAD_SPEED	Specified speed is unavailable.
CSR_BUSY	Unable to process request at this time – retry later.
CSR_GENERAL_FAILURE	An undefined error has occurred.
CSR_WRITE_PROTECTED	Media is write–protected.
CSR_BAD_ARG_LENGTH	*arglen* argument is invalid.
CSR_BAD_ARGS	Values in Argument Packet are invalid.
CSR_CONFIGURATION_LOCKED	A configuration is locked.
CSR_IN_USE	Requested resource is being used by a client.
CSR_NO_MORE_ITEMS	There are no more of the requested item.
CSR_OUT_OF_RESOURCE	Card Services has exhausted resource.
CSR_BAD_HANDLE	ClientHandle is invalid.

cfgnadd Kernel Service

Purpose

Registers a notification routine to be called when system–configurable variables are changed.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sysconfig.h>

void cfgnadd
(cbp)
struct cfgncb *cbp;
```

Parameter

cbp	Points to a **cfgncb** configuration notification control block.

Description

The **cfgnadd** kernel service adds a **cfgncb** control block to the list of **cfgncb** structures that the kernel maintains. A **cfgncb** control block contains the address of a notification routine (in its `cfgncb.func` field) to be called when a configurable variable is being changed.

The **SYS_SETPARMS sysconfig** operation allows a user with sufficient authority to change the values of configurable system parameters. The **cfgnadd** service allows kernel routines and extensions to register the notification routine that is called whenever these configurable system variables have been changed.

This notification routine is called in a two–pass process. The first pass performs validity checks on the proposed changes to the system parameters. During the second pass invocation, the notification routine performs whatever processing is needed to make these changes to the parameters. This two–pass procedure ensures that variables used by more than one kernel extension are correctly handled.

To use the **cfgnadd** service, the caller must define a **cfgncb** control block using the structure found in the **/usr/include/sys/sysconfig.h** file.

Execution Environment

The **cfgnadd** kernel service can be called from the process environment only.

The **cfgncb.func** notification routine is called in a process environment only.

Implementation Specifics

The **cfgnadd** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **sysconfig** subroutine.

The **cfgncb** configuration notification control block.

The **cfgndel** kernel service.

Kernel Extension and Device Driver Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

cfgncb Configuration Notification Control Block

Purpose

Contains the address of a notification routine that is invoked each time the **sysconfig** subroutine is called with the **SYS_SETPARMS** command.

Syntax

```
int func (cmd,
cur, new)
int cmd;
struct var *cur;
struct var *new;
```

Parameters

cmd	Indicates the current operation type. Possible values are **CFGV_PREPARE** and **CFGV_COMMIT**, as defined in the **/usr/include/sys/sysconfig.h** file.
cur	Points to a **var** structure representing the current values of system–configurable variables.
new	Points to a **var** structure representing the new or proposed values of system–configurable variables.

The *cur* and *new* **var** structures are both in the system address space.

Description

The configuration notification control block contains the address of a notification routine. This structure is intended to be used as a list element in a list of similar control blocks maintained by the kernel.

Each control block has the following definition:

```
struct cfgncb {
   struct cfgncb     *cbnext;      /* next block on chain
*/
   struct cfgncb     *cbprev;      /* prev control block on chain
*/
   int     (*func)();              /* notification function
*/
   };
```

The **cfgndel** or **cfgnadd** kernel service can be used to add or delete a **cfgncb** control block from the **cfgncb** list. To use either of these kernel services, the calling routine must define the **cfgncb** control block. This definition can be done using the **/usr/include/sys/sysconfig.h** file.

Every time a **SYS_SETPARMS sysconfig** command is issued, the **sysconfig** subroutine iterates through the kernel list of **cfgncb** blocks, invoking each notification routine with a **CFGV_PREPARE** command. This call represents the first pass of what is for the notification routine a two–pass process.

On a **CFGV_PREPARE** command, the **cfgncb.func** notification routine should determine if any values of interest have changed. All changed values should be checked for validity. If the values are valid, a return code of 0 should be returned. Otherwise, a return value indicating the byte offset of the first field in error in the *new* **var** structure should be returned.

If all registered notification routines create a return code of 0, then no value errors have been detected during validity checking. In this case, the **sysconfig** subroutine issues its second pass call to the **cfgncb.func** routine and sends the same parameters, although the

cmd parameter contains a value of **CFGV_COMMIT**. This indicates that the new values go into effect at the earliest opportunity.

An example of notification routine processing might be the following. Suppose the user wishes to increase the size of the block I/O buffer cache. On a **CFGV_PREPARE** command, the block I/O notification routine would verify that the proposed new size for the cache is legal. On a **CFGV_COMMIT** command, the notification routine would then make the additional buffers available to the user by chaining more buffers onto the existing list of buffers.

Implementation Specifics

The **cfgncb** control block is part of Base Operating System (BOS) Runtime.

Related Information

The **cfgnadd** kernel service, **cfgndel** kernel service.

The **SYS_SETPARMS** sysconfig operation.

Kernel Extension and Device Driver Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

cfgndel Kernel Service

Purpose

Removes a notification routine for receiving broadcasts of changes to configurable system variables.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sysconfig.h>

void cfgndel(cbp)
 struct cfgncb *cbp;
```

Parameter

cbp	Points to a **cfgncb** configuration notification control block.

Description

The **cfgndel** kernel service removes a previously registered **cfgncb** configuration notification control block from the list of **cfgncb** structures maintained by the kernel. This service thus allows kernel routines and extensions to remove their notification routines from the list of those called when a configurable system variable has been changed.

The address of the **cfgncb** structure passed to the **cfgndel** kernel service must be the same address used to call the **cfgnadd** service when the structure was originally added to the list. The **/usr/include/sys/sysconfig.h** file contains a definition of the **cfgncb** structure.

Execution Environment

The **cfgndel** kernel service can be called from the process environment only.

Return Values

The **cfgndel** service has no return values.

Implementation Specifics

The **cfgndel** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **sysconfig** subroutine.

The **cfgncb** configuration notification control block.

The **cfgnadd** kernel service.

Kernel Extension and Device Driver Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

check Device Queue Management Routine

Purpose

Provides a means for performing device–specific validity checking for parameters included in request queue elements.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/deviceq.h>

int check (type, ptr, length)
int type;
struct req_qe *ptr;
int length;
```

Parameters

type Specifies the type of call. The following values are used when the kernel calls the **check** routine:

CHECK_PARMS + SEND_CMD
Send command queue element.

CHECK_PARMS + START_IO
Start I/O CCB queue element.

CHECK_PARMS + GEN_PURPOSE
General purpose queue element.

ptr Specifies the address of the queue element.

length Specifies the length of the queue element.

Description

Each device queue can have a **check** routine. This routine is optional and must be specified when the device queue is created with the **creatq** service. The **enque** service calls the **check** routine before a request queue element is put on the device queue. The kernel uses the routine's return value to determine whether to put the queue element on the device queue or to stop the request.

The kernel does not call the **check** routine when an acknowledgment or control queue element is sent. Therefore, the **check** routine is only called while executing within a process.

The address of the actual queue element is passed to this routine. In the **check** routine, take care to alter only the fields that were meant to be altered. This routine does not need to be serialized with the rest of the server's routines, because it is only checking the parameters in the queue element.

The **check** routine can check the request before the request queue element is placed on the device queue. The advantage of using this routine is that you can filter out unacceptable commands before they are put on the device queue.

The routine looks at the queue element and returns **RC_GOOD** if the request is acceptable. If the return code is not **RC_GOOD**, the kernel does not place the queue element in a device queue.

Execution Environment

The **check** routine executes under the process environment of the requester. Therefore, access to data areas must be handled as if the routine were in an interrupt handler

environment. There is, however, no requirement to pin the code and data as in a normal interrupt handler environment.

Return Values

RC_GOOD Indicates successful completion.

All other return values are device–specific.

Implementation Specifics

The **check** routine is part of the Device Queue Management Kernel extension.

Related Information

The **enque** kernel service.

clrbuf Kernel Service

Purpose

Sets the memory for the specified buffer structure's buffer to all zeros.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

void clrbuf (bp)
struct buf   *bp;
```

Parameter

bp	Specifies the address of the buffer structure for the buffer to be cleared.

Description

The **clrbuf** kernel service clears the buffer associated with the specified buffer structure. The **clrbuf** service does this by setting to 0 the memory for the buffer that contains the specified buffer structure.

Execution Environment

The **clrbuf** kernel service can be called from either the process or interrupt environment.

Return Values

The **clrbuf** service has no return values.

Implementation Specifics

The **clrbuf** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Block I/O Buffer Cache Kernel Services: Overview and I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

clrjmpx Kernel Service

Purpose

Removes a saved context by popping the last saved jump buffer from the list of saved contexts.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

void clrjmpx (jump_buffer)
label_t *jump_buffer;
```

Parameter

jump_buffer	Specifies the address of the caller–supplied jump buffer that was specified on the call to the **setjmpx** service.

Description

The **clrjmpx** kernel service pops the most recent context saved by a call to the **setjmpx** kernel service. Since each **longjmpx** call automatically pops the jump buffer for the context to resume, the **clrjmpx** kernel service should be called only following:

- A normal return from the **setjmpx** service when the saved context is no longer needed
- Any code to be run that requires the saved context to be correct

The **clrjmpx** service takes the address of the jump buffer passed in the corresponding **setjmpx** service.

Execution Environment

The **clrjmpx** kernel service can be called from either the process or interrupt environment.

Return Values

The **clrjmpx** service has no return values.

Implementation Specifics

The **clrjmpx** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **longjmpx** kernel service, **setjmpx** kernel service.

Process and Exception Management Kernel Services and Understanding Exception Handling in *AIX Kernel Extensions and Device Support Programming Concepts*.

common_reclock Kernel Service

Purpose

Implements a generic interface to the record locking functions.

Syntax

```
#include <sys/types.h>
#include <sys/flock.h>

common_reclock( gp, size, offset,
lckdat, cmd, retray_fcn, retry_id, lock_fcn,
rele_fcn)
struct gnode *gp;
offset_t size;
offset_t offset;
struct eflock *lckdat;
int cmd;
int (*retry_fcn)();
ulong *retry_id;
int (*lock_fcn)();
int (*rele_fcn)();
```

Parameters

gp Points to the gnode that represents the file to lock.

size Identifies the current size of the file in bytes.

offset Specifies the current file offset. The system uses the *offset* parameter to establish where the lock region is to begin.

lckdat Points to an **eflock** structure that describes the lock operation to perform.

cmd Defines the type of operation the kernel service performs. This parameter is a bit mask consisting of the following bits:

SETFLCK If set, the system sets or clears a lock. If not set, the lock information is returned.

SLPFLCK If the lock cannot be granted immediately, wait for it. This is only valid when **SETFLCK** flag is set.

INOFLCK The caller is holding a lock on the object referred to by the gnode. The **common_reclock** kernel service calls the release function before sleeping, and the lock function on return from sleep.

When the *cmd* parameter is set to **SLPFLCK**, it indicates that if the lock cannot be granted immediately, the service should wait for it. If the *retry_fcn* parameter contains a valid pointer, the **common_reclock** kernel service does not sleep, regardless of the **SLPFLCK** flag.

retry_fcn Points to a retry function. This function is called when the lock is retried. The retry function is not used if the lock is granted immediately. When the requested lock is blocked by an existing lock, a sleeping lock is established with the retry function address stored in it. The **common_reclock** kernel service then returns a correlating ID (see the *retry_id* parameter) to the calling routine, along with an exit value of **EAGAIN**. When the sleeping lock is awakened, the retry function is called with the correlating ID as its ID argument.

If this argument is not NULL, then the **common_reclock** kernel service does not sleep, regardless of the **SLPFLCK** command flag.

retry_id Points to location to store the correlating ID. This ID is used to correlate a retry operation with a specific lock or set of locks. This parameter is used only in conjunction with retry function. The value stored in this location is an opaque value. The caller should not use this value for any purpose other than lock correlation.

lock_fcn Points to a lock function. This function is invoked by the **common_reclock** kernel service to lock a data structure used by the caller. Typically this is the data structure containing the gnode to lock. This function is necessary to serialize access to the object to lock. When the **common_reclock** kernel service invokes the lock function, it is passed the private data pointer from the gnode as its only argument.

rele_fcn Points to a release function. This function releases the lock acquired with the lock function. When the release function is invoked, it is passed the private data pointer from the gnode as its only argument.

Description

The **common_reclock** routine implements a generic interface to the record–locking functions. This service allows distributed file systems to use byte–range locking. The kernel service does the following when a requested lock is blocked by an existing lock:

- Establishes a sleeping lock with the retry function in the **lock** structure. The address of the retry function is specified by the *retry_fcn* parameter.
- Returns a correlating ID value to the caller along with an exit value of **EAGAIN**. The ID is stored in the *retry_id* parameter.
- Calls the retry function when the sleeping lock is later awakened, the retry function is called with the *retry_id* parameter as its argument.

Note: Before a call to the **common_reclock** subroutine, the **eflock** structure must be completely filled in. The *lckdat* parameter points to the **eflock** structure.

The caller can hold a serialization lock on the data object pointed to by the gnode. However, if the caller expects to sleep for a blocking–file lock and is holding the object lock, the caller must specify a lock function with the *lock_fcn* parameter and a release function with the *rele_fcn* parameter.

The lock is described by a **eflock** structure. This structure is identified by the *lckdat* parameter. If a read lock (**F_RDLCK**) or write lock (**F_WRLCK**) is set with a length of 0, the entire file is locked. Similarly, if unlock (**F_UNLCK**) is set starting at 0 for 0 length, all locks on this file are unlocked. This method is how locks are removed when a file is closed.

To allow the **common_reclock** kernel service to update the per–gnode lock list, the service takes a **GN_RECLK_LOCK** lock during processing.

Execution Environment

The **common_reclock** kernel service can be called from the process environment only.

Return Values

0 Indicates successful completion.

EAGAIN Indicates a lock cannot be granted because of a blocking lock and the caller did not request that the operation sleep.

ERRNO Indicates an error. Refer to the **fcntl** system call for the list of possible values.

Implementation Specifics

This kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **fcntl** subroutine.

The **flock.h** file.

compare_and_swap Kernel Service

Purpose

Conditionally updates or returns a single word variable atomically.

Syntax

```
#include <sys/atomic_op.h>

boolean_t compare_and_swap (word_addr, old_val_addr, new_val)
atomic_p word_addr;
int *old_val_addr;
int new_val;
```

Parameters

word_addr	Specifies the address of the single word variable.
old_val_addr	Specifies the address of the old value to be checked against (and conditionally updated with) the value of the single word variable.
new_val	Specifies the new value to be conditionally assigned to the single word variable.

Description

The **compare_and_swap** kernel service performs an atomic (uninterruptible) operation which compares the contents of a single word variable with a stored old value; if equal, a new value is stored in the single word variable, and **TRUE** is returned, otherwise the old value is set to the current value of the single word variable, and **FALSE** is returned.

The **compare_and_swap** kernel service is particularly useful in operations on singly linked lists, where a list pointer must not be updated if it has been changed by another thread since it was read.

Note: The word variable must be aligned on a full word boundary.

Execution Environment

The **compare_and_swap** kernel service can be called from either the process or interrupt environment.

Return Values

TRUE	Indicates that the single word variable was equal to the old value, and has been set to the new value.
FALSE	Indicates that the single word variable was not equal to the old value, and that its current value has been returned in the location where the old value was stored.

Implementation Specifics

The **compare_and_swap** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **fetch_and_add** kernel service, **fetch_and_and** kernel service, **fetch_and_or** kernel service.

Locking Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*

copyin Kernel Service

Purpose

Copies data between user and kernel memory.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int copyin (uaddr, kaddr, count)
char *uaddr;
char *kaddr;
int count;
```

Parameters

uaddr	Specifies the address of user data.
kaddr	Specifies the address of kernel data.
count	Specifies the number of bytes to copy.

Description

The **copyin** kernel service copies the specified number of bytes from user memory to kernel memory. This service is provided so that system calls and device driver top half routines can safely access user data. The **copyin** service ensures that the user has the appropriate authority to access the data. It also provides recovery from paging I/O errors that would otherwise cause the system to crash.

The **copyin** service should be called only while executing in kernel mode in the user process.

Execution Environment

The **copyin** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
EFAULT	Indicates that the user has insufficient authority to access the data, or the address specified in the *uaddr* parameter is not valid.
EIO	Indicates that a permanent I/O error occurred while referencing data.
ENOMEM	Indicates insufficient memory for the required paging operation.
ENOSPC	Indicates insufficient file system or paging space.

Implementation Specifics

The **copyin** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Accessing User–Mode Data While in Kernel Mode and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

The **copyinstr** kernel service, **copyout** kernel service.

copyin64 Kernel Service

Purpose

Copies data between user and kernel memory.

Syntax

```
#include <sys/types.h>
#include <sys/ernno.h>
#include <sys/uio.h>

int copyin64 (uaddr64, kaddr, count);
unsigned long long uaddr64;
char * kaddr;
int count;
```

Parameters

uaddr64	Specifies the address of user data.
kaddr	Specifies the address of kernel data.
count	Specifies the number of bytes to copy.

Description

The **copyin64** kernel service copies the specified number of bytes from user memory to kernel memory. This service is provided so that system calls and device driver top half routines can safely access user data. The **copyin64** service ensures that the user has the appropriate authority to access the data. It also provides recovery from paging I/O errors that would otherwise cause the system to crash.

This service will operate correctly for both 32–bit and 64–bit user address spaces. The *uaddr64* parameter is interpreted as being a non–remapped 32–bit address for the case where the current user address space is 32– bits. If the current user address space is 64–bits, then **uaddr64** is treated as a 64–bit address.

The **copyin64** service should be called only while executing in kernel mode in the user process.

Execution Environment

The **copyin64** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
EFAULT	Indicates that the user has insufficient authority to access the data, or the address specified in the *uaddr64* parameter is not valid.
EIO	Indicates that a permanent I/O error occurred while referencing data.
ENOMEM	Indicates insufficient memory for the required paging operation.
ENOSPC	Indicates insufficient file system or paging space.

Implementation Specifics

The **copyin64** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **copyinstr64** kernel service and **copyout64** kernel service.

Accessing User–Mode Data While in Kernel Mode and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

copyinstr Kernel Service

Purpose

Copies a character string (including the terminating null character) from user to kernel space.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int copyinstr
(from, to, max, actual)
caddt_t from;
caddt_t to;
uint max;
uint *actual;
```

Parameters

from	Specifies the address of the character string to copy.
to	Specifies the address to which the character string is to be copied.
max	Specifies the number of characters to be copied.
actual	Specifies a parameter, passed by reference, that is updated by the **copyinstr** service with the actual number of characters copied.

Description

The **copyinstr** kernel service permits a user to copy character data from one location to another. The source location must be in user space or can be in kernel space if the caller is a kernel process. The destination is in kernel space.

Execution Environment

The **copyinstr** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
E2BIG	Indicates insufficient space to complete the copy.
EIO	Indicates that a permanent I/O error occurred while referencing data.
ENOSPC	Indicates insufficient file system or paging space.
EFAULT	Indicates that the user has insufficient authority to access the data or the address specified in the *uaddr* parameter is not valid.

Implementation Specifics

The **copyinstr** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Accessing User–Mode Data While in Kernel Mode and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

copyinstr64 Kernel Service

Purpose

Copies data between user and kernel memory.

Syntax

```
#include <sys/types.h>
#include <sys/ernno.h>
#include <sys/uio.h>
int copyinstr64 (from64, to, max, actual);
unsigned long long from64;
caddr_t  to;
uint max;
uint * actual;
```

Parameters

from64	Specifies the address of character string to copy.
to	Specifies the address to which the character string is to be copied.
max	Specifies the number of characters to be copied.
actual	Specifies a parameter, passed by reference, that is updated by the **copyinstr64** service with the actual number of characters copied.

Description

The **copyinstr64** service permits a user to copy character data from one location to another. The source location must be in user space or can be in kernel space if the caller is a kernel process. The destination is in kernel space.

This service will operate correctly for both 32–bit and 64–bit user address spaces. The *from64* parameter is interpreted as being a non–remapped 32–bit address for the case where the current user address space is 32– bits. If the current user address space is 64–bits, then **from64** is treated as a 64–bit address.

Execution Environment

The **copyinstr64** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
E2BIG	Indicates insufficient space to complete the copy.
EIO	Indicates that a permanent I/O error occurred while referencing data.
ENOSPC	Indicates insufficient file system or paging space.
EFAULT	Indicates that the user has insufficient authority to access the data, or the address specified in the *from64* parameter is not valid.

Implementation Specifics

The **copyinstr64** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **copyinstr64** kernel service and **copyout64** kernel service.

Accessing User–Mode Data While in Kernel Mode and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

copyout Kernel Service

Purpose

Copies data between user and kernel memory.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int copyout (kaddr, uaddr, count)
char *kaddr;
char *uaddr;
int count;
```

Parameters

kaddr	Specifies the address of kernel data.
uaddr	Specifies the address of user data.
count	Specifies the number of bytes to copy.

Description

The **copyout** service copies the specified number of bytes from kernel memory to user memory. It is provided so that system calls and device driver top half routines can safely access user data. The **copyout** service ensures that the user has the appropriate authority to access the data. This service also provides recovery from paging I/O errors that would otherwise cause the system to crash.

The **copyout** service should be called only while executing in kernel mode in the user process.

Execution Environment

The **copyout** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
EFAULT	Indicates that the user has insufficient authority to access the data or the address specified in the *uaddr* parameter is not valid.
EIO	Indicates that a permanent I/O error occurred while referencing data.
ENOMEM	Indicates insufficient memory for the required paging operation.
ENOSPC	Indicates insufficient file system or paging space.

Implementation Specifics

The **copyout** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **copyin** kernel service, **copyinstr** kernel service.

Accessing User–Mode Data While in Kernel Mode and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

copyout64 Kernel Service

Purpose

Copies data between user and kernel memory.

Syntax

```
#include <sys/types.h>
#include <sys/ernno.h>
#include <sys/uio.h>

int copyout64 (kaddr, uaddr64, count);
char * kaddr;
unsigned long long uaddr64;
int count;
```

Parameters

kaddr	Specifies the address of kernel data.
uaddr64	Specifies the address of user data.
count	Specifies the number of bytes to copy.

Description

The **copyout64** service copies the specified number of bytes from kernel memory to user memory. It is provided so that system calls and device driver top half routines can safely access user data. The **copyout64** service ensures that the user has the appropriate authority to access the data. This service also provides recovery from paging I/O errors that would otherwise cause the system to crash.

This service will operate correctly for both 32–bit and 64–bit user address spaces. The *uaddr64* parameter is interpreted as being a non–remapped 32–bit address for the case where the current user address space is 32– bits. If the current user address space is 64–bits, then **uaddr64** is treated as a 64–bit address.

The **copyout64** service should be called only while executing in kernel mode in the user process.

Execution Environment

The **copyout64** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
EFAULT	Indicates that the user has insufficient authority to access the data, or the address specified in the *uaddr64* parameter is not valid.
EIO	Indicates that a permanent I/O error occurred while referencing data.
ENOMEM	Indicates insufficient memory for the required paging operation.
ENOSPC	Indicates insufficient file system or paging space.

Implementation Specifics

The **copyout64** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **copyinstr64** kernel service and **copyin64** kernel service.

Accessing User–Mode Data While in Kernel Mode and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

creatp Kernel Service

Purpose

Creates a new kernel process.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

pid_t creatp()
```

Description

The **creatp** kernel service creates a kernel process. It also allocates and initializes a process block for the new process. Initialization involves these three tasks:

- Assigning an identifier to the kernel process.
- Setting the process state to idle.
- Initializing its parent, child, and sibling relationships.

”Using Kernel Processes” in *AIX Kernel Extensions and Device Support Programming Concepts* has a more detailed discussion of how the **creatp** kernel service creates and initializes kernel processes.

The process calling the **creatp** service must subsequently call the **initp** kernel service to complete the process initialization. The **initp** service also makes the newly created process runnable.

Execution Environment

The **creatp** kernel service can be called from the process environment only.

Return Values

–1	Indicates an error.

Upon successful completion, the **creatp** kernel service returns the process identifier for the new kernel process.

Implementation Specifics

The **creatp** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **initp** kernel service.

CSaixLockSocket Kernel Service

Purpose

Hold or release the lock on a logical socket in Card Services.

Syntax

```
#include <sys/pcmciacs.h>
#include <sys/pcmciaAix.h>
CSaixVdrSpcPkt pkt = {
                    Socket,
                    CSaixLockSocket,
                    actioncode,              /* other members are
not used */
                    };

int status =
CardServices( CSVendorSpecific, clientID, 0, sizeof( pkt ),
&pkt);
int* clientID;
```

Parameters

Socket	Logical Socket Number to hold or release a lock.
actioncode	**_CSaixHoldLock** to hold a lock, **_CSaixReleaseLock** to release a lock.
clientID	Client ID

Description

CSaixLockSocket is called to hold or release the lock on a logical socket in Card Services. Other threads who want to access the *socket* need to wait until the lock is released. The thread who holds the lock should not wait for the events from Card Services, because event handler thread in Card Services is also blocked to access the *socket*. Also, it should not hold or release other locks in a device driver or kernel while the lock is held.

The lock is desired to hold when a client starts card's configuration or unconfiguration, because accessing PCMCIA card during card's configuration or unconfiguration by the other threads may cause the system to hang.

Return Values

CSR_SUCCESS	Specifies a successful function.
CSR_BAD_SOCKET	Socket number is wrong or the lock is already held/released.

Error Codes

0	Specifies success.

CSVendorSpecific Kernel Service

Purpose

Provide AIX unique functions in Card Services.

Syntax

```
#include <sys/pcmciacs.h>
#include <sys/pcmciacsAix.h>

int status
= CardServices(CSVendorSpecific, objectID, pointer,
argLength, argPointer);
void* objectID;
void* pointer;
int argLength;
CSaixVdrSpcPkt* argPointer;
```

Parameters

objectID	Depends on each function.
pointer	Depends on each function.
argLength	Should be more than sizeof(**CSaixVdrSpcPkt**). But, each function may require more packet size.
argPointer	The pointer to the requesting packet. **CSaixVdrSpcPkt** is defined in **<sys/pcmciacsAix.h>** as;

```
typedef struct {
int     Sockey;
int     funccode;
int     subcode;
int     rsvd;
int     opcode;
int     rsvd2;
char    subpkt[1];
} CSaixVdrSpcPkt;
```

Socket is used to specify logical socket number. funccode is used to specify function code in **CSVendorSpecific** function. Currently, **CSaixLockSocket** is prepared. subcode, opcode and subpkt may be used depending on funccode. **Rsvd** and **rsvdd** are reserved.

Description

CSVendorSpecific function parsed the packet pointed by *argPointer*, and calls each function according to the function code specified in the packet.

Return Values

CSR_SUCCESS	Specifies a successful function.
CSR_UNSUPPORTED_FUNCTION	Error on parsing packet pointed by *argPointer*.

curtime Kernel Service

Purpose

Reads the current time into a time structure.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/time.h>

void curtime (timestruct)
struct timestruc_t *timestruct;
```

Parameter

timestruct	Points to a **timestruc_t** time structure defined in the **/usr/include/sys/time.h** file. The **curtime** kernel service updates the fields in this structure with the current time.

Description

The **curtime** kernel service reads the current time into a time structure defined in the **/usr/include/sys/time.h** file. This service updates the `tv_sec` and `tv_nsec` fields in the time structure, pointed to by the *timestruct* parameter, from the hardware real–time clock. The kernel also maintains and updates a memory–mapped time **tod** structure. This structure is updated with each clock tick.

The kernel also maintains two other in–memory time values: the **lbolt** and **time** values. The three in–memory time values that the kernel maintains (the **tod**, **lbolt**, and **time** values) are available to kernel extensions. The **lbolt** in–memory time value is the number of timer ticks that have occurred since the system was booted. This value is updated once per timer tick. The **time** in–memory time value is the number of seconds since Epoch. The kernel updates the value once per second.

Note: POSIX 1003.1 defines ”seconds since Epoch” as a ”value interpreted as the number of seconds between a specified time and the Epoch”. It further specifies that a ”Coordinated Universal Time name specified in terms of seconds (*tm_sec*), minutes (*tm_min*), hours (*tm_hour*), and days since January 1 of the year (*tm_yday*), and calendar year minus 1900 (*tm_year*) is related to a time represented as seconds since the Epoch, according to the following expression: *tm_sec* + *tm_min* * 60 *tm_hour**3600 + *tm_yday* * 86400 + (*tm_year* – 70) * 31536000 ((*tm_year* – 69) / 4) * 86400 if the year is greater than or equal to 1970, otherwise it is undefined.”

The **curtime** kernel service does not page–fault if a pinned stack and input time structure are used. Also, accessing the **lbolt**, **time**, and **tod** in–memory time values does not cause a page fault since they are in pinned memory.

Execution Environment

The **curtime** kernel service can be called from either the process or interrupt environment.

The **tod**, **time**, and **lbolt** memory–mapped time values can also be read from the process or interrupt handler environment. The *timestruct* parameter and stack must be pinned when the **curtime** service is called in an interrupt handler environment.

Return Values

The **curtime** kernel service has no return values.

Implementation Specifics

The **curtime** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Timer and Time–of–Day Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

d_align Kernel Service

Purpose

Provides needed information to align a buffer with a processor cache line.

Library

Kernel Extension Runtime Routines Library (**libsys.a**)

Syntax

```
int  d_align()
```

Description

To maintain cache consistency with system memory, buffers must be aligned. The **d_align** kernel service helps provide that function by returning the maximum processor cache–line size. The cache–line size is returned in log2 form.

Execution Environment

The **d_align** service can be called from either the process or interrupt environment.

Implementation Specifics

The **d_align** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **d_cflush** kernel service, **d_clear** kernel service, **d_roundup** kernel service.

Understanding Direct Memory Access (DMA) Transfer in *AIX Kernel Extensions and Device Support Programming Concepts*.

d_cflush Kernel Service

Purpose

Flushes the processor and I/O channel controller (IOCC) data caches when mapping bus device DMA with the long–term **DMA_WRITE_ONLY** option.

Syntax

```
int d_cflush (channel_id,
baddr, count, daddr)
int  channel_id;
caddr_t  baddr;
size_t  count;
caddr_t  daddr;
```

Parameters

channel_id	Specifies the DMA channel ID returned by the **d_init** kernel service.
baddr	Designates the address of the memory buffer.
count	Specifies the length of the memory buffer transfer in bytes.
daddr	Designates the address of the device corresponding to the transfer.

Description

The **d_cflush** kernel service should be called after data has been modified in a buffer that will undergo direct memory access (DMA) processing. Through DMA processing, this data is sent to a device where the **d_master** kernel service with the **DMA_WRITE_ONLY** option has already mapped the buffer for device DMA. The **d_cflush** kernel service is not required if the **DMA_WRITE_ONLY** option is not used or if the buffer is mapped before each DMA operation by calling the **d_master** kernel service.

The **d_cflush** kernel service flushes the processor cache for the involved cache lines and invalidates any previously retrieved data that may be in the IOCC buffers for the designated channel. This most frequently occurs when using long–term buffer mapping for DMA support to or from a device.

Long–Term DMA Buffer Mapping

The long–term DMA buffer mapping approach is frequently used when a pool of buffers is defined for sending commands and obtaining responses from an adapter using bus master DMA. This approach is also used frequently in the communications field where buffers can come from a common pool such as the **mbuf** pool or a pool used for protocol headers.

When using a fixed pool of buffers, the **d_master** kernel service is used only once to map the pool's address and range. The device driver then modifies the data in the buffers. It must also flush the data from the processor and invalidate the IOCC data cache involved in transfers with the device. The IOCC cache must be invalidated because the data in the IOCC data cache may be stale due to the last DMA operation to or from the buffer area that has just been modified for the next operation.

The **d_cflush** kernel service permits the flushing of the processor cache and making the required IOCC cache not valid. The device driver should use this service after modifying the data in the buffer and before sending the command to the device to start the DMA operation.

Once DMA processing has been completed, the device driver should call the **d_complete** service to check for errors and ensure that any data read from the device has been flushed to memory.

Execution Environment

The **d_cflush** kernel service can be called from either the process or interrupt environment.

Return Values

0	Indicates that the transfer was successfully completed.
EINVAL	Indicates the presence of an invalid parameter.

Implementation Specifics

The **d_cflush** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **d_complete** kernel service, **d_init** kernel service, **d_master** kernel service.

I/O Kernel Services and Understanding Direct Memory Access (DMA) Transfer in *AIX Kernel Extensions and Device Support Programming Concepts*.

d_clear Kernel Service

Purpose

Frees a direct memory access (DMA) channel.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/dma.h>

void d_clear (channel_id)
int channel_id;
```

Parameter

channel_id DMA channel identifier returned by the **d_init** service.

Description

The **d_clear** kernel service cleans up a DMA channel. To clean up the DMA channel:

1. Mark the DMA channel specified by the *channel_id* parameter as free.
2. Reset the DMA channel.

The **d_clear** service is typically called by a device driver in its close routine. It has no return values.

Attention: The **d_clear** service, as with all DMA services, should not be called unless the DMA channel has been successfully allocated with the **d_init** service. The **d_complete** service must have been called to clean up after any DMA transfers. Otherwise, data will be lost and the system integrity will be compromised.

Execution Environment

The **d_clear** kernel service can be called from either the process or interrupt environment.

Return Values

The **d_clear** kernel service has no return values.

Implementation Specifics

The **d_clear** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **d_complete** kernel service, **d_init** kernel service.

I/O Kernel Services and Understanding Direct Memory Access (DMA) Transfers in *AIX Kernel Extensions and Device Support Programming Concepts*.

d_complete Kernel Service

Purpose

Cleans up after a direct memory access (DMA) transfer.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/dma.h>
#include <sys/xmem.h>

int d_complete
(channel_id, flags, baddr,
count, dp, daddr)
int channel_id;
int flags;
caddr_t baddr;
size_t count;
struct xmem *dp;
caddr_t daddr;
```

Parameters

channel_id	Specifies the DMA channel identifier returned by the **d_init** service.
flags	Describes the DMA transfer. The /**usr**/**include**/**dma.h** file describes these flags.
baddr	Designates the address of the memory buffer.
count	Specifies the length of the transfer in bytes.
dp	Specifies the address of the cross–memory descriptor.
daddr	Designates the address used to program the DMA master. A value of null is specified for DMA slaves.

Description

The **d_complete** kernel service completes the processing of a DMA transfer. It also indicates any DMA error detected by the system hardware. The **d_complete** service must be called after each DMA transfer.

The **d_complete** service performs machine–dependent processing, which entails:

- Flushing system DMA buffers
- Making the DMA buffer accessible to the processor

Note: When calling the **d_master** service several times for one or more of the same pages of memory, the corresponding number of **d_complete** calls must be made to reveal successfully the page or pages involved in the DMA transfers. Pages are not hidden from the processor during the DMA mapping if the **DMA_WRITE_ONLY** flag is specified on the call to the **d_master** service.

”Understanding Direct Memory Access (DMA) Transfers” in *AIX Kernel Extensions and Device Support Programming Concepts* further describes DMA transfers.

Execution Environment

The **d_complete** kernel service can be called from either the process or interrupt environment.

Return Values

DMA_SUCC	Indicates a successful completion.
DMA_INVALID	Indicates an operation that is not valid. A load or store that was not valid was performed to the I/O bus.
DMA_LIMIT	Indicates a limit check. A load or store to the I/O bus occurred that was not sufficiently authorized to access the I/O bus address.
DMA_NO_RESPONSE	Indicates no response. No device responded to the I/O bus access.
DMA_CONFLICT	Indicates an address conflict. A *daddr* parameter was specified to the **d_master** service for a system memory transfer, where this transfer conflicts with the bus memory address of an I/O bus device.
DMA_AUTHORITY	Indicates an authority error. A protection exception occurred while accessing an I/O bus memory address.
DMA_PAGE_FAULT	Indicates a page fault. A reference was made to a page not currently located in system memory.
DMA_BAD_ADDR	Indicates an address that is not valid. A bus address that is not valid or was unsupported was used. A *daddr* parameter that was not valid was specified to the **d_master** service.
DMA_CHECK	Indicates a channel check. A channel check was generated during the bus cycle. This typically occurs when a device detects a data parity error.
DMA_DATA	Indicates a system–detected data parity error.
DMA_ADDRESS	Indicates a system–detected address parity error.
DMA_EXTRA	Indicates an extra request. This typically occurs when the *count* parameter was specified incorrectly to the **d_slave** service.
DMA_SYSTEM	Indicates a system error. The system detected an internal error in system hardware. This is typically a parity error on an internal bus or register.

Implementation Specifics

The **d_complete** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **d_init** kernel service, **d_master** kernel service, **d_slave** kernel service.

I/O Kernel Services and Understanding Direct Memory Access (DMA) Transfers in *AIX Kernel Extensions and Device Support Programming Concepts*.

delay Kernel Service

Purpose

Suspends the calling process for the specified number of timer ticks.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

void delay
(ticks)
int ticks;
```

Parameter

ticks	Specifies the number of timer ticks that must occur before the process is reactivated. Many timer ticks can occur per second.

Description

The **delay** kernel service suspends the calling process for the number of timer ticks specified by the *ticks* parameter.

The HZ value in the /**usr**/**include**/**sys**/**m_param.h** file can be used to determine the number of ticks per second.

Execution Environment

The **delay** kernel service can be called from the process environment only.

Return Values

The **delay** service has no return values.

Implementation Specifics

The **delay** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Timer and Time–of–Day Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

del_domain_af Kernel Service

Purpose

Deletes an address family from the Address Family domain switch table.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/domain.h>

int
del_domain_af (domain)
struct domain *domain;
```

Parameter

domain	Specifies the address family.

Description

The **del_domain_af** kernel service deletes the address family specified by the *domain* parameter from the Address Family domain switch table.

Execution Environment

The **del_domain_af** kernel service can be called from either the process or interrupt environment.

Return Value

EINVAL	Indicates that the specified address is not found in the Address Family domain switch table.

Example

To delete an address family from the Address Family domain switch table, invoke the **del_domain_af** kernel service as follows:

```
del_domain_af(&inetdomain);
```

In this example, the family to be deleted is `inetdomain`.

Implementation Specifics

The **del_domain_af** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **add_domain_af** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

del_input_type Kernel Service

Purpose

Deletes an input type from the Network Input table.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <net/if.h>

int del_input_type
(type)
u_short type;
```

Parameter

type	Specifies which type of protocol the packet contains. This parameter is a field in a packet.

Description

The **del_input_type** kernel service deletes an input type from the Network Input table to disable the reception of the specified packet type.

Execution Environment

The **del_input_type** kernel service can be called from either the process or interrupt environment.

Return Values

0	Indicates that the type was successfully deleted.
ENOENT	Indicates that the **del_input_type** service could not find the type in the Network Input table.

Examples

1. To delete an input type from the Network Input table, invoke the **del_input_type** kernel service as follows:

   ```
   del_input_type(ETHERTYPE_IP);
   ```

   In this example, `ETHERTYPE_IP` specifies that Ethernet IP packets should no longer be processed.

2. To delete an input type from the Network Input table, invoke the **del_input_type** kernel service as follows:

   ```
   del_input_type(ETHERTYPE_ARP);
   ```

   In this example, `ETHERTYPE_ARP` specifies that Ethernet ARP packets should no longer be processed.

Implementation Specifics

The **del_input_type** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **add_input_type** kernel service, **find_input_type** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

del_netisr Kernel Service

Purpose

Deletes a network software interrupt service routine from the Network Interrupt table.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <net/netisr.h>

int del_netisr (soft_intr_level)
u_short soft_intr_level;
```

Parameter

soft_intr_level	Specifies the software interrupt service to delete. The value of *soft_intr_level* should be greater than or equal to 0 and less than a value of **NETISR_MAX3**.

Description

The **del_netisr** kernel service deletes the network software interrupt service routine specified by the *soft_intr_level* parameter from the Network Software Interrupt table.

Execution Environment

The **del_netisr** kernel service can be called from either the process or interrupt environment.

Return Values

0	Indicates that the software interrupt service was successfully deleted.
ENOENT	Indicates that the software interrupt service was not found in the Network Software Interrupt table.

Example

To delete a software interrupt service from the Network Software Interrupt table, invoke the kernel service as follows:

```
del_netisr(NETISR_IP);
```

In this example, the software interrupt routine to be deleted is `NETISR_IP`.

Implementation Specifics

The **del_netisr** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **add_netisr** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

del_netopt Macro

Purpose

Deletes a network option structure from the list of network options.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <net/netopt.h>

del_netopt (option_name_symbol)
option_name_symbol;
```

Parameter

option_name_symbol	Specifies the symbol name used to construct the **netopt** structure and default names.

Description

The **del_netopt** macro deletes a network option from the linked list of network options. After the **del_netopt** service is called, the option is no longer available to the **no** command.

Execution Environment

The **del_netopt** macro can be called from either the process or interrupt environment.

Return Values

The **del_netopt** macro has no return values.

Implementation Specifics

The **del_netopt** macro is part of Base Operating System (BOS) Runtime.

Related Information

The **no** command.

The **add_netopt** macro.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

detach Device Queue Management Routine

Purpose

Provides a means for performing device–specific processing when the **detchq** kernel service is called.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/deviceq.h>

int detach(dev_parms, path_id)
caddr_t dev_parms;
cba_id path_id;
```

Parameters

dev_parms	Passed to **creatd** service when the **detach** routine is defined.
path_id	Specifies the path identifier for the queue that is being detached from.

Description

Each device queue can have a **detach** routine. This routine is optional and must be specified when the device queue is defined with the **creatd** service. The **detchq** service calls the **detach** routine each time a path to the device queue is removed.

To ensure that the **detach** routine is not called while a queue element from this client is still in the device queue, the kernel puts a detach control queue element at the end of the device queue. The server knows by convention that a detach control queue element signifies completion of all pending queue elements for that path. The kernel calls the **detach** routine after the detach control queue element is processed.

The **detach** routine executes under the process under which the **detchq** service is called. The kernel does not serialize the execution of this service with the execution of any of the other server routines.

Execution Environment

The **detach** routine can be called from the process environment only.

Return Values

RC_GOOD	Indicates successful completion.

A return value other than **RC_GOOD** indicates an irrecoverable condition causing system failure.

Implementation Specifics

The **detach** routine is part of the Device Queue Management kernel extension.

devdump Kernel Service

Purpose

Calls a device driver dump–to–device routine.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int devdump
(devno, uiop, cmd, arg,
chan, ext)
dev_t devno;
struct uio *uiop;
int cmd, arg, ext;
```

Parameters

devno	Specifies the major and minor device numbers.
uiop	Points to the **uio** structure containing write parameters.
cmd	Specifies which dump command to perform.
arg	Specifies a parameter or address to a parameter block for the specified command.
chan	Specifies the channel ID.
ext	Specifies the extended system call parameter.

Description

The kernel or kernel extension calls the **devdump** kernel service to initiate a memory dump to a device when writing dump data and then to terminate the dump to the target device.

The **devdump** service calls the device driver's **dddump** routine, which is found in the device switch table for the device driver associated with the specified device number. If the device number (specified by the *devno* parameter) is not valid or if the associated device driver does not have a **dddump** routine, an **ENODEV** return value is returned.

If the device number is valid and the specified device driver has a **dddump** routine, the routine is called.

If the device driver's **dddump** routine is successfully called, the return value for the **devdump** service is set to the return value provided by the device's **dddump** routine.

Execution Environment

The **devdump** kernel service can be called in either the process or interrupt environment, as described under the conditions described in the **dddump** routine.

Return Values

0	Indicates a successful operation.
ENODEV	Indicates that the device number is not valid or that no **dddump** routine is registered for this device.

The **dddump** device driver routine provides other return values.

Implementation Specifics

The **devdump** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **dddump** device driver entry point.

The **dmp_prinit** kernel service.

Kernel Extension and Device Driver Management Kernel Services and How Device Drivers are Accessed in *AIX Kernel Extensions and Device Support Programming Concepts*.

devstrat Kernel Service

Purpose

Calls a block device driver's strategy routine.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int devstrat (bp)
struct buf *bp;
```

Parameter

bp	Points to the **buf** structure specifying the block transfer parameters.

Description

The kernel or kernel extension calls the **devstrat** kernel service to request a block data transfer to or from the device with the specified device number. This device number is found in the **buf** structure. The **devstrat** service can only be used for the block class of device drivers.

The **devstrat** service calls the device driver's **ddstrategy** routine. This routine is found in the device switch table for the device driver associated with the specified device number found in the `b_dev` field. The `b_dev` field is found in the **buf** structure pointed to by the *bp* parameter. The caller of the **devstrat** service must have an **iodone** routine specified in the `b_iodone` field of the **buf** structure. Following the return from the device driver's **ddstrategy** routine, the **devstrat** service returns without waiting for the I/O to be performed.

On multiprocessor systems, all **iodone** routines run by default on the first processor started when the system was booted. This ensures compatibility with uniprocessor device drivers. If the **iodone** routine has been designed to be multiprocessor–safe, set the **B_MPSAFE** flag in the `b_flags` field of the **buf** structure passed to the **devstrat** kernel service. The **iodone** routine will then run on any available processor.

If the device major number is not valid or the specified device is not a block device driver, the **devstrat** service returns the **ENODEV** return code. If the device number is valid, the device driver's **ddstrategy** routine is called with the pointer to the **buf** structure (specified by the *bp* parameter).

Execution Environment

The **devstrat** kernel service can be called from either the process or interrupt environment.

Note: The **devstrat** kernel service can be called in the interrupt environment only if its priority level is **INTIODONE** or lower.

Return Values

0	Indicates a successful operation.
ENODEV	Indicates that the device number is not valid or that no **ddstrategy** routine registered. This value is also returned when the specified device is not a block device driver. If this error occurs, the **devstrat** service can cause a page fault.

Implementation Specifics

The **devstrat** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **iodone** kernel service.

The **ddstategy** routine.

The **buf** structure.

Kernel Extension and Device Driver Management Kernel Services and How Device Drivers are Accessed in *AIX Kernel Extensions and Device Support Programming Concepts*.

devswadd Kernel Service

Purpose

Adds a device entry to the device switch table.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/device.h>

int devswadd (devno, dswptr)
dev_t devno;
struct devsw *dswptr;
```

Parameters

devno	Specifies the major and minor device numbers to be associated with the specified entry in the device switch table.
dswptr	Points to the device switch structure to be added to the device switch table.

Description

The **devswadd** kernel service is typically called by a device driver's **ddconfig** routine to add or replace the device driver's entry points in the device switch table. The device switch table is a table of device switch (**devsw**) structures indexed by the device driver's major device number. This table of structures is used by the device driver interface services in the kernel to facilitate calling device driver routines.

The major device number portion of the *devno* parameter is used to specify the index in the device switch table where the **devswadd** service must place the specified device switch entry. Before this service copies the device switch structure into the device switch table, it checks the existing entry to determine if any opened device is using it. If an opened device is currently occupying the entry to be replaced, the **devswadd** service does not perform the update. Instead, it returns an **EEXIST** error value. If the update is successful, it returns a value of 0.

Entry points in the device switch structure that are not supported by the device driver must be handled in one of two ways. If a call to an unsupported entry point should result in the return of an error code, then the entry point must be set to the **nodev** routine in the structure. As a result, any call to this entry point automatically invokes the **nodev** routine, which returns an **ENODEV** error code. The kernel provides the **nodev** routine.

Otherwise, a call to an unsupported entry point should be treated as a no–operation function. Then the corresponding entry point should be set to the **nulldev** routine. The **nulldev** routine, which is also provided by the kernel, performs no operation if called and returns a 0 return code.

On multiprocessor systems, all device driver routines run by default on the first processor started when the system was booted. This ensures compatibility with uniprocessor device drivers. If the device driver being added has been designed to be multiprocessor–safe, set the **DEV_MPSAFE** flag in the `d_opts` field of the **devsw** structure passed to the **devswadd** kernel service. The device driver routines will then run on any available processor.

All other fields within the structure that are not used should be set to 0. Some fields in the structure are for kernel use; the **devswadd** service does not copy these fields into the device switch table. These fields are documented in the /**usr**/**include**/**device.h** file.

Execution Environment

The **devswadd** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
EEXIST	Indicates that the specified device switch entry is in use and cannot be replaced.
ENOMEM	Indicates that the entry cannot be pinned due to insufficient real memory.
EINVAL	Indicates that the major device number portion of the *devno* parameter exceeds the maximum permitted number of device switch entries.

Implementation Specifics

The **devswadd** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **devswchg** kernel service, **devswdel** kernel service, **devswqry** kernel service.

The **ddconfig** device driver entry point.

Kernel Extension and Device Driver Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

devswchg Kernel Service

Purpose

Alters a device switch entry point in the device switch table.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/device.h>

int devswchg (devno, type, newfunc, oldfunc);
dev_t devno;
int type;
int (*newfunc) ();
int (**oldfunc)();
```

Parameters

devno	Specifies the major and minor device numbers of the device to be changed.
type	Specifies the device switch entry point to alter. The *type* parameter can have one of the following values: **DSW_BLOCK** Alters the **ddstrategy** entry point. **DSW_CONFIG** Alters the **ddconfig** entry point. **DSW_CREAD** Alters the **ddread** entry point. **DSW_CWRITE** Alters the **ddwrite** entry point. **DSW_DUMP** Alters the **dddump** entry point. **DSW_MPX** Alters the **ddmpx** entry point. **DSW_SELECT** Alters the **ddselect** entry point. **DSW_TCPATH** Alters the **ddrevoke** entry point.
newfunc	Specifies the new value for the device switch entry point.
oldfunc	Specifies that the old value of the device switch entry point be returned here.

Description

The **devswchg** kernel service alters the value of a device switch entry point (function pointer) after a device switch table entry has been added by the **devswadd** kernel service. The device switch entry point specified by the *type* parameter is set to the value of the *newfunc* parameter. Its previous value is returned in the memory addressed by the *oldfunc* parameter. Only one device switch entry can be altered per call.

If the **devswchg** kernel service is unsuccessful, the value referenced by the *oldfunc* parameter is not defined.

Execution Environment

The **devswchg** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
EINVAL	Indicates the *Type* command was not valid.
ENODEV	Indicates the device switch entry specified by the *devno* parameter is not defined.

Implementation Specifics

The **devswchg** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **devswadd** kernel service.

List of Kernel Extension and Device Driver Management Kernel Services and How Device Drivers are Accessed in *AIX Kernel Extensions and Device Support Programming Concepts*.

devswdel Kernel Service

Purpose

Deletes a device driver entry from the device switch table.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/device.h>

int devswdel
(devno)
dev_t devno;
```

Parameter

devno	Specifies the major and minor device numbers of the device to be deleted.

Description

The **devswdel** kernel service is typically called by a device driver's **ddconfig** routine on termination to remove the device driver's entry points from the device switch table. The device switch table is a table of device switch (**devsw**) structures indexed by the device driver's major device number. The device driver interface services use this table of structures in the kernel to facilitate calling device driver routines.

The major device number portion of the *devno* parameter is used to specify the index into the device switch table for the entry to be removed. Before the device switch structure is removed, the existing entry is checked to determine if any opened device is using it.

If an opened device is currently occupying the entry to be removed, the **devswdel** service does not perform the update. Instead, it returns an **EEXIST** return code. If the removal is successful, a return code of 0 is set.

The **devswdel** service removes a device switch structure entry from the table by marking the entry as undefined and setting all of the entry point fields within the structure to a **nodev** value. As a result, any callers of the removed device driver return an **ENODEV** error code. If the specified entry is already marked undefined, the **devswdel** service returns an **ENODEV** error code.

Execution Environment

The **devswdel** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
EEXIST	Indicates that the specified device switch entry is in use and cannot be removed.
ENODEV	Indicates that the specified device switch entry is not defined.
EINVAL	Indicates that the major device number portion of the *devno* parameter exceeds the maximum permitted number of device switch entries.

Implementation Specifics

The **devswdel** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **devswadd** kernel service, **devswchg** kernel service, **devswqry** kernel service.

Kernel Extension and Device Driver Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

devswqry Kernel Service

Purpose

Checks the status of a device switch entry in the device switch table.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/device.h>int devswqry (devno, status, dsdptr)
dev_t devno;
uint *status;
caddr_t *dsdptr;
```

Parameters

devno	Specifies the major and minor device numbers of the device to be queried.
status	Points to the status of the specified device entry in the device switch table. This parameter is passed by reference.
dsdptr	Points to device–dependent information for the specified device entry in the device switch table. This parameter is passed by reference.

Description

The **devswqry** kernel service returns the status of a specified device entry in the device switch table. The entry in the table to query is determined by the major portion of the device number specified in the *devno* parameter. The status of the entry is returned in the *status* parameter that is passed by reference on the call. If this pointer is null on entry to the **devswqry** service, then the status is not returned to the caller.

The **devswqry** service also returns the address of device–dependent information for the specified device entry in the device switch table. This address is taken from the `d_dsdptr` field for the entry and returned in the *dsdptr* parameter, which is passed by reference. If this pointer is null on entry to the **devswqry** service, then the service does not return the address from the `d_dsdptr` field to the caller.

Status Parameter Flags

The *status* parameter comprises a set of flags that can indicate the following conditions:

DSW_BLOCK	Device switch entry is defined by a block device driver. This flag is set when the device driver has a **ddstrategy** entry point.
DSW_CONFIG	Device driver in this device switch entry provides an entry point for configuration.
DSW_CREAD	Device driver in this device switch entry is providing a routine for character reads or raw input. This flag is set when the device driver has a **ddread** entry point.
DSW_CWRITE	Device driver in this device switch entry is providing a routine for character writes or raw output. This flag is set when the device driver has a **ddwrite** entry point.
DSW_DEFINED	Device switch entry is defined.
DSW_DUMP	Device driver defined by this device switch entry provides the capability to support one or more of its devices as targets for a kernel dump. This flag is set when the device driver has provided a **dddump** entry point.

DSW_MPX	Device switch entry is defined by a multiplexed device driver. This flag is set when the device driver has a **ddmpx** entry point.
DSW_OPENED	Device switch entry is in use and the device has outstanding opens. This flag is set when the device driver has at least one outstanding open.
DSW_SELECT	Device driver in this device switch entry provides a routine for handling the **select** or **poll** subroutines. This flag is set when the device driver has provided a **ddselect** entry point.
DSW_TCPATH	Device driver in this device switch entry supports devices that are considered to be in the trusted computing path and provide support for the revoke function. This flag is set when the device driver has provided a **ddrevoke** entry point.
DSW_TTY	Device switch entry is in use by a tty device driver. This flag is set when the pointer to the **d_ttys** structure is not a null character**.**
DSW_UNDEFINED	Device switch entry is not defined.

The *status* parameter is set to the **DSW_UNDEFINED** flag when a device switch entry is not in use. This is the case if either of the following are true:

- The entry has never been used. (No previous call to the **devswadd** service was made.)
- The entry has been used but was later deleted. (A call to the **devswadd** service was issued, followed by a call to the **devswdel** service.)

No other flags are set when the **DSW_UNDEFINED** flag is set.

Note: The *status* parameter must be a null character if called from the interrupt environment.

Execution Environment

The **devswqry** kernel service can be called from either the process or interrupt environment.

Return Values

0	Indicates a successful operation.
EINVAL	Indicates that the major device number portion of the *devno* parameter exceeds the maximum permitted number of device switch entries.

Implementation Specifics

The **devswqry** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **devswadd** kernel service, **devswchg** kernel service, **devswdel** kernel service.

Kernel Extension and Device Driver Management Kernel Services.

d_init Kernel Service

Purpose

Initializes a direct memory access (DMA) channel.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/dma.h>
#include <sys/adspace.h>

int d_init (channel, flags, bus_id)
int channel;
int flags;
vmhandle_t bus_id;
```

Parameters

channel	Specifies the DMA channel number.
flags	Specifies the flags that describe how the DMA channel is used. These flags are described in the /**usr**/**include**/**sys**/**dma.h** file.
bus_id	Identifies the I/O bus that the channel is to be allocated on. This parameter is normally passed to the device driver in the Device–Dependent Structure at driver initialization time.

Description

The **d_init** kernel service initializes a DMA channel. A device driver must call this service before using the DMA channel. Initializing the DMA channel entails:

- Designating the DMA channel specified by the *channel* parameter as allocated
- Personalizing the DMA channel as specified by the *flags* parameter

The **d_init** service is typically called by a device driver in its open routine when the device is not already in the opened state. A device driver must call the **d_init** service before using the DMA channel.

Execution Environment

The **d_init** kernel service can be called from either the process or interrupt environment.

Return Values

channel_id	Indicates a successful operation. This value is used as an input parameter to the other DMA routines.
DMA_FAIL	Indicates that the DMA channel is not available because it is currently allocated.

Implementation Specifics

The **d_init** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **d_clear** kernel service.

I/O Kernel Services and Understanding Direct Memory Access (DMA) Transfers in *AIX Kernel Extensions and Device Support Programming Concepts*.

Device–Dependent Structure (DDS) Overview.

disable_lock Kernel Service

Purpose

Raises the interrupt priority, and locks a simple lock if necessary.

Syntax

```
#include <sys/lock_def.h>

int disable_lock (int_pri, lock_addr)
int int_pri;
simple_lock_t lock_addr;
```

Parameters

int_pri	Specifies the interrupt priority to set.
lock_addr	Specifies the address of the lock word to lock.

Description

The **disable_lock** kernel service raises the interrupt priority, and locks a simple lock if necessary, in order to provide optimized thread–interrupt critical section protection for the system on which it is executing. On a multiprocessor system, calling the **disable_lock** kernel service is equivalent to calling the **i_disable** and **simple_lock** kernel services. On a uniprocessor system, the call to the **simple_lock** service is not necessary, and is omitted. However, you should still pass a valid lock address to the **disable_lock** kernel service. Never pass a **NULL** lock address.

Execution Environment

The **disable_lock** kernel service can be called from either the process or interrupt environment.

Return Values

The **disable_lock** kernel service returns the previous interrupt priority.

Implementation Specifics

The **disable_lock** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **i_disable** kernel service, **simple_lock_init** kernel service, **simple_lock** kernel service, **unlock_enable** kernel service.

Understanding Locking,Locking Kernel Services,Understanding Interrupts,I/O Kernel Services, and Interrupt Environment. in *AIX Kernel Extensions and Device Support Programming Concepts*.

d_map_clear Kernel Service

Purpose

Deallocates resources previously allocated on a **d_map_init** call.

Syntax

```
#include <sys/dma.h>

void d_map_clear (*handle)
struct d_handle *handle
```

Parameters

handle	Indicates the unique handle returned by the **d_map_init** kernel service.

Description

The **d_map_clear** kernel service is a bus–specific utility routine determined by the **d_map_init** service that deallocates resources previously allocated on a **d_map_init** call. This includes freeing the **d_handle** structure that was allocated by **d_map_init**.

Note: You can use the **D_MAP_CLEAR** macro provided in the /**usr**/**include**/**sys**/**dma.h** file to code calls to the **d_map_clear** kernel service.

Implementation Specifics

The **d_map_clear** kernel service is part of the base device package of your platform.

Related Information

The **d_map_init** kernel service.

d_map_disable Kernel Service

Purpose

Disables DMA for the specified handle.

Syntax

```
#include <sys/dma.h>

int d_map_disable(*handle)
struct d_handle *handle;
```

Parameters

handle	Indicates the unique handle returned by **d_map_init**.

Description

The **d_map_disable** kernel service is a bus–specific utility routine determined by the **d_map_init** kernel service that disables DMA for the specified *handle* with respect to the platform.

Note: You can use the **D_MAP_DISABLE** macro provided in the /**usr**/**include**/**sys**/**dma.h** file to code calls to the **d_map_disable** kernel service.

Return Values

DMA_SUCC	Indicates the DMA is successfully disabled.
DMA_FAIL	Indicates the DMA could not be explicitly disabled for this device or bus.

Implementation Specifics

The **d_map_disable** kernel service is part of the base device package of your platform.

Related Information

The **d_map_init** kernel service.

d_map_enable Kernel Service

Purpose

Enables DMA for the specified handle.

Syntax

```
#include <sys/dma.h>

int d_map_enable(*handle)
struct d_handle *handle;
```

Parameters

handle	Indicates the unique handle returned by **d_map_init**.

Description

The **d_map_enable** kernel service is a bus–specific utility routine determined by the **d_map_init** kernel service that enables DMA for the specified *handle* with respect to the platform.

Note: You can use the **D_MAP_ENABLE** macro provided in the /**usr**/**include**/**sys**/**dma.h** file to code calls to the **d_map_enable** kernel service.

Return Values

DMA_SUCC	Indicates the DMA is successfully enabled.
DMA_FAIL	Indicates the DMA could not be explicitly enabled for this device or bus.

Implementation Specifics

The **d_map_enable** kernel service is part of the base device package of your platform.

Related Information

The **d_map_init** kernel service.

d_map_init Kernel Service

Purpose

Allocates and initializes resources for performing DMA with PCI and ISA devices.

Syntax

```
#include <sys/dma.h>

struct d_handle* d_map_init (bid, flags, bus_flags, channel)
int bid;
int flags;
int bus_flags;
uint channel;
```

Parameters

bid	Specifies the bus identifier.
flags	Describes the mapping.
bus_flags	Specifies the target bus flags.
channel	Indicates the *channel* assignment specific to the bus.

Description

The **d_map_init** kernel service allocates and initializes resources needed for managing DMA operations and returns a unique *handle* to be used on subsequent DMA service calls. The *handle* is a pointer to a **d_handle** structure allocated by **d_map_init** from the pinned heap for the device. The device driver uses the function addresses provided in the *handle* for accessing the DMA services specific to its host bus. The **d_map_init** service returns a **DMA_FAIL** error when resources are unavailable or cannot be allocated.

The *channel* parameter is the assigned channel number for the device, if any. Some devices and or buses might not have the concept of *channels*. For example, an ISA device driver would pass in its assigned DMA channel in the *channel* parameter.

Note: The possible flag values for the *flags* parameter can be found in /**usr**/**include**/**sys**/**dma.h**. These flags can be logically ORed together to reflect the desired characteristics.

Execution Environment

The **d_map_init** kernel service should only be called from the process environment.

Return Values

DMA_FAIL	Indicates that the resources are unavailable. No registration was completed.
struct d_handle *	Indicates successful completion.

Implementation Specifics

The **d_map_init** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **d_map_clear** kernel service, **d_map_page** kernel service, **d_unmap_page** kernel service, **d_map_list** kernel service, **d_unmap_list** kernel service, **d_map_slave** kernel service, **d_unmap_slave** kernel service, **d_map_disable** kernel service, **d_map_enable** kernel service.

d_map_list Kernel Service

Purpose

Performs platform–specific DMA mapping for a list of virtual addresses.

Syntax

```
#include <sys/dma.h>

int d_map_list (*handle, flags, minxfer, *virt_list, *bus_list)
struct d_handle *handle;
int flags;
int minxfer;
struct dio *virt_list;
struct dio *bus_list;
```

Note: The following is the interface definition for **d_map_list** when the **DMA_ADDRESS_64** and **DMA_ENABLE_64** flags are set on the **d_map_init** call.

```
int d_map_list (*handle, flags, minxfer, *virt_list, *bus_list)
struct d_handle *handle;
int flags;
int minxfer;
struct dio_64 *virt_list;
struct dio_64 *bus_list;
```

Parameters

handle	Indicates the unique handle returned by the **d_map_init** kernel service.
flags	Specifies one of the following flags: **DMA_READ** Transfers from a device to memory. **BUS_DMA** Transfers from one device to another device. **DMA_BYPASS** Do not check page access.
minxfer	Specifies the minimum transfer size for the device.
virt_list	Specifies a list of virtual buffer addresses and lengths.
bus_list	Specifies a list of bus addresses and lengths.

Description

The **d_map_list** kernel service is a bus–specific utility routine determined by the **d_map_init** kernel service that accepts a list of virtual addresses and sizes and provides the resulting list of bus addresses. This service fills out the corresponding bus address list for use by the device in performing the DMA transfer. This service allows for scatter/gather capability of a device and also allows the device to combine multiple requests that are contiguous with respect to the device. The lists are passed via the **dio** structure. If the **d_map_list** service is unable to complete the mapping due to exhausting the capacity of the provided **dio** structure, the **DMA_DIOFULL** error is returned. If the **d_map_list** service is unable to complete the mapping due to exhausting resources required for the mapping, the **DMA_NORES** error is returned. In both of these cases, the *bytes_done* field of the **dio** virtual list is set to the number of bytes successfully mapped. This byte count is a multiple of the *minxfer* size for the device as provided on the call to **d_map_list**. The *resid_iov* field is set to the index of the remaining *d_iovec* fields in the list. Unless the **DMA_BYPASS** flag is set, this service verifies access permissions to each page. If an access violation is encountered on a page with the list, the **DMA_NOACC** error is returned, and the *bytes_done* field is set to the number of bytes preceding the faulting *iovec*.

Notes:

1. When the **DMA_NOACC** return value is received, no mapping is done, and the bus list is undefined. In this case, the *resid_iov* field is set to the index of the *d_iovec* that encountered the access violation.
2. You can use the **D_MAP_LIST** macro provided in the /**usr**/**include**/**sys**/**dma.h** file to code calls to the **d_map_list** kernel service.

Return Values

DMA_NORES Indicates that resources were exhausted during mapping.

Note: **d_map_list** possible partial transfer was mapped. Device driver may continue with partial transfer and submit the remainer on a subsequent **d_map_list** call, or call **d_unmap_list** to undo the partial mapping. If a partial transfer is issued, then the driver must call **d_unmap_list** when the I/O is complete.

DMA_DIOFULL Indicates that the target bus list is full.

Note: **d_map_list** possible partial transfer was mapped. Device driver may continue with partial transfer and submit the remainder on a subsequent **d_map_list** call, or call **d_unmap_list** to undo the partial mapping. If a partial transfer is issued, then the driver must call **d_unmap_list** when the I/O is complete.

DMA_NOACC Indicates no access permission to a page in the list.

Note: **d_map_list** no mapping was performed. No need for the device driver to call **d_unmap_list**, but the driver must fail the faulting I/O request, and resubmit any remainder in a subsequent **d_map_list** call.

DMA_SUCC Indicates that the entire transfer successfully mapped.

Note: **d_map_list** successful mapping was performed. Device driver must call **d_unmap_list** when the I/O is complete. In the case of a long–term mapping, the driver must call **d_unmap_list** when the long–term mapping is no longer needed.

Implementation Specifics

The **d_map_list** kernel service is part of the base device package of your platform.

Related Information

The **d_map_init** kernel service.

d_map_page Kernel Service

Purpose

Performs platform–specific DMA mapping for a single page.

Syntax

```
#include <sys/dma.h>
#include <sys/xmem.h>

int d_map_page(*handle, flags, baddr, *busaddr, *xmp)
struct d_handle *handle;
int flags;
caddr_t baddr;
uint *busaddr;
struct xmem *xmp;
```

Note: The following is the interface definition for **d_map_page** when the **DMA_ADDRESS_64** and **DMA_ENABLE_64** flags are set on the **d_map_init** call.

```
int d_map_page(*handle, flags, baddr, *busaddr, *xmp)
struct d_handle *handle;
int flags;
unsigned long long baddr;
unsigned long long *busaddr;
struct xmem *xmp;
```

Parameters

handle	Indicates the unique handle returned by the **d_map_init** kernel service.
flags	Specifies one of the following flags: **DMA_READ** Transfers from a device to memory. **BUS_DMA** Transfers from one device to another device. **DMA_BYPASS** Do not check page access.
baddr	Specifies the buffer address.
busaddr	Points to the *busaddr* field.
xmp	Cross–memory descriptor for the buffer.

Description

The **d_map_page** kernel service is a bus–specific utility routine determined by the **d_map_init** kernel service that performs platform specific mapping of a single 4KB or less transfer for DMA master devices. The **d_map_page** kernel service is a fast–path version of the **d_map_list** service. The entire transfer amount must fit within a single page in order to use this service. This service accepts a virtual address and completes the appropriate bus address for the device to use in the DMA transfer. Unless the **DMA_BYPASS** flag is set, this service also verifies access permissions to the page.

If the buffer is a global kernel space buffer, the cross–memory descriptor can be set to point to the exported **GLOBAL** cross–memory descriptor, *xmem_global*.

If the transfer is unable to be mapped due to resource restrictions, the **d_map_page** service returns **DMA_NORES**. If the transfer is unable to be mapped due to page access violations, this service returns **DMA_NOACC**.

Note: You can use the **D_MAP_PAGE** macro provided in the /**usr**/**include**/**sys**/**dma.h** file to code calls to the **d_map_page** kernel service.

Return Values

DMA_NORES Indicates that resources are unavailable.

Note: **d_map_page** no mapping is done, device driver must wait until resources are freed and attempt the **d_map_page** call again.

DMA_NOACC Indicates no access permission to the page.

Note: **d_map_page** no mapping is done, device driver must fail the corresponding I/O request.

DMA_SUCC Indicates that the *busaddr* parameter contains the bus address to use for the device transfer.

Note: **d_map_page** successful mapping was done, device driver must call **d_unmap_page** when I/O is complete, or when device driver is finished with the mapped area in the case of a long–term mapping.

Implementation Specifics

The **d_map_page** kernel service is part of the base device package of your platform.

Related Information

The **d_map_init** kernel service, **d_map_list** kernel service.

d_map_slave Kernel Service

Purpose

Accepts a list of virtual addresses and sizes and sets up the slave DMA controller.

Syntax

```
#include <sys/dma.h>

int d_map_slave (*handle, flags, minxfer, *vlist, chan_flag)
struct d_handle *handle;
int flags;
int minxfer;
struct dio *vlist;
uint chan_flag;
```

Parameters

handle	Indicates the unique handle returned by the **d_map_init** kernel service.
flags	Specifies one of the following flags: **DMA_READ** Transfers from a device to memory. **BUS_DMA** Transfers from one device to another device. **DMA_BYPASS** Do not check page access.
minxfer	Specifies the minimum transfer size for the device.
vlist	Specifies a list of buffer addresses and lengths.
chan_flag	Specifies the device and bus specific flags for the transfer.

Description

The **d_map_slave** kernel service accepts a list of virtual buffer addresses and sizes and sets up the slave DMA controller for the requested DMA transfer. This includes setting up the system address generation hardware for a specific slave channel to indicate the specified data buffers, and enabling the specific hardware channel. The **d_map_slave** kernel service is not an exported kernel service, but a bus–specific utility routine determined by the **d_map_init** kernel service and provided to the caller through the **d_handle** structure.

This service allows for scatter/gather capability of the slave DMA controller and also allows the device driver to coalesce multiple requests that are contiguous with respect to the device. The list is passed with the **dio** structure. If the **d_map_slave** kernel service is unable to complete the mapping due to resource, an error, **DMA_NORES** is returned, and the **bytes_done** field of the **dio** list is set to the number of bytes that were successfully mapped. This byte count is guaranteed to be a multiple of the *minxfer* parameter size of the device as provided to **d_map_slave**. Also, the *resid_iov* field is set to the index of the remaining *d_iovec* that could not be mapped. Unless the **DMA_BYPASS** flag is set, this service will verify access permissions to each page. If an access violation is encountered on a page within the list, an error, **DMA_NOACC** is returned and no mapping is done. The *bytes_done* field of the virtual list is set to the number of bytes preceding the faulting *iovec*. Also in this case, the *resid_iov* field is set to the index of the *d_iovec* entry that encountered the access violation.

The virtual addresses provided in the *vlist* parameter can be within multiple address spaces, distinguished by the cross–memory structure pointed to for each element of the **dio** list. Each cross–memory pointer can point to the same cross–memory descriptor for multiple buffers in the same address space, and for global space buffers, the pointers can be set to the address of the exported GLOBAL cross–memory descriptor, *xmem_global*.

The *minxfer* parameter specifies the absolute minimum data transfer supported by the device(the device blocking factor). If the device supports a minimum transfer of 512 bytes (floppy and disks, for example), the *minxfer* parameter would be set to 512. This allows the underlying services to map partial transfers to a correct multiple of the device block size.

Notes:

1. The **d_map_slave** kernel service does not support more than one outstanding DMA transfer per channel. Attempts to do multiple slave mappings on a single channel will corrupt the previous mappings.
2. You can use the **D_MAP_SLAVE** macro provided in the /**usr**/**include**/**sys**/**dma.h** file to code calls to the **d_map_clear** kernel service.
3. The possible flag values for the *chan_flag* parameter can be found in /**usr**/**include**/**sys**/**dma.h**. These flags can be logically ORed together to reflect the desired characteristics of the device and channel.
4. If the **CH_AUTOINIT** flag is used then the transfer described by the **vlist** pointer is limited to a single buffer address with a length no greater than 4K bytes.

Return Values

DMA_NORES	Indicates that resources were exhausted during the mapping.
DMA_NOACC	Indicates no access permission to a page in the list.
DMA_BAD_MODE	Indicates that the mode specified by the *chan_flag* parameter is not supported.

Implementation Specifics

The **d_map_clear** kernel service is part of the base device package of your platform.

Related Information

The **d_map_init** kernel service.

d_mask Kernel Service

Purpose

Disables a direct memory access (DMA) channel.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/dma.h>

void d_mask (channel_id)
int channel_id;
```

Parameter

channel_id	Specifies the DMA channel. This parameter is returned by the **d_init** service.

Description

The **d_mask** kernel service disables the DMA channel specified by the *channel_id* parameter.

The **d_mask** kernel service is typically called by a device driver deallocating the resources associated with its device. Some devices require it to be used during normal device operation to control DMA requests and avoid spurious DMA operations.

Note: The **d_mask** service, like all DMA services, should not be called unless the **d_init** service has allocated the DMA channel.

Execution Environment

The **d_mask** kernel service can be called from either the process or interrupt environment.

Return Values

The **d_mask** service has no return values.

Implementation Specifics

The **d_mask** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **d_init** kernel service, **d_unmask** kernel service.

I/O Kernel Services and Understanding Direct Memory Access (DMA) Transfers in *AIX Kernel Extensions and Device Support Programming Concepts*.

d_master Kernel Service

Purpose

Initializes a block–mode direct memory access (DMA) transfer for a DMA master.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/dma.h>
#include <sys/xmem.h>

void d_master
(channel_id, flags, baddr, count,
dp, daddr)
int channel_id;
int flags;
caddr_t baddr;
size_t count;
struct xmem *dp;
caddr_t daddr;
```

Parameters

channel_id	Specifies the DMA channel identifier returned by the **d_init** service.
flags	Specifies the flags that control the DMA transfer. These flags are described in the **/usr/include/sys/dma.h** file.
baddr	Designates the address of the memory buffer.
count	Indicates the length of the transfer in bytes.
dp	Specifies the address of the cross–memory descriptor.
daddr	Specifies the address used to program the DMA master.

Description

The **d_master** kernel service sets up the DMA channel specified by the *channel_id* parameter to perform a block–mode DMA transfer for a DMA master. The *flags* parameter controls the operation of the **d_master** service. ”Understanding Direct Memory Access (DMA) Transfers” in *AIX Kernel Extensions and Device Support Programming Concepts* describes DMA slaves and masters.

The **d_master** service initializes all the hardware facilities for a DMA transfer, but does not initiate the DMA transfer itself. The **d_master** service makes the specified system memory buffer available to the DMA device. The **d_unmask** service may need to be called before the DMA transfer is initiated. The **d_master** service does not enable or disable the specified DMA channel.

The **d_master** service supports three different buffer locations:

- A transfer between a buffer in user memory and the device. With this type of transfer, the *dp* parameter specifies the cross–memory descriptor used with the **xmattach** service to attach to the user buffer. The *baddr* and *count* parameters must be the same values as the *uaddr* and *count* parameters specified to the **xmattach** service.
- A transfer between a global kernel memory buffer and the device. With this type of transfer, the *dp*–>**aspace_id** variable has an **XMEM_GLOBAL** value.
- A transfer between I/O bus memory and the device. The **BUS_DMA** flag distinguishes this type of transfer from the other two types. The *dp* parameter is ignored with this type of transfer and should be set to null.

The DMA transfer starts at the *daddr* parameter bus address. The device driver should allocate only a bus address in the window associated with its DMA channel. The size and location of the window are assigned to the device during the configuration process.

Note: The device driver should ensure that the *daddr* parameter bus address provided to the device includes the page offset (low 12 bits) of the *baddr* parameter memory–buffer address.

The **d_master** service performs any required machine–dependent processing, including the following tasks:

- Managing processor memory cache.
- Updating the referenced and changed bits of memory pages involved in the transfer.
- Making the DMA buffer in memory inaccessible to the processor.

If the **DMA_WRITE_ONLY** flag is set in the *flags* parameter, the pages involved in the DMA transfer can be read by the device but cannot be written. In addition, the pages involved in the transfer are not hidden from the processor and remain accessible while the pages are a source for DMA.

If the **DMA_WRITE_ONLY** flag is not set, the pages mapped for the DMA transfer are hidden from the processor. The pages remain inaccessible to the processor until the corresponding **d_complete** service has been issued once the pages are no longer required for DMA processing.

Notes:

1. When calling the **d_master** service several times for one or more of the same pages of memory, the corresponding number of **d_complete** calls must be made to unhide successfully the page or pages involved in the DMA transfers. Pages are not hidden from the processor during the DMA mapping if the **DMA_WRITE_ONLY** flag is specified on the call to the **d_master** service.
2. The memory buffer must remain pinned once the **d_master** service is called until the DMA transfer is completed and the **d_complete** service is called.
3. The device driver must not access the buffer once the **d_master** service is called until the DMA transfer is completed and the **d_complete** service is called.
4. The **d_master** service, as with all DMA services, should not be called unless the DMA channel has been allocated with the **d_init** service.

Execution Environment

The **d_master** kernel service can be called from either the process or interrupt environment.

Implementation Specifics

The **d_master** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **d_complete** kernel service, **d_init** kernel service, **d_unmask** kernel service, **xmattach** kernel service.

I/O Kernel Services and Understanding Direct Memory Access (DMA) Transfers in *AIX Kernel Extensions and Device Support Programming Concepts*.

d_move Kernel Service

Purpose

Provides consistent access to system memory that is accessed asynchronously by both a device and the processor on the system.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/dma.h>

int d_move
(channel_id, flags, baddr, count,
dp, daddr)
int channel_id;
int flags;
void *baddr;
size_t count;
struct xmem *dp;
void *daddr;
```

Parameters

channel_id	Specifies the DMA channel ID returned by the **d_init** service.
flags	Specifies the flags that designate the direction of the move. The *flags* parameter should be set to 0 if the move is to be a write into system memory shared by a bus master device. The *flags* parameter should be set to the **DMA_READ** value if the move is to be a read from system memory shared by a bus master device. These flag values are defined in the /**usr**/**include**/**sys**/**dma.h** file.
baddr	Specifies the address of the nonshared buffer. This buffer is either the source buffer for a move to the shared buffer or the destination buffer for a move from the shared buffer. This buffer area must have an associated cross–memory descriptor attached, which is specified by the *dp* parameter.
count	Specifies the length of the transfer in bytes.
dp	Specifies the address of the cross–memory descriptor associated with the buffer that is not shared by a device. This buffer is the source buffer for a move to the shared buffer and is the destination buffer for a move from the shared buffer.
daddr	Specifies the address of the system memory buffer that is shared with the bus master device. A bus address region containing this address (which consists of the address specified by the *daddr* parameter plus at least the number of bytes specified by the *count* parameter) must have been mapped for direct memory access (DMA) by using the **d_master** service.

Description

Device handlers can use the **d_move** kernel service to access a data area in system memory that is also being accessed by a (DMA) direct memory access master. The **d_move** service uses the same I/O controller data buffers that the DMA master uses when accessing data from the shared data area in system memory. Using the same buffer keeps the processor data accesses and device data access consistent. On the system platform, this is necessary since the I/O controller provides buffer caching of data accessed by bus master devices.

A cross–memory descriptor obtained by using the **xmattach** service and a buffer address must be provided for the nonshared buffer involved in the data transfer. The **d_move** service moves the data from the nonshared buffer to the shared buffer when the *flags* parameter is set to 0. A move of the data from the shared buffer to the nonshared buffer occurs if the *flags* parameter is specified with a value of **DMA_READ**. Once the **d_move** service has returned, a call to the **d_complete** service with the specified *channel_id* parameter ensures that the **d_move** service has successfully moved the data.

Note: The **d_move** service is not supported on all the system models. If the system model is cache–consistent, the **d_move** service returns an **EINVAL** value indicating that the service is not supported. The caller should assume the system model is I/O Channel Controller (IOCC) cache–consistent and perform a direct access to the target memory.

Execution Environment

The **d_move** kernel service can be called from either the process or interrupt environment.

Return Values

EINVAL	Indicates that the **d_move** service is not supported.
XMEM_SUCC	Indicates successful completion.
XMEM_FAIL	Indicates one of these errors:

- The caller does not have appropriate access authority for the nonshared buffer.
- The nonshared buffer is located in an address range that is not valid.
- The memory region containing the nonshared buffer has been deleted.
- The cross–memory descriptor is not valid.
- A paging I/O error occurred while accessing the nonshared buffer.

An error can also occur when the **d_move** kernel service executes on an interrupt level if the nonshared buffer is not in memory.

Implementation Specifics

The **d_move** kernel service is part of Base Operating System (BOS) Runtime.

The **d_move** kernel service is available only on the system product platform.

Related Information

The **d_complete** kernel service, **d_init** kernel service, **d_master** kernel service, **xmattach** kernel service.

I/O Kernel Services and Understanding Direct Memory Access (DMA) Transfers in *AIX Kernel Extensions and Device Support Programming Concepts*.

dmp_add Kernel Service

Purpose

Specifies data to be included in a system dump by adding an entry to the master dump table.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/dump.h>

int dmp_add
(cdt_func)
struct cdt * ( (*cdt_func) (  ));
```

Parameter

cdt_func	Specifies a function that returns a pointer to a component dump table entry. The function and the component dump table entry both must reside in pinned global memory.

Description

Kernel extensions use the **dmp_add** service to register data areas to be included in a system dump. The **dmp_add** service adds an entry to the master dump table. A master dump table entry is a pointer to a function provided by the kernel extension that will be called by the kernel dump routine when a system dump occurs. The function must return a pointer to a component dump table structure.

When a dump occurs, the kernel dump routine calls the function specified by the *cdt_func* parameter twice. On the first call, an argument of 1 indicates that the kernel dump routine is starting to dump the data specified by the component dump table. On the second call, an argument of 2 indicates that the kernel dump routine has finished dumping the data specified by the component dump table. Kernel extensions should allocate and pin their component dump tables and call the **dmp_add** service during initialization. The entries in the component dump table can be filled in later. The *cdt_func* routine must not attempt to allocate memory when it is called.

The Component Dump Table

The component dump table structure specifies memory areas to be included in the system dump. The structure type (**struct cdt**) is defined in the /**usr**/**include**/**sys**/**dump.h** file. A **cdt** structure consists of a fixed–length header (**cdt_head** structure) and an array of one or more **cdt_entry** structures. The **cdt_head** structure contains a component name field, which should be filled in with the name of the kernel extension, and the length of the component dump table. Each **cdt_entry** structure describes a contiguous data area, giving a pointer to the data area, its length, a segment register, and a name for the data area. The name supplied for the data area can be used to refer to it when the **crash** command formats the dump.

Use of the Formatting Routine

Each kernel extension that includes data in the system dump can install a unique formatting routine in the /**var**/**adm**/**ras**/**dmprtns** directory. A formatting routine is a command that is called by the **crash** command. The name of the formatting routine must match the component name field of the corresponding component dump table. The **crash** command forks a child process that executes the formatting routines. If a formatting routine is not present for a component name, the **crash** command executes the **_default_dmp_fmt** default formatting routine, which prints out the data areas in hexadecimal.

The **crash** command calls the formatting routine as a command, passing the file descriptor of the open dump image file as a command line argument. The syntax for this argument is **–f***file _descriptor*.

The dump image file includes a copy of each component dump table used to dump memory. Before calling a formatting routine, the **crash** command positions the file pointer for the dump image file to the beginning of the relevant component dump table copy. A sample dump formatter is shipped with **bos.sysmgt.serv_aid** in the **/usr/samples/dumpfmt** directory.

Organization of the Dump Image File

Memory dumped for each kernel extension is laid out as follows in the dump image file. The component dump table is followed by a bit map for the first data area, then the first data area itself, then a bit map for the next data area, the next data area itself, and so on.

The bit map for a given data area indicates which pages of the data area are actually present in the dump image and which are not. Pages that were not in memory when the dump occurred were not dumped. The least significant bit of the first byte of the bit map is set to 1 (one) if the first page is present. The next least significant bit indicates the presence or absence of the second page and so on.

A macro for determining the size of a bit map is provided in the **/usr/include/sys/dump.h** file.

Execution Environment

The **dmp_add** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
–1	Indicates that the function pointer to be added is already present in the master dump table.

Implementation Specifics

The **dmp_add** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **crash** command, **exec** command.

The **dmp_del** kernel service.

RAS Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

dmp_del Kernel Service

Purpose

Deletes an entry from the master dump table.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/dump.h>

dmp_del (cdt_func_ptr)
struct cdt * ( (*cdt_func_ptr) (  ));
```

Parameter

cdt_func_ptr	Specifies a function that returns a pointer to a component dump table. The function and the component dump table must both reside in pinned global memory.

Description

Kernel extensions use the **dmp_del** kernel service to unregister data areas previously registered for inclusion in a system dump. A kernel extension that uses the **dmp_add** service to register such a data area can use the **dmp_del** service to remove this entry from the master dump table.

Execution Environment

The **dmp_del** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
–1	Indicates that the function pointer to be deleted is not in the master dump table.

Implementation Specifics

The **dmp_del** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

RAS Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

The **crash** command.

The **dmp_add** kernel service.

dmp_prinit Kernel Service

Purpose

Initializes the remote dump protocol.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/dump.h>

void dmp_prinit
(dmp_proto, proto_info)
int dmp_proto;
void *proto_info;
```

Parameters

dmp_proto	Identifies the protocol. The values for the *dmp_proto* parameter are defined in the /**usr**/**include**/**sys**/**dump.h** file.
proto_info	Points to a protocol–specific structure containing information required by the system dump services. For the TCP/IP protocol, the *proto_info* parameter contains a pointer to the ARP table.

Description

When a communications subsystem is configured, it makes itself known to the system dump services by calling the **dmp_prinit** kernel service. The **dmp_prinit** kernel service identifies the protocol and passes protocol–specific information, which is required for a remote dump.

Execution Environment

The **dmp_prinit** kernel service can be called from the process environment only.

Implementation Specifics

The **dmp_prinit** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **devdump** kernel service.

The **dddump** device driver entry point.

RAS Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

d_roundup Kernel Service

Purpose

Rounds the value length up to a given number of cache lines.

Syntax

```
int  d_roundup(length)
int  length;
```

Parameter

length	Specifies the size in bytes to be rounded.

Description

To maintain cache consistency, buffers must occupy entire cache lines. The **d_roundup** service helps provide that function by rounding the value length up to a given number in integer form.

Execution Environment

The **d_roundup** service can be called from either the process or interrupt environment.

Implementation Specifics

The **d_roundup** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **d_align** kernel service, **d_cflush** kernel service, **d_clear** kernel service.

Understanding Direct Memory Access (DMA) Transfers in *AIX Kernel Extensions and Device Support Programming Concepts*.

d_slave Kernel Service

Purpose

Initializes a block–mode direct memory access (DMA) transfer for a DMA slave.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/dma.h>
#include <sys/xmem.h>

void d_slave
(channel_id, flags, baddr,
count, dp)
int channel_id;
int flags;
caddr_t baddr;
size_t count;
struct xmem *dp;
```

Parameters

channel_id	Specifies the DMA channel identifier returned by the **d_init** service.
flags	Control the DMA transfer. The /**usr**/**include**/**sys**/**dma.h** file contains valid values for these flags.
baddr	Designates the address of the memory buffer.
count	Specifies the length of the transfer in bytes.
dp	Designates the address of the cross–memory descriptor.

Description

The **d_slave** kernel service sets up the DMA channel specified by the *channel_id* parameter to perform a block–mode DMA transfer for a DMA slave. The *flags* parameter controls the operation of the **d_slave** service. ”Understanding Direct Memory Access (DMA)” in *AIX Kernel Extensions and Device Support Programming Concepts* describes DMA slaves and masters.

The **d_slave** service does not initiate the DMA transfer. The device initiates all DMA memory references. The **d_slave** service sets up the system address–generation hardware to indicate the specified buffer.

The **d_slave** service supports three different buffer locations:

- A transfer between a buffer in user memory and the device. With this type of transfer, the *dp* parameter specifies the cross–memory descriptor used with the **xmattach** service to attach to the kernel buffer. The *baddr* and *count* parameters must be the same values as the *uaddr* and *count* parameters specified to the **xmattach** service.
- A transfer between a global kernel memory buffer and the device. With this type of transfer, the *dp***–>aspace_id** variable has an **XMEM_GLOBAL** value.
- A transfer between I/O bus memory and the device. The **BUS_DMA** flag distinguishes this type of transfer from the other two types. The *dp* parameter is ignored with this type of transfer and should be set to null.

The **d_unmask** and **d_mask** services typically do not need to be called for the DMA slave transfers. The DMA channel is automatically enabled by the **d_slave** service and automatically disabled by the hardware when the last byte specified by the *count* parameter is transferred.

The **d_slave** service performs machine–dependent processing, including the following tasks:

- Flushing the processor cache
- Updating the referenced and changed bits of memory pages involved in the transfer
- Making the buffer inaccessible to the processor

Notes:

a. The memory buffer must remain pinned from the time the **d_slave** service is called until the DMA transfer is completed and the **d_complete** service is called.

b. The device driver or device handler must not access the buffer once the **d_slave** service is called until the DMA transfer is completed and the **d_complete** service is called.

c. The **d_slave** service, as with all DMA services, should not be called unless the DMA channel has been allocated with the **d_init** service.

Execution Environment

The **d_slave** kernel service can be called from either the process or interrupt environment.

Return Values

The **d_slave** service has no return values.

Implementation Specifics

The **d_slave** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **d_complete** kernel service, **d_init** kernel service, **d_mask** kernel service, **d_unmask** kernel service, **xmattach** kernel service.

I/O Kernel Services and Understanding Direct Memory Access (DMA) Transfers in *AIX Kernel Extensions and Device Support Programming Concepts*.

DTOM Macro for mbuf Kernel Services

Purpose

Converts an address anywhere within an **mbuf** structure to the head of that **mbuf** structure.

Syntax

```
#include <sys/mbuf.h>
DTOM (bp);
```

Parameter

bp	Points to an address within an **mbuf** structure.

Description

The **DTOM** macro converts an address anywhere within an **mbuf** structure to the head of that **mbuf** structure. This macro is valid only for **mbuf** structures without an external buffer (that is, with the **M_EXT** flag not set).

This macro can be viewed as the opposite of the **MTOD** macro, which converts the address of an **mbuf** structure into the address of the actual data contained in the buffer. However, the **DTOM** macro is more general than this view implies. That is, the input parameter can point to any address within the **mbuf** structure, not merely the address of the actual data.

Example

The **DTOM** macro can be used as follows:

```
char                *bp;
struct mbuf         *m;
m = DTOM(bp);
```

Implementation Specifics

The **DTOM** macro is part of Base Operating System (BOS) Runtime.

Related Information

The **MTOD** macro for **mbuf** Kernel Services.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

d_unmap_list Kernel Service

Purpose

Deallocates resources previously allocated on a **d_map_list** call.

Syntax

```
#include <sys/dma.h>

void d_unmap_list (*handle, *bus_list)
struct d_handle *handle
struct dio *bus_list
```

Note: The following is the interface definition for **d_unmap_list** when the **DMA_ADDRESS_64** and **DMA_ENABLE_64** flags are set on the **d_map_init** call.

```
void d_unmap_list (*handle,
*bus_list)
struct d_handle *handle;
struct dio_64 *bus_list;
```

Parameters

handle	Indicates the unique handle returned by the **d_map_init** kernel service.
bus_list	Specifies a list of bus addresses and lengths.

Description

The **d_unmap_list** kernel service is a bus–specific utility routine determined by the **d_map_init** kernel service that deallocates resources previously allocated on a **d_map_list** call.

The **d_unmap_list** kernel service must be called after I/O completion involving the area mapped by the prior **d_map_list** call. Some device drivers might choose to leave pages mapped for a long–term mapping of certain memory buffers. In this case, the driver must call **d_unmap_list** when it no longer needs the long–term mapping.

Note: You can use the **D_UNMAP_LIST** macro provided in the **/usr/include/sys/dma.h** file to code calls to the **d_unmap_list** kernel service. If not, you must ensure that the **d_unmap_list** function pointer is non–**NULL** before attempting the call. Not all platforms require the unmapping service.

Implementation Specifics

The **d_unmap_list** kernel service is part of the base device package of your platform.

Related Information

The **d_map_init** kernel service, **d_map_list** kernel service.

d_unmap_slave Kernel Service

Purpose

Deallocates resources previously allocated on a **d_map_slave** call.

Syntax

```
#include <sys/dma.h>

int d_unmap_slave (*handle)
struct d_handle *handle;
```

Parameters

handle	Indicates the unique handle returned by the **d_map_init** kernel service.

Description

The **d_unmap_slave** kernel service deallocates resources previously allocated on a **d_map_slave** call, disables the physical DMA channel, and returns error and status information following the DMA transfer. The **d_unmap_slave** kernel service is not an exported kernel service, but a bus–specific utility routine that is determined by the **d_map_init** kernel service and provided to the caller through the **d_handle** structure.

Note: You can use the **D_UNMAP_SLAVE** macro provided in the **/usr/include/sys/dma.h** file to code calls to the **d_unmap_slave** kernel service. If not, you must ensure that the **d_unmap_slave** function pointer is non–**NULL** before attempting to call. No all platforms require the unmapping service.

The device driver must call **d_unmap_slave** when the I/O is complete involving a prior mapping by the **d_map_slave** kernel service.

Note: The **d_unmap_slave** kernel should be paired with a previous **d_map_slave** call. Multiple outstanding slave DMA transfers are not supported. This kernel service assumes that there is no DMA in progress on the affected channel and deallocates the current channel mapping.

Return Values

DMA_SUCC	Indicates successful transfer. The DMA controller did not report any errors and that the Terminal Count was reached.
DMA_TC_NOTREACHED	Indicates a successful partial transfer. The DMA controller reported the Terminal Count reached for the intended transfer as set up by the **d_map_slave** call. Block devices consider this an erro; however, for variable length devices this may not be an error.
DMA_FAIL	Indicates that the transfer failed. The DMA controller reported an error. The device driver assumes the transfer was unsuccessful.

Implementation Specifics

The **d_unmap_slave** kernel service is part of the base device package of your platform.

Related Information

The **d_map_init** kernel service.

d_unmap_page Kernel Service

Purpose

Deallocates resources previously allocated on a **d_unmap_page** call.

Syntax

```
#include <sys/dma.h>

void d_unmap_page (*handle, *busaddr)
struct d_handle *handle
uint *busaddr
```

Note: The following is the interface definition for **d_unmap_page** when the **DMA_ADDRESS_64** and **DMA_ENABLE_64** flags are set on the **d_map_init** call.

```
int d_unmap_page(*handle,
*busaddr)
struct d_handle *handle;
unsigned long long *busaddr;
```

Parameters

handle	Indicates the unique handle returned by the **d_map_init** kernel service.
busaddr	Points to the *busaddr* field.

Description

The **d_unmap_page** kernel service is a bus–specific utility routine determined by the **d_map_init** kernel service that deallocates resources previously allocated on a **d_map_page** call for a DMA master device.

The **d_unmap_page** service must be called after I/O completion involving the area mapped by the prior **d_map_page** call. Some device drivers might choose to leave pages mapped for a long–term mapping of certain memory buffers. In this case, the driver must call **d_unmap_page** when it no longer needs the long–term mapping.

Note: You can use the **D_UNMAP_PAGE** macro provided in the **/usr/include/sys/dma.h** file to code calls to the **d_unmap_page** kernel service. If not, you must ensure that the **d_unmap_page** function pointer is non–**NULL** before attempting the call. Not all platforms require the unmapping service.

Implementation Specifics

The **d_unmap_page** kernel service is part of the base device package of your platform.

Related Information

The **d_map_init** kernel service.

d_unmask Kernel Service

Purpose

Enables a direct memory access (DMA) channel.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/dma.h>

void d_unmask (channel_id)
int channel_id;
```

Parameter

channel_id	Indicates the DMA channel identifier returned by the **d_init** service.

Description

The **d_unmask** service enables the DMA channel specified by the *channel_id* parameter. A DMA channel must be enabled before a DMA transfer can occur.

The **d_unmask** kernel service is typically called by a device driver when allocating the resources associated with its device. Some devices require it to be used during normal device operation.

Note: The **d_unmask** service, as with all DMA services, should not be called unless the DMA channel has been successfully allocated with the **d_init** service.

Execution Environment

The **d_unmask** kernel service can be called from either the process or interrupt environment.

Return Values

The **d_unmask** service has no return values.

Implementation Specifics

The **d_unmask** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **d_complete** kernel service, **d_init** kernel service, **d_mask** kernel service.

I/O Kernel Services and Understanding Direct Memory Access (DMA) Transfers in *AIX Kernel Extensions and Device Support Programming Concepts*.

e_assert_wait Kernel Service

Purpose

Asserts that the calling kernel thread is going to sleep.

Syntax

```
#include <sys/sleep.h>

void e_assert_wait (event_word, interruptible)
int *event_word;
boolean_t interruptible;
```

Parameters

event_word	Specifies the shared event word. The kernel uses the *event_word* parameter as the anchor to the list of threads waiting on this shared event.
interruptible	Specifies if the sleep is interruptible.

Description

The **e_assert_wait** kernel service asserts that the calling kernel thread is about to be placed on the event list anchored by the *event_word* parameter. The *interruptible* parameter indicates wether the sleep can be interrupted.

This kernel service gives the caller the opportunity to release multiple locks and sleep atomically without losing the event should it occur. This call is typically followed by a call to either the **e_clear_wait** or **e_block_thread** kernel service. If only a single lock needs to be released, then the **e_sleep_thread** kernel service should be used instead.

The **e_assert_wait** kernel service has no return values.

Execution Environment

The **e_assert_wait** kernel service can be called from the process environment only.

Implementation Specifics

The **e_assert_wait** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **e_clear_wait** kernel service, **e_block_thread** kernel service, **e_sleep_thread** kernel service

Process and Exception Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

e_block_thread Kernel Service

Purpose

Blocks the calling kernel thread.

Syntax

```
#include <sys/sleep.h>

int e_block_thread ()
```

Description

The **e_block_thread** kernel service blocks the calling kernel thread. The thread must have issued a request to sleep (by calling the **e_assert_wait** kernel service). If it has been removed from its event list, it remains runnable.

Execution Environment

The **e_block_thread** kernel service can be called from the process environment only.

Return Values

The **e_block_thread** kernel service return a value that indicate how the thread was awakened. The following values are defined:

THREAD_AWAKENED	Denotes a normal wakeup; the event occurred.
THREAD_INTERRUPTED	Denotes an interruption by a signal.
THREAD_TIMED_OUT	Denotes a timeout expiration.
THREAD_OTHER	Delineates the predefined system codes from those that need to be defined at the subsystem level. Subsystem should define their own values greater than or equal to this value.

Implementation Specifics

The **e_block_thread** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **e_assert_wait** kernel service.

Process and Exception Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

e_clear_wait Kernel Service

Purpose

Clears the wait condition for a kernel thread.

Syntax

```
#include <sys/sleep.h>
void e_clear_wait (tid, result)
tid_t tid;
int result;
```

Parameters

tid Specifies the kernel thread to be awakened.

result Specifies the value returned to the awakened kernel thread. The following values can be used:

THREAD_AWAKENED
Usually generated by the **e_wakeup** or **e_wakeup_one** kernel service to indicate a normal wakeup.

THREAD_INTERRUPTED
Indicates an interrupted sleep. This value is usually generated by a signal delivery when the **INTERRUPTIBLE** flag is set.

THREAD_TIMED_OUT
Indicates a timeout expiration.

THREAD_OTHER
Delineates the predefined system codes from those that need to be defined at the subsystem level. Subsystem should define their own values greater than or equal to this value.

Description

The **e_clear_wait** kernel service clears the wait condition for the kernel thread specified by the *tid* parameter, and the thread is made runnable.

This kernel service differs from the **e_wakeup**, **e_wakeup_one**, and **e_wakeup_w_result** kernel services in the fact that it assumes the identity of the thread to be awakened. This kernel service should be used to handle exceptional cases, where a special action needs to be taken. The *result* parameter is used to specify the value returned to the awakened thread by the **e_block_thread** or **e_sleep_thread** kernel service.

The **e_clear_wait** kernel service has no return values.

Execution Environment

The **e_clear_wait** kernel service can be called from either the process environment or the interrupt environment.

Implementation Specifics

The **e_clear_wait** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **e_wakeup**, **e_wakeup_one**, or **e_wakeup_w_result** kernel services, **e_block_thread** kernel servic, **e_sleep_thread** kernel service.

Process and Exception Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

enque Kernel Service

Purpose

Sends a request queue element to a device queue.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/deviceq.h>

int enque (qe)
struct req_qe *qe;
```

Parameter

qe	Specifies the address of the request queue element.

Description

The **enque** kernel service is not part of the base kernel, but is provided by the device queue management kernel extension. This queue management kernel extension must be loaded into the kernel before loading any kernel extensions referencing these services.

The **enque** service places the queue element into a specified device queue. It is used for simple process–to–process communication within the kernel. The requester builds a copy of the queue element, indicated by the *qe* parameter, and passes this copy to the **enque** service. The kernel copies this queue element into a queue element in pinned global memory and then enqueues it on the target device queue.

The path identifier in the request queue element indicates the device queue into which the element is placed.

The **enque** service supports the sending of the following types of queue elements:

START_IO	Start I/O.
GEN_PURPOSE	General purpose.

For simple interprocess communication, general purpose queue elements are used.

The queue element priority value can range from **QE_BEST_PRTY** to **QE_WORST_PRTY**. This value is limited to the value specified when the queue was created.

The operation options in the queue element control how the queue element is processed. There are five standard operation options:

ACK_COMPLETE	Acknowledge completion in all cases.
ACK_ERRORS	Acknowledge completion if the operation results in an error.
SYNC_REQUEST	Synchronous request.
CHAINED	Chained control blocks.
CONTROL_OPT	Kernel control operation.

Note: Only one of **ACK_COMPLETE**, **ACK_ERRORS**, or **SYNC_REQUEST** can be specified. Also, all of these options are ignored if the path specifies that no acknowledgment (**NO_ACK**) should be sent.

With the **SYNC_REQUEST** synchronous request option, control does not return from the **enque** service until the request queue element is acknowledged. This performs in one step

what can also be achieved by sending a queue element with the **ACK_COMPLETE** flag on, and then calling either the **et_wait** or **waitq** kernel services.

The kernel calls the server's **check** routine, if one is defined, before a queue element is placed on the device queue. This routine can stop the operation if it detects an error.

The kernel notifies the device queue's server, if necessary, after a queue element is placed on the device queue. This is done by posting the server process (using the **et_post** kernel service) with an event control bit.

Execution Environment

The **enque** kernel service can be called from the process environment only.

Return Values

RC_GOOD	Indicates a successful operation.
RC_ID	Indicates a path identifier that is not valid.

All other error values represent errors returned by the server.

Implementation Specifics

The **enque** kernel service is part of the Device Queue Management kernel extension.

Related Information

The **et_post** kernel service, **et_wait** kernel service, **waitq** kernel service.

The **check** device queue management routine.

errsave or errlast Kernel Service

Purpose

Allows the kernel and kernel extensions to write to the error log.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/errids.h>

void errsave (buf, cnt)
char *buf;
unsigned int cnt;

void errlast (buf, cnt)
char *buf
unsigned int cnt;
```

Parameters

buf	Points to a buffer that contains an error record as described in the **/usr/include/sys/err_rec.h** file.
cnt	Specifies the number of bytes in the error record contained in the buffer pointed to by the *buf* parameter.

Description

The **errsave** kernel service allows the kernel and kernel extensions to write error log entries to the error device driver. The error record pointed to by the *buf* parameter includes the error ID resource name and detailed data.

In addition, the **errlast** kernel service disables any future error logging, thus any error logged with **errlast** will stay on NVRAM. This service is only for use prior to a pending system crash or stop. The **errlast** service should only be used in extreme circumstances where the system can not continue, such as the occurance of a machine check.

Execution Environment

The **errsave** kernel service can be called from either the process or interrupt environment.

Return Values

The **errsave** service has no return values.

Implementation Specifics

The **errsave** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **errlog** subroutine.

For more information on error device drivers, see Error Logging Special Files in *AIX Files Reference*.

RAS Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

e_sleep Kernel Service

Purpose

Forces the calling kernel thread to wait for the occurrence of a shared event.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sleep.h>

int e_sleep (event_word, flags)
int *event_word;
int flags;
```

Parameters

event_word Specifies the shared event word. The kernel uses the *event_word* parameter to anchor the list of processes sleeping on this event. The *event_word* parameter must be initialized to **EVENT_NULL** before its first use.

flags Specifies the flags that control action on occurrence of signals. These flags can be found in the **/usr/include/sys/sleep.h** file. The *flags* parameter is used to control how signals affect waiting for an event. The following flags are available to the **e_sleep** service:

EVENT_SIGRET
Indicates the termination of the wait for the event by an unmasked signal. The return value is set to **EVENT_SIG**.

EVENT_SIGWAKE
Indicates the termination of the event by an unmasked signal. This flag results in the transfer of control to the return from the last **setjmpx** service with the return value set to **EINTR**.

EVENT_SHORT
Prohibits the wait from being terminated by a signal. This flag should only be used for short, guaranteed–to–wakeup sleeps.

Description

The **e_sleep** kernel service is used to wait for the specified shared event to occur. The kernel places the current kernel thread on the list anchored by the *event_word* parameter. This list is used by the **e_wakeup** service to wake up all threads waiting for the event to occur.

The anchor for the event list, the *event_word* parameter, must be initialized to **EVENT_NULL** before its first use. Kernel extensions must not alter this anchor while it is in use.

The **e_wakeup** service does not wake up a thread that is not currently sleeping in the **e_sleep** function. That is, if an **e_wakeup** operation for an event is issued before the process calls the **e_sleep** service for the event, the thread still sleeps, waiting on the next **e_wakeup** service for the event. This implies that routines using this capability must ensure that no timing window exists in which events could be missed due to the **e_wakeup** service being called before the **e_sleep** operation for the event has been called.

Note: The **e_sleep** service can be called with interrupts disabled only if the event or lock word is pinned.

Execution Environment

The **e_sleep** kernel service can be called from the process environment only.

Return Values

EVENT_SUCC	Indicates a successful operation.
EVENT_SIG	Indicates that the **EVENT_SIGRET** flag is set and the wait is terminated by a signal.

Implementation Specifics

The **e_sleep** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **e_sleepl** kernel service, **e_wakeup** kernel service.

Process and Exception Management Kernel Services and Understanding Execution Environments in *AIX Kernel Extensions and Device Support Programming Concepts*.

e_sleepl Kernel Service

Purpose

Forces the calling kernel thread to wait for the occurrence of a shared event.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sleep.h>

int e_sleepl (lock_word, event_word, flags)
int *lock_word;
int *event_word;
int flags;
```

Parameters

lock_word	Specifies the lock word for a conventional process lock.
event_word	Specifies the shared event word. The kernel uses this word to anchor the list of kernel threads sleeping on this event. This event word must be initialized to **EVENT_NULL** before its first use.
flags	Specifies the flags that control action on occurrence of a signal. These flags are found in the /**usr/include/sys/sleep.h** file.

Description

Note: The **e_sleepl** kernel service is provided for porting old applications written for previous versions of the operating system. Use the **e_sleep_thread** kernel service when writing new applications.

The **e_sleepl** kernel service waits for the specified shared event to occur. The kernel places the current kernel thread on the list anchored by the *event_word* parameter. The **e_wakeup** service wakes up all threads on the list.

The **e_wakeup** service does not wake up a thread that is not currently sleeping in the **e_sleepl** function. That is, if an **e_wakeup** operation for an event is issued before the thread calls the **e_sleepl** service for the event, the thread still sleeps, waiting on the next **e_wakeup** operation for the event. This implies that routines using this capability must ensure that no timing window exists in which events could be missed due to the **e_wakeup** service being called before the **e_sleepl** service for the event has been called.

The **e_sleepl** service also unlocks the conventional lock specified by the *lock_word* parameter before putting the thread to sleep. It also reacquires the lock when the thread wakes up.

The anchor for the event list, specified by the *event_word* parameter, must be initialized to **EVENT_NULL** before its first use. Kernel extensions must not alter this anchor while it is in use.

Note: The **e_sleepl** service can be called with interrupts disabled, only if the event or lock word is pinned.

Values for the flags Parameter

The *flags* parameter controls how signals affect waiting for an event. There are three flags available to the **e_sleepl** service:

EVENT_SIGRET	Indicates the termination of the wait for the event by an unmasked signal. The return value is set to EVENT_SIG.
EVENT_SIGWAKE	Indicates the termination of the event by an unmasked signal. This flag also indicates the transfer of control to the return from the last **setjmpx** service with the return value set to **EINTR**.
EVENT_SHORT	Indicates that signals cannot terminate the wait. Use the **EVENT_SHORT** flag for only short, guaranteed–to–wakeup sleeps.

Note: The **EVENT_SIGRET** flag overrides the **EVENT_SIGWAKE** flag.

Execution Environment

The **e_sleepl** kernel service can be called from the process environment only.

Return Values

EVENT_SUCC	Indicates successful completion.
EVENT_SIG	Indicates that the **EVENT_SIGRET** flag is set and the wait is terminated by a signal.

Implementation Specifics

The **e_sleepl** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **e_sleep** kernel service, **e_wakeup** kernel service.

Process and Exception Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

Interrupt Environment in *AIX Kernel Extensions and Device Support Programming Concepts*.

e_sleep_thread Kernel Service

Purpose

Forces the calling kernel thread to wait for the occurrence of a shared event.

Syntax

```
#include <sys/sleep.h>

int e_sleep_thread (event_word, lock_word, flags)
int *event_word;
void *lock_word;
int flags;
```

Parameters

event_word	Specifies the shared event word. The kernel uses the *event_word* parameter as the anchor to the list of threads waiting on this shared event.
lock_word	Specifies simple or complex lock to unlock.
flags	Specifies lock and signal handling options.

Description

The **e_sleep_thread** kernel service forces the calling thread to wait until a shared event occurs. The kernel places the calling thread on the event list anchored by the *event_word* parameter. This list is used by the **e_wakeup**, **e_wakeup_one**, and **e_wakeup_w_result** kernel services to wakeup some or all threads waiting for the event to occur.

A lock can be specified; it will be unlocked when the kernel service is entered, just before the thread blocks. This lock can be a simple or a complex lock, as specified by the *flags* parameter. When the kernel service exits, the lock is re–acquired.

Flags

The *flags* parameter specifies options for the kernel service. Several flags can be combined with the bitwise OR operator. They are described below.

The four following flags specify the lock type. If the *lock_word* parameter is not **NULL**, exactly one of these flags must be used.

LOCK_HANDLER	*lock_word* specifies a simple lock protecting a thread–interrupt or interrupt–interrupt critical section.
LOCK_SIMPLE	*lock_word* specifies a simple lock protecting a thread–thread critical section.
LOCK_READ	*lock_word* specifies a complex lock in shared–read mode.
LOCK_WRITE	*lock_word* specifies a complex lock in exclusive write mode.

The following flag specify the signal handling. By default, while the thread sleeps, signals are held pending until it wakes up.

INTERRUPTIBLE	The signals must be checked while the kernel thread is sleeping. If a signal needs to be delivered, the thread is awakened.

Return Values

The **e_sleep_thread** kernel service return a value that indicate how the kernel thread was awakened. The following values are defined:

THREAD_AWAKENED	Denotes a normal wakeup; the event occurred.
THREAD_INTERRUPTED	Denotes an interruption by a signal. This value can be returned even if the **INTERRUPTIBLE** flag is not set since it may be also generated by the **e_clear_wait** or **e_wakeup_w_result** kernel services.
THREAD_TIMED_OUT	Denotes a timeout expiration. The **e_sleep_thread** has no timeout. However, the **e_clear_wait** or **e_wakeup_w_result** kernel services may generate this return value.
THREAD_OTHER	Delineates the predefined system codes from those that need to be defined at the subsystem level. Subsystem should define their own values greater than or equal to this value.

Execution Environment

The **e_sleep_thread** kernel service can be called from the process environment only.

Implementation Specifics

The **e_sleep_thread** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **e_wakeup**, **e_wakeup_one**, or **e_wakeup_w_result** kernel services, **e_block_thread** kernel service, **e_clear_wait** kernel service.

Process and Exception Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

Understanding Locking in *AIX Kernel Extensions and Device Support Programming Concepts*

Locking Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*

et_post Kernel Service

Purpose

Notifies a kernel thread of the occurrence of one or more events.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sleep.h>

void et_post (events, tid)
unsigned long events;
tid_t tid;
```

Parameters

events	Identifies the masks of events to be posted.
tid	Specifies the thread identifier of the kernel thread to be notified.

Description

The **et_post** kernel service is used to notify a kernel thread that one or more events occurred.

The **et_post** service provides the fastest method of interprocess communication, although only the event numbers are passed.

The event numbers must be known by the cooperating components, either through programming convention or the passing of initialization parameters.

The **et_post** service is performed automatically when sending a request to a device queue serviced by a kernel thread or when sending an acknowledgment.

The **EVENT_KERNEL** mask defines the event bits reserved for use by the kernel. For example, a bit with a value of 1 indicates an event bit reserved for the kernel. Kernel extensions should assign their events starting with the most significant bits and working down. If threads using the **et_post** service are also using the device queue management kernel extensions, care must be taken not to use the event bits registered for device queue management.

The **et_wait** service does not sleep but returns immediately if a specified event has already been posted by the **et_post** service.

Execution Environment

The **et_post** kernel service can be called from either the process or interrupt environment.

Return Values

The **et_post** service has no return values.

Implementation Specifics

The **et_post** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **et_wait** kernel service.

Process and Exception Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

et_wait Kernel Service

Purpose

Forces the calling kernel thread to wait for the occurrence of an event.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sleep.h>

unsigned long
 et_wait (wait_mask, clear_mask, flags)
unsigned long wait_mask;
unsigned long clear_mask;
int flags;
```

Parameters

wait_mask Specifies the mask of events to await.

clear_mask Specifies the mask of events to clear.

flags Specifies the flags controling actions on occurrence of a signal. The *flags* parameter is used to control how signals affect waiting for an event. There are two flag values:

EVENT_SIGRET
Causes the wait for the event to be ended by an unmasked signal and the return value set to **EVENT_SIG**.

EVENT_SIGWAKE
Causes the event to be ended by an unmasked signal and control transferred to the return from the last **setjmpx** call, with the return value set to **EXSIG**.

EVENT_SHORT
Prohibits the wait from being terminated by a signal. This flag should only be used for short, guaranteed–to–wakeup sleeps.

Note: The **EVENT_SIGRET** flag overrides the **EVENT_SIGWAKE** flag.

Description

The **et_wait** kernel service forces the calling kernel thread to wait for specified events to occur.

The *wait_mask* parameter indicates a mask, where each bit set equal to 1 represents an event for which the thread must wait. The *clear_mask* parameter indicates a mask of events that must clear when the wait is complete. Subsequent calls to the **et_wait** service return immediately unless you clear the bits, which ends the wait.

Note: The **et_wait** service can be called with interrupts disabled only if the event or lock word is pinned.

Strategies for Using et_wait

Calling the **et_wait** kernel service with the **EVENT_SIGRET** flag clears the the pending events field when the signal is received. If **et_wait** is called again by the same kernel thread, the thread waits indefinitely for an event that has already occurred. When this

happens, the thread does not run to completion. This problem occurs only if the event and signal are posted at the same time.

To avoid this problem, use one of the following programming methods:

- Use the **EVENT_SHORT** flag to prevent signals from waking the thread up.
- Mask signals prior to the call of **et_wait** by using the limit_sigs kernel service. Then call **et_wait**. Invoke the sigprocmask call to restore the signal mask by using the mask returned previously by limit_sigs.

The **et_wait** service is also used to clear events without waiting for them to occur. This is accomplished by doing one of the following:

- Set the *wait_mask* parameter to **EVENT_NDELAY**.
- Set the bits in the *clear_mask* parameter that correspond with the events to be cleared to 1.

Because the **et_wait** service returns an event mask indicating those events that were actually cleared, these methods can be used to poll the events.

Execution Environment

The **et_wait** kernel service can be called from the process environment only.

Return Values

Upon successful completion, the **et_wait** service returns an event mask indicating the events that terminated the wait. If an **EVENT_NDELAY** value is specified, the returned event mask indicates the pending events that were cleared by this call. Otherwise, it returns the following error code:

EVENT_SIG	Indicates that the **EVENT_SIGRET** flag is set and the wait is terminated by a signal.

Implementation Specifics

The **et_wait** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **et_post** kernel service, **setjmpx** kernel service.

e_wakeup, e_wakeup_one, or e_wakeup_w_result Kernel Service

Purpose

Notifies kernel threads waiting on a shared event of the event's occurrence.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sleep.h>

void e_wakeup (event_word)
int *event_word;

void e_wakeup_one (event_word)
int *event_word;

void e_wakeup_w_result (event_word, result)
int *event_word;
int result;
```

Parameters

event_word Specifies the shared event designator. The kernel uses the *event_word* parameter as the anchor to the list of threads waiting on this shared event.

result Specifies the value returned to the awakened kernel thread. The following values can be used:

THREAD_AWAKENED
Indicates a normal wakeup. This is the value automatically generated by the **e_wakeup** or **e_wakeup_one** kernel services.

THREAD_INTERRUPTED
Indicates an interrupted sleep. This value is usually generated by a signal delivery when the **INTERRUPTIBLE** flag is set.

THREAD_TIMED_OUT
Indicates a timeout expiration.

THREAD_OTHER
Delineates the predefined system codes from those that need to be defined at the subsystem level. Subsystem should define their own values greater than or equal to this value.

Description

The **e_wakeup** and **e_wakeup_w_result** kernel services wake up all kernel threads sleeping on the event list anchored by the *event_word* parameter. The **e_wakeup_one** kernel service wakes up only the most favored thread sleeping on the event list anchored by the *event_word* parameter.

When threads are awakened, they return from a call to either the **e_block_thread** or **e_sleep_thread** kernel service. The return value depends on the kernel service called to wake up the threads (the wake–up kernel service):

- **THREAD_AWAKENED** is returned if the **e_wakeup** or **e_wakeup_one** kernel service is called

- The value of the *result* parameter is returned if the **e_wakeup_w_result** kernel service is called.

If a signal is delivered to a thread being awakened by one of the wake–up kernel services, and if the thread specified the **INTERRUPTIBLE** flag, the signal delivery takes precedence. The thread is awakened with a return value of **THREAD_INTERRUPTED**, regardless of the called wake–up kernel service.

The **e_wakeup** and **e_wakeup_w_result** kernel services set the *event_word* parameter to **EVENT_NULL**.

The **e_wakeup**, **e_wakeup_one**, and **e_wakeup_w_result** kernel services have no return values.

Execution Environment

The **e_wakeup**, **e_wakeup_one**, and **e_wakeup_w_result** kernel services can be called from either the process environment or the interrupt environment.

When called by an interrupt handler, the *event_word* parameter must be located in pinned memory.

Implementation Specifics

The **e_wakeup**, **e_wakeup_one**, and **e_wakeup_w_result** kernel services are part of the Base Operating System (BOS) Runtime.

Related Information

The **e_block_thread** kernel service, **e_clear_wait** kernel service, **e_sleep_thread** kernel service.

Process and Exception Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

e_wakeup_w_sig Kernel Service

Purpose

Posts a signal to sleeping kernel threads.

Syntax

```
#include <sys/sleep.h>

void e_wakeup_w_sig (event_word, sig)
int *event_word;
int sig;
```

Parameters

event_word	Specifies the shared event word. The kernel uses the *event_word* parameter as the anchor to the list of threads waiting on this shared event.
sig	Specifies the signal number to post.

Description

The **e_wakeup_w_sig** kernel service posts the signal *sig* to each kernel thread sleeping interruptible on the event list anchored by the *event_word* parameter.

The **e_wakeup_w_sig** kernel service has no return values.

Execution Environment

The **e_wakeup_w_sig** kernel service can be called from either the process environment or the interrupt environment.

Implementation Specifics

The **e_wakeup_w_sig** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **e_block_thread** kernel service, **e_clear_wait** kernel service.

Process and Exception Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

fetch_and_add Kernel Service

Purpose

Increments a single word variable atomically.

Syntax

#include <sys/atomic_op.h>

int fetch_and_add (*word_addr*, *value***)**
atomic_p *word_addr***;**
int *value***;**

Parameters

word_addr	Specifies the address of the word variable to be incremented.
value	Specifies the value to be added to the word variable.

Description

The **fetch_and_add** kernel service atomically increments a single word. This operation is useful when a counter variable is shared between several kernel threads, since it ensures that the fetch, update, and store operations used to increment the counter occur atomically (are not interruptible).

Note: The word variable must be aligned on a full word boundary.

Execution Environment

The **fetch_and_add** kernel service can be called from either the process or interrupt environment.

Return Values

The **fetch_and_add** kernel service returns the original value of the word.

Implementation Specifics

The **fetch_and_add** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **fetch_and_and** kernel service, **fetch_and_or** kernel service, **compare_and_swap** kernel service.

Locking Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*

fetch_and_and or fetch_and_or Kernel Service

Purpose

Clears and sets bits in a single word variable atomically.

Syntax

#include <sys/atomic_op.h>

uint fetch_and_and (*word_addr***,** *mask***)**
atomic_p *word_addr***;**
int *mask***;**

uint fetch_and_or (*word_addr***,** *mask***)**
atomic_p *word_addr***;**
int *mask***;**

Parameters

word_addr	Specifies the address of the single word variable whose bits are to be cleared or set.
mask	Specifies the bit mask which is to be applied to the single word variable.

Description

The **fetch_and_and** and **fetch_and_or** kernel services respectively clear and set bits in one word, according to a bit mask, as a single atomic operation. The **fetch_and_and** service clears bits in the word which correspond to clear bits in the bit mask, and the **fetch_and_or** service sets bits in the word which correspond to set bits in the bit mask.

These operations are useful when a variable containing bit flags is shared between several kernel threads, since they ensure that the fetch, update, and store operations used to clear or set a bit in the variable occur atomically (are not interruptible).

Note: The word containing the bit flags must be aligned on a full word boundary.

Execution Environment

The **fetch_and_and** and **fetch_and_or** kernel services can be called from either the process or interrupt environment.

Return Values

The **fetch_and_and** and **fetch_and_or** kernel services return the original value of the word.

Implementation Specifics

The **fetch_and_and** and **fetch_and_or** kernel services are part of the Base Operating System (BOS) Runtime.

Related Information

The **fetch_and_add** kernel service, **compare_and_swap** kernel service.

Locking Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*

fidtovp Kernel Service

Purpose

Maps a file system structure to a file ID.

Maps a file identifier to a mode.

Syntax

```
#include <sys/types.h>
#include <sys/vnode.h>

int fidtovp(fsid, fid, vpp)
fsid_t *fsid;
struct fileid *fid;
struct vnode **vpp;
```

Parameters

fsid	Points to a file system ID structure. The system uses this structure to determine which virtual file system (VFS) contains the requested file.
fid	Points to a file ID structure. The system uses this pointer to locate the specific file within the VFS.
vpp	Points to a location to store the file's vnode pointer upon successful return of the **fidtovp** kernel service.

Description

The **fidtovp** kernel service returns a pointer to a vnode for the file identified by **fsid** and **fid**, and increments the count on the vnode so the file is not removed. Subroutines that call the **fidtovp** kernel service must call VNOP_RELE to release the vnode pointer.

This kernel service is designed for use by the server side of distributed file systems.

Execution Environment

The **fidtovp** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
ESTALE	Indicates the requested file or file system was removed or recreated since last access with the given file system ID or file ID.

Implementation Specifics

This kernel service is part of Base Operating System (BOS) Runtime.

find_input_type Kernel Service

Purpose

Finds the given packet type in the Network Input Interface switch table and distributes the input packet according to the table entry for that type.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <net/if.h>

int find_input_type (type, m, ac, header_pointer)
ushort type;
struct mbuf *m;
struct arpcom *ac;
caddr_t header_pointer;
```

Parameters

type	Specifies the protocol type.
m	Points to the **mbuf** buffer containing the packet to distribute.
ac	Points to the network common portion (**arpcom**) of the network interface on which the packet was received. This common portion is defined as follows: `in net/if_arp.h`
header_pointer	Points to the buffer containing the input packet header.

Description

The **find_input_type** kernel service finds the given packet type in the Network Input table and distributes the input packet contained in the **mbuf** buffer pointed to by the *m* value. The *ac* parameter is passed to services that do not have a queued interface.

Execution Environment

The **find_input_type** kernel service can be called from either the process or interrupt environment.

Return Values

0	Indicates that the protocol type was successfully found.
ENOENT	Indicates that the service could not find the type in the Network Input table.

Implementation Specifics

The **find_input_type** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **add_input_type** kernel service, **del_input_type** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

fp_access Kernel Service

Purpose

Checks for access permission to an open file.

Syntax

#include <sys/types.h>
#include <sys/errno.h>

int fp_access (*fp***,** *perm***)**
struct file ****fp*;**
int *perm***;**

Parameters

fp	Points to a file structure returned by the **fp_open** or **fp_opendev** kernel service.
perm	Indicates which read, write, and execute permissions are to be checked. The **/usr/include/sys/mode.h** file contains pertinent values (IREAD, IWRITE, IEXEC).

Description

The **fp_access** kernel service is used to see if either the read, write, or exec bit is set anywhere in a file's permissions mode. Set *perm* to one of the following constants from **mode.h**:

IREAD
IWRITE
IEXEC

Execution Environment

The **fp_access** kernel service can be called from the process environment only.

Return Values

0	Indicates that the calling process has the requested permission.
EACCES	Indicates all other conditions.

Implementation Specifics

The **fp_access** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **access** subroutine.

Logical File System Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

fp_close Kernel Service

Purpose

Closes a file.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int fp_close (fp)
struct file *fp;
```

Parameter

fp	Points to a file structure returned by the **fp_open**, **fp_getf**, or **fp_opendev** kernel service.

Description

The **fp_close** kernel service is a common service for closing files used by both the file system and routines outside the file system.

Execution Environment

The **fp_close** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.

If an error occurs, one of the values from the /**usr/include/sys/error.h** file is returned.

Implementation Specifics

The **fp_close** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **close** subroutine.

Logical File System Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

fp_close Kernel Service for Data Link Control (DLC) Devices

Purpose

Allows kernel to close the generic data link control (GDLC) device manager using a file pointer.

Syntax

int fp_close(***fp*****,** ***ext*****)**
struct file ****fp***;**

Parameters

fp	Specifies the file pointer of the GDLC being closed.
ext	Specifies the extension parameter. This parameter is ignored by GDLC.

Description

The **fp_close** kernel service disables a GDLC channel. If this is the last channel to close on a port, the GDLC device manager resets to an idle state on that port and the communications device handler is closed.

Return Values

0	Indicates a successful completion.
ENXIO	Indicates an invalid file pointer. This value is defined in the **/usr/include/sys/errno.h** file.

Implementation Specifics

Each GDLC supports the **fp_close** kernel service by way of its **dlcclose** entry point. The **fp_close** kernel service may be called from the process environment only.

Related Information

The **fp_close** kernel service.

The **fp_open** kernel service for data link control (DLC) devices.

Generic Data Link Control (GDLC) Environment Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

fp_fstat Kernel Service

Purpose

Gets the attributes of an open file.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int fp_fstat (fp, statbuf, statsz, segflag)
struct file *fp;
caddr_t statbuf;
unsigned int statsz;
unsigned int segflag;
```

Parameters

fp	Points to a file structure returned by the **fp_open** kernel service.
statbuf	Points to a buffer defined to be of **stat** or **fullstat** type structure. The *statsz* parameter indicates the buffer type.
statsz	Indicates the size of the **stat** or **fullstat** structure to be returned. The /**usr**/**include**/**sys**/**stat.h** file contains information about the **stat** structure.
segflag	Specifies the flag indicating where the information represented by the *statbuf* parameter is located: **SYS_ADSPACE** Buffer is in kernel memory. **USER_ADSPACE** Buffer is in user memory.

Description

The **fp_fstat** kernel service is an internal interface to the function provided by the **fstatx** subroutine.

Execution Environment

The **fp_fstat** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.

If an error occurs, one of the values from the /**usr**/**include**/**sys**/**errno.h** file is returned.

Implementation Specifics

The **fp_fstat** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **fstatx** subroutine.

Logical File System Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

fp_getdevno Kernel Service

Purpose

Gets the device number or channel number for a device.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/file.h>

int fp_getdevno (fp, devp, chanp)
struct file *fp;
dev_t *devp;
chan_t *chanp;
```

Parameters

fp	Points to a file structure returned by the **fp_open** or **fp_opendev** service.
devp	Points to a location where the device number is to be returned.
chanp	Points to a location where the channel number is to be returned.

Description

The **fp_getdevno** service finds the device number and channel number for an open device that is associated with the file pointer specified by the *fp* parameter. If the value of either *devp* or *chanp* parameter is null, this service does not attempt to return any value for the argument.

Execution Environment

The **fp_getdevno** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
EINVAL	Indicates that the pointer specified by the *fp* parameter does not point to a file structure for an open device.

Implementation Specifics

The **fp_getdevno** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Logical File System Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

fp_getf Kernel Service

Purpose

Retrieves a pointer to a file structure.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int fp_getf (fd, fpp)
int fd;
struct file **fpp;
```

Parameters

fd	Specifies a file descriptor.
fpp	Points to the location where the file pointer is to be returned.

Description

A process calls the **fp_getf** kernel service when it has a file descriptor for an open file, but needs a file pointer to use other Logical File System services.

The **fp_getf** kernel service uses the file descriptor as an index into the process's open file table. From this table it extracts a pointer to the associated file structure.

As a side effect of the call to the **fp_getf** kernel service, the reference count on the file descriptor is incremented. This count must be decremented when the caller has completed its use of the returned file pointer. The file descriptor reference count is decremented by a call to the **ufdrele** kernel service.

Execution Environment

The **fp_getf** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
EBADF	Indicates that either the file descriptor is invalid or not currently used in the process.

Implementation Specifics

The **fp_getf** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **ufdrele** kernel service.

Logical File System Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

fp_hold Kernel Service

Purpose

Increments the open count for a specified file pointer.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int fp_hold (fp)
struct file *fp;
```

Parameter

fp	Points to a file structure previously obtained by calling the **fp_open**, **fp_getf**, or **fp_opendev** kernel service.

Description

The **fp_hold** kernel service increments the use count in the file structure specified by the *fp* parameter. This results in the associated file remaining opened even when the original open is closed.

If this function is used, and access to the file associated with the pointer specified by the *fp* parameter is no longer required, the **fp_close** kernel service should be called to decrement the use count and close the file as required.

Execution Environment

The **fp_hold** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
EINVAL	Indicates that the *fp* parameter is not a valid file pointer.

Implementation Specifics

The **fp_hold** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Logical File System Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

fp_ioctl Kernel Service

Purpose

Issues a control command to an open device or file.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int fp_ioctl (fp, cmd, arg, ext)
struct file *fp;
unsigned int cmd;
caddr_t arg;
int ext;
```

Parameters

fp	Points to a file structure returned by the **fp_open** or **fp_opendev** kernel service.
cmd	Specifies the specific control command requested.
arg	Indicates the data required for the command.
ext	Specifies an extension argument required by some device drivers. Its content, form, and use are determined by the individual driver.

Description

The **fp_ioctl** kernel service is an internal interface to the function provided by the **ioctl** subroutine.

Execution Environment

The **fp_ioctl** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.

If an error occurs, one of the values from the /**usr/include/sys/errno.h** file is returned. The **ioctl** subroutine contains valid **errno** values.

Implementation Specifics

The **fp_ioctl** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **ioctl** subroutine.

Logical File System Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

fp_ioctl Kernel Service for Data Link Control (DLC) Devices

Purpose

Transfers special commands from the kernel to generic data link control (GDLC) using a file pointer.

Syntax

```
#include <sys/gdlextcb.h>
#include <fcntl.h>

int fp_ioctl
(fp, cmd, arg, ext)

struct file *fp;
unsigned int cmd;
caddr_t arg;
int ext;
```

Parameters

fp Specifies the file pointer of the target GDLC.

cmd Specifies the operation to be performed by GDLC. For a listing of all possible operators, see "ioctl Operations (op) for DLC"*AIX Technical Reference, Volume 3: Communications*.

arg Specifies the address of the parameter block. The argument for this parameter must be in the kernel space. For a listing of possible values, see "Parameter Blocks by ioctl Operation for DLC"*AIX Technical Reference, Volume 3: Communications*.

ext Specifies the extension parameter. This parameter is ignored by GDLC.

Description

Various GDLC functions can be initiated using the **fp_ioctl** kernel service, such as changing configuration parameters, contacting the remote, and testing a link. Most of these operations can be completed before returning to the user synchronously. Some operations take longer, so asynchronous results are returned much later using the **exception** function handler. GDLC calls the kernel user's exception handler to complete these results.

Note: The **DLC_GET_EXCEP** ioctl operation is not used since all exception conditions are passed to the kernel user through the exception handler.

Return Values

0	Indicates a successful completion.
ENXIO	Indicates an invalid file pointer.
EINVAL	Indicates an invalid value.
ENOMEM	Indicates insufficient resources to satisfy the **ioctl** subroutine.

These return values are defined in the **/usr/include/sys/errno.h** file.

Implementation Specifics

Each GDLC supports the **fp_ioctl** kernel service by way of its **dlcioctl** entry point. The **fp_ioctl** kernel service may be called from the process environment only.

Related Information

The **fp_ioctl** kernel service.

The **ioctl** subroutine.

The **ioctl** subroutine interface for DLC devices.

Generic Data Link Control (GDLC) Environment Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

fp_lseek, fp_llseek Kernel Service

Purpose

Changes the current offset in an open file.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int fp_lseek  (fp, offset, whence)
struct file *fp;
off_t offset;
int whence;

int fp_llseek
(fp, offset, whence)
struct file *fp
offset_t offset;
int whence;
```

Parameters

fp	Points to a file structure returned by the **fp_open** kernel service.
offset	Specifies the number of bytes (positive or negative) to move the file pointer.
whence	Indicates how to use the offset value:
	SEEK_SET Sets file pointer equal to the number of bytes specified by the *offset* parameter.
	SEEK_CUR Adds the number of bytes specified by the *offset* parameter to current file pointer.
	SEEK_END Adds the number of bytes specified by the *offset* parameter to current end of file.

Description

The **fp_lseek** and **fp_llseek** kernel services are internal interfaces to the function provided by the **lseek** and **llseek** subroutines.

Execution Environment

The **fp_lseek** and **fp_llseek** kernel services can be called from the process environment only.

Return Values

0	Indicates a successful operation.
ERRNO	Returns an error number from the /**usr**/**include**/**sys**/**errno.h** file on failure.

Implementation Specifics

The **fp_lseek** and **fp_llseek** kernel services are parts of Base Operating System (BOS) Runtime.

Related Information

The **lseek**, **llseek** subroutine.

Logical File System Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

fp_open Kernel Service

Purpose

Opens special and regular files or directories.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int fp_open (path, oflags, cmode, ext, segflag, fpp)
char *path;
unsigned oflags;
unsigned cmode;
int ext;
unsigned segflag;
struct file **fpp,
```

Parameters

path	Points to the file name of the file to be opened.
oflags	Specifies open mode flags as described in the **open** subroutine.
cmode	Specifies the mode (permissions) value to be given to the file if the file is to be created.
ext	Specifies an extension argument required by some device drivers. Individual drivers determine its content, form, and use.
segflag	Specifies the flag indicating where the pointer specified by the *path* parameter is located: **SYS_ADSPACE** The pointer specified by the *path* parameter is stored in kernel memory. **USER_ADSPACE** The pointer specified by the *path* parameter is stored in application memory.
fpp	Points to the location where the file structure pointer is to be returned by the **fp_open** service.

Description

The **fp_open** kernel service provides a common service used by:

- The file system for the implementation of the **open** subroutine
- Kernel routines outside the file system that must open files

Execution Environment

The **fp_open** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.

Also, the *fpp* parameter points to an open file structure that is valid for use with the other Logical File System services. If an error occurs, one of the values from the **/usr/include/sys/errno.h** file is returned. The discussion of the **open** subroutine contains possible **errno** values.

Implementation Specifics

The **fp_open** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **open** subroutine.

Logical File System Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

fp_open Kernel Service for Data Link Control (DLC) Devices

Purpose

Allows kernel to open the generic data link control (GDLC) device manager by its device name.

Syntax

```
#include <sys/gdlextcb.h>
#include <fcntl.h>

fp_open (path, oflags, cmode, ext, segflag, fpp)
char path;
unsigned int oflags;
unsigned int cmode;
int ext;
unsigned int segflag;
struct file **fpp;
```

Parameters

path Consists of a character string containing the /**dev** special file name of the GDLC device manager, with the name of the communications device handler appended. The format is shown in the following example:

```
/dev/dlcether/ent0
```

oflags Specifies a value to set the file status flag. The GDLC device manager ignores all but the following values:

O_RDWR Open for reading and writing. This must be set for GDLC or the open will not be successful.

O_NDELAY, **O_NONBLOCK** Subsequent writes return immediately if no resources are available. The calling process is not put to sleep.

cmode Specifies the **O_CREAT** mode parameter. This is ignored by GDLC.

ext Specifies the extended kernel service parameter. This is a pointer to the **dlc_open_ext** extended I/O structure for open subroutines. The argument for this parameter must be in the kernel space. ”open Subroutine Extended Parameters for DLC”*AIX Technical Reference, Volume 3: Communications* provides more information on the extension parameter.

segflag Specifies the segment flag indicating where the *path* parameter is located:

FP_SYS The *path* parameter is stored in kernel memory.

FP_USR The *path* parameter is stored in application memory.

fpp Specifies the returned file pointer. This parameter is passed by reference and updated by the file I/O subsystem to be the file pointer for this **open** subroutine.

Description

The **fp_open** kernel service allows the kernel user to open a GDLC device manager by specifying the special file names of both the DLC and the communications device handler. Since the GDLC device manager is multiplexed, more than one process can open it (or the same process multiple times) and still have unique channel identifications.

Each open carries the communications device handler's special file name so that the DLC knows which port to transfer data on.

The kernel user must also provide functional entry addresses in order to obtain receive data and exception conditions. "Using GDLC Special Kernel Services" in *AIX Communications Programming Concepts* provides additional information.

Return Values

Upon successful completion, this service returns a value of 0 and a valid file pointer in the *fpp* parameter.

ECHILD	Indicates that the service cannot create a kernel process.
EINVAL	Indicates an invalid value.
ENODEV	Indicates that no such device handler is present.
ENOMEM	Indicates insufficient resources to satisfy the open.
EFAULT	Indicates that the kernel service, such as the **copyin** or **initp** service, has failed.

These return values are defined in the **/usr/include/sys/errno.h** file.

Implementation Specifics

Each GDLC supports the **fp_open** kernel service via its **dlcopen** entry point. The **fp_open** kernel service may be called from the process environment only.

Related Information

The **copyin** kernel service, **fp_open** kernel service, **initp** kernel service.

The **fp_close** kernel service for data link control (DLC) devices.

open Subroutine Extended Parameters for DLC in *AIX Technical Reference, Volume 3: Communications*.

Generic Data Link Control (GDLC) Environment Overview and Using GDLC Special Kernel Services in *AIX Communications Programming Concepts*.

fp_opendev Kernel Service

Purpose

Opens a device special file.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int fp_opendev (devno, devflag, channame, ext, fpp)
dev_t devno;
int devflag;
char *channame;
int ext;
struct file**fpp;
```

Parameters

devno	Specifies the major and minor device number of device driver to open.
devflag	Specifies one of the following values: **DREAD** The device is being opened for reading only. **DWRITE** The device is being opened for writing. **DNDELAY** The device is being opened in nonblocking mode.
channame	Points to a channel specifying a character string or a null value.
ext	Specifies an extension argument required by some device drivers. Its content, form, and use are determined by the individual driver.
fpp	Specifies the returned file pointer. This parameter is passed by reference and is updated by the **fp_opendev** service to be the file pointer for this open instance. This file pointer is used as input to other Logical File System services to specify the open instance.

Description

The kernel or kernel extension calls the **fp_opendev** kernel service to open a device by specifying its device major and minor number. The **fp_opendev** kernel service provides the correct semantics for opening the character or multiplexed class of device drivers.

If the specified device driver is nonmultiplexed:

- An in–core i–node is found or created for this device.
- The i–node reference count is incremented.
- The device driver's **ddopen** entry point is called with the *devno*, *devflag*, and *ext* parameters. The unused *chan* parameter on the call to the **ddopen** routine is set to 0.

If the device driver is a multiplexed character device driver (that is, its **ddmpx** entry point is defined), an in–core i–node is created for this channel. The device driver's **ddmpx** routine is also called with the *channame* pointer to the channel identification string if non–null. If the *channame* pointer is null, the **ddmpx** device driver routine is called with the pointer to a null character string.

If the device driver can allocate the channel, the **ddmpx** routine returns a channel ID, represented by the *chan* parameter. If the device driver cannot allocate a channel, the **fp_opendev** kernel service returns an **ENXIO** error code. If successful, the i–node reference count is incremented. The device driver's **ddopen** routine is also called with the *devno*, *devflag*, *chan* (provided by **ddmpx** routine), and *ext* parameters.

If the return value from the specified device driver's **ddopen** routine is nonzero, it is returned as the return code for the **fp_opendev** kernel service. If the return code from the device driver's **ddopen** routine is 0, the **fp_opendev** service returns the file pointer corresponding to this open of the device.

The **fp_opendev** kernel service can only be called in the process environment or device driver top half. Interrupt handlers cannot call it. It is assumed that all arguments to the **fp_opendev** kernel service are in kernel space.

The file pointer (*fpp*) returned by the **fp_opendev** kernel service is only valid for use with a subset of the Logical File System services. These nine services can be called:

- **fp_close**
- **fp_ioctl**
- **fp_poll**
- **fp_select**
- **fp_read**
- **fp_readv**
- **fp_rwuio**
- **fp_write**
- **fp_writev**

Other services return an **EINVAL** return value if called.

Execution Environment

The **fp_opendev** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.

The **fpp* field also points to an open file structure that is valid for use with the other Logical File System services. If an error occurs, one of the following values from the **/usr/include/sys/errno.h** file is returned:

EINVAL	Indicates that the major portion of the *devno* parameter exceeds the maximum number allowed, or the *devflags* parameter is not valid.
ENODEV	Indicates that the device does not exist.
EINTR	Indicates that the signal was caught while processing the **fp_opendev** request.
ENFILE	Indicates that the system file table is full.
ENXIO	Indicates that the device is multiplexed and unable to allocate the channel.

The **fp_opendev** service also returns any nonzero return code returned from a device driver **ddopen** routine.

Implementation Specifics

The **fp_opendev** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **ddopen** Device Driver Entry Point.

The **fp_close** kernel service, **fp_ioctl** kernel service, **fp_poll** kernel service, **fp_read** kernel service, **fp_readv** kernel service, **fp_rwuio** kernel service, **fp_select** kernel service, **fp_write** kernel service, **fp_writev** kernel service.

Logical File System Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

fp_poll Kernel Service

Purpose

Checks the I/O status of multiple file pointers, file descriptors, and message queues.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/poll.h>

int fp_poll (listptr, nfdsmsgs, timeout, flags)
void *listptr;
unsigned long nfdsmsgs;
long timeout;
uint flags;
```

Parameters

listptr	Points to an array of **pollfd** or **pollmsg** structures, or to a single **pollist** structure. Each structure specifies a file pointer, file descriptor, or message queue ID. The events of interest for this file or message queue are also specified.
nfdsmsgs	Specifies the number of files and message queues to check. The low–order 16 bits give the number of elements present in the array of **pollfd** structures. The high–order 16 bits give the number of elements present in the array of **pollmsg** structures. If either half of the *nfdsmsgs* parameter is equal to 0, then the corresponding array is presumed abse1e.
timeout	Specifies how long the service waits for a specified event to occur. If the value of this parameter is –1, the **fp_poll** kernel service does not return until at least one of the specified events has occurred. If the time–out value is 0, the **fp_poll** kernel service does not wait for an event to occur. Instead, the service returns immediately even if none of the specified events have occurred. For any other value of the *timeout* parameter, the **fp_poll** kernel service specifies the maximum length of time (in milliseconds) to wait for at least one of the specified events to occur.
flags	Specifies the type of data in the *listptr* parameter: **POLL_FDMSG** Input is a file descriptor and/or message queue. **0** Input is a file pointer.

Description

Note: The **fp_poll** service applies only to character devices, pipes, message queues, and sockets. Not all character device drivers support the **fp_poll** service.

The **fp_poll** kernel service checks the specified file pointers/descriptors and message queues to see if they are ready for reading or writing, or if they have an exceptional condition pending.

The **pollfd**, **pollmsg**, and **pollist** structures are defined in the /**usr**/**include**/**sys**/**poll.h** file. These are the same structures described for the **poll** subroutine. One difference is that the **fd** field in the **pollfd** structure contains a file pointer when the *flags* parameter on the **fp_poll** kernel service equals 0 (zero). If the *flags* parameter is set to a **POLL_FDMSG** value, the field is taken as a file descriptor in all processed **pollfd** structures. If either the **fd** or **msgid** fields in their respective structures has a negative value, the processing for that structure is skipped.

When performing a poll operation on both files and message queues, the *listptr* parameter points to a **pollist** structure, which can specify both files and message queues. To construct a **pollist** structure, use the **POLLIST** macro as described in the **poll** subroutine.

If the number of **pollfd** elements in the *nfdsmsgs* parameter is 0, then the *listptr* parameter must point to an array of **pollmsg** structures.

If the number of **pollmsg** elements in the *nfdsmsgs* parameter is 0, then the *listptr* parameter must point to an array of **pollfd** structures.

If the number of **pollmsg** and **pollfd** elements are both nonzero in the *nfdsmsgs* parameter, the *listptr* parameter must point to a **pollist** structure as previously defined.

Execution Environment

The **fp_poll** kernel service can be called from the process environment only.

Return Values

Upon successful completion, the **fp_poll** kernel service returns a value that indicates the total number of files and message queues that satisfy the selection criteria. The return value is similar to the *nfdsmsgs* parameter in the following ways:

- The low–order 16 bits give the number of files.
- The high–order 16 bits give the number of message queue identifiers that have nonzero *revents* values.

Use the **NFDS** and **NMSGS** macros to separate these two values from the return value. A return code of 0 (zero) indicates that:

- The call has timed out.
- None of the specified files or message queues indicates the presence of an event.

In other words, all *revents* fields are 0 (zero).

When the return code from the **fp_poll** kernel service is negative, it is set to the following value:

EINTR	Indicates that a signal was caught during the **fp_poll** kernel service.

Implementation Specifics

The **fp_poll** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **poll** subroutine.

The **selreg** kernel service.

Logical File System Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

fp_read Kernel Service

Purpose

Performs a read on an open file with arguments passed.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int fp_read (fp, buf, nbytes, ext, segflag, countp)
struct file *fp;
char *buf;
int nbytes;
int  ext;
int segflag;
int *countp;
```

Parameters

fp	Points to a file structure returned by the **fp_open** or **fp_opendev** kernel service.
buf	Points to the buffer where data read from the file is to be stored.
nbytes	Specifies the number of bytes to be read from the file into the buffer.
ext	Specifies an extension argument required by some device drivers. Its content, form, and use are determined by the individual driver.
segflag	Indicates in which part of memory the buffer specified by the *buf* parameter is located: **SYS_ADSPACE** The buffer specified by the *buf* parameter is in kernel memory. **USER_ADSPACE** The buffer specified by the *buf* parameter is in application memory.
countp	Points to the location where the count of bytes actually read from the file is to be returned.

Description

The **fp_read** kernel service is an internal interface to the function provided by the **read** subroutine.

Execution Environment

The **fp_read** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.

If an error occurs, one of the values from the **/usr/include/sys/errno.h** file is returned.

Implementation Specifics

The **fp_read** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **read** subroutine.

Logical File System Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

fp_readv Kernel Service

Purpose

Performs a read operation on an open file with arguments passed in **iovec** elements.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int fp_readv
(fp, iov, iovcnt, ext,
segflag, countp)
struct file *fp;
char *iov;
int iovcnt;
int  ext;
int segflag;
int *countp;
```

Parameters

fp Points to a file structure returned by the **fp_open** kernel service.

iov Points to an array of **iovec** elements. Each **iovec** element describes a buffer where data to be read from the file is to be stored.

iovcnt Specifies the number of **iovec** elements in the array pointed to by the *iov* parameter.

ext Specifies an extension argument required by some device drivers. Its content, form, and use are determined by the individual driver.

segflag

Indicates in which part of memory the array specified by the *iov* parameter is located:

SYS_ADSPACE
The array specified by the *iov* parameter is in kernel memory.

USER_ADSPACE
The array specified by the *iov* parameter is in application memory.

countp Points to the location where the count of bytes actually read from the file is to be returned.

Description

The **fp_readv** kernel service is an internal interface to the function provided by the **readv** subroutine.

Execution Environment

The **fp_readv** kernel service can be called from the process environment only.

Return Values

0 Indicates a successful operation.

If an error occurs, one of the values from the /**usr**/**include**/**sys**/**errno.h** file is returned.

Implementation Specifics

The **fp_readv** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **readv** subroutine.

Logical File System Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

fp_rwuio Kernel Service

Purpose

Performs read and write on an open file with arguments passed in a **uio** structure.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int fp_rwuio
(fp, rw, uiop, ext)
struct file *fp;
enum uio_rw rw;
struct uio *uiop;
int ext;
```

Parameters

fp	Points to a file structure returned by the **fp_open** or **fp_opendev** kernel service.
rw	Indicates whether this is a read operation or a write operation. It has a value of **UIO_READ** or **UIO_WRITE**.
uiop	Points to a **uio** structure, which contains information such as where to move data and how much to move.
ext	Specifies an extension argument required by some device drivers. Its content, form, and use are determined by the individual driver.

Description

The **fp_rwuio** kernel service is not the preferred interface for read and write operations. The **fp_rwuio** kernel service should only be used if the calling routine has been passed a **uio** structure. If the calling routine has not been passed a **uio** structure, it should not attempt to construct one and call the **fp_rwuio** kernel service with it. Rather, it should pass the requisite **uio** components to the **fp_read** or **fp_write** kernel services.

Execution Environment

The **fp_rwuio** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.

If an error occurs, one of the values from the /**usr**/**include**/**sys**/**errno.h** file is returned.

Implementation Specifics

The **fp_rwuio** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **uio** structure.

Logical File System Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

fp_select Kernel Service

Purpose

Provides for cascaded, or redirected, support of the select or poll request.

Syntax

#include <sys/types.h>
#include <sys/errno.h>

int fp_select (*fp***,** *events***,** *rtneventp***,** *notify***)**
struct file ****fp*;**
ushort *events***;**
ushort ****rtneventp*;**
void (**notify***)();**

Parameters

fp — Points to the open instance of the device driver, socket, or pipe for which the low–level select operation is intended.

events — Identifies the events that are to be checked. There are three standard event flags defined for the **poll** and **select** functions and one informational flag. The **/usr/include/sys/poll.h** file details the event bit definition. The four basic indicators are:

- **POLLIN** — Input is present for the specified object.
- **POLLOUT** — The specified file object is capable of accepting output.
- **POLLPRI** — An exception condition has occurred on the specified object.
- **POLLSYNC** — This is a synchronous request only. If none of the requested events are true, the selected routine should not remember this request as pending. That is, the routine does not need to call the **selnotify** service because of this request.

rtneventp — Indicates the returned events pointer. This parameter, passed by reference, is used to indicate which selected events are true at the current time. The returned event bits include the requested events plus an additional error event indicator:

- **POLLERR** — An error condition was indicated by the object's select routine. If this flag is set, the nonzero return code from the specified object's select routine is returned as the return code from the **fp_select** kernel service.

notify — Points to a routine to be called when the specified object invokes the **selnotify** kernel service for an outstanding asynchronous select or poll event request. If no routine is to be called, this parameter must be NULL.

Description

The **fp_select** kernel service is a low–level service used by kernel extensions to perform a select operation for an open device, socket, or named pipe. The **fp_select** kernel service can be used for both synchronous and asynchronous select requests. Synchronous requests report on the current state of a device, and asynchronous requests allow the caller to be notified of future events on a device.

Invocation from a Device Driver's ddselect Routine

A device driver's **ddselect** routine can call the **fp_select** kernel service to pass select/poll requests to other device drivers. The **ddselect** routine for one device invokes the **fp_select** kernel service, which calls the **ddselect** routine for a second device, and so on. This is required when event information for the original device depends upon events occurring on other devices. A cascaded chain of select requests can be initiated that involves more than two devices, or a single device can issue **fp_select** calls to several other devices.

Each **ddselect** routine should preserve, in its call to the **fp_select** kernel service, the same **POLLSYNC** indicator that it received when previously called by the **fp_select** kernel service.

Invocation from Outside a Device Driver's ddselect Routine

If the **fp_select** kernel service is invoked outside of the device driver's **ddselect** routine, the **fp_select** kernel service sets the **POLLSYNC** flag, always making the request synchronous. In this case, no notification of future events for the specified device occurs, nor is a **notify** routine called, if specified. The **fp_select** kernel service can be used in this manner (unrelated to a poll or select request in progress) to check an object's current status.

Asynchronous Processing and the Use of the notify Routine

For asynchronous requests, the **fp_select** kernel service allows its callers to register a **notify** routine to be called by the kernel when specified events become true. When the relevant device driver detects that one or more pending events have become true, it invokes the **selnotify** kernel service. The **selnotify** kernel service then calls the **notify** routine, if one has been registered. Thus, the **notify** routine is called at interrupt time and must be programmed to run in an interrupt environment.

Use of a **notify** routine affects both the calling sequence at interrupt time and how the requested information is actually reported. Generalized asynchronous processing entails the following sequence of events:

1. A select request is initiated on a device and passed on (by multiple **fp_select** kernel service invocations) to further devices. Eventually, a device driver's **ddselect** routine that is not dependent on other devices for information is reached. This **ddselect** routine finds that none of the requested events are true, but remembers the asynchronous request, and returns to the caller. In this way, the entire chain of calls is backed out, until the origin of the select request is reached. The kernel then puts the originating process to sleep.
2. Later, one or more events become true for the device remembering the asynchronous request. The device driver routine (possibly an interrupt handler) calls the **selnotify** kernel service.
3. If the events are still being waited on, the **selnotify** kernel service responds in one of two ways. If no **notify** routine was registered when the select request was made for the device, then all processes waiting for events on this device are awakened. If a **notify** routine exists for the device, then this routine is called. The **notify** routine determines whether the original requested event should be reported as true, and if so, calls the **selnotify** kernel service on its own.

The following example details a cascaded scenario involving several devices. Suppose that a request has been made for Device A, and Device A depends on Device B, which depends on Device C. When specified events become true at Device C, the **selnotify** kernel service called from Device C's device driver performs differently depending on whether a **notify** routine was registered at the time of the request.

Cascaded Processing without the Use of notify Routines

If no **notify** routine was registered from Device B, then the **selnotify** kernel service determines that the specified events are to be considered true for the device driver at the head of the cascading chain. (The head of the chain, in this case Device A, is the first

device driver to issue the **fp_select** kernel service from its select routine.) The **selnotify** kernel service awakens all processes waiting for events that have occurred on Device A.

It is important to note that when no **notify** routine is used, any device driver in the calling chain that reports an event with the **selnotify** kernel service causes that event to appear true for the first device in the chain. As a result, any processes waiting for events that have occurred on that first device are awakened.

Cascaded Processing with notify Routines

If, on the other hand, **notify** routines have been registered throughout the chain, then each interrupting device (by calling the **selnotify** kernel service) invokes the **notify** routine for the device above it in the calling chain. Thus in the preceding example, the **selnotify** kernel service for Device C calls the **notify** routine registered when Device B's **ddselect** routine invoked the **fp_select** kernel service. Device B's **notify** routine must then decide whether to again call the **selnotify** kernel service to alert Device A's **notify** routine. If so, then Device A's **notify** routine is called, and makes its own determination whether to call another **selnotify** routine. If it does, the **selnotify** kernel service wakes up all the processes waiting on occurred events for Device A.

A variation on this scenario involves a cascaded chain in which only some device drivers have registered **notify** routines. In this case, the **selnotify** kernel service at each level calls the **notify** routine for the level above, until a level is encountered for which no **notify** routine was registered. At this point, all events of interest are determined to be true for the device driver at the head of the cascading chain. If any **notify** routines were registered in levels above the current level, they are never called.

Returning from the fp_select Kernel Service

The **fp_select** kernel service does not wait for any selected events to become true, but returns immediately after the call to the object's **ddselect** routine has completed.

If the object's select routine is successfully called, the return code for the **fp_select** kernel service is set to the return code provided by the object's **ddselect** routine.

Execution Environment

The **fp_select** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
EAGAIN	Indicates that the allocation of internal data structures failed. The *rtneventp* parameter is not updated.
EINVAL	Indicates that the *fp* parameter is not a valid file pointer. The *rtneventp* parameter has the **POLLNVAL** flag set.

The **fp_select** kernel service can also be set to the nonzero return code from the specified object's **ddselect** routine. The *rtneventp* parameter has the **POLLERR** flag set.

Implementation Specifics

This kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **fp_poll** kernel service, **selnotify** kernel service, **selreg** kernel service.

The **fp_select kernel service notify** routine.

The **poll** subroutine, **select** subroutine.

Logical File System Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

fp_select Kernel Service notify Routine

Purpose

Registers the **notify** routine.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

void notify (id, sub_id, rtnevents, pid)
int id;
int sub_id;
ushort rtnevents;
pid_t pid;
```

Parameters

id	Indicates the selected function ID specified by the routine that made the call to the **selnotify** kernel service to indicate the occurrence of an outstanding event. For device drivers, this parameter is equivalent to the *devno* (device major and minor number) parameter.
sub_id	Indicates the unique ID specified by the routine that made the call to the **selnotify** kernel service to indicate the occurrence of an outstanding event. For device drivers, this parameter is equivalent to the *chan* parameter: channel for multiplexed drivers; 0 for nonmultiplexed drivers.
rtnevents	Specifies the *rtnevents* parameter supplied by the routine that made the call to the **selnotify** service indicating which events are designated as true.
pid	Specifies the process ID of a process waiting for the event corresponding to this call of the **notify** routine.

When a **notify** routine is provided for a cascaded function, the **selnotify** kernel service calls the specified **notify** routine instead of posting the process that was waiting on the event. It is up to this **notify** routine to determine if another **selnotify** call should be made to notify the waiting process of an event.

The **notify** routine is not called if the request is synchronous (that is, if the **POLLSYNC** flag is set in the *events* parameter) or if the original poll or select request is no longer outstanding.

Note: When more than one process has requested notification of an event and the **fp_select** kernel service is used with a **notify** routine specified, the notification of the event causes the **notify** routine to be called once for each process that is currently waiting on one or more of the occurring events.

Description

The **fp_select** kernel service **notify** routine is registered by the caller of the **fp_select** kernel service to be called by the kernel when specified events become true. The option to register this **notify** routine is available in a cascaded environment. The **notify** routine can be called at interrupt time.

Execution Environment

The **fp_select** kernel service **notify** routine can be called from either the process or interrupt environment.

Implementation Specifics

The **fp_select** kernel service **notify** routine is part of Base Operating System (BOS) Runtime.

Related Information

The **fp_select** kernel service, **selnotify** kernel service.

Logical File System Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

fp_write Kernel Service

Purpose

Performs a write operation on an open file with arguments passed.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int fp_write (fp, buf, nbytes, ext, segflag, countp)
struct file * fp;
 char *buf;
 int nbytes,
 int ext;
int segflag;
 int *countp;
```

Parameters

fp	Points to a file structure returned by the **fp_open** or **fp_opendev** kernel service.
buf	Points to the buffer where data to be written to a file is located.
nbytes	Indicates the number of bytes to be written to the file.
ext	Specifies an extension argument required by some device drivers. Its content, form, and use are determined by the individual driver.
segflag	Indicates in which part of memory the buffer specified by the *buf* parameter is located: **SYS_ADSPACE** The buffer specified by the *buf* parameter is in kernel memory. **USER_ADSPACE** The buffer specified by the *buf* parameter is in application memory.
countp	Points to the location where count of bytes actually written to the file is to be returned.

Description

The **fp_write** kernel service is an internal interface to the function provided by the **write** subroutine.

Execution Environment

The **fp_write** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
ERRNO	Returns an error number from the **/usr/include/sys/errno.h** file on failure.

Implementation Specifics

The **fp_write** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **write** subroutine.

Logical File System Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

fp_write Kernel Service for Data Link Control (DLC) Devices

Purpose

Allows kernel data to be sent using a file pointer.

Syntax

#include <sys/gdlextcb.h>
#include <sys/fp_io.h>

int fp_write (*fp***,** *buf***,** *nbytes***,** *ext***,** *segflag***,** *countp***)**
struct file **fp*;*
char **buf*;*
int *nbytes*;
int *ext*;
int *segflag*;
int **countp*;>

Parameters

fp	Specifies file pointer returned from the **fp_open** kernel service.
buf	Points to a kernel **mbuf** structure.
nbytes	Contains the byte length of the write data. It is not necessary to set this field to the actual length of write data, however, since the **mbuf** contains a length field. Instead, this field can be set to any non–negative value (generally set to 0).
ext	Specifies the extended kernel service parameter. This is a pointer to the **dlc_io_ext** extended I/O structure for writes. The argument for this parameter must be in the kernel space. For more information on this parameter, see ”write Subroutine Extended Parameters for DLC”*AIX Technical Reference, Volume 3: Communications*.
segflag	Specifies the segment flag indicating where the *path* parameter is located. The only valid value is: **FP_SYS** The *path* parameter is stored in kernel memory.
countp	Points to the location where a count of bytes actually written is to be returned (must be in kernel space). GDLC does not provide this information for a kernel user since **mbufs** are used, but the file system requires a valid address and writes a copy of the *nbytes* parameter to that location.

Description

Four types of data can be sent to generic data link control (GDLC). Network data can be sent to a service access point (SAP), and normal, exchange identification (XID) or datagram data can be sent to a link station (LS).

Kernel users pass a communications memory buffer (**mbuf**) directly to GDLC on the **fp_write** kernel service. In this case, a **uiomove** kernel service is not required, and maximum performance can be achieved by merely passing the buffer pointer to GDLC. Each write buffer is required to have the proper buffer header information and enough space for the data link headers to be inserted. A write data offset is passed back to the kernel user at start LS completion for this purpose.

All data must fit into a single packet for each write call. That is, GDLC does not separate the user’s write data area into multiple transmit packets. A maximum write data size is passed back to the user at **DLC_ENABLE_SAP** completion and at **DLC_START_LS** completion for this purpose.

Normally, a write subroutine can be satisfied immediately by GDLC by completing the data link headers and sending the transmit packet down to the device handler. In some cases, however, transmit packets can be blocked by the particular protocol's flow control or a resource outage. GDLC reacts to this differently, based on the system blocked/nonblocked file status flags (set by the file system and based on the **O_NDELAY** and **O_NONBLOCKED** values passed on the **fp_open** kernel service). Nonblocked **write** subroutines that cannot get enough resources to queue the communications memory buffer (**mbuf**) return an error indication. Blocked write subroutines put the calling process to sleep until the resources free up or an error occurs.

Return Values

0	Indicates a successful operation.
EAGAIN	Indicates that transmit is temporarily blocked, and the calling process cannot be put to sleep.
EINTR	Indicates that a signal interrupted the kernel service before it could complete successfully.
EINVAL	Indicates an invalid argument, such as too much data for a single packet.
ENXIO	Indicates an invalid file pointer.

These return values are defined in the /**usr**/**include**/**sys**/**errno.h** file.

Implementation Specifics

Each GDLC supports the **fp_write** kernel service via its **dlcwrite** entry point. The **fp_write** kernel service may be called from the process environment only.

Related Information

The **fp_open** kernel service, **fp_write** kernel service.

The **uiomove** subroutine.

Generic Data Link Control (GDLC) Environment Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

Parameter Blocks by ioctl Operation for DLC.

read Subroutine Extended Parameters for DLC.

fp_writev Kernel Service

Purpose

Performs a write operation on an open file with arguments passed in **iovec** elements.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int fp_writev (fp, iov, iovcnt, ext, segflag, countp)
struct file *fp;
struct iovec *iov;
int iovcnt;
int ext;
int segflag;
int *countp;
```

Parameters

fp	Points to a file structure returned by the **fp_open** kernel service.
iov	Points to an array of **iovec** elements. Each **iovec** element describes a buffer containing data to be written to the file.
iovcnt	Specifies the number of **iovec** elements in an array pointed to by the *iov* parameter.
ext	Specifies an extension argument required by some device drivers. Its content, form, and use are determined by the individual driver.
segflag	Indicates which part of memory the information designated by the *iov* parameter is located in: **SYS_ADSPACE** The information designated by the *iov* parameter is in kernel memory. **USER_ADSPACE** The information designated by the *iov* parameter is in application memory.
countp	Points to the location where the count of bytes actually written to the file is to be returned.

Description

The **fp_writev** kernel service is an internal interface to the function provided by the **writev** subroutine.

Execution Environment

The **fp_writev** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.

If an error occurs, one of the values from the **/usr/include/sys/errno.h** file is returned.

Implementation Specifics

The **fp_writev** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **writev** subroutine.

Logical File System Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

fubyte Kernel Service

Purpose

Retrieves a byte of data from user memory.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int fubyte (uaddr)
uchar *uaddr;
```

Parameter

uaddr	Specifies the address of the user data.

Description

The **fubyte** kernel service fetches, or retrieves, a byte of data from the specified address in user memory. It is provided so that system calls and device heads can safely access user data. The **fubyte** service ensures that the user has the appropriate authority to:

- Access the data.
- Protect the operating system from paging I/O errors on user data.

The **fubyte** service should be called only while executing in kernel mode in the user process.

Execution Environment

The **fubyte** kernel service can be called from the process environment only.

Return Values

When successful, the **fubyte** service returns the specified byte.

–1	Indicates a *uaddr* parameter that is not valid.

The access is not valid under the following circumstances:

- The user does not have sufficient authority to access the data.
- The address is not valid.
- An I/O error occurs while referencing the user data.

Implementation Specifics

The **fubyte** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **fuword** kernel service, **subyte** kernel service, **suword** kernel service.

Accessing User–Mode Data while in Kernel Mode and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

fubyte64 Kernel Service

Purpose

Retrieves a byte of data from user memory.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/uio.h>

int fubyte64 (uaddr64)
unsigned long long uaddr64;
```

Parameter

uaddr64	Specifies the address of user data.

Description

The **fubyte64** kernel service fetches, or retrieves, a byte of data from the specified address in user memory. It is provided so that system calls and device heads can safely access user data. The **fubyte64** service ensures that the user has the appropriate authority to:

- Access the data.
- Protect the operating system from paging I/O errors on user data.

This service will operate correctly for both 32–bit and 64–bit user address spaces. The *uaddr64* parameter is interpreted as being a non–remapped 32–bit address for the case where the current user address space is 32– bits. If the current user address space is 64–bits, then **uaddr64** is treated as a 64–bit address.

The **fubyte64** service should be called only while executing in kernel mode in the user process.

Execution Environment

The **fubyte64** kernel service can be called from the process environment only.

Return Values

When successful, the **fubyte64** service returns the specified byte.

–1	Indicates a *uaddr64* parameter that is not valid.

The access is not valid under the following circumstances:

- The user does not have sufficient authority to access the data.
- The address is not valid.
- An I/O error occurs while referencing the user data.

Implementation Specifics

The **fubyte64** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **fuword64** kernel service, **subyte64** kernel service, and **suword64** kernel service.

Accessing User–Mode Data While in Kernel Mode and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

fuword Kernel Service

Purpose

Retrieves a word of data from user memory.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int fuword (uaddr)
int *uaddr;
```

Parameter

uaddr	Specifies the address of user data.

Description

The **fuword** kernel service retrieves a word of data from the specified address in user memory. It is provided so that system calls and device heads can safely access user data. The **fuword** service ensures that the user had the appropriate authority to:

- Access the data.
- Protect the operating system from paging I/O errors on user data.

The **fuword** service should be called only while executing in kernel mode in the user process.

Execution Environment

The **fuword** kernel service can be called from the process environment only.

Return Values

When successful, the **fuword** service returns the specified word of data.

–1	Indicates a *uaddr* parameter that is not valid.

The access is not valid under the following circumstances:

- The user does not have sufficient authority to access the data.
- The address is not valid.
- An I/O error occurred while referencing the user data.

For the **fuword** service, a retrieved value of –1 and a return code of –1 are indistinguishable.

Implementation Specifics

The **fuword** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **fubyte** kernel service, **subyte** kernel service, **suword** kernel service.

Accessing User–Mode Data while in Kernel Mode and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

fuword64 Kernel Service

Purpose

Retrieves a word of data from user memory.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/uio.h>

int fuword64 (uaddr64)
unsigned long long uaddr64;
```

Parameter

uaddr64	Specifies the address of user data.

Description

The **fuword64** kernel service retrievesa word of data from the specified address in user memory. It is provided so that system calls and device heads can safely access user data. The **fuword64** service ensures that the user has the appropriate authority to:

- Access the data.
- Protect the operating system from paging I/O errors on user data.

This service will operate correctly for both 32–bit and 64–bit user address spaces. The *uaddr64* parameter is interpreted as being a non–remapped 32–bit address for the case where the current user address space is 32– bits. If the current user address space is 64–bits, then **uaddr64** is treated as a 64–bit address.

The **fuword64** service should be called only while executing in kernel mode in the user process.

Execution Environment

The **fuword64** kernel service can be called from the process environment only.

Return Values

When successful, the **fuword64** service returns the word of data.

–1	Indicates a *uaddr64* parameter that is not valid.

The access is not valid under the following circumstances:

- The user does not have sufficient authority to access the data.
- The address is not valid.
- An I/O error occurs while referencing the user data.

Implementation Specifics

The **fuword64** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **fubyte64** kernel service, **subyte64** kernel service, and **suword64** kernel service.

Accessing User–Mode Data While in Kernel Mode and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

getadsp Kernel Service

Purpose

Obtains a pointer to the current process's address space structure for use with the **as_att** and **as_det** kernel services.

Syntax

```
#include <sys/types.h>#include <sys/errno.h>
#include <sys/vmuser.h>
#include <sys/adspace.h>

adspace_t *getadsp ()
```

Description

The **getadsp** kernel service returns a pointer to the current process's address space structure for use with the **as_att** and **as_det** kernel services. This routine distinguishes between kernel processes (kprocs) and ordinary processes. It returns the correct address space pointer for the current process.

Execution Environment

The **getadsp** kernel service can be called from the process environment only.

Return Values

The **getadsp** service returns a pointer to the current process's address space structure.

Implementation Specifics

The **getadsp** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **as_att** kernel service, **as_det** kernel service, **as_geth** kernel service, **as_getsrva**l kernel service, **as_puth** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

getblk Kernel Service

Purpose

Assigns a buffer to the specified block.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/buf.h>

struct buf *getblk
(dev, blkno)
dev_t  dev;
daddr_t  blkno;
```

Parameters

dev	Specifies the device containing the block to be allocated.
blkno	Specifies the block to be allocated.

Description

The **getblk** kernel service first checks whether the specified buffer is in the buffer cache. If the buffer resides there, but is in use, the **e_sleep** service is called to wait until the buffer is no longer in use. Upon waking, the **getblk** service tries again to access the buffer. If the buffer is in the cache and not in use, it is removed from the free list and marked as busy. Its buffer header is then returned. If the buffer is not in the buffer cache, another buffer is taken from the free list and returned.

Execution Environment

The **getblk** kernel service can be called from the process environment only.

Return Values

The **getblk** service returns a pointer to the buffer header. A nonzero value for **B_ERROR** in the `b_flags` field of the buffer header (**buf** structure) indicates an error. If this occurs, the caller should release the block's buffer using the **brelse** kernel service.

Implementation Specifics

The **getblk** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Block I/O Buffer Cache Kernel Services: Overview in *AIX Kernel Extensions and Device Support Programming Concepts* summarizes how the **bread**, **brelse**, and **getblk** services uniquely manage the block I/O buffer cache.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

getc Kernel Service

Purpose

Retrieves a character from a character list.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <cblock.h>

int getc (header)
struct clist *header;
```

Parameter

header	Specifies the address of the **clist** structure that describes the character list.

Description

Attention: The caller of the **getc** service must ensure that the character list is pinned. This includes the **clist** header and all the **cblock** character buffers. Otherwise, the system may crash.

The **getc** kernel service returns the character at the front of the character list. After returning the last character in the buffer, the **getc** service frees that buffer.

Execution Environment

The **getc** kernel service can be called from either the process or interrupt environment.

Return Values

–1	Indicates that the character list is empty.

Implementation Specifics

The **getc** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

getcb Kernel Service

Purpose

Removes the first buffer from a character list and returns the address of the removed buffer.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <cblock.h>

struct cblock *getcb
(header)
struct clist *header;
```

Parameter

header Specifies the address of the **clist** structure that describes the character list.

Description

Attention: The caller of the **getcb** service must ensure that the character list is pinned. This includes the **clist** header and all the **cblock** character buffers. Character buffers acquired from the **getcf** service are pinned. Otherwise, the system may crash.

The **getcb** kernel service returns the address of the character buffer at the start of the character list and removes that buffer from the character list. The user must free the buffer with the **putcf** service when finished with it.

Execution Environment

The **getcb** kernel service can be called from either the process or interrupt environment.

Return Values

A null address indicates the character list is empty.

The **getcb** service returns the address of the character buffer at the start of the character list when the character list is not empty.

Implementation Specifics

The **getcb** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **getcf** kernel service.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

getcbp Kernel Service

Purpose

Retrieves multiple characters from a character buffer and places them at a designated address.

Syntax

```
#include <cblock.h>

int getcbp (header, dest, n)
struct clist *header;
char *dest;
int n;
```

Parameters

header	Specifies the address of the **clist** structure that describes the character list.
dest	Specifies the address where the characters obtained from the character list are to be placed.
n	Specifies the number of characters to be read from the character list.

Description

Attention: The caller of the **getcbp** services must ensure that the character list is pinned. This includes the **clist** header and all the **cblock** character buffers. Character buffers acquired from the **getcf** service are pinned. Otherwise, the system may crash.

The **getcbp** kernel service retrieves as many as possible of the *n* characters requested from the character buffer at the start of the character list. The **getcbp** service then places them at the address pointed to by the *dest* parameter.

Execution Environment

The **getcbp** kernel service can be called from either the process or interrupt environment.

Return Values

The **getcbp** service returns the number of characters retrieved from the character buffer.

Implementation Specifics

The **getcbp** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **getcf** kernel service.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

getcf Kernel Service

Purpose

Retrieves a free character buffer.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <cblock.h

struct cblock *getcf ( )
```

Description

The **getcf** kernel service retrieves a character buffer from the list of available ones and returns that buffer's address. The returned character buffer is pinned. If you use the **getcf** service to get a character buffer, be sure to free the space when you have finished using it. The buffers received from the **getcf** service should be freed by using the **putcf** kernel service.

Before starting the **getcf** service, the caller should request enough **clist** resources by using the **pincf** kernel service. The proper use of the **getcf** service ensures that there are sufficient pinned buffers available to the caller.

If the **getcf** service indicates that there is no available character buffer, the **waitcfree** service can be called to wait until a character buffer becomes available.

The **getcf** service has no parameters.

Execution Environment

The **getcf** kernel service can be called from either the process or interrupt environment.

Return Values

Upon successful completion, the **getcf** service returns the address of the allocated character buffer.

A null pointer indicates no buffers are available.

Implementation Specifics

The **getcf** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **pincf** kernel service, **putcf** kernel service, **waitcfree** kernel service.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

getcx Kernel Service

Purpose

Returns the character at the end of a designated list.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <cblock.h>

int getcx (header)
struct clist *header;
```

Parameter

header Specifies the address of the **clist** structure that describes the character list.

Description

Attention: The caller of the **getcx** service must ensure that the character list is pinned. This includes the **clist** header and all the **cblock** character buffers. Character buffers acquired from the **getcf** service are pinned.

The **getcx** kernel service is identical to the **getc** service, except that the **getcx** service returns the character at the end of the list instead of the character at the front of the list. The character at the end of the list is the last character in the first buffer, not in the last buffer.

Execution Environment

The **getcx** kernel service can be called from either the process or interrupt environment.

Return Values

The **getcx** service returns the character at the end of the list instead of the character at the front of the list.

Implementation Specifics

The **getcx** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **getcf** kernel service.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

geteblk Kernel Service

Purpose

Allocates a free buffer.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/buf.h>

struct buf *geteblk ( )
```

Description

Attention: The use of the **geteblk** service by character device drivers is strongly discouraged. As an alternative, character device drivers can use the **xmalloc** service to allocate the memory space directly, or the character I/O kernel services such as the **getcb** or **getcf** services.

The **geteblk** kernel service allocates a buffer and buffer header and returns the address of the buffer header. If no free buffers are available, then the **geteblk** service waits for one to become available. Block device drivers can retrieve buffers using the **geteblk** service.

In the header, the `b_forw`, `b_back`, `b_flags`, `b_bcount`, `b_dev`, and `b_un` fields are used by the system and cannot be modified by the driver. The `av_forw` and `av_back` fields are available to the user of the **geteblk** service for keeping a chain of buffers by the user of the **geteblk** service. (This user could be the kernel file system or a device driver.) The `b_blkno` and `b_resid` fields can be used for any purpose.

The **brelse** service is used to free this type of buffer.

The **geteblk** service has no parameters.

Execution Environment

The **geteblk** kernel service can be called from the process environment only.

Return Values

The **geteblk** service returns a pointer to the buffer header. There are no error codes because the **geteblk** service waits until a buffer header becomes available.

Implementation Specifics

The **geteblk** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **brelse** kernel service, **xmalloc** kernel service.

Block I/O Buffer Cache Kernel Services: Overview, I/O Kernel Services, buf Structure, Device Driver Concepts Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

geterror Kernel Service

Purpose

Determines the completion status of the buffer.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/buf.h>

int geterror (bp)
struct buf  *bp;
```

Parameter

bp	Specifies the address of the buffer structure whose status is to be checked.

Description

The **geterror** kernel service checks the specified buffer to see if the **b_error** flag is set. If that flag is not set, the **geterror** service returns 0. Otherwise, it returns the nonzero **B_ERROR** value or the **EIO** value (if **b_error** is 0).

Execution Environment

The **geterror** kernel service can be called from either the process or interrupt environment.

Return Values

0	Indicates that no I/O error occurred on the buffer.
b_error value	Indicates that an I/O error occurred on the buffer.
EIO	Indicates that an unknown I/O error occurred on the buffer.

Implementation Specifics

The **geterror** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Block I/O Buffer Cache Kernel Services: Overview and I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

getexcept Kernel Service

Purpose

Allows kernel exception handlers to retrieve additional exception information.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/except.h>

void getexcept
(exceptp)
struct except *exceptp;
```

Parameter

exceptp	Specifies the address of an **except** structure, as defined in the /**usr**/**include**/**sys**/**except.h** file. The **getexcept** service copies detailed exception data from the current machine–state save area into this caller–supplied structure.

Description

The **getexcept** kernel service provides exception handlers the capability to retrieve additional information concerning the exception from the machine–state save area.

The **getexcept** service should only be used by exception handlers when called to handle an exception. The contents of the structure pointed at by the *exceptp* parameter is platform–specific, but is described in the /**usr**/**include**/**sys**/**except.h** file for each type of exception that provides additional data. This data is typically included in any error logging data for the exception. It can be also used to attempt to handle or recover from the exception.

Execution Environment

The **getexcept** kernel service can be called from either the process or interrupt environment. It should be called only when handling an exception.

Return Values

The **getexcept** service has no return values.

Implementation Specifics

The **getexcept** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Kernel Extension and Device Driver Management Kernel Services and in *AIX Kernel Extensions and Device Support Programming Concepts*.

getfslimit Kernel Service

Purpose

Returns the maximum file size limit of the current process.

Syntax

#include <sys/types.h>
offset_t getfslimit (void)

Description

The **getfslimit** kernel service returns the file size limit of the current process as a 64 bit integer. This can be used by file systems to implement the checks needed to enforce limits. The **getfslimit** kernel service is called from the process environment.

Return Values

The **getfslimit** kernel service returns the the file size limit, there are no error values.

Implementation Specifics

The **getfslimit** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **ulimit** subroutine, **getrlimit** subroutine, **setrlimit** subroutine.

The **ulimit** command.

getpid Kernel Service

Purpose

Gets the process ID of the current process.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

pid_t getpid ()
```

Description

The **getpid** kernel service returns the process ID of the calling process.

The **getpid** service can also be used to check the environment that the routine is being executed in. If the caller is executing in the interrupt environment, the **getpid** service returns a process ID of –1. If a routine is executing in a process environment, the **getpid** service obtains the current process ID.

Execution Environment

The **getpid** kernel service can be called from either the process or interrupt environment.

Return Values

–1	Indicates that the **getpid** service was called from an interrupt environment.

The **getpid** service returns the process ID of the current process if called from a process environment.

Implementation Specifics

The **getpid** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

Process and Exception Management Kernel Services and Understanding Execution Environments in *AIX Kernel Extensions and Device Support Programming Concepts*.

getppidx Kernel Service

Purpose

Gets the parent process ID of the specified process.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

pid_t getppidx (ProcessID)
pid_t ProcessID;
```

Parameter

ProcessID	Specifies the process ID. If this parameter is 0, then the parent process ID of the calling process will be returned.

Description

The **getppidx** kernel service returns the parent process ID of the specified process.

Execution Environment

The **getppidx** kernel service can be called from the process environment only.

Return Values

–1	Indicates that the *ProcessID* parameter is invalid.

The **getppidx** service returns the parent process ID of the calling process.

Implementation Specifics

The **getppidx** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **getpid** kernel service.

Process and Exception Management Kernel Services and Understanding Execution Environments in *AIX Kernel Extensions and Device Support Programming Concepts*.

getuerror Kernel Service

Purpose

Allows kernel extensions to retrieve the current value of the `ut_error` field.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int getuerror ()
```

Description

The **getuerror** kernel service allows a kernel extension in a process environment to retrieve the current value of the current thread's `ut_error` field. Kernel extensions can use the **getuerror** service when using system calls or other kernel services that return error information in the `ut _error` field.

For system calls, the system call handler copies the value of the `u t _error` field in the per thread **uthread** structure to the **errno** global variable before returning to the caller. However, when kernel services use available system calls, the system call handler is bypassed. The **getuerror** service must then be used to obtain error information.

Execution Environment

The **getuerror** kernel service can be called from the process environment only.

Return Values

0 Indicates a successful operation.

When an error occurs, the **getuerror** kernel service returns the current value of the `u t _error` field in the per thread **uthread** structure. Possible return values for this field are defined in the **/usr/include/sys/errno.h** file.

Implementation Specifics

The **getuerror** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **setuerror** kernel service.

Kernel Extension and Device Driver Management Kernel Services and Understanding System Call Execution in *AIX Kernel Extensions and Device Support Programming Concepts*.

getufdflags and setufdflags Kernel Services

Purpose

Queries and sets file–descriptor flags.

Syntax

#include <sys/user.h>

int getufdflags(*fd, flagsp***)**
int *fd***;**
int **flagsp***;*

#include <sys/user.h>

int setufdflags(*fd*, *flags*)
int *fd*;
int *flags***;**

Parameters

fd	Identifies the file descriptor.
flags	Sets attribute flags for the specified file descriptor. Refer to the **sys/user.h** file for the list of valid flags.
flagsp	Points to an integer field where the flags associated with the file descriptor are stored on successful return.

Description

The **setufdflags** and **getufdflags** kernel services set and query the file descriptor flags. The file descriptor flags are listed in **fontl.h**.

Execution Environment

These kernel services can be called from the process environment only.

Return Values

0	Indicates successful completion.
EBADF	Indicates that the *fd* parameter is not a file descriptor for an open file.

Implementation Specifics

These kernel services are part of Base Operating System (BOS) Runtime.

Related Information

The **ufdhold** and **ufdrele** kernel services.

get_umask Kernel Service

Purpose

Queries the file mode creation mask.

Syntax

```
int get_umask(void)
```

Description

The **get_umask** service gets the value of the file mode creation mask currently set for the process.

Note: There is no corresponding kernel service to set the umask because kernel routines that need to set the umask can call the **umask** subroutine.

Execution Environment

The **get_umask** kernel service can be called from the process environment only.

Return Values

The **get_umask** kernel service always completes successfully. Its return value is the current value of the **umask**.

Implementation Specifics

This kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **umask** subroutine.

gfsadd Kernel Service

Purpose

Adds a file system type to the **gfs** table.

Syntax

#include <sys/types.h>
#include <sys/errno.h>

int gfsadd (*gfsno*,*gfsp***)**
int *gfsno***;**
struct gfs ****gfsp*;**

Parameters

gfsno	Specifies the file system number. This small integer value is either defined in the **/usr/include/sys/vmount.h** file or a user–defined number of the same order.
gfsp	Points to the file system description structure.

Description

The **gfsadd** kernel service is used during configuration of a file system. The configuration routine for a file system invokes the **gfsadd** kernel service with a **gfs** structure. This structure describes the file system type.

The **gfs** structure type is defined in the **/usr/include/sys/gfs.h** file. The **gfs** structure must have the following fields filled in:

`gfs_type`	Specifies the integer type value. The predefined types are listed in the **/usr/include/sys/vmount.h** file.
`gfs_name`	Specifies the character string name of the file system. The maximum length of this field is 16 bytes. Shorter names must be null–padded.
`gfs_flags`	Specifies the flags that define the capabilities of the file system. The following flag values are defined:

GFS_SYS5DIR	File system that uses the System V–type directory structure.
GFS_REMOTE	File system is remote (ie. NFS).
GFS_FUMNT	File system supports forced unmount.
GFS_NOUMASK	File system applies umask when creating new objects.
GFS_VERSION4	File system supports AIX Version 4 V–node interface.
GFS_VERSION42	File system supports AIX Version 4.2 V–node interface. (new vnode op: vn_seek)
GFS_VERSION421	File system supports AIX Version 4.2.1 V–node interface.(new vnode ops: vn_sync_range, vn_create_attr, vn_finfo, vn_map_lloff, vn_readdir_eofp, vn_rdwr_attr))
GFS_VERSION43	File system supports AIX Version 4.3 V–node interface. (new file flag for vn_sync_range:FMSYNC)

`gfs_ops`	Specifies the array of pointers to **vfs** operation implementations.
`gn_ops`	Specifies the array of pointers to v–node operation implementations.

The file system description structure can also specify:

`gfs_init`	Points to an initialization routine to be called by the **gfsadd** kernel service. This field must be null if no initialization routine is to be called.
`gfs_data`	Points to file system private data.

Execution Environment

The **gfsadd** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
EBUSY	Indicates that the file system type has already been installed.
EINVAL	Indicates that the *gfsno* value is larger than the system–defined maximum. The system–defined maximum is indicated in the **/usr/include/sys/vmount.h** file.

Implementation Specifics

The **gfsadd** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **gfsdel** kernel service.

gfsdel Kernel Service

Purpose

Removes a file system type from the **gfs** table.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int gfsdel (gfsno)
int  gfsno;
```

Parameter

gfsno	Specifies the file system number. This value identifies the type of the file system to be deleted.

Description

The **gfsdel** kernel service is called to delete a file system type. It is not valid to mount any file system of the given type after that type has been deleted.

Execution Environment

The **gfsdel** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
ENOENT	Indicates that the indicated file system type was not installed.
EINVAL	Indicates that the *gfsno* value is larger than the system–defined maximum. The system–defined maximum is indicated in the **/usr/include/sys/vmount.h** file.
EBUSY	Indicates that there are active **vfs** structures for the file system type being deleted.

Implementation Specifics

The **gfsdel** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Virtual File System Overview, Virtual File System Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

The **gfsadd** kernel service.

i_clear Kernel Service

Purpose

Removes an interrupt handler.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/intr.h>

void i_clear (handler)
struct intr *handler;
```

Parameter

handler	Specifies the address of the interrupt handler structure passed to the **i_init** service.

Description

The **i_clear** service removes the interrupt handler specified by the *handler* parameter from the set of interrupt handlers that the kernel knows about. ”Coding an Interrupt Handler” in *AIX Kernel Extensions and Device Support Programming Concepts* contains a brief description of interrupt handlers.

The **i_mask** service is called by the **i_clear** service to disable the interrupt handler’s bus interrupt level when this is the last interrupt handler for the bus interrupt level. The **i_clear** service removes the interrupt handler structure from the list of interrupt handlers. The kernel maintains this list for that bus interrupt level.

Execution Environment

The **i_clear** kernel service can be called from the process environment only.

Return Values

The **i_clear** service has no return values.

Implementation Specifics

The **i_clear** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **i_init** kernel service.

I/O Kernel Services, Understanding Interrupts in *AIX Kernel Extensions and Device Support Programming Concepts*.

i_disable Kernel Service

Purpose

Disables interrupt priorities.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/intr.h>

int i_disable (new)
int new;
```

Parameter

new	Specifies the new interrupt priority.

Description

Attention: The **i_disable** service has two side effects that result from the replaceable and pageable nature of the kernel. First, it prevents process dispatching. Second, it ensures, within limits, that the caller's stack is in memory. Page faults that occur while the interrupt priority is not equal to **INTBASE** crash the system.

Note: The **i_disable** service is very similar to the standard UNIX **spl** service.

The **i_disable** service sets the interrupt priority to a more favored interrupt priority. The interrupt priority is used to control which interrupts are allowed.

A value of **INTMAX** is the most favored priority and disables all interrupts. A value of **INTBASE** is the least favored and disables only interrupts not in use. The **/usr/include/sys/intr.h** file defines valid interrupt priorities.

The interrupt priority is changed only to serialize code executing in more than one environment (that is, process and interrupt environments).

For example, a device driver typically links requests in a list while executing under the calling process. The device driver's interrupt handler typically uses this list to initiate the next request. Therefore, the device driver must serialize updating this list with device interrupts. The **i_disable** and **i_enable** services provide this ability. The **I_init** kernel service contains a brief description of interrupt handlers.

Note: When serializing such code in a multiprocessor–safe kernel extension, locking must be used as well as interrupt control. For this reason, new code should call the **disable_lock** kernel service instead of **i_disable**. The **disable_lock** service performs locking only on multiprocessor systems, and helps ensure that code is portable between uniprocessor and multiprocessor systems.

The **i_disable** service must always be used with the **i_enable** service. A routine must always return with the interrupt priority restored to the value that it had upon entry.

The **i_mask** service can be used when a routine must disable its device across a return.

Because of these side effects, the caller of the **i_disable** service should ensure that:

- The reference parameters are pinned.
- The code executed during the disable operation is pinned.
- The amount of stack used during the disable operation is less than 1KB.
- The called programs use less than 1KB of stack.

In general, the caller of the **i_disable** service should also call only services that can be called by interrupt handlers. However, processes that call the **i_disable** service can call the **e_sleep**, **e_wait**, **e_sleepl**, **lockl**, and **unlockl** services as long as the event word or lockword is pinned.

The kernel's first–level interrupt handler sets the interrupt priority for an interrupt handler before calling the interrupt handler. The interrupt priority for a process is set to **INTBASE** when the process is created and is part of each process's state. The dispatcher sets the interrupt priority to the value associated with the process to be executed.

Execution Environment

The **i_disable** kernel service can be called from either the process or interrupt environment.

Return Value

The **i_disable** service returns the current interrupt priority that is subsequently used with the **i_enable** service.

Implementation Specifics

The **i_disable** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **disable_lock** kernel service, **i_enable** kernel service, **i_mask** kernel service.

I/O Kernel Services, Understanding Execution Environments, Understanding Interrupts in *AIX Kernel Extensions and Device Support Programming Concepts*.

i_enable Kernel Service

Purpose

Enables interrupt priorities.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/intr.h>

void i_enable (old)
int old;
```

Parameter

old	Specifies the interrupt priority returned by the **i_disable** service.

Description

The **i_enable** service restores the interrupt priority to a less–favored value. This value should be the value that was in effect before the corresponding call to the **i_disable** service.

Note: When serializing a thread with an interrupt handler in a multiprocessor–safe kernel extension, locking must be used as well as interrupt control. For this reason, new code should call the **unlock_enable** kernel service instead of **i_enable**. The **unlock_enable** service performs locking only on multiprocessor systems, and helps ensure that code is portable between uniprocessor and multiprocessor systems.

Execution Environment

The **i_enable** kernel service can be called from either the process or interrupt environment.

Return Values

The **i_enable** service has no return values.

Implementation Specifics

The **i_enable** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **i_disable** kernel service, **unlock_enable** kernel service.

Understanding Interrupts, I/O Kernel Services, Understanding Execution Environments in *AIX Kernel Extensions and Device Support Programming Concepts*.

ifa_ifwithaddr Kernel Service

Purpose

Locates an interface based on a complete address.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/socket.h>
#include  <net/if.h>
#include  <net/af.h>

struct ifaddr * ifa_ifwithaddr (addr)
struct sockaddr *addr;
```

Parameter

addr	Specifies a complete address.

Description

The **ifa_ifwithaddr** kernel service is passed a complete address and locates the corresponding interface. If successful, the **ifa_ifwithaddr** service returns the **ifaddr** structure associated with that address.

Execution Environment

The **ifa_ifwithaddr** kernel service can be called from either the process or interrupt environment.

Return Values

If successful, the **ifa_ifwithaddr** service returns the corresponding **ifaddr** structure associated with the address it is passed. If no interface is found, the **ifa_ifwithaddr** service returns a null pointer.

Example

To locate an interface based on a complete address, invoke the **ifa_ifwithaddr** kernel service as follows:

```
ifa_ifwithaddr((struct sockaddr *)&ipaddr);
```

Implementation Specifics

The **ifa_ifwithaddr** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **ifa_ifwithdstaddr** kernel service, **ifa_ifwithnet** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

ifa_ifwithdstaddr Kernel Service

Purpose

Locates the point–to–point interface with a given destination address.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/socket.h>
#include <net/if.h>

struct ifaddr * ifa_ifwithdstaddr (addr)
struct sockaddr *addr;
```

Parameter

addr	Specifies a destination address.

Description

The **ifa_ifwithdstaddr** kernel service searches the list of point–to–point addresses per interface and locates the connection with the destination address specified by the *addr* parameter.

Execution Environment

The **ifa_withdstaddr** kernel service can be called from either the process or interrupt environment.

Return Values

If successful, the **ifa_ifwithdstaddr** service returns the corresponding **ifaddr** structure associated with the point–to–point interface. If no interface is found, the **ifa_ifwithdstaddr** service returns a null pointer.

Example

To locate the point–to–point interface with a given destination address, invoke the **ifa_ifwithdstaddr** kernel service as follows:

```
ifa_ifwithdstaddr((struct sockaddr *)&ipaddr);
```

Implementation Specifics

The **ifa_ifwithdstaddr** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **ifa_ifwithaddr** kernel service, **ifa_ifwithnet** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

ifa_ifwithnet Kernel Service

Purpose

Locates an interface on a specific network.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/socket.h>
#include <net/if.h>

struct ifaddr * ifa_ifwithnet (addr)
register struct sockaddr *addr;
```

Parameter

addr	Specifies the address.

Description

The **ifa_ifwithnet** kernel service locates an interface that matches the network specified by the address it is passed. If more than one interface matches, the **ifa_ifwithnet** service returns the first interface found.

Execution Environment

The **ifa_ifwithnet** kernel service can be called from either the process or interrupt environment.

Return Values

If successful, the **ifa_ifwithnet** service returns the **ifaddr** structure of the correct interface. If no interface is found, the **ifa_ifwithnet** service returns a null pointer.

Example

To locate an interface on a specific network, invoke the **ifa_ifwithnet** kernel service as follows:

```
ifa_ifwithnet((struct sockaddr *)&ipaddr);
```

Implementation Specifics

The **ifa_ifwithnet** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **ifa_ifwithaddr** kernel service, **ifa_ifwithdstaddr** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

if_attach Kernel Service

Purpose

Adds a network interface to the network interface list.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <net/if.h>

if_attach (ifp)
struct ifnet  *ifp;
```

Parameter

ifp	Points to the interface network (**ifnet**) structure that defines the network interface.

Description

The **if_attach** kernel service registers a Network Interface Driver (NID) in the network interface list.

Execution Environment

The **if_attach** kernel service can be called from either the process or interrupt environment.

Return Values

The **if_attach** kernel service has no return values.

Implementation Specifics

The **if_attach** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **if_detach** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

if_detach Kernel Service

Purpose

Deletes a network interface from the network interface list.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <net/if.h>

if_detach (ifp)
struct ifnet *ifp;
```

Parameter

ifp	Points to the interface network (**ifnet**) structure that describes the network interface to delete.

Description

The **if_detach** kernel service deletes a Network Interface Driver (NID) entry from the network interface list.

Execution Environment

The **if_detach** kernel service can be called from either the process or interrupt environment.

Return Values

0	Indicates that the network interface was successfully deleted.
ENOENT	Indicates that the **if_detach** kernel service could not find the NID in the network interface list.

Implementation Specifics

The **if_detach** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **if_attach** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

if_down Kernel Service

Purpose

Marks an interface as down.

Syntax

#include <sys/types.h>
#include <sys/errno.h>
#include <net/if.h>

void if_down (*ifp***)**
register struct ifnet ****ifp*;**

Parameter

ifp	Specifies the **ifnet** structure associated with the interface array.

Description

The **if_down** kernel service:

- Marks an interface as down by setting the `flags` field of the **ifnet** structure appropriately.
- Notifies the protocols of the transaction.
- Flushes the output queue.

The *ifp* parameter specifies the **ifnet** structure associated with the interface as the structure to be marked as down.

Execution Environment

The **if_down** kernel service can be called from either the process or interrupt environment.

Return Values

The **if_down** service has no return values.

Example

To mark an interface as down, invoke the **if_down** kernel service as follows:

```
if_down(ifp);
```

Implementation Specifics

The **if_down** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

if_nostat Kernel Service

Purpose

Zeroes statistical elements of the interface array in preparation for an attach operation.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <net/if.h>

void if_nostat (ifp)
struct ifnet *ifp;
```

Parameter

ifp	Specifies the **ifnet** structure associated with the interface array.

Description

The **if_nostat** kernel service zeroes the statistic elements of the **ifnet** structure for the interface. The *ifp* parameter specifies the **ifnet** structure associated with the interface that is being attached. The **if_nostat** service is called from the interface attach routine.

Execution Environment

The **if_nostat** kernel service can be called from either the process or interrupt environment.

Return Values

The **if_nostat** service has no return values.

Example

To zero statistical elements of the interface array in preparation for an attach operation, invoke the **if_nostat** kernel service as follows:

```
if_nostat(ifp);
```

Implementation Specifics

The **if_nostat** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

ifunit Kernel Service

Purpose

Returns a pointer to the **ifnet** structure of the requested interface.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <net/if.h>

struct ifnet *
ifunit (name)
char *name;
```

Parameter

name	Specifies the name of an interface (for example, `en0`).

Description

The **ifunit** kernel service searches the list of configured interfaces for an interface specified by the *name* parameter. If a match is found, the **ifunit** service returns the address of the **ifnet** structure for that interface.

Execution Environment

The **ifunit** kernel service can be called from either the process or interrupt environment.

Return Values

The **ifunit** kernel service returns the address of the **ifnet** structure associated with the named interface. If the interface is not found, the service returns a null value.

Example

To return a pointer to the **ifnet** structure of the requested interface, invoke the **ifunit** kernel service as follows:

```
ifp = ifunit(”en0”);
```

Implementation Specifics

The **ifunit** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

i_init Kernel Service

Purpose

Defines an interrupt handler.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/intr.h>

int i_init
(handler)
struct intr *handler;
```

Parameter

handler	Designates the address of the pinned interrupt handler structure.

Description

Attention: The interrupt handler structure must not be altered between the call to the **i_init** service to define the interrupt handler and the call to the **i_clear** service to remove the interrupt handler. The structure must also stay pinned. If this structure is altered at those times, a kernel panic may result.

The **i_init** service allows device drivers to define an interrupt handler to the kernel. The interrupt handler **intr** structure pointed to by the *handler* parameter describes the interrupt handler. The caller of the **i_init** service must initialize all the fields in the **intr** structure. The /**usr**/**include**/**sys**/**intr.h** file defines these fields and their valid values.

The **i_init** service enables interrupts by linking the interrupt handler structure to the end of the list of interrupt handlers defined for that bus level. If this is the first interrupt handler for the specified bus interrupt level, the **i_init** service enables the bus interrupt level by calling the **i_unmask** service.

The interrupt handler can be called before the **i_init** service returns if the following two conditions are met:

- The caller of the **i_init** service is executing at a lower interrupt priority than the one defined for the interrupt.
- An interrupt for the device or another device on the same bus interrupt level is already pending.

On multiprocessor systems, all interrupt handlers defined with the **i_init** kernel service run by default on the first processor started when the system was booted. This ensures compatibility with uniprocessor interrupt handlers. If the interrupt handler being defined has been designed to be multiprocessor–safe, or is an EPOW (Early Power–Off Warning) or off–level interrupt handler, set the **INTR_MPSAFE** flag in the `flags` field of the **intr** structure passed to the **i_init** kernel service. The interrupt handler will then run on any available processor.

Coding an Interrupt Handler

The kernel calls the interrupt handler when an enabled interrupt occurs on that bus interrupt level. The interrupt handler is responsible for determining if the interrupt is from its own device and processing the interrupt. The interface to the interrupt handler is as follows: **int** *interrupt_handler* **(***handler***)**
struct intr ***handler*****;**

The *handler* parameter points to the same interrupt handler structure specified in the call to the **i_init** kernel service. The device driver can pass additional parameters to its interrupt

handler by declaring the interrupt handler structure to be part of a larger structure that contains these parameters.

The interrupt handler can return one of two return values. A value of **INTR_SUCC** indicates that the interrupt handler processed the interrupt and reset the interrupting device. A value of **INTR_FAIL** indicates that the interrupt was not from this interrupt handler's device.

Registering Early Power–Off Warning (EPOW) Routines

The **i_init** kernel service can also be used to register an EPOW (Early Power–Off Warning) notification routine.

The return value from the EPOW interrupt handler should be **INTR_SUCC**, which indicates that the interrupt was successfully handled. All registered EPOW interrupt handlers are called when an EPOW interrupt is indicated.

Execution Environment

The **i_init** kernel service can be called from the process environment only.

Return Values

INTR_SUCC	Indicates a successful completion.
INTR_FAIL	Indicates an unsuccessful completion. The **i_init** service did not define the interrupt handler. An unsuccessful completion occurs when there is a conflict between a shared and a nonshared bus interrupt level. An unsuccessful completion also occurs when more than one interrupt priority is assigned to a bus interrupt level.

Implementation Specifics

The **i_init** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Understanding Interrupts, I/O Kernel Services, in *AIX Kernel Extensions and Device Support Programming Concepts*.

i_mask Kernel Service

Purpose

Disables a bus interrupt level.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/intr.h>

void i_mask (handler)
struct intr *handler;
```

Parameter

handler	Specifies the address of the interrupt handler structure that was passed to the **i_init** service.

Description

The **i_mask** service disables the bus interrupt level specified by the *handler* parameter.

The **i_disable** and **i_enable** services are used to serialize the execution of various device driver routines with their device interrupts.

The **i_init** and **i_clear** services use the **i_mask** and **i_unmask** services internally to configure bus interrupt levels.

Device drivers can use the **i_disable**, **i_enable**, **i_mask**, and **i_unmask** services when they must perform off–level processing with their device interrupts disabled. Device drivers also use these services to allow process execution when their device interrupts are disabled.

Execution Environment

The **i_mask** kernel service can be called from either the process or interrupt environment.

Return Values

The **i_mask** service has no return values.

Implementation Specifics

The **i_mask** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **i_unmask** kernel service.

Understanding Interrupts, I/O Kernel Services, in *AIX Kernel Extensions and Device Support Programming Concepts*.

init_heap Kernel Service

Purpose

Initializes a new heap to be used with kernel memory management services.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/xmalloc.h>
#include <sys/malloc.h>

heapaddr_t init_heap (area, size, heapp)
caddr_t area;
int size;
heapaddr_t *heapp;
```

Parameters

area	Specifies the virtual memory address used to define the starting memory area for the heap. This address must be page–aligned.
size	Specifies the size of the heap in bytes. This value must be an integral number of system pages.
heapp	Points to the external heap descriptor. This must have a null value. The base kernel uses this field is used to specify special heap characteristics that are unavailable to kernel extensions.

Description

The **init_heap** kernel service is most commonly used by a kernel process to initialize and manage an area of virtual memory as a private heap. Once this service creates a private heap, the returned **heapaddr_t** value can be used with the **xmalloc** or **xmfree** service to allocate or deallocate memory from the private heap. Heaps can be created within other heaps, a kernel process private region, or even on a stack.

Few kernel extensions ever require the **init_heap** service because the exported global **kernel_heap** and **pinned_heap** are normally used for memory allocation within the kernel. However, kernel processes can use the **init_heap** service to create private nonglobal heaps within their process private region for controlling kernel access to the heap and possibly for performance considerations.

Execution Environment

The **init_heap** kernel service can be called from the process environment only.

Implementation Specifics

The **init_heap** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **xmalloc** kernel service, **xmfree** kernel service.

Memory Kernel Services and Using Kernel Processes in *AIX Kernel Extensions and Device Support Programming Concepts*.

initp Kernel Service

Purpose

Changes the state of a kernel process from idle to ready.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int initp
(pid, func, init_parms,
parms_length, name)
pid_t pid;
void (func) (int
flag,
void* init_parms, int parms_length );
void *init_parms;
int parms_length;
char *name;
```

Parameters

pid	Specifies the process identifier of the process to be initialized.
func	Specifies the process's initialization routine.
init_parm	Specifies the pointer to the initialization parameters.
parms_length	Specifies the length of the initialization parameters.
name	Specifies the process name.

Description

The **initp** kernel service completes the transition of a kernel process from idle to ready. The idle state for a process is represented by **p_status == SIDL**. Before calling the **initp** service, the **creatp** service is called to create the process. The **creatp** service allocates and initializes a process table entry.

The **initp** service creates and initializes the process–private segment. The process is marked as a kernel process by a bit set in the **p_flag** field in the process table entry. This bit, the SKPROC bit, signifies that the process is a kernel process.

The process calling the **initp** service to initialize a newly created process must be the same process that called the **creatp** service to create the new process.

"Using Kernel Processes" in *AIX Kernel Extensions and Device Support Programming Concepts* further explains how the **initp** kernel service completes the initialization process begun by the **creatp** service.

The *pid* parameter identifies the process to be initialized. It must be valid and identify a process in the SIDL (idle) state.

The *name* parameter points to a character string that names the process. The leading characters of this string are copied to the user structure. The number of characters copied is implementation–dependent, but at least four are always copied.

The *func* parameter indicates the main entry point of the process. The new process is made ready to run this function. If the *init_parms* parameter is not null, it points to data passed to this routine. The parameter structure must be agreed upon between the initializing and initialized process. The **initp** service copies the data specified by the *init_parm* parameter (with the exact number of bytes specified by the *parms_length* parameter) of data to the new process's stack.

Execution Environment

The **initp** kernel service can be called from the process environment only.

Example

To initialize the kernel process running the function *main_kproc*, enter:

```
{
.
.
.
pid = creatp();
initp(pid, main_kproc, &node_num, sizeof(int), ”tkproc”);

.
.
}
void
main_kproc(int flag, void* init_parms, int parms_length)
{
        .
        .
        .
        int i;
        i = *( (int *)init_parms );
        .
        .
        .
}
```

Return Values

0	Indicates a successful operation.
ENOMEM	Indicates that there was insufficient memory to initialize the process.
EINVAL	Indicates an *pid* parameter that was not valid.

Implementation Specifics

The **initp** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **creatp** kernel service.

The **func** subroutine.

Introduction to Kernel Processes and Process and Exception Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

initp Kernel Service func Subroutine

Purpose

Directs the process initialization routine.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

void func (flag, init_parms, parms_length)
int flag;
void *init_parms;
int parms_length;
```

Parameters

func	Specifies the process's initialization routine.
flag	Has a 0 value if this subroutine is executed as a result of initializing a process with the **initp** service.
init_parms	Specifies the pointer to the initialization parameters.
parms_length	Specifies the length of the initialization parameters.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **initp** kernel service.

Process and Exception Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

io_att Kernel Service

Purpose

Selects, allocates, and maps a region in the current address space for I/O access.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
#include <sys/adspace.h>

caddr_t io_att (iohandle, offset)
vmhandle_t iohandle;
caddr_t offset;
```

Parameters

iohandle	Specifies a handle for the I/O object to be mapped in the current address space.
offset	Specifies the address offset in both the I/O space and the virtual memory region to be mapped.

Description

Attention: The **io_att** service will crash the kernel if there are no more free regions.

The **io_att** kernel service performs these four tasks:

- Selects an unallocated virtual memory region.
- Allocates it.
- Maps the I/O address space specified by the *iohandle* parameter with the access permission specified in the handle.
- Constructs the address specified by the *offset* parameter in the current address space.

The **io_att** kernel service assumes an address space model of fixed–size I/O objects and virtual memory address space regions.

Execution Environment

The **io_att** kernel service can be called from either the process or interrupt environment.

Return Values

The **io_att** kernel service returns an address for the offset in the virtual memory address space.

Implementation Specifics

The **io_att** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **io_det** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

io_det Kernel Service

Purpose

Unmaps and deallocates the region in the current address space at the given address.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
#include <sys/adspace.h>

void io_det (eaddr)
caddr_t eaddr;
```

Parameter

eaddr	Specifies the effective address for the virtual memory region that is to be detached. This address should be the same address that was previously obtained by using the **io_att** kernel service to attach the virtual memory region.

Description

The **io_det** kernel service unmaps the region containing the address specified by the *eaddr* parameter and deallocates the region. This service then adds the region to the free list for the current address space.

The **io_det** service assumes an address space model of fixed–size I/O objects and address space regions.

Execution Environment

The **io_det** kernel service can be called from either the process or interrupt environment.

Return Values

The **io_det** kernel service has no return values.

Implementation Specifics

The **io_det** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **io_att** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

iodone Kernel Service

Purpose

Performs block I/O completion processing.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/buf.h>

void iodone (bp)
struct buf *bp;
```

Parameter

bp	Specifies the address of the **buf** structure for the buffer whose I/O has completed.

Description

A device driver calls the **iodone** kernel service when a block I/O request is complete. The device driver must not reference or alter the buffer header or buffer after calling the **iodone** service.

The **iodone** service takes one of two actions, depending on the current interrupt level. Either it invokes the caller's individual **iodone** routine directly, or it schedules I/O completion processing for the buffer to be performed off–level, at the **INTIODONE** interrupt level. The interrupt handler for this level then calls the iodone routine for the individual device driver. In either case, the individual iodone routine is defined by the `b_iodone` buffer header field in the buffer header. This iodone routine is set up by the caller of the device's strategy routine.

For example, the file I/O system calls set up a routine that performs buffered I/O completion processing. The **uphysio** service sets up a routine that performs raw I/O completion processing. Similarly, the pager sets up a routine that performs page–fault completion processing.

Setting up an iodone Routine

Under certain circumstances, a device driver can set up an **iodone** routine. For example, the logical volume device driver can follow this procedure:

1. Take a request for a logical volume.
2. Allocate a buffer header.
3. Convert the logical volume request into a physical volume request.
4. Update the allocated buffer header with the information about the physical volume request. This includes setting the `b_iodone` buffer header field to the address of the individual iodone routine.
5. Call the physical volume device driver strategy routine.

   Here, the caller of the logical volume strategy routine has set up an iodone routine that is started when the logical volume request is complete. The logical volume strategy routine in turn sets up an iodone routine that is invoked when the physical volume request is complete.

The key point of this example is that only the caller of a strategy routine can set up an iodone routine and even then, this can only be done while setting up the request in the buffer header.

The interface for the **iodone** routine is identical to the interface to the **iodone** service.

Execution Environment

The **iodone** kernel service can be called from either the process or interrupt environment.

Return Values

The **iodone** service has no return values.

Implementation Specifics

The **iodone** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **iowait** kernel service.

The **buf** structure.

Understanding Interrupts and I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

iomem_att Kernel Service

Purpose

Establishes access to memory–mapped I/O.

Syntax

```
#include <sys/types.h>
#include <sys/adspace.h>

void *iomem_att (io_map_ptr)
struct io_map *io_map_ptr;

struct io_map {
   int key;
   int flags;
   int size;
   int BID;
   long long busaddress;
}
```

Parameters

The address of the **io_map** structure passes the following parameters to the **iomem_att** kernel service:

key	Set to `IO_MEM_MAP`.
flags	Describes the mapping.
size	Specifies the number of bytes to map.
bid	Specifies the bus identifier.
busaddress	Specifies the address of the bus.

Description

Note: The **iomem_att** kernel service is only supported on PowerPC–based machines. All mappings are done with storage attributes: cache inhibited, guarded, and coherent. It is a violation of the PowerPC architecture to access memory with multiple storage modes. The caller of **iomem_att** must ensure no mappings using other storage attributes exist in the system.

Calling this function on a POWER–based machine causes the system to crash.

The **iomem_att** kernel service provides temporary addressability to memory–mapped I/O. The **iomem_att** kernel service does the following:

- Allocates one segment of kernel address space
- Establishes kernel addressability
- Maps a contiguous region of memory mapped I/O into that segment.

The addressability is valid only for the context that called **iomem_att**. The memory is addressable until **iomem_det** is called. I/O memory must be mapped each time a context is entered and freed before returning.

Note: Kernel address space is an exhaustible resource. When exhausted the system crashes. A driver must never map more than two I/O regions at once, or call another driver with an **iomem_att** outstanding. DMA, interrupt, and PIO kernel services can be called with up to two I/O regions mapped.

The *size* parameter supports from 4096 bytes to 256 MB. The caller can specify a minimum of *size* bytes, but may choose to map up to 256 MB. The caller must not reference memory beyond *size* bytes. The *size* parameter should be set to the minimum value required to address the target device.

Specifying **IOM_RDONLY** in the *flags* parameter results in a read–only mapping. A store to memory, mapped in this mode, results in a data storage interrupt. If the *flag* parameter is **0** (zero) the memory is mapped read–write. All mappings are read–write on 601–based machines.

Execution Environment

The **iomem_att** kernel service can be called from either the process or interrupt environment.

Return Values

The **iomem_att** kernel service returns the effective address that can be used to address the I/O memory.

Implementation Specifics

The **iomem_att** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **iomem_det** Kernel Service.

Kernel Extension and Device Driver Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

iomem_det Kernel Service

Purpose

Releases access to memory–mapped IO.

Syntax

```
#include <sys/types.h>
#include <sys/adspace.h>

void iomem_det (ioaddr)
void *ioaddr
```

Parameters

ioaddr	Specifies the effective address returned by the **iomem_att** kernel service.

Description

The **iomem_det** kernel service releases memory–mapped I/O addressability. A call to the **iomem_det** kernel service must be made for every **iomem_att** call, with the address that **iomem_att** returned.

Execution Evironment

The **iomem_det** kernel service can be called from either the process or interrupt environment.

Return Values

The **iomem_det** kernel service returns no return values.

Implementation Specifics

The **iomem_det** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **iomem_att** kernel service.

Kernel Extension and Device Driver Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

iostadd Kernel Service

Purpose

Registers an I/O statistics structure used for updating I/O statistics reported by the **iostat** subroutine.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/iostat.h>
#include <sys/devinfo.h>

int iostadd (devtype, devstatp)
int devtype;
union {
       struct ttystat *ttystp;
       struct dkstat  *dkstp;
       } devstatp;
```

Parameters

devtype — Specifies the type of device for which I/O statistics are kept. The various device types are defined in the **/usr/include/sys/devinfo.h** file. Currently, I/O statistics are only kept for disks, CD–ROMs, and tty devices. Possible values for this parameter are:

DD_DISK	For disks
DD_CD–ROM	For CD–ROMs
DD_TTY	For tty devices

devstatp — Points to an I/O statistics structure for the device type specified by the *devtype* parameter. For a *devtype* parameter of **DD_tty**, the address of a pinned **ttystat** structure is returned. For a *devtype* parameter of **DD_DISK** or **DD_CD–ROM**, the parameter is an input parameter pointing to a **dkstat** structure previously allocated by the caller.

Description

The **iostadd** kernel service is used to register the I/O statistics structure required to maintain statistics on a device. The **iostadd** service is typically called by a tty, disk, or CD–ROM device driver to provide the statistical information used by the **iostat** subroutine. The **iostat** subroutine displays statistic information for tty and disk devices on the system. The **iostadd** service should be used once for each configured device.

For tty devices, the *devtype* parameter has a value of **DD_tty**. In this case, the **iostadd** service uses the *devstatp* parameter to return a pointer to a **ttystat** structure.

For disk or CD–ROM devices with a *devtype* value of **DD_DISK** or **DD_CD–ROM**, the caller must provide a pinned and initialized **dkstat** structure as an input parameter. This structure is pointed to by the *devstatp* parameter on entry to the **iostadd** kernel service.

If the device driver support for a device is terminated, the **dkstat** or **ttystat** structure registered with the **iostadd** kernel service should be deregistered by calling the **iostdel** kernel service.

I/O Statistics Structures

The **iostadd** kernel service uses two structures that are found in the **usr/include/sys/iostat.h** file: the **ttystat** structure and the **dkstat** structure.

The **ttystat** structure contains the following tty–related fields:

rawinch	Count of raw characters received by the tty device
caninch	Count of canonical characters generated from canonical processing
outch	Count of the characters output to a tty device

The second structure used by the **iostadd** kernel service is the **dkstat** structure, which contains information about disk devices. This structure contains the following fields:

diskname	32–character string name for the disk's logical device
dknextp	Pointer to the next **dkstat** structure in the chain
dk_status	Disk entry–status flags
dk_time	Time the disk is active
dk_bsize	Number of bytes in a block
dk_xfers	Number of transfers to or from the disk
dk_rblks	Number of blocks read from the disk
dk_wblks	Number of blocks written to the disk
dk_seeks	Number of seek operations for disks

tty Device Driver Support

The rawinch field in the **ttystat** structure should be incremented by the number of characters received by the tty device. The caninch field in the **ttystat** structure should be incremented by the number of input characters generated from canonical processing. The outch field is increased by the number of characters output to tty devices. These fields should be incremented by the device driver, but never be cleared.

Disk Device Driver Support

A disk device driver must perform these four tasks:

- Allocate and pin a **dkstat** structure during device initialization.
- Update the dkstat.diskname field with the device's logical name.
- Update the dkstat.dk_bsize field with the number of bytes in a block on the device.
- Set all other fields in the structure to 0.

If the device supports discrete seek commands, the dkstat.dk_xrate field in the structure should be set to the transfer rate capability of the device (KB/sec). The device's **dkstat** structure should then be registered using the **iostadd** kernel service.

During drive operation update, the dkstat.dk_status field should show the busy/nonbusy state of the device. This can be done by setting and resetting the **IOST_DK_BUSY** flag. The dkstat.dk_xfers field should be incremented for each transfer initiated to or from the device. The dkstat.dk_rblks and dkstat.dk_wblks fields should be incremented by the number of blocks read or written.

If the device supports discrete seek commands, the dkstat.dk_seek field should be incremented by the number of seek commands sent to the device. If the device does not support discrete seek commands, both the dkstat.dk_seek and dkstat.dk_xrate fields should be left with a value of 0.

The base kernel updates the dkstat.dk_nextp and dkstat.dk_time fields. They should not be modified by the device driver after initialization.

Note: The same **dkstat** structure must not be registered more than once.

Execution Environment

The **iostadd** kernel service can be called from the process environment only.

Return Values

0	Indicates that no error has been detected.
EINVAL	Indicates that the *devtype* parameter specified a device type that is not valid.

Implementation Specifics

The **iostadd** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **iostat** command.

The **iostdel** kernel service.

Kernel Extension and Device Driver Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

iostdel Kernel Service

Purpose

Removes the registration of an I/O statistics structure used for maintaining I/O statistics on a particular device.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/iostat.h>

void iostdel (devstatp)
union {
        struct ttystat *ttystp;
        struct dkstat  *dkstp;
      } devstatp;
```

Parameter

devstatp	Points to an I/O statistics structure previously registered using the **iostadd** kernel service.

Description

The **iostdel** kernel service removes the registration of an I/O statistics structure for a device being terminated. The device's **ttystat** or **dkstat** structure should have previously been registered using the **iostadd** kernel service. Following a return from the **iostdel** service, the **iostat** command will no longer display statistics for the device being terminated.

Execution Environment

The **iostdel** kernel service can be called from the process environment only.

Return Values

The **iostdel** service has no return values.

Implementation Specifics

The **iostdel** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **iostat** command.

The **iostadd** kernel service.

Kernel Extension and Device Driver Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

iowait Kernel Service

Purpose

Waits for block I/O completion.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/buf.h>

int iowait (bp)
struct buf *bp;
```

Parameter

bp	Specifies the address of the **buf** structure for the buffer with in–process I/O.

Description

The **iowait** kernel service causes a process to wait until the I/O is complete for the buffer specified by the *bp* parameter. Only the caller of the strategy routine can call the **iowait** service. The **B_ASYNC** bit in the buffer's `b_flags` field should not be set.

The **iodone** kernel service must be called when the block I/O transfer is complete. The **buf** structure pointed to by the *bp* parameter must specify an iodone routine. This routine is called by the iodone interrupt handler in response to the call to the **iodone** kernel service. This iodone routine must call the **e_wakeup** service with the `bp->b_events` field as the event. This action awakens all processes waiting on I/O completion for the **buf** structure using the **iowait** service.

Execution Environment

The **iowait** kernel service can be called from the process environment only.

Return Values

The **iowait** service uses the **geterror** service to determine which of the following values to return:

0	Indicates that I/O was successful on this buffer.
EIO	Indicates that an I/O error has occurred.
b_error value	Indicates that an I/O error has occurred on the buffer.

Implementation Specifics

The **iowait** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **geterror** kernel service, **iodone** kernel service.

The **buf** structure.

ip_fltr_in_hook, ip_fltr_out_hook, ipsec_decap_hook Kernel Service

Purpose

Contains hooks for IP filtering.

Syntax

```
#define FIREWALL_OK          0  /* Accept IP packet
     */
#define FIREWALL_NOTOR       1  /* Drop IP packet
     */
#define FIREWALL_OK_NOTSEC   2  /* Accept non-encapsulated IP
packet
                                   (ipsec_decap_hook only)
*/
#include <sys/mbuf.h>
#include <net/if.h>

int (*ip_fltr_in_hook)(struct mbuf **pkt, void **arg)

int (*ipsec_decap_hook)(struct mbuf **pkt, void **arg)

int (*ip_fltr_out_hook)(struct ifnet *ifp, struct mbuf **pkt, int
flags)
```

Parameters

pkt	Points to the mbuf chain containing the IP packet to be received (**ip_fltr_in_hook**, **ipsec_decap_hook**) or transmitted **(ip_fltr_out_hook**). The *pkt* parameter may be examined and/or changed in any of the three hook functions.
arg	Is the address of a pointer to *void* that is locally defined in the function where **ip_fltr_in_hook** and **ipsec_decap_hook** are called. The *arg* parameter is initially set to NULL, but the address of this pointer is passed to the two hook functions, **ip_fltr_in_hook** and **ipsec_decap_hook**. The *arg* parameter may be set by either of these functions, thereby allowing a void pointer to be shared between them.
ifp	Is the outgoing interface on which the IP packet will be transmitted for the **ip_fltr_out_hook** function.
flags	Indicates the ip_output flags passed by a transport layer protocol. Valid flags are currently defined in the **/usr/include/netinet/ip_var.h** files. See the Flags section below.

Description

These routines provide kernel–level hooks for IP packet filtering enabling IP packets to be selectively accepted, rejected, or modified during reception, transmission, and

decapsulation. These hooks are initially NULL, but are exported by the netinet kernel extension and will be invoked if assigned non–NULL values.

The **ip_fltr_in_hook** routine is used to filter incoming IP packets, the **ip_fltr_out_hook** routine filters outgoing IP packets, and the **ipsec_decap_hook** routine filters incoming encapsulated IP packets.

The **ip_fltr_in_hook** function is invoked for every IP packet received by the host, whether addressed directly to this host or not. It is called after verifying the integrity and consistency of the IP packet. The function is free to examine or change the IP packet (*pkt*) or the pointer shared with **ipsec_decap_hook** (*arg*). The return value of the **ip_fltr_in_hook** indicates whether *pkt* should be accepted or dropped. The return values are described in Expected Return Values below. If *pkt* is accepted (a return value of **FIREWALL_OK**) and it is addressed directly to the host, the **ipsec_decap_hook** function is invoked next. If *pkt* is accepted, but is not directly addressed to the host, it is forwarded if IP forwarding is enabled. If **ip_fltr_in_hook** indicates *pkt* should be dropped (a return value of **FIREWALL_NOTOK**), it is neither delivered nor forwarded.

The **ipsec_decap_hook** function is called after reassembly of any IP fragments (the **ip_fltr_in_hook** function will have examined each of the IP fragments) and is invoked only for IP packets that are directly addressed to the host. The **ipsec_decap_hook** function is free to examine or change the IP packet (*pkt*) or the pointer shared with **ipsec_decap_hook** (*arg*). The hook function should perform decapsulation if necessary, back into *pkt* and return the proper status so that the IP packet can be processed appropriately. See the Expected Return Values section below. For acceptable encapsulated IP packets (a return value of **FIREWALL_OK**), the decapsulated packet is processed again by jumping to the beginning of the IP input processing loop. Consequently, the decapsulated IP packet will be examined first by **ip_fltr_in_hook** and, if addressed to the host, by **ipsec_decap_hook**. For acceptable non–encapsulated IP packets (a return value of **FIREWALL_OK_NOTSEC**), IP packet delivery simply continues and *pkt* is processed by the transport layer. A return value of **FIREWALL_NOTOK** indicates that *pkt* should be dropped.

The **ip_fltr_out_hook** function is called for every IP packet to be transmitted, provided the outgoing IP packet's destination IP address is NOT an IP multicast address. If it is, it is sent immediately, bypassing the **ip_fltr_out_hook** function. This hook function is invoked after inserting the IP options from the upper protocol layers, constructing the complete IP header, and locating a route to the destination IP address. The **ip_fltr_out_hook** function may modify the outgoing IP packet (*pkt*), but the interface and route have already been assigned and may not be changed. The return value from the **ip_fltr_out_hook** function indicates whether *pkt* should be transmitted or dropped. See the Expected Return Values section below. If *pkt* is not dropped (**FIREWALL_OK**), it's source address is verified to be local and, if *pkt* is to be broadcast, the ability to broadcast is confirmed. Thereafter, *pkt* is enqueued on the interface's (*ifp*) output queue. If *pk*t is dropped (**FIREWALL_NOTOK**), it is not transmitted and **EACCES** is returned to the process.

Flags

IP_FORWARDING	Indicates that most of the IP headers exist.
IP_RAWOUTPUT	Indicates that the raw IP header exists.
IP_MULTICAST_OPTS	Indicates that multicast options are present.
IP_ROUTETOIF	Contains bypass routing tables.
IP_ALLOWBROADCAST	Provides capability to send broadcast packets.
IP_BROADCASTOPTS	Contains broadcast options inside.
IP_PMTUOPTS	Provides PMTU discovery options.
IP_GROUP_ROUTING	Contains group routing gidlist.

Expected Return Values

FIREWALL_OK	Indicates that *pkt* is acceptable for any of the filtering functions. It will be delivered, forwarded, or transmitted as appropriate.
FIREWALL_NOTOK	Indicates that *pkt* should be dropped. It will not be received (**ip_fltr_in_hook**, **ipsec_decap_hook**) or transmitted (**ip_fltr_out_hook**).
FIREWALL_OK_NOTSEC	Indicates a return value only valid for the **ipsec_decap_hook** function. This indicates that *pkt* is acceptable according to the filtering rules, but is not encapsulated; *pkt* will be processed by the transport layer rather than processed as a decapsulated IP packet.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

See Network Kernel Services *AIX Kernel Extensions and Device Support Programming Concepts*.

i_pollsched Kernel Service

Purpose

Queue a pseudo interrupt to an interrupt handler list.

Syntax

```
#include <sys/intr.h>
int i_pollsched (handler, pril)
struct intr *handler;
int pril;
```

Parameters

handler	Pointer to the **intr** structure for which the interrupt is to be queued.
pril	Processor level to queue logical interrupt for.

Description

The **i_pollsched** service allows device drivers to queue a pseudo interrupt to another interrupt handler. The calling arguements are mutually exclusive. If *handler* is not NULL then it is used to generate a *pril* value, via **pal_i_genplvl** subroutine. If the *handler* is NULL then the value in *pril* represents the processor level of the target interrupt *handler.*

This service will not queue an interrupt to a funneled, or nonMPSAFE interrupt *handler*, unless the service is executing on the MPMASTER processor. INTR_FAIL will be returned if not executing on MPMASTER processor and the target interrupt handler is not MPSAFE.

This service should only be called on an RSPC based platform. Calling this service on a non–RSPC machine will always result in a failure return code.

Execution Environment

The **i_pollsched** kernel service can be called from either the process of interrupt environments.

Return Values

INTR_SUCC	Interrupted was queued.
INTR_FAIL	Interrupt was not queued. This can be returned when the target list was NULL or the service was called on an invalid platform.

Implementation Specifics

The **i_pollsched** kernel service is part of the Base Operating System (BOS) Runtime.

i_reset Kernel Service

Purpose

Resets a bus interrupt level.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/intr.h>

void i_reset (handler)
struct intr *handler;
```

Parameter

handler	Specifies the address of an interrupt handler structure passed to the **i_init** service.

Description

The **i_reset** service resets the bus interrupt specified by the *handler* parameter. A device interrupt handler calls the **i_reset** service after resetting the interrupt at the device on the bus. See **i_init** kernel service for a brief description of interrupt handlers.

Execution Environment

The **i_reset** kernel service can be called from either the process or interrupt environment.

Return Values

The **i_reset** service has no return values.

Implementation Specifics

The **i_reset** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **i_init** kernel service.

Understanding Interrupts, I/O Kernel Services, Processing Interrupts in *AIX Kernel Extensions and Device Support Programming Concepts*.

i_sched Kernel Service

Purpose

Schedules off–level processing.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/intr.h>

void i_sched (handler)
struct intr *handler;
```

Parameter

handler	Specifies the address of the pinned interrupt handler structure.

Description

The **i_sched** service allows device drivers to schedule some of their work to be processed at a less–favored interrupt priority. This capability allows interrupt handlers to run as quickly as possible, avoiding interrupt–processing delays and overrun conditions. See the **i_init** kernel service for a brief description of interrupt handlers.

Processing can be scheduled off–level in the following situations:

- The interrupt handler routine for a device driver must perform time–consuming processing.
- This work does not need to be performed immediately.

  Attention: The caller cannot alter any fields in the **intr** structure from the time the **i_sched** service is called until the kernel calls the off–level routine. The structure must also stay pinned. Otherwise, the system may crash.

The interrupt handler structure pointed to by the *handler* parameter describes an off–level interrupt handler. The caller of the **i_sched** service must set up all fields in the **intr** structure. The **INIT_OFFL***n* macros in the **/usr/include/sys/intr.h** file can be used to initialize the *handler* parameter. The *n* value represents the priority class that the off–level handler should run at. Currently, classes from 0 to 3 are defined.

Use of the **i_sched** service has two additional restrictions:

First, the **i_sched** service will not re–register an **intr** structure that is already registered for off–level handling. Since **i_sched** has no return value, the service will simply return normally without registering the specified structure if it was already registered but not yet executed. The kernel removes the **intr** structure from the registration list immediately prior to calling the off–level handler specified in the structure. It is therefore possible for the off–level handler to use the structure again to register another off–level request.

Care must be taken when scheduling off–level requests from a second–level interrupt handler (SLIH). If the off–level request is already registered but has not yet executed, a second registration will be ignored. If the off–level handler is currently executing, or has already run, a new request will be registered. Users of this service should be aware of these timing considerations and program accordingly.

Second, the kernel uses the `flags` field in the specified **intr** structure to determine if this structure is already registered. This field should be initialized once before the first call to the **i_sched** service and should remain unmodified for future calls to the **i_sched** service.

Note: Off–level interrupt handler path length should not exceed 5,000 instructions. If it does exceed this number, real–time support is adversely affected.

Execution Environment

The **i_sched** kernel service can be called from either the process or interrupt environment.

Return Values

The **i_sched** service has no return values.

Implementation Specifics

The **i_sched** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **i_init** kernel service.

Understanding Interrupts, I/O Kernel Services, Processing Interrupts in *AIX Kernel Extensions and Device Support Programming Concepts*.

i_unmask Kernel Service

Purpose

Enables a bus interrupt level.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/intr.h>

void i_unmask (handler)
struct intr *handler;
```

Parameter

handler	Specifies the address of the interrupt handler structure that was passed to the **i_init** service.

Description

The **i_unmask** service enables the bus interrupt level specified by the *handler* parameter.

Execution Environment

The **i_unmask** kernel service can be called from either the process or interrupt environment.

Return Values

The **i_unmask** service has no return values.

Implementation Specifics

The **i_unmask** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **i_init** kernel service, **i_mask** kernel service.

Understanding Interrupts, I/O Kernel Services, Processing Interrupts in *AIX Kernel Extensions and Device Support Programming Concepts*.

IS64U Kernel Service

Purpose

Determines if the current user–address space is 64–bit or not.

Syntax

#include <sys/types.h>
#include <sys/user.h>

int IS64U

Description

The **IS64U** kernel service returns 1 if the current user–address space is 64–bit. It returns 0 otherwise.

Execution Environment

The **IS64U** kernel service can be called from a process or interrupt handler environment. In either case, it will operate only on the current user–address space.

Return Values

0	The current user–address space is 32–bits.
1	The current user–address space is 64–bits.

Implementation Specifics

The **IS64U** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **as_att** kernel service, **as_det** kernel service, **as_geth** kernel service, **as_getsrval** kernel service, **as_puth** kernel service, **getadsp** kernel service, and **as_att64** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

kgethostname Kernel Service

Purpose

Retrieves the name of the current host.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int
kgethostname (name,
namelen)
char *name;
int *namelen;
```

Parameters

name	Specifies the address of the buffer in which to place the host name.
namelen	Specifies the address of a variable in which the length of the host name will be stored. This parameter should be set to the size of the buffer before the **kgethostname** kernel service is called.

Description

The **kgethostname** kernel service returns the standard name of the current host as set by the **sethostname** subroutine. The returned host name is null–terminated unless insufficient space is provided.

Execution Environment

The **kgethostname** kernel service can be called from either the process or interrupt environment.

Return Value

0	Indicates successful completion.

Implementation Specifics

The **kgethostname** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **sethostname** subroutine.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

kgettickd Kernel Service

Purpose

Retrieves the current status of the systemwide time–of–day timer–adjustment values.

Syntax

```
#include <sys/types.h>

int kgettickd (timed, tickd, time_adjusted)
int *timed;
int *tickd;
int *time_adjusted;
```

Parameters

timed	Specifies the current amount of time adjustment in microseconds remaining to be applied to the systemwide timer.
tickd	Specifies the time–adjustment rate in microseconds.
time_adjusted	Indicates if the systemwide timer has been adjusted. A value of True indicates that the timer has been adjusted by a call to the **adjtime** or **settimer** subroutine. A value of False indicates that it has not. The use of the **ksettimer** kernel service has no effect on this flag. This flag can be changed by the **ksettickd** kernel service.

Description

The **kgettickd** kernel service provides kernel extensions with the capability to determine if the **adjtime** or **settimer** subroutine has adjusted or changed the systemwide timer.

The **kgettickd** kernel service is typically used only by kernel extensions providing time synchronization functions. This includes coordinated network time (which is the periodic synchronization of all system clocks to a common time by a time server or set of time servers on a network), where use of the **adjtime** subroutine is insufficient.

Execution Environment

The **kgettickd** kernel service can be called from either the process or interrupt environment.

Return Values

The **kgettickd** service always returns a value of 0.

Implementation Specifics

The **kgettickd** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **ksettimer** kernel service.

The **adjtime** subroutine, **settimer** subroutine.

Timer and Time–of–Day Kernel Services and Using Fine Granularity Timer Services and Structures in *AIX Kernel Extensions and Device Support Programming Concepts*.

kmod_entrypt Kernel Service

Purpose

Returns a function pointer to a kernel module's entry point.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/ldr.h>

void (*(kmod_entrypt (kmid, flags)))( )
mid_t kmid;
uint flags;
```

Parameters

kmid	Specifies the kernel module ID of the object file for which the entry point is requested. This parameter is the kernel module ID returned by the **kmod_load** kernel service.
flags	Flag specifying entry point options. The following flag is defined: **0** Returns a function pointer to the specified module's entry point as specified in the module header.

Description

The **kmod_entrypt** kernel service obtains a function pointer to a specified module's entry point. This function pointer is typically used to invoke a routine in the module for initializing or terminating its functions. Initialization and termination occurs after loading and before unloading. The module for which the entry point is requested is specified by the kernel module ID represented by the *kmid* parameter.

Execution Environment

The **kmod_entrypt** kernel service can be called from the process environment only.

Return Values

A nonnull function pointer indicates a successful completion. This function pointer contains the module's entry point. A null function pointer indicates an error.

Implementation Specifics

The **kmod_entrypt** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **kmod_load** kernel service.

Kernel Extension and Device Driver Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

kmod_load Kernel Service

Purpose

Loads an object file into the kernel or queries for an object file already loaded.

Syntax

```
#include <sys/ldr.h>
#include <sys/types.h>
#include <sys/errno.h>

int kmod_load (pathp, flags,libpathp, kmidp)
caddr_t pathp;
uint flags;
caddr_t libpathp;
mid_t *kmidp;
```

Parameters

pathp Points to a character string containing the path name of the object file to load or query.

flags Specifies a set of loader flags describing which loader options to invoke. The following flags are defined:

LD_USRPATH The character strings pointed to by the *pathp* and *libpathp* parameters are in user address space. If the **LD_USRPATH** flag is not set, the character strings are assumed to be in kernel, or system, space.

LD_KERNELEX Puts this object file's exported symbols into the **/usr/lib/boot/unix** name space. Additional object files loaded due to symbol resolution for the specified file do not have their exported symbols placed in kernel name space.

LD_SINGLELOAD When this flag is set, the object file specified by the *pathp* parameter is loaded into the kernel only if an object file with the same path name has not already been loaded. If an object file with the same path name has already been loaded, its module ID is returned (using the *kmidp* parameter) and its load count incremented. If the object file is yet not loaded, this service performs the load as if the flag were not set.

This option is useful in supporting global kernel routines where only one copy of the routine and its data can be present. Typically, routines that export symbols to be added to kernel name space are of this type.

Note: A path–name comparison is done to determine whether the same object file has already been loaded. This service will erroneously load a new copy of the object file into the kernel if the path name to the object file is expressed differently than it was on a previous load request.

If neither this flag nor the **LD_QUERY** flag is set, this service loads a new copy of the object file into the kernel. This occurs even if other copies of the object file have previously been loaded.

LD_QUERY This flag specifies that a query operation will determine if the object file specified by the *pathp* parameter is loaded. If not loaded, a kernel module ID of 0 is returned using the *kmidp* parameter. Otherwise, the kernel module ID assigned to the object file is returned.

If multiple instances of this file have been loaded into the kernel, the kernel module ID of the most recently loaded object file is returned.

The *libpathp* parameter is not used for this option.

Note: A path–name comparison is done to determine whether the same object file has been loaded. This service will erroneously return a `not loaded` condition if the path name to the object file is expressed differently than it was on a previous load request.

If this flag is set, no object file is loaded and the **LD_SINGLELOAD** and **LD_KERNELEX** flags are ignored, if set.

libpathp	Points to a character string containing the search path to use for finding object files required to complete symbol resolution for this load. If the parameter is null, the search path is set from the specification in the object file header for the object file specified by the *pathp* parameter.
kmidp	Points to an area where the kernel module ID associated with this load of the specified module is to be returned. The data in this area is not valid if the **kmod_load** service returns a nonzero return code.

Description

The **kmod_load** kernel service loads into the kernel a kernel extension object file specified by the *pathp* parameter. This service returns a kernel module ID for that instance of the module.

You can specify flags to request a single load, which ensures that only one copy of the object file is loaded into the kernel. An additional option is simply to query for a given object file (specified by path name). This allows the user to determine if a module is already loaded and then access its assigned kernel module ID.

The **kmod_load** service also provides load–time symbol resolution of the loaded module's imported symbols. The **kmod_load** service loads additional kernel object modules if required for symbol resolution.

Loader Symbol Binding Support

Symbols imported from the kernel name space are resolved with symbols that exist in the kernel name space at the time of the load. (Symbols are imported from the kernel name space by specifying the **#!/unix** character string as the first field in an import list at link–edit time.)

Kernel modules can also import symbols from other kernel object modules. These other kernel object modules are loaded along with the specified object module if they are needed to resolve the imported symbols.

Any symbols exported by the specified kernel object module are added to the kernel name space if the *flags* parameter has the **LD_KERNELEX** flag set. This makes the symbols available to other subsequently loaded kernel object modules. Kernel object modules loaded on behalf of the specified kernel object module (to resolve imported symbols) do not have their exported symbols added to the kernel name space.

Kernel export symbols specified (at link–edit time) with the **SYSCALL** keyword in the primary module's export list are added to the system call table. These kernel export symbols are available to application programs as system calls.

Finding Shared Object Modules for Resolving Symbol References

The search path search string is taken from the module header of the object module specified by the *pathp* parameter if the *libpathp* parameter is null. The module header of the object module specified by the *pathp* parameter is used.

If the module header contains an unqualified base file name for the symbol (no / [slash] characters in the name), a search string is used to find the location of the shared object module required to resolve the import. This search string can be taken from one of two places. If the *libpathp* parameter on the call to the **kmod_load** service is not null, then it points to a character string specifying the search path to be used. However, if the *libpathp* parameter is null, then the search path is to be taken from the module header for the object module specified by the *pathp* parameter.

The search path specification found in object modules loaded to resolve imported symbols is not used. The kernel loader service does not support deferred symbol resolution. The load of the kernel module is terminated with an error if any imported symbols cannot be resolved.

Execution Environment

The **kmod_load** kernel service can be called from the process environment only.

Return Values

If the object file is loaded without error, the module ID is returned in the location pointed to by the *kmidp* parameter and the return code is set to 0.

Error Codes

If an error results, the module is not loaded, and no kernel module ID is returned. The return code is set to one of the following return values:

EACCES	Indicates that an object module to be loaded is not an ordinary file or that the mode of the object module file denies read–only access.
EACCES	Search permission is denied on a component of the path prefix.
EFAULT	Indicates that the calling process does not have sufficient authority to access the data area described by the *pathp* or *libpathp* parameters when the **LD_USRPATH** flag is set. This error code is also returned if an I/O error occurs when accessing data in this area.
ENOEXEC	Indicates that the program file has the appropriate access permission, but has an XCOFF indicator that is not valid in its header. The **kmod_load** kernel service supports loading of XCOFF (Extended Common Object File Format) object files only. This error code is also returned if the loader is unable to resolve an imported symbol.
EINVAL	Indicates that the program file has a valid XCOFF indicator in its header, but the header is either damaged or incorrect for the machine on which the file is to be loaded.
ENOMEM	Indicates that the load requires more kernel memory than allowed by the system–imposed maximum.
ETXTBSY	Indicates that the object file is currently open for writing by some process.
ENOTDIR	Indicates that a component of the path prefix is not a directory.
ENOENT	Indicates that no such file or directory exists or the path name is null.
ESTALE	Indicates that the caller's root or current directory is located in a virtual file system that has been unmounted.
ELOOP	Indicates that too many symbolic links were encountered in translating the *path* or *libpathp* parameter.
ENAMETOOLONG	Indicates that a component of a path name exceeded 255 characters, or an entire path name exceeded 1023 characters.
EIO	Indicates that an I/O error occurred during the operation.

Implementation Specifics

The **kmod_unload** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **kmod_unload** kernel service.

Kernel Extension and Device Driver Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

kmod_unload Kernel Service

Purpose

Unloads a kernel object file.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/ldr.h>

int kmod_unload (kmid, flags)
mid_t kmid;
uint flags;
```

Parameters

kmid	Specifies the kernel module ID of the object file to be unloaded. This kernel module ID is returned when using the **kmod_load** kernel service.
flags	Flags specifying unload options. The following flag is defined: **0** Unloads the object module specified by its *kmid* parameter and any object modules that were loaded as a result of loading the specified object file if this file is not still in use.

Description

The **kmod_unload** kernel service unloads a previously loaded kernel extension object file. The object to be unloaded is specified by the *kmid* parameter. Upon successful completion, the following objects are unloaded or marked *unload pending*:

- The specified object file
- Any imported kernel object modules that were loaded as a result of the loading of the specified module

Users of these exports or system calls are modules bound to this module's exported symbols. If there are no users of any of the module's kernel exports or system calls, the module is immediately unloaded. If there are users of this module, the module is not unloaded but marked *unload pending*.

Marking a module *unload pending* removes the module's exported symbols from the kernel name space. Any system calls exported by this module are also removed. This prohibits new users of these symbols. The module is unloaded only when all current users have been unloaded.

If the unload is successfully completed or marked *pending*, a value of 0 is returned. When an error occurs, the specified module and any imported modules are not unloaded. A nonzero return value indicates the error.

Execution Environment

The **kmod_unload** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
EINVAL	Indicates that the *kmid* parameter, which specifies the kernel module, is not valid or does not correspond to a currently loaded module.

Implementation Specifics

The **kmod_unload** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **kmod_load** kernel service.

Kernel Extension and Device Driver Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

kmsgctl Kernel Service

Purpose

Provides message–queue control operations.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int kmsgctl (msqid, cmd, buf)
int msqid, cmd;
struct msqid_ds *buf;
```

Parameters

msqid	Specifies the message queue ID, which indicates the message queue for which the control operation is being requested for.
cmd	Specifies which control operation is being requested. There are three valid commands.
buf	Points to the **msqid_ds** structure provided by the caller of the **kmsgctl** service. Data is obtained either from this structure or from status returned in this structure, depending on the *cmd* parameter. The **msqid_ds** structure is defined in the /**usr**/**include**/**sys**/**msg.h** file.

Description

The **kmsgctl** kernel service provides a variety of message–queue control operations as specified by the *cmd* parameter. The **kmsgctl** kernel service provides the same functions for user–mode processes in kernel mode as the **msgctl** subroutine performs for kernel processes or user–mode processes in user mode. The **kmsgctl** service can be called by a user–mode process in kernel mode or by a kernel process. A kernel process can also call the **msgctl** subroutine to provide the same function.

The following three commands can be specified with the *cmd* parameter:

IPC_STAT Sets only documented fields. See the **msgctl** subroutine.

IPC_SET Sets the value of the following fields of the data structure associated with the *msqid* parameter to the corresponding values found in the structure pointed to by the *buf* parameter:

- `msg_perm.uid`
- `msg_perm.gid`
- `msg_perm.mode` (only the low–order 9 bits)
- `msg_qbytes`

To perform the **IPC_SET** operation, the current process must have an effective user ID equal to the value of the `msg_perm.uid` or `msg_perm.cuid` field in the data structure associated with the *msqid* parameter. To raise the value of the `msg_qbytes` field, the calling process must have the appropriate system privilege.

IPC_RMID Removes from the system the message–queue identifier specified by the *msqid* parameter. This operation also destroys both the message queue and the data structure associated with it. To perform this operation, the current process must have an effective user ID equal to the value of the `msg_perm.uid` or `msg_perm.cuid` field in the data structure associated with the *msqid* parameter.

Execution Environment

The **kmsgctl** kernel service can be called from the process environment only.

Return Values

0 Indicates successful completion.

EINVAL Indicates either

- The identifier specified by the *msqid* parameter is not a valid message queue identifier.
- The command specified by the *cmd* parameter is not a valid command.

EACCES The command specified by the *cmd* parameter is equal to **IPC_STAT** and read permission is denied to the calling process.

EPERM The command specified by the *cmd* parameter is equal to **IPC_RMID**, **IPC_SET**, and the effective user ID of the calling process is not equal to that of the value of the `msg_perm.uid` field in the data structure associated with the *msqid* parameter.

EPERM Indicates the following conditions:

- The command specified by the *cmd* parameter is equal to **IPC_SET**.
- An attempt is being made to increase to the value of the `msg_qbytes` field, but the calling process does not have the appropriate system privilege.

Implementation Specifics

The **kmsgctl** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **msgctl** subroutine.

Message Queue Kernel Services and Understanding System Call Execution in *AIX Kernel Extensions and Device Support Programming Concepts*.

kmsgget Kernel Service

Purpose

Obtains a message queue identifier.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int kmsgget (key, msgflg, msqid)
key_t key;
int msgflg;
int *msqid;
```

Parameters

key — Specifies either a value of **IPC_PRIVATE** or an IPC key constructed by the **ftok** subroutine (or a similar algorithm).

msgflg — Specifies that the *msgflg* parameter is constructed by logically ORing one or more of these values:

- **IPC_CREAT** — Creates the data structure if it does not already exist.
- **IPC_EXCL** — Causes the **kmsgget** kernel service to fail if **IPC_CREAT** is also set and the data structure already exists.
- **S_IRUSR** — Permits the process that owns the data structure to read it.
- **S_IWUSR** — Permits the process that owns the data structure to modify it.
- **S_IRGRP** — Permits the process group associated with the data structure to read it.
- **S_IWGRP** — Permits the process group associated with the data structure to modify it.
- **S_IROTH** — Permits others to read the data structure.
- **S_IWOTH** — Permits others to modify the data structure.

The values that begin with **S_I**... are defined in the **/usr/include/sys/stat.h** file. They are a subset of the access permissions that apply to files.

msqid — A reference parameter where a valid message–queue ID is returned if the **kmsgget** kernel service is successful.

Description

The **kmsgget** kernel service returns the message–queue identifier specified by the *msqid* parameter associated with the specified *key* parameter value. The **kmsgget** kernel service provides the same functions for user–mode processes in kernel mode as the **msgget** subroutine performs for kernel processes or user–mode processes in user mode. The **kmsgget** service can be called by a user–mode process in kernel mode or by a kernel process. A kernel process can also call the **msgget** subroutine to provide the same function.

Execution Environment

The **kmsgget** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion. The *msqid* parameter is set to a valid message–queue identifier.

If the **kmsgget** kernel service fails, the *msqid* parameter is not valid and the return code is one of these four values:

EACCES	Indicates that a message queue ID exists for the *key* parameter but operation permission as specified by the *msgflg* parameter cannot be granted.
ENOENT	Indicates that a message queue ID does not exist for the *key* parameter and the **IPC_CREAT** command is not set.
ENOSPC	Indicates that a message queue ID is to be created but the system–imposed limit on the maximum number of allowed message queue IDs systemwide will be exceeded.
EEXIST	Indicates that a message queue ID exists for the value specified by the *key* parameter, and both the **IPC_CREAT** and **IPC_EXCL** commands are set.

Implementation Specifics

The **kmsgget** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **msgget** subroutine.

Message Queue Kernel Services and Understanding System Call Execution in *AIX Kernel Extensions and Device Support Programming Concepts*.

kmsgrcv Kernel Service

Purpose

Reads a message from a message queue.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int kmsgrcv
(msqid, msgp, msgsz,
msgtyp, msgflg, flags,
bytes)
int msqid;
struct msgxbuf *msgp;
   or struct msgbuf *msgp;
int msgsz;
mtyp_t msgtyp;
int msgflg;
int flags;
int *bytes;
```

Parameters

msqid Specifies the message queue from which to read.

msgp Points to either an **msgxbuf** or an **msgbuf** structure where the message text is placed. The type of structure pointed to is determined by the values of the *flags* parameter. These structures are defined in the /**usr**/**include**/**sys**/**msg.h** file.

msgsz Specifies the maximum number of bytes of text to be received from the message queue. The received message is truncated to the size specified by the *msgsz* parameter if the message is longer than this size and **MSG_NOERROR** is set in the *msgflg* parameter. The truncated part of the message is lost and no indication of the truncation is given to the calling process.

msgtyp Specifies the type of message requested as follows:

- If the *msgtyp* parameter is equal to 0, the first message on the queue is received.
- If the *msgtyp* parameter is greater than 0, the first message of the type specified by the *msgtyp* parameter is received.
- If the *msgtyp* parameter is less than 0, the first message of the lowest type that is less than or equal to the absolute value of the *msgtyp* parameter is received.

msgflg Specifies a value of 0, or is constructed by logically ORing one of several values:

MSG_NOERROR Truncates the message if it is longer than the number of bytes specified by the *msgsz* parameter.

IPC_NOWAIT Specifies the action to take if a message of the desired type is not on the queue:

- If **IPC_NOWAIT** is set, then the **kmsgrcv** service returns an **ENOMSG** value.
- If **IPC_NOWAIT** is not set, then the calling process suspends execution until one of the following occurs:
  - A message of the desired type is placed on the queue.
  - The message queue ID specified by the *msqid* parameter is removed from the system. When this occurs, the **kmsgrcv** service returns an **EIDRM** value.
  - The calling process receives a signal that is to be caught. In this case, a message is not received and the **kmsgrcv** service returns an **EINTR** value.

flags Specifies a value of 0 if a normal message receive is to be performed. If an extended message receive is to be performed, this flag should be set to an **XMSG** value. With this flag set, the **kmsgrcv** service functions as the **msgxrcv** subroutine would. Otherwise, the **kmsgrcv** service functions as the **msgrcv** subroutine would.

bytes Specifies a reference parameter. This parameter contains the number of message–text bytes read from the message queue upon return from the **kmsgrcv** service.

If the message is longer than the number of bytes specified by the *msgsz* parameter bytes but **MSG_NOERROR** is not set, then the **kmsgrcv** kernel service fails and returns an **E2BIG** return value.

Description

The **kmsgrcv** kernel service reads a message from the queue specified by the *msqid* parameter and stores the message into the structure pointed to by the *msgp* parameter. The **kmsgrcv** kernel service provides the same functions for user–mode processes in kernel mode as the **msgrcv** and **msgxrcv** subroutines perform for kernel processes or user–mode processes in user mode.

The **kmsgrcv** service can be called by a user–mode process in kernel mode or by a kernel process. A kernel process can also call the **msgrcv** and **msgxrcv** subroutines to provide the same functions.

Execution Environment

The **kmsgrcv** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
EINVAL	Indicates that the ID specified by the *msqid* parameter is not a valid message queue ID.
EACCES	Indicates that operation permission is denied to the calling process.
EINVAL	Indicates that the value of the *msgsz* parameter is less than 0.
E2BIG	Indicates that the message text is greater than the maximum length specified by the *msgsz* parameter and **MSG_NOERROR** is not set.

ENOMSG	Indicates that the queue does not contain a message of the desired type and **IPC_NOWAIT** is set.
EINTR	Indicates that the **kmsgrcv** service received a signal.
EIDRM	Indicates that the message queue ID specified by the *msqid* parameter has been removed from the system.

Implementation Specifics

The **kmsgrcv** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **msgrcv** subroutine, **msgxrcv** subroutine.

Message Queue Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

Understanding System Call Execution in *AIX Kernel Extensions and Device Support Programming Concepts*.

kmsgsnd Kernel Service

Purpose

Sends a message using a previously defined message queue.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int kmsgsnd (msqid, msgp, msgsz, msgflg)
int msqid;
struct msgbuf * msgp;
int msgsz, msgflg;
```

Parameters

msqid	Specifies the message queue ID that indicates which message queue the message is to be sent on.
msgp	Points to an **msgbuf** structure containing the message. The **msgbuf** structure is defined in the /**usr**/**include**/**sys**/**msg.h** file.
msgsz	Specifies the size of the message to be sent in bytes. The *msgsz* parameter can range from 0 to a system–imposed maximum.
msgflg	Specifies the action to be taken if the message cannot be sent for one of several reasons.

Description

The **kmsgsnd** kernel service sends a message to the queue specified by the *msqid* parameter. The **kmsgsnd** kernel service provides the same functions for user–mode processes in kernel mode as the **msgsnd** subroutine performs for kernel processes or user–mode processes in user mode. The **kmsgsnd** service can be called by a user–mode process in kernel mode or by a kernel process. A kernel process can also call the **msgsnd** subroutine to provide the same function.

There are two reasons why the **kmsgsnd** kernel service cannot send the message:

- The number of bytes already on the queue is equal to the **msg_qbytes** member.
- The total number of messages on all queues systemwide is equal to a system–imposed limit.

There are several actions to take when the **kmsgsnd** kernel service cannot send the message:

- If the *msgflg* parameter is set to **IPC_NOWAIT**, then the message is not sent, and the **kmsgsnd** service fails and returns an **EAGAIN** value.
- If the *msgflg* parameter is 0, then the calling process suspends execution until one of the following occurs:
  - The condition responsible for the suspension no longer exists, in which case the message is sent.
  - The message queue ID specified by the *msqid* parameter is removed from the system. When this occurs, the **kmsgsnd** service fails and an **EIDRM** value is returned.
  - The calling process receives a signal that is to be caught. In this case, the message is not sent and the calling process resumes execution as described in the **sigaction** kernel service.

Execution Environment

The **kmsgsnd** kernel service can be called from the process environment only.

The calling process must have write permission to perform the **kmsgsnd** operation.

Return Values

0	Indicates a successful operation.
EINVAL	Indicates that the *msqid* parameter is not a valid message queue ID.
EACCES	Indicates that operation permission is denied to the calling process.
EAGAIN	Indicates that the message cannot be sent for one of the reasons stated previously, and the *msgflg* parameter is set to **IPC_NOWAIT**.
EINVAL	Indicates that the *msgsz* parameter is less than 0 or greater than the system–imposed limit.
EINTR	Indicates that the **kmsgsnd** service received a signal.
EIDRM	Indicates that the message queue ID specified by the *msqid* parameter has been removed from the system.
ENOMEM	Indicates that the system does not have enough memory to send the message.

Implementation Specifics

The **kmsgsnd** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **msgsnd** subroutine.

Message Queue Kernel Services and Understanding System Call Execution in *AIX Kernel Extensions and Device Support Programming Concepts*.

ksettickd Kernel Service

Purpose

Sets the current status of the systemwide timer–adjustment values.

Syntax

```
#include <sys/types.h>

int ksettickd (timed, tickd,
time_adjusted)
int *timed;
int *tickd;
int *time_adjusted;
```

Parameters

timed	Specifies the number of microseconds by which the systemwide timer is to be adjusted unless set to a null pointer.
tickd	Specifies the adjustment rate of the systemwide timer unless set to a null pointer. This rate determines the number of microseconds that the systemwide timer is adjusted with each timer tick. Adjustment continues until the time has been corrected by the amount specified by the *timed* parameter.
time_adjusted	Sets the kernel–maintained time adjusted flag to True or False. If the *time_adjusted* parameter is a null pointer, calling the **ksettickd** kernel service always sets the kernel's *time_adjusted* parameter to False.

Description

The **ksettickd** kernel service provides kernel extensions with the capability to update the *time_adjusted* parameter, and set or change the systemwide time–of–day timer adjustment amount and rate. The timer–adjustment values indicated by the *timed* and *tickd* parameters are the same values used by the **adjtime** subroutine. A call to the **settimer** or **adjtime** subroutine for the systemwide time–of–day timer sets the *time_adjusted* parameter to True, as read by the **kgettickd** kernel service.

This kernel service is typically used only by kernel extensions providing time synchronization functions such as coordinated network time where the **adjtime** subroutine is insufficient.

Note: The **ksettickd** service provides no serialization with respect to the **adjtime** and **settimer** subroutines, the **ksettimer** kernel service, or the timer interrupt handler, all of which also use and update these values. The caller of this kernel service must provide the necessary serialization to ensure appropriate operation.

Execution Environment

The **ksettickd** kernel service can be called from either the process or interrupt environment.

Return Value

The **ksettickd** kernel service always returns a value of 0.

Implementation Specifics

The **ksettickd** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **kgettickd** kernel service, **ksettimer** kernel service.

The **adjtime** subroutine, **settimer** subroutine.

Timer and Time–of–Day Kernel Services and Using Fine Granularity Timer Services and Structures in *AIX Kernel Extensions and Device Support Programming Concepts*.

ksettimer Kernel Service

Purpose

Sets the systemwide time–of–day timer.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/time.h>

int ksettimer (nct)
struct timestruc_t *nct;
```

Parameter

nct	Points to a **timestruc_t** structure, which contains the new current time to be set. The nanoseconds member of this structure is valid only if greater than or equal to 0, and less than the number of nanoseconds in a second.

Description

The **ksettimer** kernel service provides a kernel extension with the capability to set the systemwide time–of–day timer. Kernel extensions typically use this kernel service to support network coordinated time, which is the periodic synchronization of all system clocks to a common time by a time server or set of time servers on a network. The newly set ”current” time must represent the amount of time since 00:00:00 GMT, January 1, 1970.

Execution Environment

The **ksettimer** kernel service can be called from the process environment only.

Return Values

0	Indicates success.
EINVAL	Indicates that the new current time specified by the *nct* parameter is outside the range of the systemwide timer.
EIO	Indicates that an error occurred while this kernel service was accessing the timer device.

Implementation Specifics

The **ksettimer** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Using Fine Granularity Timer Services and Structures andTimer and Time–of–Day Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

kthread_kill Kernel Service

Purpose

Posts a signal to a specified kernel–only thread.

Syntax

```
#include <sys/thread.h>

void kthread_kill (tid, sig)
tid_t tid;
int sig;
```

Parameters

tid	Specifies the target kernel–only thread. If its value is –1, the signal is posted to the calling thread.
sig	Specifies the signal number to post.

Description

The **kthread_kill** kernel service posts the signal *sig* to the kernel thread specified by the *tid* parameter. When the service is called from the process environment, the target thread must be in the same process as the calling thread. When the service is called from the interrupt environment, the signal is posted to the target thread, without a permission check.

Execution Environment

The **kthread_kill** kernel service can be called from either the process environment or the interrupt environment.

Return Values

The **kthread_kill** kernel service has no return values.

Implementation Specifics

The **kthread_kill** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **sig_chk** kernel service.

kthread_start Kernel Service

Purpose

Starts a previously created kernel–only thread.

Syntax

```
#include <sys/thread.h>

int kthread_start (tid, i_func, i_data_addr, i_data_len,
i_stackaddr, i_sigmask)
tid_t tid;
int (*i_func) (void *);
void *i_data_addr;
size_t i_data_len;
void *i_stackaddr;
sigset_t *i_sigmask;
```

Parameters

tid	Specifies the kernel–only thread to start.
i_func	Points to the entry–point routine of the kernel–only thread.
i_data_addr	Points to data that will be passed to the entry–point routine.
i_data_len	Specifies the length of the data chunk.
i_stackaddr	Specifies the stack's base address for the kernel–only thread.
i_sigmask	Specifies the set of signal to block from delivery when the new kernel–only thread begins execution.

Description

The **kthread_start** kernel service starts the kernel–only thread specified by the *tid* parameter. The thread must have been previously created with the **thread_create** kernel service, and its state must be **TSIDL**.

This kernel service initializes and schedules the thread for the processor. Its state is changed to **TSRUN**. The thread is initialized so that it begins executing at the entry point specified by the *i_func* parameter, and that the signals specified by the *i_sigmask* parameter are blocked from delivery.

The thread's entry point gets one parameter, a pointer to a chunk of data that is copied to the base of the thread's stack. The *i_data_addr* and *i_data_len* parameters specify the location and quantity of data to copy. The format of the data must be agreed upon by the initializing and initialized thread.

The thread's stack's base address is specified by the *i_stackaddr* parameter. If a value of zero is specified, the kernel will allocate the memory for the stack (96K). This memory will be reclaimed by the system when the thread terminates. If a non–zero value is specified, then the caller should allocate the backing memory for the stack. Since stacks grow from high addresses to lower addresses, the *i_stackaddr* parameter specifies the highest address for the thread's stack.

The thread will be automatically terminated when it returns from the entry point routine. If it is the last thread in the process, then the process will be exited.

Execution Environment

The **kthread_start** kernel service can be called from the process environment only.

Return Values

The **kthread_start** kernel service returns one of the following values:

0	Indicates a successful start.
ESRCH	Indicates that the *tid* parameter is not valid.

Implementation Specifics

The **kthread_start** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **thread_create** kernel service.

Process and Exception Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

limit_sigs or sigsetmask Kernel Service

Purpose

Changes the signal mask for the calling kernel thread.

Syntax

```
#include <sys/encap.h>

void limit_sigs (siglist, old_mask)
sigset_t *siglist;
sigset_t *old_mask;

void sigsetmask (old_mask)
sigset_t *old_mask;
```

Parameters

siglist	Specifies the signal set to deliver.
old_mask	Points to the old signal set.

Description

The **limit_sigs** kernel service changes the signal mask for the calling kernel thread such that only the signals specified by the *siglist* parameter will be delivered, unless they are currently being blocked or ignored.

The old signal mask is returned via the *old_mask* parameter. If the *siglist* parameter is **NULL**, the signal mask is not changed; it can be used for getting the current signal mask.

The **sigsetmask** kernel service should be used to restore the set of blocked signals for the calling thread. The typical usage of these services is the following:

```
sigset_t allowed = limited set of signals
sigset_t old;

/* limits the set of delivered signals */
limit_sigs (&allowed, &old);

   /* do something with a limited set of delivered signals */

/* restore the original set */
sigsetmask (&old);
```

Execution Environment

The **limit_sigs** and **sigsetmask** kernel services can be called from the process environment only.

Return Values

The **limit_sigs** and **sigsetmask** kernel services have no return values.

Implementation Specifics

The **limit_sigs** and **sigsetmask** kernel services are part of the Base Operating System (BOS) Runtime.

Related Information

The **kthread_kill** kernel service.

lock_alloc Kernel Service

Purpose

Allocates system memory for a simple or complex lock.

Syntax

```
#include <sys/lock_def.h>
#include <sys/lock_alloc.h>

void lock_alloc (lock_addr, flags, class, occurrence)
void *lock_addr;
int flags;
short class;
short occurrence;
```

Parameters

lock_addr	Specifies a valid simple or complex lock address.
flags	Specifies whether the memory allocated is to be pinned or pageable. Set this parameter as follows: **LOCK_ALLOC_PIN** Allocate pinned memory; use if it is not permissible to take a page fault while calling a locking kernel service for this lock. **LOCK_ALLOC_PAGED** Allocate pageable memory; use if it is permissible to take a page fault while calling a locking kernel service for this lock.
class	Specifies the family which the lock belongs to.
occurrence	Identifies the instance of the lock within the family. If only one instance of the lock is defined, this parameter should be set to –1.

Description

The **lock_alloc** kernel service allocates system memory for a simple or complex lock. The **lock_alloc** kernel service must be called for each simple or complex before the lock is initialized and used. The memory allocated is for internal lock instrumentation use, and is not returned to the caller; no memory is allocated if instrumentation is not used.

Execution Environment

The **lock_alloc** kernel service can be called from the process environment only.

Return Values

The **lock_alloc** kernel service has no return values.

Implementation Specifics

The **lock_alloc** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **lock_free** kernel service, **lock_init** kernel service, **simple_lock_init** kernel service.

Understanding Locking and Locking Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*

lock_clear_recursive Kernel Service

Purpose

Prevents a complex lock from being acquired recursively.

Syntax

```
#include <sys/lock_def.h>

void lock_clear_recursive (lock_addr)
complex_lock_t lock_addr;
```

Parameter

lock_addr	Specifies the address of the lock word which is no longer to be acquired recursively.

Description

The **lock_clear_recursive** kernel service prevents the specified complex lock from being acquired recursively. The lock must have been made recursive with the **lock_set_recursive** kernel service. The calling thread must hold the specified complex lock in write–exclusive mode.

Execution Environment

The **lock_clear_recursive** kernel service can be called from the process environment only.

Return Values

The **lock_clear_recursive** kernel service has no return values.

Implementation Specifics

The **lock_clear_recursive** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **lock_init** kernel service, **lock_done** kernel service, **lock_read** kernel service, **lock_read_to_write** kernel service, **lock_write** kernel service, **lock_set_recursive** kernel service.

Understanding Locking and Locking Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*

lock_done Kernel Service

Purpose

Unlocks a complex lock.

Syntax

```
#include <sys/lock_def.h>

void lock_done (lock_addr)
complex_lock_t lock_addr;
```

Parameter

lock_addr	Specifies the address of the lock word to unlock.

Description

The **lock_done** kernel services unlocks a complex lock. The calling kernel thread must hold the lock either in shared–read mode or exclusive–write mode. If one or more kernel threads are waiting to acquire the lock in exclusive–write mode, one of these kernel threads (the one with the highest priority) is made runnable and may compete for the lock. Otherwise, any kernel threads which are waiting to acquire the lock in shared–read mode are made runnable. If there was at least one kernel thread waiting for the lock, the priority of the calling kernel thread is recomputed.

If the lock is held recursively, it is not actually released until the **lock_done** kernel service has been called once for each time that the lock was locked.

Execution Environment

The **lock_done** kernel service can be called from the process environment only.

Return Values

The **lock_done** kernel service has no return values.

Implementation Specifics

The **lock_done** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **lock_alloc** kernel service, **lock_free** kernel service, **lock_init** kernel service.

Understanding Locking and Locking Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*

lock_free Kernel Service

Purpose

Frees the memory of a simple or complex lock.

Syntax

```
#include <sys/lock_def.h>
#include <sys/lock_alloc.h>

void lock_free (lock_addr)
void *lock_addr;
```

Parameter

lock_addr	Specifies the address of the lock word whose memory is to be freed.

Description

The **lock_free** kernel service frees the memory of a simple or complex lock. The memory freed is the internal operating system memory which was allocated with the **lock_alloc** kernel service.

Note: It is only necessary to call the **lock_free** kernel service when the memory that the corresponding lock was protecting is released. For example, if you allocate memory for an i–node which is to be protected by a lock, you must allocate and initialize the lock before using it. The memory may be used with several i–nodes, each taken from, and returned to, the free i–node pool; the **lock_init** kernel service must be called each time this is done.The **lock_free** kernel service must be called when the memory allocated for the inode is finally freed.

Execution Environment

The **lock_free** kernel service can be called from the process environment only.

Return Values

The **lock_free** kernel service has no return values.

Implementation Specifics

The **lock_free** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **lock_alloc** kernel service.

Understanding Locking and Locking Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*

lock_init Kernel Service

Purpose

Initializes a complex lock.

Syntax

```
#include <sys/lock_def.h>

void lock_init (lock_addr, can_sleep)
complex_lock_t lock_addr;
boolean_t can_sleep;
```

Parameters

lock_addr	Specifies the address of the lock word.
can_sleep	This parameter is ignored.

Description

The **lock_init** kernel service initializes the specified complex lock. This kernel service must be called for each complex lock before the lock is used. The complex lock must previously have been allocated with the **lock_alloc** kernel service.

Execution Environment

The **lock_init** kernel service can be called from the process environment only.

Return Values

The **lock_init** kernel service has no return values.

Implementation Specifics

The **lock_init** kernel service is part of the Base Operating System (BOS) Runtime.

The *can_sleep* parameter is included for compatibility with OSF/1 1.1, but is ignored. Using a value of **TRUE** for this parameter will maintain OSF/1 1.1 semantics.

Related Information

The **lock_alloc** kernel service, **lock_free** kernel service.

Understanding Locking and Locking Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*

lock_islocked Kernel Service

Purpose

Tests whether a complex lock is locked.

Syntax

```
#include <sys/lock_def.h>

int lock_islocked (lock_addr)
complex_lock_t lock_addr;
```

Parameter

lock_addr	Specifies the address of the lock word to test.

Description

The **lock_islocked** kernel service determines whether the specified complex lock is free, or is locked in either shared–read or exclusive–write mode.

Execution Environment

The **lock_islocked** kernel service can be called from the process environment only.

Return Values

TRUE	Indicates that the lock was locked.
FALSE	Indicates that the lock was free.

Implementation Specifics

The **lock_islocked** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **lock_init** kernel service.

Understanding Locking and Locking Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*

lockl Kernel Service

Purpose

Locks a conventional process lock.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/lockl.h>

int lockl (lock_word, flags)
lock_t *lock_word;
int flags;
```

Parameters

lock _word Specifies the address of the lock word.

flags Specifies the flags that control waiting for a lock. The *flags* parameter is used to control how signals affect waiting for a lock. The four flags are:

LOCK_NDELAY Controls whether the caller waits for the lock. Setting the flag causes the request to be terminated. The lock is assigned to the caller. Not setting the flag causes the caller to wait until the lock is not owned by another process before the lock is assigned to the caller.

LOCK_SHORT Prevents signals from terminating the wait for the lock. **LOCK_SHORT** is the default flag for the **lockl** Kernel Service. This flag causes non–preemptive sleep.

LOCK_SIGRET Causes the wait for the lock to be terminated by an unmasked signal.

LOCK_SIGWAKE Causes the wait for the lock to be terminated by an unmasked signal and control transferred to the return from the last operation by the **setjmpx** kernel service.

Note: The **LOCK_SIGRET** flag overrides the **LOCK_SIGWAKE** flag.

Description

Note: The **lockl** kernel service is provided for compatibility only and should not be used in new code, which should instead use simple locks or complex locks.

The **lockl** kernel service locks a conventional lock

The lock word can be located in shared memory. It must be in the process's address space when the **lockl** or **unlockl** services are called. The kernel accesses the lock word only while executing under the caller's process.

The *lock_word* parameter is typically part of the data structure that describes the resource managed by the lock. This parameter must be initialized to the **LOCK_AVAIL** value before the first call to the **lockl** service. Only the **lockl** and **unlockl** services can alter this parameter while the lock is in use.

The **lockl** service is nestable. The caller should use the **LOCK_SUCC** value for determining when to call the **unlockl** service to unlock the conventional lock.

The **lockl** service temporarily assigns the owner the process priority of the most favored waiter for the lock.

A process must release all locks before terminating or leaving kernel mode. Signals are not delivered to kernel processes while those processes own any lock. ”Understanding System Call Execution” in *AIX Kernel Extensions and Device Support Programming Concepts* discusses how system calls can use the **lockl** service when accessing global data.

Execution Environment

The **lockl** kernel service can be called from the process environment only.

Return Values

LOCK_SUCC	Indicates that the process does not already own the lock or the lock is not owned by another process when the *flags* parameter is set to **LOCK_NDELAY**.
LOCK_NEST	Indicates that the process already owns the lock or the lock is not owned by another process when the *flags* parameter is set to **LOCK_NDELAY**.
LOCK_FAIL	Indicates that the lock is owned by another process when the *flags* parameter is set to **LOCK_NDELAY**.
LOCK_SIG	Indicates that the wait is terminated by a signal when the *flags* parameter is set to **LOCK_SIGRET**.

Implementation Specifics

The **lockl** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **unlockl** kernel service.

Understanding Locking in *AIX Kernel Extensions and Device Support Programming Concepts*.

Locking Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*

lock_mine Kernel Service

Purpose

Checks whether a simple or complex lock is owned by the caller.

Syntax

```
#include <sys/lock_def.h>

boolean_t lock_mine (lock_addr)
void *lock_addr;
```

Parameter

lock_addr	Specifies the address of the lock word to check.

Description

The **lock_mine** kernel service checks whether the specified simple or complex lock is owned by the calling kernel thread. Because a complex lock held in shared–read mode has no owner, the service returns FALSE in this case. This kernel service is provided to assist with debugging.

Execution Environment

The **lock_mine** kernel service can be called from the process environment only.

Return Values

TRUE	Indicates that the calling kernel thread owns the lock.
FALSE	Indicates that the calling kernel thread does not own the lock, or that a complex lock is held in shared–read mode.

Implementation Specifics

The **lock_mine** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **lock_init** kernel service, **lock_islocked** kernel service, **lock_read** kernel service, **lock_write** kernel service, **simple_lock** kernel service.

Understanding Locking and Locking Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*

lock_read or lock_try_read Kernel Service

Purpose

Locks a complex lock in shared–read mode.

Syntax

```
#include <sys/lock_def.h>

void lock_read (lock_addr)
complex_lock_t lock_addr;

boolean_t lock_try_read (lock_addr)
complex_lock_t lock_addr;
```

Parameter

lock_addr	Specifies the address of the lock word to lock.

Description

The **lock_read** kernel service locks the specified complex lock in shared–read mode; it blocks if the lock is locked in exclusive–write mode. The lock must previously have been initialized with the **lock_init** kernel service. The **lock_read** kernel service has no return values.

The **lock_try_read** kernel service tries to lock the specified complex lock in shared–read mode; it returns immediately if the lock is locked in exclusive–write mode, otherwise it locks the lock in shared–read mode. The lock must previously have been initialized with the **lock_init** kernel service.

Execution Environment

The **lock_read** and **lock_try_read** kernel services can be called from the process environment only.

Return Values

The **lock_try_read** kernel service has the following return values:

TRUE	Indicates that the lock was successfully acquired in shared–read mode.
FALSE	Indicates that the lock was not acquired.

Implementation Specifics

The **lock_read** and **lock_try_read** kernel services are part of the Base Operating System (BOS) Runtime.

Related Information

The **lock_init** kernel service, **lock_islocked** kernel service, **lock_done** kernel service.

Understanding Locking and Locking Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*

lock_read_to_write or lock_try_read_to_write Kernel Service

Purpose

Upgrades a complex lock from shared–read mode to exclusive–write mode.

Syntax

```
#include <sys/lock_def.h>

boolean_t lock_read_to_write (lock_addr)
complex_lock_t lock_addr;

boolean_t lock_try_read_to_write (lock_addr)
complex_lock_t lock_addr;
```

Parameter

lock_addr	Specifies the address of the lock word to be converted from read–shared to write–exclusive mode.

Description

The **lock_read_to_write** and **lock_try_read_to_write** kernel services try to upgrade the specified complex lock from shared–read mode to exclusive–write mode. The lock is successfully upgraded if no other thread has already requested write–exclusive access for this lock. If the lock cannot be upgraded, it is no longer held on return from the **lock_read_to_write** kernel service; it is still held in shared–read mode on return from the **lock_try_read_to_write** kernel service.

The calling kernel thread must hold the lock in shared–read mode.

Execution Environment

The **lock_read_to_write** and **lock_try_read_to_write** kernel services can be called from the process environment only.

Return Values

The following only apply to **lock_read_to_write**:

TRUE	Indicates that the lock was not upgraded and is no longer held.
FALSE	Indicates that the lock was successfully upgraded to exclusive–write mode.

The following only apply to **lock_try_read_to_write**:

TRUE	Indicates that the lock was successfully upgraded to exclusive–write mode.
FALSE	Indicates that the lock was not upgraded and is held in read mode.

Implementation Specifics

The **lock_read_to_write** and **lock_try_read_to_write** kernel services are part of the Base Operating System (BOS) Runtime.

Related Information

The **lock_init** kernel service, **lock_islocked** kernel service, **lock_done** kernel service.

Understanding Locking and Locking Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*

lock_set_recursive Kernel Service

Purpose

Prepares a complex lock for recursive use.

Syntax

```
#include <sys/lock_def.h>

void lock_set_recursive (lock_addr)
complex_lock_t lock_addr;
```

Parameter

lock_addr	Specifies the address of the lock word to be prepared for recursive use.

Description

The **lock_set_recursive** kernel service prepares the specified complex lock for recursive use. A complex lock cannot be nested until the **lock_set_recursive** kernel service is called for it. The calling kernel thread must hold the specified complex lock in write–exclusive mode.

When a complex lock is used recursively, the **lock_done** kernel service must be called once for each time that the thread is locked in order to unlock the lock.

Only the kernel thread which calls the **lock_set_recursive** kernel service for a lock may acquire that lock recursively.

Execution Environment

The **lock_set_recursive** kernel service can be called from process environment only.

Return Values

The **lock_set_recursive** kernel service has no return values.

Implementation Specifics

The **lock_set_recursive** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **lock_init** kernel service, **lock_done** kernel service, **lock_write** kernel service, **lock_clear_recursive** kernel service.

Understanding Locking and Locking Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*

lock_write or lock_try_write Kernel Service

Purpose

Locks a complex lock in exclusive–write mode.

Syntax

```
#include <sys/lock_def.h>

void lock_write (lock_addr)
complex_lock_t lock_addr;

boolean_t lock_try_write (lock_addr)
complex_lock_t lock_addr;
```

Parameter

lock_addr	Specifies the address of the lock word to lock.

Description

The **lock_write** kernel service locks the specified complex lock in exclusive–write mode; it blocks if the lock is busy. The lock must have been previously initialized with the **lock_init** kernel service. The **lock_write** kernel service has no return values.

The **lock_try_write** kernel service tries to lock the specified complex lock in exclusive–write mode; it returns immediately without blocking if the lock is busy. The lock must have been previously initialized with the **lock_init** kernel service.

Execution Environment

The **lock_write** and **lock_try_write** kernel services can be called from the process environment only.

Return Values

The **lock_try_write** kernel service has the following parameters:

TRUE	Indicates that the lock was successfully acquired.
FALSE	Indicates that the lock was not acquired.

Implementation Specifics

The **lock_write** and **lock_try_write** kernel services are part of the Base Operating System (BOS) Runtime.

Related Information

The **lock_init** kernel service, **lock_islocked** kernel service, **lock_done** kernel service, **lock_read_to_write** kernel service, **lock_try_read_to_write** kernel service, **lock_write_to_read** kernel service.

Understanding Locking and Locking Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*

lock_write_to_read Kernel Service

Purpose

Downgrades a complex lock from exclusive–write mode to shared–read mode.

Syntax

```
#include <sys/lock_def.h>

void lock_write_to_read (lock_addr)
complex_lock_t lock_addr;
```

Parameter

lock_addr	Specifies the address of the lock word to be downgraded from exclusive–write to shared–read mode.

Description

The **lock_write_to_read** kernel service downgrades the specified complex lock from exclusive–write mode to shared–read mode. The calling kernel thread must hold the lock in exclusive–write mode.

Once the lock has been downgraded to shared–read mode, other kernel threads will also be able to acquire it in shared–read mode.

Execution Environment

The **lock_write_to_read** kernel service can be called from the process environment only.

Return Values

The **lock_write_to_read** kernel service has no return values.

Implementation Specifics

The **lock_write_to_read** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **lock_init** kernel service, **lock_islocked** kernel service, **lock_done** kernel service, **lock_read_to_write** kernel service, **lock_try_read_to_write** kernel service, **lock_try_write** kernel service, **lock_write** kernel service.

Understanding Locking and Locking Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*

loifp Kernel Service

Purpose

Returns the address of the software loopback interface structure.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

struct ifnet *loifp ()
```

Description

The **loifp** kernel service returns the address of the **ifnet** structure associated with the software loopback interface. The interface address can be used to examine the interface flags. This address can also be used to determine whether the **looutput** kernel service can be called to send a packet through the loopback interface.

Execution Environment

The **loifp** kernel service can be called from either the process or interrupt environment.

Return Values

The **loifp** service returns the address of the **ifnet** structure describing the software loopback interface.

Implementation Specifics

The **loifp** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **looutput** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

longjmpx Kernel Service

Purpose

Allows exception handling by causing execution to resume at the most recently saved context.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int longjmpx (ret_val)
int ret_val;
```

Parameters

ret_val	Specifies the return value to be supplied on the return from the **setjmpx** kernel service for the resumed context. This value normally indicates the type of exception that has occurred.

Description

The **longjmpx** kernel service causes the normal execution flow to be modified so that execution resumes at the most recently saved context. The kernel mode lock is reacquired if it is necessary. The interrupt priority level is reset to that of the saved context.

The **longjmpx** service internally calls the **clrjmpx** service to remove the jump buffer specified by the *jump_buffer* parameter from the list of contexts to be resumed. The **longjmpx** service always returns a nonzero value when returning to the restored context. Therefore, if the value of the *ret_val* parameter is 0, the **longjmpx** service returns an **EINTR** value to the restored context.

If there is no saved context to resume, the system crashes.

Execution Environment

The **longjmpx** kernel service can be called from either the process or interrupt environment.

Return Values

A successful call to the **longjmpx** service does not return to the caller. Instead, it causes execution to resume at the return from a previous **setjmpx** call with the return value of the *ret_val* parameter.

Implementation Specifics

The **longjmpx** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **clrjmpx** kernel service, **setjmpx** kernel service.

Understanding Exception Handling in *AIX Kernel Extensions and Device Support Programming Concepts*.

Process and Exception Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

lookupvp Kernel Service

Purpose

Retrieves the v–node that corresponds to the named path.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int lookupvp (namep, flags, vpp, crp)
char *namep;
int flags;
struct vnode **vpp;
struct ucred *crp;
```

Parameters

crp Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

namep Points to a character string path name.

flags Specifies lookup directives, including these six flags:

- **L_LOC** The path–name resolution must not cross a mount point into another file system implementation.
- **L_NOFOLLOW** If the final component of the path name resolves to a symbolic link, the link is not to be traversed.
- **L_NOXMOUNT** If the final component of the path name resolves to a mounted–over object, the mounted–over object, rather than the root of the next virtual file system, is to be returned.
- **L_CRT** The object is to be created.
- **L_DEL** The object is to be deleted.
- **L_EROFS** An error is to be returned if the object resides in a read–only file system.

vpp Points to the location where the v–node pointer is to be returned to the calling routine.

Description

The **lookupvp** kernel service provides translation of the path name provided by the *namep* parameter into a virtual file system node. The **lookupvp** service provides a flexible interface to path–name resolution by regarding the *flags* parameter values as directives to the lookup process. The lookup process is a cooperative effort between the logical file system and underlying virtual file systems (VFS). Several v–node and VFS operations are employed to:

- Look up individual name components
- Read symbolic links
- Cross mount points

The **lookupvp** kernel service determines the process's current and root directories by consulting the `u_cdir` and `u_rdir` fields in the **u** structure. Information about the virtual file system and file system installation for transient v–nodes is obtained from each name component's **vfs** or **gfs** structure.

Execution Environment

The **lookupvp** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
errno	Indicates an error. This number is defined in the **/usr/include/sys/errno.h** file.

Implementation Specifics

The **lookupvp** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Understanding Data Structures and Header Files for Virtual File Systems in *AIX Kernel Extensions and Device Support Programming Concepts*.

Virtual File System Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

Virtual File System (VFS) Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

looutput Kernel Service

Purpose

Sends data through a software loopback interface.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int looutput (ifp, m0, dst)
struct  ifnet *ifp;
struct  mbuf *m0;
struct  sockaddr *dst;
```

Parameters

ifp	Specifies the address of an **ifnet** structure describing the software loopback interface.
m0	Specifies an **mbuf** chain containing output data.
dst	Specifies the address of a **sockaddr** structure that specifies the destination for the data.

Description

The **looutput** kernel service sends data through a software loopback interface. The data in the *m0* parameter is passed to the input handler of the protocol specified by the *dst* parameter.

Execution Environment

The **looutput** kernel service can be called from either the process or interrupt environment.

Return Values

0	Indicates that the data was successfully sent.
ENOBUFS	Indicates that resource allocation failed.
EAFNOSUPPORT	Indicates that the address family specified by the *dst* parameter is not supported.

Implementation Specifics

The **looutput** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **loifp** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

ltpin Kernel Service

Purpose

Pins the address range in the system (kernel) space and frees the page space for the associated pages.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/pin.h>

int ltpin (addr, length)
caddr_t  addr;
int      length;
```

Parameters

addr	Specifies the address of the first byte to pin.
length	Specifies the number of bytes to pin.

Description

The **ltpin** (long term pin) kernel service pins the real memory pages touched by the address range specified by the *addr* and *length* parameters in the system (kernel) address space. It pins the real–memory pages to ensure that page faults do not occur for memory references in this address range. The **ltpin** kernel service increments the long–term pin count for each real–memory page. While either the long–term or short–term pin count is nonzero, the page cannot be paged out of real memory.

The **ltpin** kernel service pins either the entire address range or none of it. Only a limited number of pages are pinned in the system. If there are not enough unpinned pages in the system, the **ltpin** kernel service returns an error code.

Note: The operating system pins only whole pages at a time. Therfore, if the requested range is not aligned on a page boundary, then memory outside this range is also pinned.

The **ltpin** kernel service can only be called for addresses within the system (kernel) address space.

Return Values

0	Indicates successful completion.
EINVAL	Indicates that the *length* parameter has a negative value. Otherwise, the area of memory beginning at the address of the first byte to pin (the **addr** parameter) and extending for the number of bytes specified by the *length* parameter is not defined.
EIO	Indicates that a permanent I/O error occurred while referencing data.
ENOMEM	Indicates that the **pin** kernel service was unable to pin due to insufficient real memory or exceeding the system–wide pin count.
ENOSPC	Indicates insufficient file system or paging space.

Implementation Specifics

The **ltpin** kernel service is not a published interface.

Related Information

The **ltunpin** kernel service.

ltunpin Kernel Service

Purpose

Unpins the address range in system (kernel) address space and reallocates paging space for the specified region.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/pin.h>

int   ltunpin (addr, length)
caddr_t  addr;
int      length;
```

Parameters

addr	Specifies the address of the first byte to unpin.
length	Specifies the number of bytes to unpin.

Description

The **ltunpin** kernel service decreases the long–term pin count of each page in the address range. When the long–term pin count becomes 0, the backing storage (paging space) for the memory region is allocated and assigned to the pages. When both the long–term and short–term pin counts are 0, the page is no longer pinned and the **ltunpin** kernel service will assert. If allocating backing pages would put the system below the low paging space threshold, the call waits until paging space becomes available.

The **ltunpin** kernel service can only be called with addresses in the system (kernel) address space from the process environment.

Return Values

0	Indicates successful completion.
EINVAL	Indicates that the *length* parameter is a negative value.
EIO	Indicates that a permanent I/O error occurred while referencing data.

Related Information

The **ltpin** kernel service.

m_adj Kernel Service

Purpose

Adjusts the size of an **mbuf** chain.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/mbuf.h>

void m_adj (m, diff)
struct mbuf *m;
int diff;
```

Parameters

m	Specifies the **mbuf** chain to be adjusted.
diff	Specifies the number of bytes to be removed.

Description

The **m_adj** kernel service adjusts the size of an **mbuf** chain by the number of bytes specified by the *diff* parameter. If the number specified by the *diff* parameter is nonnegative, the bytes are removed from the front of the chain. If this number is negative, the alteration is done from back to front.

Execution Environment

The **m_adj** kernel service can be called from either the process or interrupt environment.

Return Values

The **m_adj** service has no return values.

Implementation Specifics

The **m_adj** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

mbreq Structure for mbuf Kernel Services

Purpose

Contains **mbuf** structure registration information for the **m_reg** and **m_dereg** kernel services.

Syntax

```
#include <sys/mbuf.h>

struct mbreq {
                    int low_mbuf;
                    int low_clust;
                    int initial_mbuf;
                    int initial_clust;
}
```

Parameters

low_mbuf	Specifies the **mbuf** structure low–water mark.
low_clust	Specifies the page–sized **mbuf** structure low–water mark.
initial_mbuf	Specifies the initial allocation of **mbuf** structures.
initial_clust	Specifies the initial allocation of page–sized **mbuf** structures.

Description

The **mbreq** structure specifies the **mbuf** structure usage expectations for a user of **mbuf** kernel services.

Implementation Specifics

The **mbreq** structure is part of Base Operating System (BOS) Runtime.

Related Information

The **m_dereg** kernel service, **m_reg** kernel service.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

mbstat Structure for mbuf Kernel Services

Purpose

Contains **mbuf** usage statistics.

Syntax

```
#include <sys/mbuf.h>

struct mbstat {
ulong m_mbufs;
ulong m_clusters;
ulong m_spare;
ulong m_clfree;
ulong m_drops;
ulong m_wait;
ulong m_drain;
short m_mtypes[256];
}
```

Parameters

m_mbufs	Specifies the number of **mbuf** structures allocated.
m_clusters	Specifies the number of clusters allocated.
m_spare	Specifies the spare field.
m_clfree	Specifies the number of free clusters.
m_drops	Specifies the times failed to find space.
m_wait	Specifies the times waited for space.
m_drain	Specifies the times drained protocols for space.
m_mtypes	Specifies the type–specific **mbuf** structure allocations.

Description

The **mbstat** structure provides usage information for the **mbuf** services. Statistics can be viewed through the **netstat –m** command.

Implementation Specifics

The **mbstat** structure is part of Base Operating System (BOS) Runtime.

Related Information

The **netstat** command.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

m_cat Kernel Service

Purpose

Appends one **mbuf** chain to the end of another.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/mbuf.h>

void m_cat (m, n)
struct mbuf *m;
struct mbuf *n;
```

Parameters

m	Specifies the **mbuf** chain to be appended to.
n	Specifies the **mbuf** chain to append.

Description

The **m_cat** kernel service appends an **mbuf** chain specified by the *n* parameter to the end of **mbuf** chain specified by the *m* parameter. Where possible, compaction is performed.

Execution Environment

The **m_cat** kernel service can be called from either the process or interrupt environment.

Return Values

The **m_cat** service has no return values.

Implementation Specifics

The **m_cat** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

m_clattach Kernel Service

Purpose

Allocates an **mbuf** structure and attaches an external cluster.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/mbuf.h>

struct mbuf *
m_clattach(ext_buf, ext_free, ext_size, ext_arg, wait)
caddr_t ext_buf;
int (*ext_free)();
int ext_size;
int ext_arg;
int wait;
```

Parameters

ext_buf	Specifies the address of the external data area.
ext_free	Specifies the address of a function to be called when this **mbuf** structure is freed.
ext_size	Specifies the length of the external data area.
ext_arg	Specifies an argument to pass to the above function.
wait	Specifies either the **M_WAIT** or **M_DONTWAIT** value.

Description

The **m_clattach** kernel service allocates an **mbuf** structure and attaches the cluster specified by the *ext_buf* parameter. This data is owned by the caller. The `m_data` field of the returned **mbuf** structure points to the caller's data. Interrupt handlers can call this service only with the *wait* parameter set to **M_DONTWAIT**.

Note: The **m_clattach** kernel service replaces the **m_clgetx** kernel service, which is no longer supported.

The calling function is required to fill out the mbuf structure sufficiently to support normal usage. This includes support for the DMA functions during network transmission. To support DMA functions, the **ext_hasxm** flag field needs to be set to true and the **ext_xmemd** structure needs to be filled out. For buffers allocated from the kernel pinned heap, the **ext_xmemd.aspace_id** field should be set to XMEM_GLOBAL.

Execution Environment

The **m_clattach** kernel service can be called from either the process or interrupt environment.

Return Values

The **m_clattach** kernel service returns the address of an allocated **mbuf** structure. If the *wait* parameter is set to **M_DONTWAIT** and there are no free **mbuf** structures, the **m_clattach** service returns null.

Implementation Specifics

The **m_clattach** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

m_clget Macro for mbuf Kernel Services

Purpose

Allocates a page–sized **mbuf** structure cluster.

Syntax

```
#include <sys/mbuf.h>

int m_clget (m)
struct mbuf *m;
```

Parameter

m	Specifies the **mbuf** structure with which the cluster is to be associated.

Description

The **m_clget** macro allocates a page–sized **mbuf** cluster and attaches it to the given **mbuf** structure. If successful, the length of the **mbuf** structure is set to **CLBYTES**.

Execution Environment

The **m_clget** macro can be called from either the process or interrupt environment.

Return Values

1	Indicates successful completion.
0	Indicates an error.

Implementation Specifics

The **m_clget** macro is part of Base Operating System (BOS) Runtime.

Related Information

The **m_clgetm** kernel service.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

m_clgetm Kernel Service

Purpose

Allocates and attaches an external buffer.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/mbuf.h>

int
m_clgetm(m, how, size)
struct mbuf *m;
int  how;
int  size;
```

Parameters

m	Specifies the **mbuf** structure that the cluster will be associated with.
how	Specifies either the **M_DONTWAIT** or **M_WAIT** value.
size	Specifies the size of external cluster to attach. Valid sizes are listed in the /**usr**/**include**/**sys**/**mbuf.h** file

Description

The **m_clgetm** service allocates an **mbuf** cluster of the specified number of bytes and attaches it to the **mbuf** structure indicated by the *m* parameter. If successful, the **m_clgetm** service sets the **M_EXT** flag.

Execution Environment

The **m_clgetm** kernel service can be called from either the process or interrupt environment.

An interrupt handler can specify the *wait* parameter as **M_DONTWAIT** only.

Return Values

1	Indicates a successful operation.

If there are no free **mbuf** structures, the **m_clgetm** kernel service returns a null value.

Implementation Specifics

The **m_clgetm** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **m_free** kernel service, **m_freem** kernel service, **m_get** kernel service.

The **m_clget** macro.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

m_collapse Kernel Service

Purpose

Guarantees that an **mbuf** chain contains no more than a given number of **mbuf** structures.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/mbuf.h>

struct mbuf *m_collapse (m, size)
struct mbuf *m;
int size;
```

Parameters

m	Specifies the **mbuf** chain to be collapsed.
size	Denotes the maximum number of **mbuf** structures allowed in the chain.

Description

The **m_collapse** kernel service reduces the number of **mbuf** structures in an **mbuf** chain to the number of **mbuf** structures specified by the *size* parameter. The **m_collapse** service accomplishes this by copying data into page–sized **mbuf** structures until the chain is of the desired length. (If required, more than one page–sized **mbuf** structure is used.)

Execution Environment

The **m_collapse** kernel service can be called from either the process or interrupt environment.

Return Values

If the chain cannot be collapsed into the number of **mbuf** structures specified by the *size* parameter, a value of null is returned and the original chain is deallocated. Upon successful completion, the head of the altered **mbuf** chain is returned.

Implementation Specifics

The **m_collapse** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

m_copy Macro for mbuf Kernel Services

Purpose

Creates a copy of all or part of a list of **mbuf** structures.

Syntax

```
#include <sys/mbuf.h>

struct mbuf *m_copy (m, off, len)
struct mbuf *m;
int off;
int len;
```

Parameters

m	Specifies the **mbuf** structure, or the head of a list of **mbuf** structures, to be copied.
off	Specifies an offset into data from which copying starts.
len	Denotes the total number of bytes to copy.

Description

The **m_copy** macro makes a copy of the structure specified by the *m* parameter. The copy begins at the specified bytes (represented by the *off* parameter) and continues for the number of bytes specified by the *len* parameter. If the *len* parameter is set to **M_COPYALL**, the entire **mbuf** chain is copied.

Execution Environment

The **m_copy** macro can be called from either the process or interrupt environment.

Return Values

Upon successful completion, the address of the copied list (the **mbuf** structure that heads the list) is returned. If the copy fails, a value of null is returned.

Implementation Specifics

The **m_copy** macro is part of Base Operating System (BOS) Runtime.

Related Information

The **m_copydata** kernel service, **m_copym** kernel service.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

m_copydata Kernel Service

Purpose

Copies data from an **mbuf** chain to a specified buffer.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/mbuf.h>

void m_copydata (m, off, len, cp)
struct mbuf  *m;
int  off;
int  len;
caddr_t  cp;
```

Parameters

m	Indicates the **mbuf** structure, or the head of a list of **mbuf** structures, to be copied.
off	Specifies an offset into data from which copying starts.
len	Denotes the total number of bytes to copy.
cp	Points to a data buffer into which to copy the **mbuf** data.

Description

The **m_copydata** kernel service makes a copy of the structure specified by the *m* parameter. The copy begins at the specified bytes (represented by the *off* parameter) and continues for the number of bytes specified by the *len* parameter. The data is copied into the buffer specified by the *cp* parameter.

Execution Environment

The **m_copydata** kernel service can be called from either the process or interrupt environment.

Return Values

The **mcopydata** service has no return values.

Implementation Specifics

The **m_copydata** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **m_copy** macro.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

m_copym Kernel Service

Purpose

Creates a copy of all or part of a list of **mbuf** structures.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/mbuf.h>

struct mbuf *
m_copym(m, off, len, wait)
struct mbuf  m;
int  off;
int  len;
int  wait;
```

Parameters

m	Specifies the **mbuf** structure to be copied.
off	Specifies an offset into data from which copying will start.
len	Specifies the total number of bytes to copy.
wait	Specifies either the **M_DONTWAIT** or **M_WAIT** value.

Description

The **m_copym** kernel service makes a copy of the **mbuf** structure specified by the *m* parameter starting at the specified offset from the beginning and continuing for the number of bytes specified by the *len* parameter. If the *len* parameter is set to **M_COPYALL**, the entire **mbuf** chain is copied.

If the **mbuf** structure specified by the *m* parameter has an external buffer attached (that is, the **M_EXT** flag is set), the copy is done by reference to the external cluster. In this case, the data must not be altered or both copies will be changed. Interrupt handlers can specify the *wait* parameter as **M_DONTWAIT** only.

Execution Environment

The **m_copym** kernel service can be called from either the process or interrupt environment.

Return Values

The address of the copy is returned upon successful completion. If the copy fails, null is returned. If the *wait* parameter is set to **M_DONTWAIT** and there are no free **mbuf** structures, the **m_copym** kernel service returns a null value.

Implementation Specifics

The **m_copym** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **m_copydata** kernel service.

The **m_copy** macro.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

m_dereg Kernel Service

Purpose

Deregisters expected **mbuf** structure usage.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/mbuf.h>

void m_dereg (mbp)
struct mbreq mbp;
```

Parameter

mbp Defines the address of an **mbreq** structure that specifies expected **mbuf** usage.

Description

The **m_dereg** kernel service deregisters requirements previously registered with the **m_reg** kernel service. The **m_dereg** service is mandatory if the **m_reg** service is called.

Execution Environment

The **m_dereg** kernel service can be called from the process environment only.

Return Values

The **m_dereg** service has no return values.

Implementation Specifics

The **m_dereg** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **mbreq** Structure for **mbuf** Kernel Services.

The **m_reg** kernel service.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

m_free Kernel Service

Purpose

Frees an **mbuf** structure and any associated external storage area.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/mbuf.h>

struct mbuf *m_free(m)
struct mbuf *m;
```

Parameter

m	Specifies the **mbuf** structure to be freed.

Description

The **m_free** kernel service returns an **mbuf** structure to the buffer pool. If the **mbuf** structure specified by the *m* parameter has an attached cluster (that is, a paged–size **mbuf** structure), the **m_free** kernel service also frees the associated external storage.

Execution Environment

The **m_free** kernel service can be called from either the process or interrupt environment.

Return Values

If the **mbuf** structure specified by the *m* parameter is the head of an **mbuf** chain, the **m_free** service returns the next **mbuf** structure in the chain. A null value is returned if the structure specified by the *m* parameter is not part of an **mbuf** chain.

Implementation Specifics

The **m_free** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **m_get** kernel service.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

m_freem Kernel Service

Purpose

Frees an entire **mbuf** chain.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/mbuf.h>

void m_freem (m)
struct mbuf *m;
```

Parameter

m	Indicates the head of the **mbuf** chain to be freed.

Description

The **m_freem** kernel service starts the **m_free** kernel service for each **mbuf** structure in the chain headed by the head specified by the *m* parameter.

Execution Environment

The **m_freem** kernel service can be called from either the process or interrupt environment.

Return Values

The **m_freem** service has no return values.

Implementation Specifics

The **m_freem** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **m_free** kernel service, **m_get** kernel service.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

m_get Kernel Service

Purpose

Allocates a memory buffer (mbuf) from the **mbuf** pool.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/mbuf.h>

struct mbuf *m_get (wait, type)
int  wait;
int  type;
```

Parameters

wait	Indicates the action to be taken if there are no free **mbuf** structures. Possible values are:
	M_DONTWAIT Called from either an interrupt or process environment.
	M_WAIT Called from a process environment.
type	Specifies a valid **mbuf** type, as listed in the **/usr/include/sys/mbuf.h** file.

Description

The **m_get** kernel service allocates an **mbuf** structure of the specified type. If the buffer pool is empty and the *wait* parameter is set to **M_WAIT**, the **m_get** kernel service does not return until an **mbuf** structure is available.

Execution Environment

The **m_get** kernel service can be called from either the process or interrupt environment.

An interrupt handler can specify the *wait* parameter as **M_DONTWAIT** only.

Return Values

Upon successful completion, the **m_get** service returns the address of an allocated **mbuf** structure. If the *wait* parameter is set to **M_DONTWAIT** and there are no free **mbuf** structures, the **m_get** kernel service returns a null value.

Implementation Specifics

The **m_get** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **m_free** kernel service, **m_freem** kernel service.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

m_getclr Kernel Service

Purpose

Allocates and zeroes a memory buffer from the **mbuf** pool.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/mbuf.h>

struct mbuf *m_getclr (wait, type)
int wait;
int type;
```

Parameters

wait This flag indicates the action to be taken if there are no free **mbuf** structures. Possible values are:

M_DONTWAIT Called from either an interrupt or process environment.

M_WAIT Called from a process environment only.

type Specifies a valid **mbuf** type, as listed in the /**usr**/**include**/**sys**/**mbuf.h** file.

Description

The **m_getclr** kernel service allocates an **mbuf** structure of the specified type. If the buffer pool is empty and the *wait* parameter is set to **M_WAIT** value, the **m_getclr** service does not return until an **mbuf** structure is available.

The **m_getclr** kernel service differs from the **m_get** kernel service in that the **m_getclr** service zeroes the data portion of the allocated **mbuf** structure.

Execution Environment

The **m_getclr** kernel service can be called from either the process or interrupt environment. Interrupt handlers can call the **m_getclr** service only with the *wait* parameter set to the **M_DONTWAIT** value.

Return Values

The **m_getclr** kernel service returns the address of an allocated **mbuf** structure. If the *wait* parameter is set to the **M_DONTWAIT** value and there are no free **mbuf** structures, the **m_getclr** kernel service returns a null value.

Implementation Specifics

The **m_getclr** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **m_free** kernel service, **m_freem** kernel service, **m_get** kernel service.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

m_getclust Macro for mbuf Kernel Services

Purpose

Allocates an **mbuf** structure from the **mbuf** buffer pool and attaches a page–sized cluster.

Syntax

```
#include <sys/mbuf.h>

struct mbuf *m_getclust (wait, type)
int  wait;
int  type;
```

Parameters

wait Indicates the action to be taken if there are no available **mbuf** structures. Possible values are:

M_DONTWAIT Called from either an interrupt or process environment.

M_WAIT Called from a process environment only.

type Specifies a valid **mbuf** type from the /**usr**/**include**/**sys**/**mbuf.h** file.

Description

The **m_getclust** macro allocates an **mbuf** structure of the specified type. If the allocation succeeds, the **m_getclust** macro then attempts to attach a page–sized cluster to the structure.

If the buffer pool is empty and the *wait* parameter is set to **M_WAIT**, the **m_getclust** macro does not return until an **mbuf** structure is available.

Execution Environment

The **m_getclust** macro can be called from either the process or interrupt environment.

Return Values

The address of an allocated **mbuf** structure is returned on success. If the *wait* parameter is set to **M_DONTWAIT** and there are no free **mbuf** structures, the **m_getclust** macro returns a null value.

Implementation Specifics

The **m_getclust** macro is part of Base Operating System (BOS) Runtime.

Related Information

The **m_getclustm** kernel service.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

m_getclustm Kernel Service

Purpose

Allocates an **mbuf** structure from the **mbuf** buffer pool and attaches a cluster of the specified size.

Syntax

```
#include <sys/mbuf.h>

struct mbuf *
m_getclustm(wait, type, size)
int wait;
int type;
int size;
```

Parameters

wait	Specifies either the **M_DONTWAIT** or **M_WAIT** value.
type	Specifies a valid **mbuf** type from the **/usr/include/sys/mbuf.h** file.
size	Specifies the size of the external cluster to attach. Valid sizes are in the **/usr/include/sys/mbuf.h** file.

Description

The **m_getclustm** service allocates an **mbuf** structure of the specified type. If successful, the **m_getclustm** service then attempts to attach a cluster of the indicated size (specified by the *size* parameter) to the **mbuf** structure. If the buffer pool is empty and the *wait* parameter is set to **M_WAIT**, the **m_get** service does not return until an **mbuf** structure is available. Interrupt handlers should call this service only with the *wait* parameter set to **M_DONTWAIT**.

Execution Environment

The **m_getclustm** kernel service can be called from either the process or interrupt environment.

An interrupt handler can specify the *wait* parameter as **M_DONTWAIT** only.

Return Values

The **m_getclustm** kernel service returns the address of an allocated **mbuf** structure on success. If the *wait* parameter is set to **M_DONTWAIT** and there are no free **mbuf** structures, the **m_getclustm** kernel service returns null.

Implementation Specifics

The **m_getclustm** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **m_clget** kernel service, **m_free** kernel service, **m_freem** kernel service, **m_get** kernel service.

The **m_getclust** macro.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

m_gethdr Kernel Service

Purpose

Allocates a header memory buffer from the **mbuf** pool.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/mbuf.h>

struct mbuf *
m_gethdr (wait, type)
int  wait;
int  type;
```

Parameters

wait	Specifies either the **M_DONTWAIT** or **M_WAIT** value.
type	Specifies the valid **mbuf** type from the **/usr/include/sys/mbuf.h** file.

Description

The **m_gethdr** kernel service allocates an **mbuf** structure of the specified type. If the buffer pool is empty and the *wait* parameter is set to **M_WAIT**, the **m_gethdr** kernel service will not return until an **mbuf** structure is available. Interrupt handlers should call this kernel service only with the *wait* parameter set to **M_DONTWAIT**. The **M_PKTHDR** flag is set for the returned **mbuf** structure.

Execution Environment

The **m_gethdr** kernel service can be called from either the process or interrupt environment.

An interrupt handler can specify the *wait* parameter as **M_DONTWAIT** only.

Return Values

The address of an allocated **mbuf** structure is returned on success. If the *wait* parameter is set to **M_DONTWAIT** and there are no free **mbuf** structure, the **m_gethdr** kernel service returns null.

Implementation Specifics

The **m_gethdr** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **m_free** kernel service, **m_freem** kernel service.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

M_HASCL Macro for mbuf Kernel Services

Purpose

Determines if an **mbuf** structure has an attached cluster.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/mbuf.h>

struct mbuf *m;
M_HASCL (m);
```

Parameter

m	Indicates the address of the **mbuf** structure in question.

Description

The **M_HASCL** macro determines if an **mbuf** structure has an attached cluster.

Execution Environment

The **M_HASCL** macro can be called from either the process or interrupt environment.

Example

The **M_HASCL** macro can be used as in the following example:

```
struct mbuf  *m;
if (M_HASCL(m))
   printf("mbuf has attached cluster");
```

Implementation Specifics

The **M_HASCL** macro is part of Base Operating System (BOS) Runtime.

Related Information

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

m_pullup Kernel Service

Purpose

Adjusts an **mbuf** chain so that a given number of bytes is in contiguous memory in the data area of the head **mbuf** structure.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/mbuf.h>

struct mbuf *m_pullup (m, size)
struct mbuf *m;
int size;
```

Parameters

m	Specifies the **mbuf** chain to be adjusted.
size	Specifies the number of bytes to be contiguous.

Description

The **m_pullup** kernel service guarantees that the **mbuf** structure at the head of a chain has in contiguous memory within its data area at least the number of data bytes specified by the *size* parameter.

Execution Environment

The **m_pullup** kernel service can be called from either the process or interrupt environment.

Return Values

Upon successful completion, the head structure in the altered **mbuf** chain is returned.

A value of null is returned and the original chain is deallocated under the following circumstances:

- The size of the chain is less than indicated by the *size* parameter.
- The number indicated by the *size* parameter is greater than the data portion of the head–size **mbuf** structure.

Implementation Specifics

The **m_pullup** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

m_reg Kernel Service

Purpose

Registers expected **mbuf** usage.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/mbuf.h>

void m_reg (mbp)
struct mbreq mbp;
```

Parameter

mbp Defines the address of an **mbreq** structure that specifies expected **mbuf** usage.

Description

The **m_reg** kernel service lets users of **mbuf** services specify initial requirements. The **m_reg** kernel service also allows the buffer pool low–water and deallocation marks to be adjusted based on expected usage. Its use is recommended for better control of the buffer pool.

When the number of free **mbuf** structures falls below the low–water mark, the total **mbuf** pool is expanded. When the number of free **mbuf** structures rises above the deallocation mark, the total **mbuf** pool is contracted and resources are returned to the system.

Execution Environment

The **m_reg** kernel service can be called from the process environment only.

Return Values

The **m_reg** service has no return values.

Implementation Specifics

The **m_reg** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **mbreq** structure for **mbuf** kernel services, the **m_dereg** kernel service.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

md_restart_block_read Kernel Service

Purpose

A copy of the **RESTART_BLOCK** structure in the **NVRAM** header will be placed in the caller's buffer.

Syntax

```
#include <sys/mdio.h>

int md_restart_block_read (md)
                struct mdio *md;
```

Parameters

md Specifies the address of the **mdio** structure. The **mdio** structure contains the following fields:

md_data	Pointer to the data buffer.
md_size	Number of bytes in the data buffer.
md_addr	Contains the value PMMode on return in the least significant byte.

Description

The RestartBlock which is in the **NVRAM** header will be copied to the user supplied buffer. This block is a communication vehicle for the Power Management software and the firmware.

Return Values

Returns 0 for successful completion.

ENOMEM	Indicates that there was not enough room in the user supplied buffer to contain the RestartBlock.
EINVAL	Indicates this is not a PowerPC reference platform.

Prerequisite Information

Kernel Extensions and Device Driver Management Kernel Services in Kernel Extensions and Device Support Programming Concepts.

Implementation Specifics

The **md_restart_block_read** Kernel Service is part of the Base Operating System (BOS) Runtime.

Related Information

Machine Device Driver in *AIX Version 4.1 Technical Reference, Volume 6: Kernel and Subsystems*.

md_restart_block_upd Kernel Service

Purpose

The caller supplied RestartBlock will be copied to the **NVRAM** header.

Syntax

```
#include <sys/mdio.h>

int md_restart_block_upd (md, pmmode)
                struct mdio *md;
                unsigned char pmmode;
```

Description

The 8–bit value in *pmmode* will be stored into the **NVRAM** header at the PMMode offset.The RestartBlock which is in the caller's buffer will be copied to the **NVRAM** after the RestartBlock checksum is calculated and a new Crc1 value is computed.

Parameters

md	Specifies the address of the **mdio** structure. The **mdio** structure contains the following fields: **md_data** Pointer to the RestartBlock structure..
pmmode	Value to be stored into PMMode in the **NVRAM** header.

Return Values

Returns 0 for successful completion.

EINVAL	Indicates this is not a PowerPC reference platform.

Prerequisite Information

Kernel Extensions and Device Driver Management Kernel Services in Kernel Extensions and Device Support Programming Concepts.

Implementation Specifics

The **md_restart_block_upd** Kernel Service is part of the Base Operating System (BOS) Runtime.

Related Information

Machine Device Driver in *AIX Version 4.1 Technical Reference, Volume 6: Kernel and Subsystems*.

MTOCL Macro for mbuf Kernel Services

Purpose

Converts a pointer to an **mbuf** structure to a pointer to the head of an attached cluster.

Syntax

```
#include <sys/mbuf.h>

struct mbuf *m;
MTOCL (m);
```

Parameter

m	Indicates the address of the **mbuf** structure in question.

Description

The **MTOCL** macro converts a pointer to an **mbuf** structure to a pointer to the head of an attached cluster.

The **MTOCL** macro can be used as in the following example:

```
caddr_t attcls;
struct mbuf      *m;
attcls = (caddr_t) MTOCL(m);
```

Execution Environment

The **MTOCL** macro can be called from either the process or interrupt environment.

Implementation Specifics

The **MTOCL** macro is part of Base Operating System (BOS) Runtime.

Related Information

The **M_HASCL** macro for **mbuf** kernel services.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

MTOD Macro for mbuf Kernel Services

Purpose

Converts a pointer to an **mbuf** structure to a pointer to the data stored in that **mbuf** structure.

Syntax

```
#include <sys/mbuf.h>
MTOD (m, type);
```

Parameters

m	Identifies the address of an **mbuf** structure.
type	Indicates the type to which the resulting pointer should be cast.

Description

The **MTOD** macro converts a pointer to an **mbuf** structure into a pointer to the data stored in the **mbuf** structure. This macro can be used as in the following example:

```
char    *bufp;
         bufp = MTOD(m, char *);
```

Execution Environment

The **MTOD** macro can be called from either the process or interrupt environment.

Implementation Specifics

The **MTOD** macro is part of Base Operating System (BOS) Runtime.

Related Information

The **DTOM** macro for **mbuf** Kernel Services.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

M_XMEMD Macro for mbuf Kernel Services

Purpose

Returns the address of an **mbuf** cross–memory descriptor.

Syntax

```
#include   <sys/mbuf.h>
#include   <sys/xmem.h>

struct mbuf *m;

M_XMEMD (m);
```

Parameter

m	Specifies the address of the **mbuf** structure in question.

Description

The **M_XMEMD** macro returns the address of an **mbuf** cross–memory descriptor.

Execution Environment

The **M_XMEMD** macro can be called from either the process or interrupt environment.

Example

The **M_XMEMD** macro can be used as in the following example:

```
struct mbuf   *m;
struct xmem   *xmemd;

xmemd = M_XMEMD(m);
```

Implementation Specifics

The **M_XMEMD** macro is part of Base Operating System (BOS) Runtime.

Related Information

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

net_attach Kernel Service

Purpose

Opens a communications I/O device handler.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <aixif/net_if.h>
#include <sys/comio.h>

int net_attach (kopen_ext, device_req, netid, netfpp)
struct  kopen_ext *kopen_ext;
struct  device_req *device_req;
struct  netid_list *netid;
struct file  **netfpp;
```

Parameters

kopen_ext	Specifies the device handler kernel open extension.
device_req	Indicates the address of the device description structure.
netid	Indicates the address of the network ID list.
netfpp	Specifies the address of the variable that will hold the returned file pointer.

Description

The **net_attach** kernel service opens the device handler specified by the *device_req* parameter and then starts all the network IDs listed in the address specified by the *netid* parameter. The **net_attach** service then sleeps and waits for the asynchronous `start completion` notifications from the **net_start_done** kernel service.

Execution Environment

The **net_attach** kernel service can be called from the process environment only.

Return Values

Upon success, a value of 0 is returned and a file pointer is stored in the address specified by the *netfpp* parameter. Upon failure, the **net_attach** service returns either the error codes received from the **fp_opendev** or **fp_ioctl** kernel service, or the value **ETIMEDOUT**. The latter value is returned when an open operation times out.

Implementation Specifics

The **net_attach** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **net_detach** kernel service, **net_start** kernel service, **net_start_done** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

net_detach Kernel Service

Purpose

Closes a communications I/O device handler.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <aixif/net_if.h>

int net_detach (netfp)
struct file   *netfp;
```

Parameter

netfp Points to an open file structure obtained from the **net_attach** kernel service.

Description

The **net_detach** kernel service closes the device handler associated with the file pointer specified by the *netfp* parameter.

Execution Environment

The **net_detach** kernel service can be called from the process environment only.

Return Values

The **net_detach** service returns the value it obtains from the **fp_close** service.

Implementation Specifics

The **net_detach** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **fp_close** kernel service, **net_attach** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

net_error Kernel Service

Purpose

Handles errors for communication network interface drivers.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <net/if.h>
#include <sys/comio.h>

net_error (ifp, error_code, netfp)
struct  ifnet *ifp;
int error_code;
struct  file *netfp;
```

Parameters

error_code	Specifies the error code listed in the **/usr/include/sys/comio.h** file.
ifp	Specifies the address of the **ifnet** structure for the device with an error.
netfp	Specifies the file pointer for the device with an error.

Description

The **net_error** kernel service provides generic error handling for AIX communications network interface (**if**) drivers. Network interface (**if**) kernel extensions call this service to trace errors and, in some instances, perform error recovery.

Errors traced include those:

- Received from the communications adapter drivers.
- Occurring during input and output packet processing.

Execution Environment

The **net_error** kernel service can be called from either the process or interrupt environment.

Return Values

The **net_error** service has no return values.

Implementation Specifics

The **net_error** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **net_attach** kernel service, **net_detach** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

net_sleep Kernel Service

Purpose

Sleeps on the specified wait channel.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/pri.h>

net_sleep (chan, flags)
int chan;
int flags;
```

Parameters

chan	Specifies the wait channel to sleep upon.
flags	Sleep flags described in the **sleep** kernel service.

Description

The **net_sleep** kernel service puts the caller to sleep waiting on the specified wait channel. If the caller holds the network lock, the **net_sleep** kernel service releases the lock before sleeping and reacquires the lock when the caller is awakened.

Execution Environment

The **net_sleep** kernel service can be called from the process environment only.

Return Values

0	Indicates that the sleeping process was not awakened by a signal.
1	Indicates that the sleeper was awakened by a signal.

Implementation Specifics

The **net_sleep** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **net_wakeup** kernel service, **sleep** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

net_start Kernel Service

Purpose

Starts network IDs on a communications I/O device handler.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <aixif/net_if.h>
#include <sys/comio.h>

struct  file *net_start (netfp, netid)
struct  file *netfp;
struct  netid_list *netid;
```

Parameters

netfp	Specifies the file pointer of the device handler.
netid	Specifies the address of the network ID list.

Description

The **net_start** kernel service starts all the network IDs listed in the list specified by the *netid* parameter. This service then waits for the asynchronous notification of completion of starts.

Execution Environment

The **net_start** kernel service can be called from the process environment only.

Return Values

The **net_start** service uses the return value returned from a call to the **fp_ioctl** service requesting the **CIO_START** operation.

ETIMEDOUT	Indicates that the start for at least one network ID timed out waiting for start–done notifications from the device handler.

Implementation Specifics

The **net_start** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **fp_ioctl** kernel service, **net_attach** kernel service, **net_start_done** kernel service,.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

net_start_done Kernel Service

Purpose

Starts the done notification handler for communications I/O device handlers.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <aixif/net_if.h>
#include <sys/comio.h>

void net_start_done (netid, sbp)
struct  netid_list *netid;
struct  status_block *sbp;
```

Parameters

netid	Specifies the address of the network ID list for the device being started.
sbp	Specifies the status block pointer returned from the device handler.

Description

The **net_start_done** kernel service is used to mark the completion of a network ID start operation. When all the network IDs listed in the *netid* parameter have been started, the **net_attach** kernel service returns to the caller. The **net_start_done** service should be called when a **CIO_START_DONE** status block is received from the device handler. If the status block indicates an error, the start process is immediately aborted.

Execution Environment

The **net_start_done** kernel service can be called from either the process or interrupt environment.

Return Values

The **net_start_done** service has no return values.

Implementation Specifics

The **net_start_done** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **net_attach** kernel service, **net_start** kernel service.

The **CIO_START_DONE** status block.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

net_wakeup Kernel Service

Purpose

Wakes up all sleepers waiting on the specified wait channel.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

net_wakeup (chan)
int chan;
```

Parameter

chan	Specifies the wait channel.

Description

The **net_wakeup** service wakes up all network processes sleeping on the specified wait channel.

Execution Environment

The **net_wakeup** kernel service can be called from either the process or interrupt environment.

Return Values

The **net_wakeup** service has no return values.

Implementation Specifics

The **net_wakeup** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **net_sleep** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

net_xmit Kernel Service

Purpose

Transmits data using a communications device handler .

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <aixif/net_if.h>

int net_xmit (ifp, m, netfp, lngth, m_ext)
struct  ifnet *ifp;
struct  mbuf *m;
struct  file *netfp;
int  lngth;
struct mbuf *m_ext;
```

Parameters

ifp	Indicates an address of the **ifnet** structure for this interface.
m	Specifies the address of an **mbuf** structure containing the data to transmit.
netfp	Indicates the open file pointer obtained from the **net_attach** kernel service.
lngth	Indicates the total length of the buffer being transmitted.
m_ext	Indicates the address of an **mbuf** structure containing a write extension.

Description

The **net_xmit** kernel service builds a **uio** structure and then invokes the **fp_rwuio** service to transmit a packet. The **net_xmit_trace** kernel service is an alternative for network interfaces that choose not to use the **net_xmit** kernel service.

Execution Environment

The **net_xmit** kernel service can be called from either the process or interrupt environment.

Return Values

0	Indicates that the packet was transmitted successfully.
ENOBUFS	Indicates that buffer resources were not available.

The **net_xmit** kernel service returns a value from the **fp_rwuio** service when an error occurs during a call to that service.

Implementation Specifics

The **net_xmit** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **fp_rwuio** kernel service, **net_xmit_trace** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

net_xmit_trace Kernel Service

Purpose

Traces transmit packets.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int net_xmit_trace ( ifp, mbuf )
struct ifnet *ifp;
struct mbuf *mbuf;
```

Parameters

ifp	Designates the address of the **ifnet** structure for this interface.
mbuf	Designates the address of the **mbuf** structure to be traced.

Description

The **net_xmit_trace** kernel service traces the data pointed to by the *mbuf* parameter. This kernel service was added for those network interfaces that choose not to use the **net_xmit** kernel service to transmit packets. An application program (the **iptrace** command) reads the trace data and writes it to a file for the **ipreport** command to interpret.

Execution Environment

The **net_xmit_trace** kernel service can be called from either the process or interrupt environment.

Return Values

The **net_xmit_trace** kernel service has no return values.

Implementation Specifics

The **net_xmit_trace** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **net_xmit** kernel service.

The **ipreport** command.

The **iptrace** daemon.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

NLuprintf Kernel Service

Purpose

Submits a request to print an internationalized message to a process' controlling terminal.

Syntax

```
#include <sys/uprintf.h>

int NLuprintf (Uprintf)
struct uprintf *Uprintf;
```

Parameters

Uprintf	Points to a **uprintf** request structure.

Description

The **NLuprintf** kernel service submits a internationalized kernel message request with the **uprintf** request structure specified by the *Uprintf* parameter as input. Once the request has been successfully submitted, the **uprintfd** daemon retrieves, converts, formats, and writes the message described by the **uprintf** request structure to a process' controlling terminal.

The caller must initialize the **uprintf** request structure before calling the **NLuprintf** kernel service. Fields in the **uprintf** request structure use several constants. The following constants are defined in the /**usr**/**include**/**sys**/**uprintf.h** file:

- **UP_MAXSTR**
- **UP_MAXARGS**
- **UP_MAXCAT**
- **UP_MAXMSG**

The **uprintf** request structure consists of the following fields:

`Uprintf->upf_defmsg` Points to a default message format. The default message format is a character string that contains either or both of two types of objects:

- Plain characters, which are copied to the message output stream
- Conversion specifications, each of which causes zero or more items to be fetched from the *Uprintf–>arg* value parameter array

Each conversion specification consists of a % (percent sign) followed by a character that indicates the type of conversion to be applied:

% Performs no conversion. Prints a % character.

d, **i** Accepts an integer value and converts it to signed decimal notation.

u Accepts an integer value and converts it to unsigned decimal notation.

o Accepts an integer value and converts it to unsigned octal notation.

x Accepts an integer value and converts it to unsigned hexadecimal notation.

c Accepts and prints a **char** value.

s Accepts a value as a string (character pointer). Characters from the string are printed until a \0 (null character) is encountered.

Field–width or precision conversion specifications are not supported.

The maximum length of the default message–format string pointed to by the `Uprintf->upf_defmsg` field is the number of characters specified by the **UP_MAXSTR** constant. The `Uprintf->upf_defmsg` field must be a nonnull character.

The default message format is used in constructing the kernel message if the message format described by the `Uprintf->upf_NLsetno` and `Uprint->upf_NLmsgno` fields cannot be retrieved from the message catalog specified by `Uprintf->upf_NLcatname`. The conversion specifications contained within the default message format should match those contained in the message format specified by the `upf_NLsetno` and `upf_NLmsgno` fields.

`Uprintf->upf_arg[UP_MAXARGS]` Specifies from zero to the number of value parameters specified by the **UP_MAXARGS** constant. A *Value* parameter may be a integer value, a character value, or a string value (character pointer). Strings are limited in length to the number of characters specified by the **UP_MAXSTR** constant. String value parameters must be nonnull characters. The number, type, and order of items in the *Value* parameter array should match the conversion specifications within the message format string.

`Uprintf->upf_NLcatname` Points to the message catalog file name. If the catalog file name referred to by the `Uprintf->upf_NLcatname` field begins with a / (slash), it is assumed to be an absolute path name. If the catalog file name is not an absolute path name, the process environment determines the directory paths to search. The maximum length of the catalog file name is limited to the number of characters specified by the **UP_MAXCAT** constant. The value of the `Uprintf->upf_NLcatname` field must be a nonnull character.

`Uprintf->upf_NLsetno`	Specifies the set ID.
`Uprintf->upf_NLmsgno`	Specifies the message ID. The `Uprintf->upf_NLsetno` and `Uprintf->upf_NLmsgno` fields specify a particular message format string to be retrieved from the message catalog specified by the `Uprintf->upf_NLcatname` field. The maximum length of the constructed kernel message is limited to the number of characters specified by the **UP_MAXMSG** constant. Messages larger then the number of characters specified by the **UP_MAXMSG** constant are discarded.

Execution Environment

The **NLuprintf** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
ENOMEM	Indicates that memory is not available to buffer the request.
ENODEV	Indicates that a controlling terminal does not exist for the process.
ESRCH	Indicates the **uprintfd** daemon is not active. No requests may be submitted.
EINVAL	Indicates that the message catalog file–name pointer is null or the catalog file name is greater than the number of characters specified by the **UP_MAXCAT** constant.
EINVAL	Indicates that a string–value parameter pointer is null or the string–value parameter is greater than the number of characters specified by the **UP_MAXCAT** constant.
EINVAL	Indicates one of the following: • Default message format pointer is null. • Number of characters in the default message format is greater than the number specified by the **UP_MAXSTR** constant. • Number of conversion specifications contained within the default message format is greater than the number specified by the **UP_MAXARGS** constant.

Implementation Specifics

This **NLuprintf** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **uprintf** kernel service.

The **uprintfd** daemon.

Process and Exception Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

ns_add_demux Network Kernel Service

Purpose

Adds a demuxer for the specified type of network interface.

Syntax

```
#include <sys/ndd.h>
#include <sys/cdli.h>

int ns_add_demux (ndd_type, demux)
        u_long ndd_type;
        struct ns_demuxer *demux;
```

Parameters

ndd_type	Specifies the interface type of the demuxer to be added.
demux	Specifies the pointer to an **ns_demux** structure that defines the demuxer.

Description

The **ns_add_demux** network service adds the specified demuxer to the list of available network demuxers. Only one demuxer per network interface type can exist. An interface type describes a certain class of network devices that have the same characteristics (such as ethernet or token ring). The values of the *ndd_type* parameter listed in the **/usr/include/sys/ndd.h** file are the numbers defined by Simple Network Management Protocol (SNMP). If the desired type is not in the **ndd.h** file, the SNMP value should be used if it is defined. Otherwise, any undefined type above **NDD_MAX_TYPE** may be used.

Note: The **ns_demuxer** structure must be allocated and pinned by the network demuxer.

Examples

The following example illustrates the **ns_add_demux** network service:

```
struct ns_demuxer demuxer;
bzero (&demuxer, sizeof (demuxer));
demuxer.nd_add_filter = eth_add_filter;
demuxer.nd_del_filter = eth_del_filter;
demuxer.nd_add_status = eth_add_status;
demuxer.nd_del_status = eth_del_status;
demuxer.nd_receive = eth_receive;
demuxer.nd_status = eth_status;
demuxer.nd_response = eth_response;
demuxer.nd_use_nsdnx = 1;
ns_add_demux(NDD_ISO88023, &demuxer);
```

Return Values

0	Indicates the operation was successful.
EEXIST	Indicates a demuxer already exists for the given type.

Related Information

The **ns_del_demux** network service.

ns_add_filter Network Service

Purpose

Registers a receive filter to enable the reception of packets.

Syntax

```
#include <sys/cdli.h>
#include <sys/ndd.h>

int ns_add_filter (nddp, filter, len, ns_user)
       struct ndd *nddp;
       caddr_t filter;
       int len;
       struct ns_user *ns_user;
```

Parameters

nddp	Specifies the **ndd** structure to which this add request applies.
filter	Specifies the pointer to the receive filter.
len	Specifies the length in bytes of the receive filter to which the *filter* parameter points.
ns_user	Specifies the pointer to a **ns_user** structure that defines the user.

Description

The **ns_add_filter** network service registers a receive filter for the reception of packets and enables a network demuxer to route packets to the appropriate users. The **add** request is passed on to the **nd_add_filter** function of the demuxer for the specified NDD. The caller of the **ns_add_filter** network service is responsible for relinquishing filters before calling the **ns_free** network service.

Examples

The following example illustrates the **ns_add_filter** network service:

```
struct ns_8022 dl;
struct ns_user ns_user;

dl.filtertype = NS_LLC_DSAP_SNAP;
dl.dsap = 0xaa;
dl.orgcode[0] = 0x0;
dl.orgcode[1] = 0x0;
dl.orgcode[2] = 0x0;
dl.ethertype = 0x0800;
ns_user.isr = ipintr;
ns_user.protoq = &ipintrq;
ns_user.netisr = NETISR_IP;
ns_user.ifp = ifp;
ns_user.pkt_format = NS_PROTO_SNAP;
ns_add_filter(nddp, &dl, sizeof(dl), &ns_user);
```

Return Values

0	Indicates the operation was successful.

The network demuxer may supply other return values.

Related Information

The **ns_del_filter** network service.

ns_add_status Network Service

Purpose

Adds a status filter for the routing of asynchronous status.

Syntax

```
#include <sys/cdli.h>
#include <sys/ndd.h>

int ns_add_status (nddp, statfilter, len, ns_statuser)
        struct ndd *nddp;
        caddr_t statfilter;
        int len;
        struct ns_statuser *ns_statuser;
```

Parameters

nddp	Specifies a pointer to the **ndd** structure to which this add request applies.
statfilter	Specifies a pointer to the status filter.
len	Specifies the length, in bytes, of the value of the *statfilter* parameter.
ns_statuser	Specifies a pointer to an **ns_statuser** structure that defines this user.

Description

The **ns_add_status** network service registers a status filter. The add request is passed on to the **nd_add_status** function of the demuxer for the specified network device driver (NDD). This network service enables the user to receive asynchronous status information from the specified device.

Note: The user's status processing function is specified by the `isr` field of the **ns_statuser** structure. The network demuxer calls the user's status processing function directly when asynchronous status information becomes available. Consequently; the status processing function cannot be a scheduled routine. The caller of the **ns_add_status** network service is responsible for relinquishing status filters before calling the **ns_free** network service.

Examples

The following example illustrates the **ns_add_status** network service:

```
struct ns_statuser   user;
struct ns_com_status   filter;

filter.filtertype = NS_STATUS_MASK;
filter.mask = NDD_HARD_FAIL;
filter.sid = 0;
user.isr = status_fn;
user.isr_data = whatever_makes_sense;

error = ns_add_status(nddp, &filter, sizeof(filter), &user);
```

Return Values

0	Indicates the operation was successful.

The network demuxer may supply other return values.

Related Information

The **ns_del_status** network service.

ns_alloc Network Service

Purpose

Allocates use of a network device driver (NDD).

Syntax

```
#include <sys/ndd.h>

int ns_alloc (nddname, nddpp)
        char *nddname;
        struct ndd **nddpp;
```

Parameters

nddname	Specifies the device name to be allocated.
nddpp	Indicates the address of the pointer to a **ndd** structure.

Description

The **ns_alloc** network service searches the Network Service (NS) device chain to find the device driver with the specified *nddname* parameter. If the service finds a match, it increments the reference count for the specified device driver. If the reference count is incremented to 1, the **ndd_open** subroutine specified in the **ndd** structure is called to open the device driver.

Examples

The following example illustrates the **ns_alloc** network service:

```
struct ndd   *nddp;
error = ns_alloc(”en0”, &nddp);
```

Return Values

If a match is found and the **ndd_open** subroutine to the device is successful, a pointer to the **ndd** structure for the specified device is stored in the *nddpp* parameter. If no match is found or the open of the device is unsuccessful, a non–zero value is returned.

0	Indicates the operation was successful.
ENODEV	Indicates an invalid network device.
ENOENT	Indicates no network demuxer is available for this device.

The **ndd_open** routine may specify other return values.

Related Information

The **ns_free** network service.

ns_attach Network Service

Purpose

Attaches a network device to the network subsystem.

Syntax

```
#include <sys/ndd.h>

int ns_attach (nddp)
       struct ndd *nddp;
```

Parameters

nddp Specifies a pointer to an **ndd** structure describing the device to be attached.

Description

The **ns_attach** network service places the device into the available network service (NS) device chain. The network device driver (NDD) should be prepared to be opened after the **ns_attach** network service is called.

Note: The **ndd** structure is allocated and initialized by the device. It should be pinned.

Examples

The following example illustrates the **ns_attach** network service:

```
struct ndd ndd;
ndd.ndd_name = ”en0”;
ndd.ndd_addrlen = 6;
ndd.ndd_hdrlen = 14;
ndd.ndd_mtu = ETHERMTU;
ndd.ndd_mintu = 60;
ndd.ndd_type = NDD_ETHER;
ndd.ndd_flags =
   NDD_BROADCAST | NDD_SIMPLEX;
ndd.ndd_open = entopen;
ndd.ndd_output = entwrite;
ndd.ndd_ctl = entctl;
ndd.ndd_close = entclose;
.
.
.
ns_attach(&ndd);
```

Return Values

0	Indicates the operation was successful.
EEXIST	Indicates the device is already in the available NS device chain.

Related Information

The **ns_detach** network service.

ns_del_demux Network Service

Purpose

Deletes a demuxer for the specified type of network interface.

Syntax

```
#include <sys/ndd.h>

int ns_del_demux (ndd_type)
        u_long ndd_type;
```

Parameters

ndd_type	Specifies the network interface type of the demuxer that is to be deleted.

Description

If the demuxer is not currently in use, the **ns_del_demux** network service deletes the specified demuxer from the list of available network demuxers. A demuxer is in use if a network device driver (NDD) is open for the demuxer.

Examples

The following example illustrates the **ns_del_demux** network service:

```
ns_del_demux(NDD_ISO88023);
```

Return Values

0	Indicates the operation was successful.
ENOENT	Indicates the demuxer of the specified type does not exist.

Related Information

The **ns_add_demux** network service.

ns_del_filter Network Service

Purpose

Deletes a receive filter.

Syntax

```
#include <sys/cdli.h>
#include <sys/ndd.h>

int ns_del_filter (nddp, filter, len)
        struct ndd *nddp;
        caddr_t filter;
        int len;
```

Parameters

nddp	Specifies the **ndd** structure that this delete request is for.
filter	Specifies the pointer to the receive filter.
len	Specifies the length in bytes of the receive filter.

Description

The **ns_del_filter** network service deletes the receive filter from the corresponding network demuxer. This disables packet reception for packets that match the filter. The delete request is passed on to the **nd_del_filter** function of the demuxer for the specified network device driver (NDD).

Examples

The following example illustrates the **ns_del_filter** network service:

```
struct ns_8022 dl;

dl.filtertype = NS_LLC_DSAP_SNAP;
dl.dsap = 0xaa;
dl.orgcode[0] = 0x0;
dl.orgcode[1] = 0x0;
dl.orgcode[2] = 0x0;
dl.ethertype = 0x0800;
ns_del_filter(nddp, &dl, sizeof(dl));
```

Return Values

0	Indicates the operation was successful.

The network demuxer may supply other return values.

Related Information

The **ns_add_filter** network service, **ns_alloc** network service.

ns_del_status Network Service

Purpose

Deletes a previously added status filter.

Syntax

```
#include <sys/cdli.h>
#include <sys/ndd.h>

int ns_del_status (nddp, statfilter, len)
        struct ndd *nddp;
        caddr_t statfilter;
        int len;
```

Parameters

nddp	Specifies the pointer to the **ndd** structure to which this delete request applies.
statfilter	Specifies the pointer to the status filter.
len	Specifies the length, in bytes, of the value of the *statfilter* parameter.

Description

The **ns_del_status** network service deletes a previously added status filter from the corresponding network demuxer. The delete request is passed on to the **nd_del_status** function of the demuxer for the specified network device driver (NDD). This network service disables asynchronous status notification from the specified device.

Examples

The following example illustrates the **ns_del_status** network service:

```
error = ns_add_status(nddp, &filter,
sizeof(filter));
```

Return Values

0	Indicates the operation was successful.

The network demuxer may supply other return values.

Related Information

The **ns_add_status** network service.

ns_detach Network Service

Purpose

Removes a network device from the network subsystem.

Syntax

```
#include <sys/ndd.h>

int ns_detach (nddp)
        struct ndd *nddp;
```

Parameters

nddp Specifies a pointer to an **ndd** structure describing the device to be detached.

Description

The **ns_detach** service removes the **ndd** structure from the chain of available NS devices.

Examples

The following example illustrates the **ns_detach** network service:

```
ns_detach(nddp);
```

Return Values

0	Indicates the operation was successful.
ENOENT	Indicates the specified *ndd* structure was not found.
EBUSY	Indicates the network device driver (NDD) is currently in use.

Related Information

The **ns_attach** network service.

ns_free Network Service

Purpose

Relinquishes access to a network device.

Syntax

```
#include <sys/ndd.h>

void ns_free (nddp)
       struct ndd *nddp;
```

Parameters

nddp Specifies the **ndd** structure of the network device that is to be freed from use.

Description

The **ns_free** network service relinquishes access to a network device. The **ns_free** network service also decrements the reference count for the specified **ndd** structure. If the reference count becomes 0, the **ns_free** network service calls the **ndd_close** subroutine specified in the **ndd** structure.

Examples

The following example illustrates the **ns_free** network service:

```
struct ndd *nddp
ns_free(nddp);
```

Files

net/cdli.c

Related Information

The **ns_alloc** network service.

panic Kernel Service

Purpose

Crashes the system.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

panic (s)
char *s;
```

Parameter

s	Points to a character string to be written to the error log.

Description

The **panic** kernel service is called when a catastrophic error occurs and the system can no longer continue to operate. The **panic** service performs these two actions:

- Writes the character string pointed to by the *s* parameter to the error log.
- Performs a system dump.

The system halts after the dump. You should wait for the dump to complete, reboot the system, and then save and analyze the dump.

Execution Environment

The **panic** kernel service can be called from either the process or interrupt environment.

Return Values

The **panic** kernel service has no return values.

Implementation Specifics

The **panic** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

RAS Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

pci_cfgrw Kernel Service

Purpose

Reads and writes PCI bus slot configuration registers.

Syntax

```
#include <sys/mdio.h>

int pci_cfgrw(bid, md, write_flag)
int bid;
struct mdio *md;
int write_flag;
```

Description

The **pci_cfgrw** kernel service provides serialized access to the configuration registers for a PCI bus. To ensure data integrity in a multi–processor environment, a lock is required before accessing the configuration registers. Depending on the value of the *write_flag* parameter, a read or write to the configuration register is performed at offset *md_addr* for the device identified by *md_sla*.

The **pci_cfgrw** kernel service provides for kernel extensions the same services as the **MIOPCFGET** and **MIOPCFPUT** ioctls provides for applications. The **pci_cfgrw** kernel service can be called from either the process or the interrupt environment.

Parameters

bid	Specifies the bus identifier.	
md	Specifies the address of the *mdio* structure. The *mdio* structure contains the following fields:	
	md_addr	Starting offset of the configuration register to access (**0** to **0xFF**).
	ms_data	Pointer to the data buffer.
	md_size	Number of items of size specified by the *md_incr* parameter. The maximum size is 256 bytes.
	md_incr	Access types, **MV_BYTE**, **MV_WORD**, or **MV_SHORT**.
	md_sla	Device Number and Function Number. (Device Number * 8) + Function.
write_flag	Set to **1** for write and **0** for read.	

Return Values

Returns 0 for successful completion.

ENOMEM	Indicates no memory could be allocated.
EINVAL	Indicated that the bus, device/function, or size is not valid.

Implementation Specifics

The **pci_cfgrw** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

Machine Device Driver in *AIX Technical Reference, Volume 6: Kernel and Subsystems*

pfctlinput Kernel Service

Purpose

Invokes the **ctlinput** function for each configured protocol.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/domain.h>

void pfctlinput (cmd, sa)
int cmd;
struct  sockaddr *sa;
```

Parameters

cmd	Specifies the command to pass on to protocols.
sa	Indicates the address of a **sockaddr** structure that is passed to the protocols.

Description

The **pfctlinput** kernel service searches through the protocol switch table of each configured domain and invokes the protocol **ctlinput** function if defined. Both the *cmd* and *sa* parameters are passed as parameters to the protocol function.

Execution Environment

The **pfctlinput** kernel service can be called from either the process or interrupt environment.

Return Values

The **pfctlinput** service has no return values.

Implementation Specifics

The **pfctlinput** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

pffindproto Kernel Service

Purpose

Returns the address of a protocol switch table entry.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/domain.h>

struct protosw *pffindproto (family, protocol, type)
int family;
int protocol;
int type;
```

Parameters

family	Specifies the address family for which to search.
protocol	Indicates the protocol within the address family.
type	Specifies the type of socket (for example, **SOCK_RAW**).

Description

The **pffindproto** kernel service first searches the domain switch table for the address family specified by the *family* parameter. If found, the **pffindproto** service then searches the protocol switch table for that domain and checks for matches with the *type* and *protocol* parameters.

If a match is found, the **pffindproto** service returns the address of the protocol switch table entry. If the *type* parameter is set to **SOCK_RAW**, the **pffindproto** service returns the first entry it finds with protocol equal to 0 and type equal to **SOCK_RAW**.

Execution Environment

The **pffindproto** kernel service can be called from either the process or interrupt environment.

Return Values

The **pffindproto** service returns a null value if a protocol switch table entry was not found for the given search criteria. Upon success, the **pffindproto** service returns the address of a protocol switch table entry.

Implementation Specifics

The **pffindproto** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

Understanding Socket Header Files in *AIX Communications Programming Concepts*.

pgsignal Kernel Service

Purpose

Sends a signal to all of the processes in a process group.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

void pgsignal (pid, sig)
pid_t pid;
int sig;
```

Parameters

pid Specifies the process ID of a process in the group of processes to receive the signal.

sig Specifies the signal to send.

Description

The **pgsignal** kernel service sends a signal to each member in the process group to which the process identified by the *pid* parameter belongs. The *pid* parameter must be the process identifier of the member of the process group to be sent the signal. The *sig* parameter specifies which signal to send.

Device drivers can get the value for the *pid* parameter by using the **getpid** kernel service. This value is the process identifier for the currently executing process.

The **sigaction** subroutine contains a list of the valid signals.

Execution Environment

The **pgsignal** kernel service can be called from either the process or interrupt environment.

Return Values

The **pgsignal** service has no return values.

Implementation Specifics

The **pgsignal** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **getpid** kernel service, **pidsig** kernel service.

The **sigaction** subroutine.

Process and Exception Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

pidsig Kernel Service

Purpose

Sends a signal to a process.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

void pidsig (pid, sig)
pid_t pid;
int sig;
```

Parameters

pid	Specifies the process ID of the receiving process.
sig	Specifies the signal to send.

Description

The **pidsig** kernel service sends a signal to a process. The *pid* parameter must be the process identifier of the process to be sent the signal. The *sig* parameter specifies the signal to send. See the **sigaction** subroutine for a list of the valid signals.

Device drivers can get the value for the *pid* parameter by using the **getpid** kernel service. This value is the process identifier for the currently executing process.

The **pidsig** kernel service can be called from an interrupt handler execution environment if the process ID is known.

Execution Environment

The **pidsig** kernel service can be called from either the process or interrupt environment.

Return Values

The **pidsig** service has no return values.

Implementation Specifics

The **pidsig** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **getpid** kernel service, **pgsignal** kernel service.

The **sigaction** subroutine.

Process and Exception Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

pin Kernel Service

Purpose

Pins the address range in the system (kernel) space.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/pin.h>

int pin (addr, length)
caddr_t  addr;
int length;
```

Parameters

addr	Specifies the address of the first byte to pin.
length	Specifies the number of bytes to pin.

Description

The **pin** service pins the real memory pages touched by the address range specified by the *addr* and *length* parameters in the system (kernel) address space. It pins the real–memory pages to ensure that page faults do not occur for memory references in this address range. The **pin** service increments the pin count for each real–memory page. While the pin count is nonzero, the page cannot be paged out of real memory.

The **pin** routine pins either the entire address range or none of it. Only a limited number of pages can be pinned in the system. If there are not enough unpinned pages in the system, the **pin** service returns an error code.

Note: If the requested range is not aligned on a page boundary, then memory outside this range is also pinned. This is because the operating system pins only whole pages at a time.

The **pin** service can only be called for addresses within the system (kernel) address space. The **pinu** service should be used for addresses within kernel or user space.

Execution Environment

The **pin** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
EINVAL	Indicates that the value of the *length* parameter is negative or 0. Otherwise, the area of memory beginning at the address of the first byte to pin (the *addr* parameter) and extending for the number of bytes specified by the *length* parameter is not defined.
EIO	Indicates that a permanent I/O error occurred while referencing data.
ENOMEM	Indicates that the **pin** service was unable to pin due to insufficient real memory or exceeding the systemwide pin count.
ENOSPC	Indicates insufficient file system or paging space.

Implementation Specifics

The **pin** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **unpin** kernel service.

Understanding Execution Environments and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

pincf Kernel Service

Purpose

Manages the list of free character buffers.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <cblock.h>

int pincf (delta)
int delta;
```

Parameter

delta Specifies the amount by which to change the number of free–pinned character buffers.

Description

The **pincf** service is used to control the size of the list of free–pinned character buffers. A positive value for the *delta* parameter increases the size of this list, while a negative value decreases the size.

All device drivers that use character blocks need to use the **pincf** service. These drivers must indicate with a positive delta value the maximum number of character blocks they expect to be using concurrently. Device drivers typically call this service with a positive value when the **ddopen** routine is called. They should call the **pincf** service with a negative value of the same amount when they no longer need the pinned character blocks. This occurs typically when the **ddclose** routine is called.

Execution Environment

The **pincf** kernel service can be called in the process environment only.

Return Values

The **pincf** service returns a value representing the amount by which the service changed the number of free–pinned character buffers.

Implementation Specifics

The **pincf** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **waitcfree** kernel service.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

pincode Kernel Service

Purpose

Pins the code and data associated with a loaded object module.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/pin.h>

int pincode (func)
int (*func) ();
```

Parameter

func Specifies an address used to determine the object module to be pinned. The address is typically that of a function exported by this object module.

Description

The **pincode** service uses the **pin** service to pin the specified object module. The loader entry for the object module is used to determine the size of both the code and data.

Execution Environment

The **pincode** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
EINVAL	Indicates that the *func* parameter is not a valid pointer to the function.
ENOMEM	Indicates that the **pincode** service was unable to pin the module due to insufficient real memory.

When an error occurs, the **pincode** service returns without pinning any pages.

Implementation Specifics

The **pincode** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **pin** kernel service.

Understanding Execution Environments and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

pinu Kernel Service

Purpose

Pins the specified address range in user or system memory.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/uio.h>

int pinu (base, len, segflg)
caddr_t  base;
int len;
short segflg;
```

Parameters

base	Specifies the address of the first byte to pin.
len	Indicates the number of bytes to pin.
segflg	Specifies whether the data to pin is in user space or system space. The values for this flag are defined in the /**usr**/**include**/**sys**/**uio.h** file. This value can be one of the following:

UIO_SYSSPACE	Indicates the region is mapped into the kernel address space.
UIO_USERSPACE	Indicates the region is mapped into the user address space.

Description

The **pinu** kernel service is used to pin pages backing a specified memory region which is defined in either system or user address space. Pinning a memory region prohibits the pager from stealing pages from the pages backing the pinned memory region. Once a memory region is pinned, accessing that region does not result in a page fault until the region is subsequently unpinned.

The **pinu** kernel service will not work on a mapped file.

If the caller has a valid cross–memory descriptor for the address range, the **xmempin** and **xmemunpin** kernel services can be used instead of **pinu** and **unpinu**, and result in less pathlength.

Execution Environment

The **pinu** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
EFAULT	Indicates that the memory region as specified by the *base* and *len* parameters is not within the address space specified by the *segflg* parameter.

EINVAL	Indicates that the value of the *length* parameter is negative or 0. Otherwise, the area of memory beginning at the byte specified by the *base* parameter and extending for the number of bytes specified by the *len* parameter is not defined.
ENOMEM	Indicates that the **pinu** service is unable to pin the region due to insufficient real memory or because it has exceeded the systemwide pin count.

Implementation Specifics

The **pinu** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **pin** kernel service, **unpinu** kernel service, **xmempin** kernel service, **xmemunpin** kernel service.

Understanding Execution Environments and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

pio_assist Kernel Service

Purpose

Provides a standardized programmed I/O exception handling mechanism for all routines performing programmed I/O.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int pio_assist (ioparms, iofunc, iorecov)
caddr_t ioparms;
int (*iofunc)( );
int (*iorecov)( );
```

Parameters

ioparms	Points to parameters for the I/O routine.
iofunc	Specifies the I/O routine function pointer.
iorecov	Specifies the I/O recovery routine function pointer.

Description

The **pio_assist** kernel service assists in handling exceptions caused by programmed I/O. Use of the **pio_assist** service standardizes the programmed I/O exception handling for all routines performing programmed I/O. The **pio_assist** service is built upon other kernel services that routines access to provide their own exception handling if the **pio_assist** service should not be used.

Using the pio_assist Kernel Service

To use the **pio_assist** service, the device handler writer must provide a callable routine that performs the I/O operation. The device handler writer can also optionally provide a routine that can recover and log I/O errors. The mainline device handler code would then call the **pio_assist** service with the following parameters:

- A pointer to the parameters needed by the I/O routine
- The function pointer for the routine performing I/O
- A pointer for the I/O recovery routine (or a null pointer, if there is no I/O recovery routine)

If the pointer for the I/O recovery routine is a null character, the *iofunc* routine is recalled to recover from I/O exceptions. The I/O routine for error retry should only be re–used if the I/O routine can handle being recalled when an error occurs, and if the sequence of I/O instructions can be reissued to recover from typical bus errors.

The *ioparms* parameter points to the parameters needed by the I/O routine. It is passed to the I/O routine when the **pio_assist** service calls the I/O routine. It is also passed to the I/O recovery routine when the I/O recovery routine is invoked by the **pio_assist** service. If any of the parameters found in the structure pointed to by the *ioparms* parameter are modified by the *iofunc* routine and needed by the *iorecov* or recalled *iofunc* routine, they must be declared as *volatile*.

Requirements for Coding the Caller–Provided I/O Routine

The *iofunc* parameter is a function pointer to the routine performing the actual I/O. It is called by the **pio_assist** service with the following parameters:

```
int iofunc (ioparms)
caddr_t ioparms;            /* pointer to parameters */
```

The *ioparms* parameter points to the parameters used by the I/O routine that was provided on the call to the **pio_assist** kernel service.

If the **pio_assist** kernel service is used with a null pointer to the *iorecov* I/O recovery routine, the *iofunc* I/O routine is called to retry all programmed I/O exceptions. This is useful for devices that have I/O operations that can be re–sent without concern for hardware state synchronization problems.

Upon return from the I/O, the return code should be 0 if no error was encountered by the I/O routine itself. If a nonzero return code is presented, it is used as the return code from the **pio_assist** kernel service.

Requirements for Coding the Caller–Provided I/O Recovery Routine

The *iorecov* parameter is a function pointer to the device handler's I/O recovery routine. This *iorecov* routine is responsible for logging error information, if required, and performing the necessary recovery operations to complete the I/O, if possible. This may in fact include calling the original I/O routine. The *iorecov* routine is called with the following parameters when an exception is detected during execution of the I/O routine:

```
int iorecov (parms, action, infop)
caddr_t parms;/* pointer to parameters passed to iofunc*/
int action;            /* action indicator */
struct pio_except *infop;        /* pointer to exception info */
```

The *parms* parameter points to the parameters used by the I/O routine that were provided on the call to the **pio_assist** service.

The *action* parameter is an operation code set by the **pio_assist** kernel service to one of the following:

PIO_RETRY	Log error and retry I/O operations, if possible.
PIO_NO_RETRY	Log error but do not retry the I/O operation.

The **pio_except** structure containing the exception information is platform–specific and defined in the /**usr**/**include**/**sys**/**except.h** file. The fields in this structure define the type of error that occurred, the bus address on which the error occurred, and additional platform–specific information to assist in the handling of the exception.

The *iorecov* routine should return with a return code of 0 if the exception is a type that the routine can handle. A **EXCEPT_NOT_HANDLED** return code signals that the exception is a type not handled by the *iorecov* routine. This return code causes the **pio_assist** kernel service to invoke the next exception handler on the stack of exception handlers. Any other nonzero return code signals that the *iorecov* routine handled the exception but could not successfully recover the I/O. This error code is returned as the return code from the **pio_assist** kernel service.

Return Codes by the pio_assist Kernel Service

The **pio_assist** kernel service returns a return code of 0 if the *iofunc* I/O routine does not indicate any errors, or if programmed I/O exceptions did occur but were successfully handled by the *iorecov* I/O recovery routine. If an I/O exception occurs during execution of the *iofunc* or *iorecov* routines and the exception count has not exceeded the maximum value, the *iorecov* routine is called with an *op* value of **PIO_RETRY**.

If the number of exceptions that occurred during this operation exceeds the maximum number of retries set by the platform–specific value of **PIO_RETRY_COUNT**, the **pio_assist** kernel service calls the *iorecov* routine with an *op* value of **PIO_NO_RETRY**. This indicates that the I/O operation should not be retried. In this case, the **pio_assist** service returns a return code value of **EIO** indicating failure of the I/O operation.

If the exception is not an I/O–related exception or if the *iorecov* routine returns with the return code of **EXCEPT_NOT_HANDLED** (indicating that it could not handle the exception), the **pio_assist** kernel service does not return to the caller. Instead, it invokes the next

exception handler on the stack of exception handlers for the current process or interrupt handler. If no other exception handlers are on the stack, the default exception handler is invoked. The normal action of the default exception handler is to cause a system crash.

Execution Environment

The **pio_assist** kernel service can be called from either the process or interrupt environment.

Return Values

0	Indicates that either no errors were encountered, or PIO errors were encountered and successfully handled.
EIO	Indicates that the I/O operation was unsuccessful because the maximum number of I/O retry operations was exceeded.

Implementation Specifics

The **pio_assist** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Kernel Extension and Device Driver Management Kernel Services, User–Mode Exception Handling, Kernel–Mode Exception Handling in *AIX Kernel Extensions and Device Support Programming Concepts*.

pm_planar_control Kernel Service

Purpose

Controls power of a specified device on the planar.

Syntax

```
#include <sys/pm.h>
int pm_planar_control(DevNumber,DevID,command)
dev_t DevNumber;
int DevID;
int command;
```

Parameters

DevNumber	Specifies the major minor number of the caller device. If caller is not a device, *DevNumber* is 0.
DevID	Specifies the planar device ID.
command	Specifies one of the following: **PM_PLANAR_QUERY** Queries supported commands of the kernel service. **PM_PLANAR_ON** Turns the device on. **PM_PLANAR_OFF** Turns the device off. **PM_PLANAR_LOWPOWER1** Brings the device into low power 1. **PM_PLANAR_LOWPOWER2** Brings the device into low power 2.

Description

The **pm_planar_control** kernel service turns a device on the planar either on or off.

Return Values

If *command* is **PM_PLANAR_QUERY**, OR of the following is returned:

PM_PLANAR_ON

PM_PLANAR_OFF

PM_PLANAR_LOWPOWER1

or **PM_PLANAR_LOWPOWER2**

If *command* is other than **PM_PLANAR_QUERY** the following is returned:

PM_SUCCESS	Indicates successful completion.
PM_ERROR	Indicates an error condition.

Execution Environment

The **pm_planar_control** kernel service can be called from either the process or interrupt environment.

Implementation Specifics

The **pm_planar_control** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **pm_control_parameter** subroutine, **pm_control_state** subroutine, **pm_event_query** subroutine, **pm_battery_control** subroutine.

The **pm_register_handle** kernel service, **pm_reqister_planar_contr ol_handle** kernel service.

pm_register_handle Kernel Service

Purpose

Registers and unregisters Power Management handle.

Syntax

```
#include <sys/pm.h>
int pm_register_handle(pmh, command)
struct pm_handle *pmh;
int command;
```

Parameters

pmh	Points to a **pm_handle** structure.
command	Specifies **PM_REGISTER** or **PM_UNREGISTER**.

Description

The **pm_register_handle** kernel service registers and unregisters a Power Management (PM) handle to a PM core. This kernel service need to be called from **config** entry point of each PM aware device driver.

Return Values

PM_SUCCESS	Indicates successful completion.
PM_ERROR	Indicates an error condition.

Implementation Specifics

The **pm_register_handle** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **pm_control_parameter** subroutine, **pm_control_state** subroutine, **pm_event_query** subroutine, **pm_battery_control** subroutine.

The **pm_register_planar_control_handle** kernel service, **pm_planar_control** kernel service.

pm_register_planar_control_handle Kernel Service

Purpose

Registers and unregisters a planar control subroutine.

Syntax

```
#include <sys/pm.h>
int pm_register_planar_control_handle(ppch, command)
struct pm_planar_control_handle *ppch;
int command;
```

Parameters

```
ppch        Points a
pm_planar_control_handle structure as follows.
            The structure and the control subroutine must be
pinned.

struct pm_planar_control_handle {
            int              DevID;
            int              (*control)(dev_t DevNumber, int
DevID,
int command);
             struct pm_planar_control_handle          *next;

command      Specifies one of the following:
             PM_REGISTER
             PM_UNREGISTER
```

Description

The **pm_register_planar_control_handle** kernel service registers and unregisters planar control subroutines.

The planar device ID, *DevID*, are stored in the ODM. The Power Management aware device driver retrieves the values at device configuration if planar_control operation is required to control the low power or off state of its own device on the planar as follows:

- Display indicator

LCD	PMDEV_LCD	0x00010000
CRT	PMDEV_CRT	0x00010100

- Video Controller

Graphic controller	PMDEV_GCC	0x00020000
DAC	PMDEV_DAC	0x00020100
VRAM	PMDEV_VRAM	0x00020200

- Multi media

Video capture	PMDEV_VCAP	0x00030000
Playback	PMDEV_VPLAY	0x00030100

CCD camera	PMDEV_CAMERA	0x00030200
Audio	PMDEV_AUDIO	0x00030300

- Graphical input

Internal keyboard	PMDEV_INTKBD	0x00040000
External keyboard	PMDEV_EXTKBD	0x00040100
Internal mouse	PMDEV_INTMOUSE	0x00040200
External mouse	PMDEV_EXTMOUSE	0x00040300

- Communication device

Serial	PMDEV_SERIAL	0x00050000
Parallell	PMDEV_PARALLEL	0x00050100

- SCSI adapter

SCSI controller	PMDEV_SCSIC	0x00060000
SCSI bus terminator	PMDEV_SCSIT	0x00060100

- Internal SCSI device

Device ID 0–6	PMDEV_SCSI*n*(*n*=0–6)	0x00070000–0x00070600

- Internal IDE device

Device ID 0–3	PMDEV_IDE*n*(*n*=0–3)	0x00080000–0x00080300

- CPU local bus

CPU	PMDEV_CPU	0x00090000
L2 cache	PMDEV_L2	0x00090100
Others	PMDEV_LOCAL*n(*n=2–f*)*	0x00090200–0x00090f00

- Extended bus slot

ISA bus slot 0–f	PMDEV_ISA*n(*n=0–f*)*	0x000a0000–0x000a000f
Micro Channel slot 0–f	PMDEV_MCA*n(*n=0–f*)*	0x000a0100–0x000a010f
PCI slot 0–f	PMDEV_PCI*n(*n=0–f*)*	0x000a0200–0x000a020f
PCMCIA slot 0–f	PMDEV_PCMCIA*n(*n=0–f*)*	0x000a0300–0x000a030f

- Power supply unit (PSU)

Suspend power	PMDEV_PSUSUS	0x000b0000
Main power	PMDEV_PSUMAIN	0x000b0100

- Others

FDD/FDC	PMDEV_FDD	0x000f0000
Internal CD–ROM	PMDEV_CDROM	0x000f0100

Return Values

PM_SUCCESS Indicates successful completion.

PM_ERROR Indicates an error condition.

Implementation Specifics

The **pm_register_planar_control_handle** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **pm_planar_control** kernel service.

probe or kprobe Kernel Service

Purpose

Logs errors with symptom strings.

Library (for probe)

Run–time Services Library.

Syntax

```
#include <sys/probe.h>
or
#include <sys/sysprobe.h>
int probe (probe_p)
probe_t *probe_p

int kprobe (probe_p)
probe_t *probe_p
```

Description

The probe subroutine logs an entry to the error log. The entry consists of an error log entry as defined in the **errlog** subroutine and the **err_rec.h** header file, and a symptom string.

The **probe** subroutine is called from an application, while **kprobe** is called from the Kernel and Kernel extensions. Both **probe** and **kprobe** have the same interfaces, except for return codes.

IBM software should use the **sys**/**sysprobe.h** header file while non–IBM programs should include the **sys**/**probe.h file**. This is because IBM symptom strings must conform to different rules than non–IBM strings. It also tells any electronic support application whether or not to route the symptom string to IBM's Retain database.

Parameters

probe_p is a pointer to the data structure which contains the pointer and length of the error record, and the data for the probe. The error record is described under the **errlog** subroutine and defined in **err_rec.h**.

The first word of the structure is a magic number to identify this version of the structure. The magic number should be set to PROBE_MAGIC.

Note: PROBE_MAGIC is different between **probe.h** and **sysprobe.h** to distinguish an IBM symptom string from a non–IBM string.

The probe data consists of flags which control probe handling, the number of symptom string keywords, followed by an array consisting of one element for each keyword.

Flags

SSNOSEND indicates this symptom string shouldn't be forwarded to automatic problem opening facilities. An example where **SSNOSEND** should be used is in symptom data used for debugging purposes.

nsskwd This gives the number of keywords specified (i.e.), the number of elements in the sskwds array .

sskwds This is an array of keyword/value pairs. The keywords and their values are in the following table. The **I/S** value indicates whether the *keyword* and *value* are informational or are part of the logged symptom string. The number in parenthesis indicates, where applicable, the maximum string length.

keyword	I/S	value	type	Description
SSKWD_LONGNAME	I	char *	(30)	Product's long name
SSKWD_OWNER	I	char *	(16)	Product's owner
SSKWD_PIDS	S	char *	(11)	product id.(required for IBM symptom strings)
SSKWD_LVLS	S	char *	(5)	product level (required for IBM symptom strings)
SSKWD_APPLID	I	char *	(8)	application id.
SSKWD_PCSS	S	char *	(8)	probe id (required for all symptom strings)
SSKWD_DESC	I	char *	(80)	problem description
SSKWD_SEV	I	int		severity from 1 (highest) to 4 (lowest). 3 is the default.
SSKWD_AB	S	char *	(5)	abend code
SSKWD_ADRS	S	void *		address. If used at all this should be a relative address.
SSKWD_DEVS	S	char *	(6)	Device type
SSKWD_FLDS	S	char *	(9)	arbitrary chararacter string. This is usually a field name and the SSKWD_VALUE keyword specifies the value.
SSKWD_MS	S	char *	(11)	Message number
SSKWD_OPCS	S	char *	(8)	OP code
SSKWD_OVS	S	char *	(9)	overwritten storage
SSKWD_PRCS	S			unsigned long return code
SSKWD_REGS	S	char *	(4)	Register name (e.g.) GR15 or LR unsigned long Value
SSKWD_VALU	S			
SSKWD_RIDS	S	char *	(8)	resource or module id.
SSKWD_SIG	S .	int		Signal number
SSKWD_SN	S	char *	(7)	Serial Number
SSKWD_SRN	S	char *	(9)	Service Req. Number If specified, and no error is logged, a hardware error is assumed.
SSKWD_WS	S	char *	(10)	Coded wait

Note: The **SSKWD_PCCS** value is always required. This is the probe id. Additionally, for IBM symptom strings, the **SSKWD_PIDS** and **SSKWD_LVLS** keywords are also required.

If either the **erecp** or **erecl** fields in the **probe_rec** structure is 0 then no error logging record is being passed, and one of the default templates for symptom strings is used. The default template indicating a software error is used unless the **SSKWD_SRN** keyword is specified. If it is, the error is assumed to be a hardware error. If you don't wish to log your own error with a symptom string, and you wish to have a hardware error, and don't want to use the **SSKWD_SRN** value, then you can supply an error log record using the error identifier of **ERRID_HARDWARE_SYMPTOM**, see the **/usr/include/sys/errids.h** file.

Return Values for probe Subroutine

0	Successful
–1	Error. The errno variable is set to
EINVAL	Indicates an invalid parameter
EFAULT	Indicates an invalid address

Return Values for kprobe Kernal Service

0	Successful
EINVAL	Indicates an invalid parameter

Implementation Specifics

These subroutines are part of AIX Base Operating System (BOS) Run Time.

Execution Environment

probe is executed from the application environment.

kprobe is executed from the Kernel and Kernel extensions. Currently, **kprobe** must not be called with interrupts disabled.

Files

/usr/include/sys/probe.h	Contains parameter definition.

Related Information

Error Logging Overview.

The **errlog** subroutines.

The **errsave** or **errlast** subroutines.

Process State–Change Notification Routine

Purpose

Allows kernel extensions to be notified of major process state transitions.

Syntax

```
void handler (term, type, pid)
struct proch *term;
int type;
pid_t pid;
```

Parameters

term	Points to a **proch** structure used in the **prochadd** call.
type	Defines the process's state transition: initialization, termination, swap in, or swap out. These four values, defined in the /**usr**/**include**/**sys**/**proc.h** file, are as follows:
	PROCH_INITIALIZE Process is initializing.
	PROCH_TERMINATE Process is terminating.
	PROCH_SWAPIN Process has been swapped in.
	PROCH_SWAPOUT Process is about to be swapped out.
pid	Defines the process ID of the process.

Description

For process initialization, the process state–change notification routine is called in the execution environment of a parent process for the initialization of a newly created child process. For kernel processes, the notification routine is called when the **initp** kernel service is called to complete initialization.

For process termination, the notification routines are called before the kernel handles default termination procedures. They are called in a LIFO order. The routines must be written so as not to allocate any resources under the terminating process. The notification routine is called under the process image of the terminating process.

The notification routine is activated for a swap in when a process has just been swapped in and is about to be put on the ready–to–run queue. At the point of call to the notification routine, the process's **u** block has been pinned.

The notification routine is activated for a swap out when a process is about to be swapped out. At the point of call to the notification routine, the process **u** block has not yet been unpinned.

Implementation Specifics

This routine is part of Base Operating System (BOS) Runtime.

Related Information

The **prochadd** kernel service, **prochdel** kernel service.

Kernel Extension and Device Driver Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

prochadd Kernel Service

Purpose

Adds a system–wide process state–change notification routine.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/proc.h>

void prochadd (term)
struct proch *term;
```

Parameters

term	Points to a **proch** structure containing a notification routine to be added from the chain of systemwide notification routines.

Description

The **prochadd** kernel service allows kernel extensions to register for notification of major process state transitions. The **prochadd** service allows the caller to be notified when a process:

- Has just been created.
- Is about to be terminated.
- Is executing a new program.

The complete list of callouts is:

PROCH_INITIALIZE	Process (pid) created (**initp**, **kforkx**)
PROCH_TERMINATE	Process (pid) terminated (**kexitx**)
PROCH_EXEC	Process (pid) executing (**execvex**)
THREAD_INITIALIZE	Thread (tid) created (**kforkx**, **thread_create**)
THREAD_TERMINATE	Thread (tid) created (**kexitx**, **thread_terminate**)

The **prochadd** service is typically used to allow recovery or reassignment of resources when processes undergo major state changes.

The caller should allocate a **proch** structure and update the `proch.handler` field with the entry point of a caller–supplied notification routine before calling the **prochadd** kernel service. This notification routine is called once for each process in the system undergoing a major state change.

The **proch** structure has the following form:

```
struct proch
{
        struct proch *next
        void                    *handler ();
}
```

Execution Environment

The **prochadd** kernel service can be called from the process environment only.

Implementation Specifics

The **prochadd** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **prochdel** kernel service.

The Process State–Change Notification Routine.

Kernel Extension and Driver Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

prochdel Kernel Service

Purpose

Deletes a process state change notification routine.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/proc.h>

void prochdel (term)
struct proch *term;
```

Parameter

term	Points to a **proch** structure containing a notification routine to be removed from the chain of system–wide notification routines. This structure was previously registered by using the **prochadd** kernel service.

Description

The **prochdel** kernel service removes a process change notification routine from the chain of system–wide notification routines. The registered notification routine defined by the handler field in the **proch** structure is no longer to be called by the kernel when major process state changes occur.

If the **proch** structure pointed to by the *term* parameter is not found in the chain of structures, the **prochdel** service performs no operation.

Execution Environment

The **prochdel** kernel service can be called from the process environment only.

Implementation Specifics

The **prochdel** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **prochadd** kernel service.

The Process State–Change Notification Routine.

Kernel Extension and Driver Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

purblk Kernel Service

Purpose

Purges the specified block from the buffer cache.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/buf.h>

void purblk (dev, blkno)
dev_t dev;
daddr_t blkno;
```

Parameters

dev	Specifies the device containing the block to be purged.
blkno	Specifies the block to be purged.

Description

The **purblk** kernel service purges (that is, makes unreclaimable by marking the block with a value of **STALE**) the specified block from the buffer cache.

Execution Environment

The **purblk** kernel service can be called from the process environment only.

Return Values

The **purblk** service has no return values.

Implementation Specifics

The **purblk** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **brelse** kernel service, **geteblk** kernel service.

Block I/O Buffer Cache Kernel Services: Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

putc Kernel Service

Purpose

Places a character at the end of a character list.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <cblock.h>

int putc (c, header)
char c;
struct clist *header;
```

Parameters

c	Specifies the character to place on the character list.
header	Specifies the address of the **clist** structure that describes the character list.

Description

Attention: The caller of the **putc** service must ensure that the character list is pinned. This includes the **clist** header and all the **cblock** character buffers. Character blocks acquired from the **getcf** service are also pinned. Otherwise, the system may crash.

The **putc** kernel service puts the character specified by the *c* parameter at the end of the character list pointed to by the *header* parameter.

If the **putc** service indicates that there are no more buffers available, the **waitcfree** service can be used to wait until a character block is available.

Execution Environment

The **putc** kernel service can be called from either the process or interrupt environment.

Return Values

0	Indicates successful completion.
–1	Indicates that the character list is full and no more buffers are available.

Implementation Specifics

The **putc** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **getcb** kernel service, **getcf** kernel service, **pincf** kernel service, **putcf** kernel service, **putcfl** kernel service, **waitcfree** kernel service.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

putcb Kernel Service

Purpose

Places a character buffer at the end of a character list.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <cblock.h>

void putcb (p, header)
struct cblock *p;
struct clist *header;
```

Parameters

p	Specifies the address of the character buffer to place on the character list.
header	Specifies the address of the **clist** structure that describes the character list.

Description

Attention: The caller of the **putcb** service must ensure that the character list is pinned. This includes the **clist** header and all the **cblock** character buffers. Character blocks acquired from the **getcf** service are pinned. Otherwise, the system may crash.

The **putcb** kernel service places the character buffer pointed to by the *p* parameter on the end of the character list specified by the *header* parameter. Before calling the **putcb** service, you must load this new buffer with characters and set the `c_first` and `c_last` fields in the **cblock** structure. The *p* parameter is the address returned by either the **getcf** or the **getcb** service.

Execution Environment

The **putcb** kernel service can be called from either the process or interrupt environment.

Return Values

0	Indicates successful completion.
–1	Indicates that the character list is full and no more buffers are available.

Implementation Specifics

The **putcb** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **getcb** kernel service, **getcf** kernel service, **pincf** kernel service, **putcf** kernel service, **putcfl** kernel service, **waitcfree** kernel service.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

putcbp Kernel Service

Purpose

Places several characters at the end of a character list.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <cblock.h>

int putcbp (header, source, n)
struct clist *header;
char *source;
int n;
```

Parameters

header	Specifies the address of the **clist** structure that describes the character list.
source	Specifies the address from which characters are read to be placed on the character list.
n	Specifies the number of characters to be placed on the character list.

Description

Attention: The caller of the **putcbp** service must ensure that the character list is pinned. This includes the **clist** header and all of the **cblock** character buffers. Character blocks acquired from the **getcf** service are pinned. Otherwise, the system may crash.

The **putcbp** kernel service operates on the characters specified by the *n* parameter starting at the address pointed to by the *source* parameter. This service places these characters at the end of the character list pointed to by the *header* parameter. The **putcbp** service then returns the number of characters added to the character list. If the character list is full and no more buffers are available, the **putcbp** service returns a 0. Otherwise, it returns the number of characters written.

Execution Environment

The **putcbp** kernel service can be called from either the process or interrupt environment.

Return Values

The **putcbp** service returns the number of characters written or a value of 0 if the character list is full, and no more buffers are available.

Implementation Specifics

The **putcbp** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **getcb** kernel service, **getcf** kernel service, **pincf** kernel service, **putcf** kernel service, **putcfl** kernel service, **waitcfree** kernel service.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

putcf Kernel Service

Purpose

Frees a specified buffer.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <cblock.h>

void putcf (p)
struct cblock *p;
```

Parameter

p	Identifies which character buffer to free.

Description

The **putcf** kernel service unpins the indicated character buffer.

The **putcf** service returns the specified buffer to the list of free character buffers.

Execution Environment

The **putcf** kernel service can be called from either the process or interrupt environment.

Return Values

The **putcf** service has no return values.

Implementation Specifics

The **putcf** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

putcfl Kernel Service

Purpose

Frees the specified list of buffers.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <cblock.h>

void putcfl (header)
struct clist *header;
```

Parameter

header	Identifies which list of character buffers to free.

Description

The **putcfl** kernel service returns the specified list of buffers to the list of free character buffers. The **putcfl** service unpins the indicated character buffer.

Note: The caller of the **putcfl** service must ensure that the header and **clist** structure are pinned.

Execution Environment

The **putcfl** kernel service can be called from either the process or interrupt environment.

Return Values

The **putcfl** service has no return values.

Implementation Specifics

The **putcfl** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

putcx Kernel Service

Purpose

Places a character on a character list.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/cblock.h>

int putcx (c, header)
char c;
struct clist *header;
```

Parameters

c	Specifies the character to place at the front of the character list.
header	Specifies the address of the **clist** structure that describes the character list.

Description

The **putcx** kernel service puts the character specified by the *c* parameter at the front of the character list pointed to by the *header* parameter. The **putcx** service is identical to the **putc** service, except that it puts the character at the front of the list instead of at the end.

If the **putcx** service indicates that there are no more buffers available, the **waitcfree** service can be used to wait until a character buffer is available.

Note: The caller of the **putcx** service must ensure that the character list is pinned. This includes the **clist** header and all the **cblock** character buffers. Character blocks acquired from the **getcf** service are pinned.

Execution Environment

The **putcx** kernel service can be called from either the process or interrupt environment.

Return Values

0	Indicates successful completion.
–1	Indicates that the character list is full and no more buffers are available.

Implementation Specifics

The **putcx** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **getcb** kernel service, **getcf** kernel service, **pincf** kernel service, **putcf** kernel service, **putcfl** kernel service, **waitcfree** kernel service.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

raw_input Kernel Service

Purpose

Builds a **raw_header** structure for a packet and sends both to the raw protocol handler.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

void raw_input (m0, proto, src, dst)
struct  mbuf *m0;
struct  sockproto *proto;
struct  sockaddr *src;
struct  sockaddr *dst;
```

Parameters

m0	Specifies the address of an **mbuf** structure containing input data.
proto	Specifies the protocol definition of data.
src	Identifies the **sockaddr** structure indicating where data is from.
dst	Identifies the **sockaddr** structure indicating the destination of the data.

Description

The **raw_input** kernel service accepts an input packet, builds a **raw_header** structure (as defined in the /**usr**/**include**/**net**/**raw_cb.h** file), and passes both on to the raw protocol input handler.

Execution Environment

The **raw_input** kernel service can be called from either the process or interrupt environment.

Return Values

The **raw_input** service has no return values.

Implementation Specifics

The **raw_input** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

raw_usrreq Kernel Service

Purpose

Implements user requests for raw protocols.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

void raw_usrreq (so, req, m, nam, control)
struct  socket *so;
int req;
struct  mbuf *m;
struct  mbuf *nam;
struct  mbuf *control;
```

Parameters

so	Identifies the address of a raw socket.
req	Specifies the request command.
m	Specifies the address of an **mbuf** structure containing data.
nam	Specifies the address of an **mbuf** structure containing the **sockaddr** structure.
control	This parameter should be set to a null value.

Description

The **raw_usrreq** kernel service implements user requests for the raw protocol.

The **raw_usrreq** service supports the following commands:

PRU_ABORT	Aborts (fast DISCONNECT, DETACH).
PRU_ACCEPT	Accepts connection from peer.
PRU_ATTACH	Attaches protocol to up.
PRU_BIND	Binds socket to address.
PRU_CONNECT	Establishes connection to peer.
PRU_CONNECT2	Connects two sockets.
PRU_CONTROL	Controls operations on protocol.
PRU_DETACH	Detaches protocol from up.
PRU_DISCONNECT	Disconnects from peer.
PRU_LISTEN	Listens for connection.
PRU_PEERADDR	Fetches peer's address.
PRU_RCVD	Have taken data; more room now.
PRU_RCVOOB	Retrieves out of band data.
PRU_SEND	Sends this data.
PRU_SENDOOB	Sends out of band data.
PRU_SENSE	Returns status into m.
PRU_SOCKADDR	Fetches socket's address.
PRU_SHUTDOWN	Will not send any more data.

Any unrecognized command causes the **panic** kernel service to be called.

Execution Environment

The **raw_userreq** kernel service can be called from either the process or interrupt environment.

Return Values

EOPNOTSUPP	Indicates an unsupported command.
EINVAL	Indicates a parameter error.
EACCESS	Indicates insufficient authority to support the **PRU_ATTACH** command.
ENOTCONN	Indicates an attempt to detach when not attached.
EISCONN	Indicates that the caller tried to connect while already connected.

Implementation Specifics

The **raw_userreq** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **panic** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

remap_64 Kernel Service

Purpose

Registers the input remapping of one or more addresses for the duration of a system call for a 64–bit process.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/adspace.h>
#include <sys/remap.h>

int remap_64 (rh)
remap_handle rh;
```

Parameter

rh	Specifies the **remap_handle** structure containing a single 64–bit remapping, two 64–bit remappings, or a pointer to an array of remap structures (**kremap** structure).

Description

The **remap_64** service will register an input remapping of one or more 64–bit user addresses as specified by the input **remap** structure.

The remapping process involves creating a 32–bit name, or alias for each corresponding 64–bit address so that the kernel code, which is 32–bit, can reference user–mode data for 64–bit applications. The **__remap** subroutine creates the remappings. The **remap_64** subroutine registers them with the kernel, so they will be honored when used.

The return value from the subroutine **__remap** is what gets passed down as input to this routine. The size of this structure depends on the number of parameters passed on the system call. The REMAP_HANDLEx macro is used on the library side to call the kernel extension with x parameters following the **remap_handle**. This macro ensures that the structure is the correct size and is correctly split into 32 bit registers for the kernel. The R_HANDLEx macro is used on the kernel extension side to declare the correct size **remap_handle** structure. This structure is passed on to **remap_64**. Therefore, the last half of the **remap_handle** structure passed to **remap_64** may be invalid. The **handle_type** field in the first half of the structure (filled in by **__remap**) indicates how much of the structure is valid.

If the **handle_type** field indicates R_ONE_REMAP, R_TWO_REMAP, or R_NO_REMAP, the remapping(s) is/are passed in–line in the **remap_handle** and there is no **copyin64** of a **kremap** structure required. If the **handle_type** field indicates R_N_REMAP, the remappings could not be passed in–line, and a **copyin64** of the **kremap** structure is required.

This kernel service must be called, in kernel mode, only when the current user process is 64–bits. It does not work for 32–bit user processes.

Execution Environment

The **remap_64** kernel service can be called from the process environment only.

Return Values

0	Successful completion.
–1	Indicates an error occurred while accessing the **kremap** struct. For example, the user has insufficient authority to access the data (or) **remap_64** has already been called on this system call. errno is set to EFAULT.
–1	Indicates that the remap struct was invalid (or) current user process not 64–bits. errno is set to EINVAL.

Implementation Specifics

The **remap_64** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **as_remap64** kernel service, **as_unremap64** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

rmalloc Kernel Service

Purpose

Allocates an area of memory.

Syntax

```
#include <sys/types.h>

caddr_t rmalloc (size, align)
int size
int align
```

Parameters

size	Specifies the number of bytes to allocate.
align	Specifies alignment characteristics.

Description

The **rmalloc** kernel service allocates an area of memory from the contiguous real memory heap. This area is the number of bytes in length specified by the *size* parameter and is aligned on the byte boundary specified by the *align* parameter. The *align* parameter is actually the log base 2 of the desired address boundary. For example, an *align* value of 4 requests that the allocated area be aligned on a 16–byte boundary.

The contiguous real memory heap, **real_heap**, is a heap of contiguous real memory pages located in the low 16MB of real memory. This heap is mapped virtually into the kernel extension segment. By nature, this heap is implicitly pinned, so no explicit pinning of allocated regions is necessary. This heap is provided primarily for device drivers whose devices can only address 0 to 16MB of real memory, so they must ”bounce” the I/O in and out of a buffer **rmalloc**’ed from this heap. Also, this heap is useful for devices that require greater than 4KB transfers, but do not support scatter/gather.

The **real_heap** is only supported on platforms with an ISA bus. On unsupported platforms, the **rmalloc** service returns **NULL** if the requested memory cannot be allocated.

The **rmfree** kernel service should be called to free allocation from a previous **rmalloc** call. The **rmalloc** kernel service can be called from the process environment only.

Return Values

Upon successful completion, the **rmalloc** kernel service returns the address of the allocated area. A **NULL** pointer is returned if the requested memory cannot be allocated.

Implementation Specifics

The **rmalloc** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **rmfree** kernel service.

rmfree Kernel Service

Purpose

Frees memory allocated by the **rmalloc** kernel service.

Syntax

```
#include <sys/types.h>

int rmfree (pointer, size)
caddr_t pointer
int size
```

Parameters

pointer	Specifies the address of the area in memory to free.
size	Specifies the size of the area in memory to free.

Description

The **rmfree** kernel service frees the area of memory pointed to by the *pointer* parameter in the contiguous real memory heap. This area of memory must be allocated with the **rmalloc** kernel service, and the *pointer* must be the pointer returned from the corresponding **rmalloc** kernel service call. Also, the *size* must be the same size that was used on the corresponding **rmalloc** call.

Any memory allocated in a prior **rmalloc** call must be explicitly freed with an **rmfree** call. This service can be called from the process environment only.

Return Values

0	Indicates successful completion.
–1	Indicates one of the following: • The area was not allocated by the **rmalloc** kernel service. • The heap was not initialized for memory allocation.

Implementation Specifics

The **rmfree** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **rmalloc** kernel service.

rmmap_create Kernel Service

Purpose

Defines an Effective Address [EA] to Real Address [RA] translation region.

Syntax

```
#include <sys/ioacc.h>
#include <sys/adspace.h>

int rmmap_create (eaddrp, iomp, flags)
void **eaddrp;
struct io_map *iomp;
int flags;
```

Parameters

eaddr Desired process effective address of the mapping region.

iomp The bus memory to which the effective address described by the *eaddr* parameter should correspond. For real memory, the bus id should be set to **REALMEM_BID** and the bus address should be set to the real memory address. The size field must be at least **PAGESIZE**, no larger than **SEGSIZE**, and a multiple of **PAGESIZE**. The key should be set to **IO_MEM_MAP**. The flags field is not used.

flags The flags select page and segment attributes of the translation.

flags

The flags select page and segment attributes of the translation. Not all page attribute flags are compatible. See below for the valid combinations of page attribute flags.

RMMAP_PAGE_W
PowerPC ”Write Through” page attribute. Valid with all other flags. If set, page operates write–through. If clear, operates write–back.

RMMAP_PAGE_W
PowerPC ”Write Through” page attribute. Write–through mode is not supported, and if this flag is set, **EINVAL** is reported.

RMMAP_PAGE_I
PowerPC ”Cache Inhibited” page attribute. Valid with all other flags. If set, page operates cache inhibited. If clear, page is considered cacheable.

RMMAP_PAGE_I
PowerPC ”Cache Inhibited” page attribute. This flag is valid for I/O mappings, but is not allowed for real memory mappings.

RMMAP_PAGE_M
PowerPC ”Memory Coherency Required” page attribute. Valid with all other flags. If set, accesses to a location are serialized within the processor complex. Otherwise, there is no guaranteed ordering. The default operating mode of AIX for real memory pages has this bit set.

RMMAP_PAGE_G
PowerPC ”Guarded” page attribute. This flag is optional for I/O mappings, and must be 0 for real memory mappings. Note that although optional for I/O, it is strongly recommended that this be set for I/O mappings. When set, the processor will not make unnecessary (speculative) references to the page. This includes out of order read/write operations and branch fetching. When clear, normal PowerPC speculative execution rules apply. This bit does not exist on the 601 microprocessor and is ignored.

RMMAP_RDONLY
When set, the page protection bits used in the **HTAB** will not allow write operations regardless of the setting of the key bit in the associated segment register. Exactly one of **RMMAP_RDONLY** and **RMMAP_RDWR** must be specified.

RMMAP_RDWR
When set, the page protection bits used in the **HTAB** will allow read and write operations regardless of the setting of the key bit in the associated segment register. Exactly one of: **RMMAP_RDONLY**, and **RMMAP_RDWR** must be specified.

RMMAP_PRELOAD
When set, the protection attributes of this region will be entered immediately into the hardware page table. This is very slow initially, but prevents each referenced page in the region from faulting in separately. This is only advisory. The **rmmap_create64** reserves the right to preload regions which do not specify this flag and to ignore the flag on regions which do. This flag is not maintained as an attribute of the map region, it is used only during the current call.

RMMAP_INHERIT
When set, this specifies that the translation region created by this **rmmap_create** invocation should be inherited on a **fork** operation, to the child process. This inheritance is achieved with copy–semantics. That is to say that the child will have its own private mapping to the same I/O or real memory address range as the parent.

Description

The translation regions created with **rmmap_create** kernel service are maintained in I/O mapping segments. Any single such segment may translate up to 256 Megabytes of I/O mapped memory in a single region. The only granularity for which the **rmmap_remove** service may be invoked is a single mapping created by a single call to the **rmmap_create**kernel service.

The translation regions created with **rmmap_create** kernel service are maintained in I/O mapping segments. Any single such segment may translate up to 256 Megabytes of real memory or memory mapped I/O in a single region. The only granularity for which the **rmmap_remove** service may be invoked is a single mapping created by a single call to the **rmmap_create**.

There are constraints on the size of the mapping and the *flags* parameter, described later, which will cause the call to fail regardless of whether adequate effective address space exists.

If **rmmap_create** kernel service is called with the effective address of zero (0), the function will attempt to find free space in the process address space. If successful, the effective

address (which is passed by reference) is changed to the effective address which is mapped to the first page of the *iomp* memory.

If **rmmap_create** kernel service is called with the effective address of zero (0), the function attempts to find free space in the process address space. If successful, an I/O mapping segment is created and the effective address (which is passed by reference) is changed to the effective address which is mapped to the first page of the *iomp* memory.

If **rmmap_create** kernel service is called with a non–zero effective address, it is taken as the desired effective address that should translate to the passed *iomp* memory. This function verifies that the region identified by the effective address is free. If not, it fails and returns **EINVAL**. If the mapping at the effective address is not contained in a single segment, the function fails and returns **ENOSPC**. Otherwise, the region is allocated and the effective address is not modified. The effective address is mapped to the first page of the *iomp* memory. References outside of the mapped regions but within the same segment are invalid.

If **rmmap_create** kernel service is called with a non–zero effective address, it is taken as the desired effective address which should translate to the passed *iomp* memory. This function verifies that the requested range is free. If not, it fails and returns **EINVAL**. If the mapping at the effective address is not contained in a single segment, the function fails and returns **ENOSPC**. Otherwise, the region is allocated and the effective address is not modified. The effective address is mapped to the first page of the *iomp* memory. References outside of the mapped regions but within the same segment are invalid.

The effective address (if provided) and the bus address must be a multiple of **PAGESIZE** or **EINVAL** is returned.

If **rmmap_create** kernel service is called with a length which is either not a multiple of **PAGESIZE**, is less than **PAGESIZE**, or is greater than **SEGSIZE**, **EINVAL** is returned. This return code takes precedence in cases where otherwise the segment would overflow and **ENOSPC** is returned.

I/O mapping segments are not inherited by child processes after a **fork** subroutine.

I/O mapping segments are not inherited by child processes after a **fork** subroutine, except when **RMMAP_INHERIT** is specified. These segments are deleted by **exec**, **exit**, or **rmmap_remove** of the last range in a segment. Only certain combinations of flags are permitted, depending on the type of memory being mapped. For real memory mappings, **RMMAP_PAGE_M** is required while **RMMAP_PAGE_W**, **RMMAP_PAGE_I**, and **RMMAP_PAGE_G** are not allowed. For I/O mappings, it is valid to specify only **RMMAP_PAGE_M**, with no other page attribute flags. It is also valid to specify **RMMAP_PAGE_I** and optionally, either or both of **RMMAP_PAGE_M**, and **RMMAP_PAGE_G**. **RMMAP_PAGE_W** is never allowed.

Execution Environment

The **rmmap_create** kernel service can only be called from the process environment.

Return Values

On successful completion, **rmmap_create** kernel service returns zero and modifies the effective address to the value at which the newly created mapping region was attached to the process address space. Otherwise, it returns one of:

EINVAL	Some type of parameter error occurred. These include, but are not limited to, size errors and mutually exclusive flag selections.
ENOMEM	The operating system could not allocate the necessary data structures to represent the mapping.

ENOSPC	Effective address space exhausted in the region indicated by *eaddr*.
EPERM	This hardware platform does not implement this service.

Implementation Specifics

This service must be called from the process level.

The **rmmap.create** kernel service is part of the Base Operating System (BOS) Runtime.

This service only functions on PowerPC microprocessors.

The real address range described by the *iomp* parameter must be unique within this I/O mapping segment.

Related Information

The **rmmap_remove** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

rmmap_create64 Kernel Service

Purpose

Defines an Effective Address [EA] to Real Address [RA] translation region for either 64–bit or 32–bit Effective Addresses.

Syntax

```
#include <sys/ioacc.h>
#include <sys/adspace.h>

int rmmap_create64(eaddrp, iomp, flags )
unsigned long long *eaddrp;
 struct io_map *iomp;
int flags;
```

Parameters

eaddrp Desired process effective address of the mapping region. This address is interpreted as a 64–bit quantity if the current user address space is 64–bits, and is interpreted as a 32–bit (not remapped) quantity if the current user address space is 32–bits.

iomp The bus memory to which the effective address described by the **eaddr** parameter should correspond. The size field must be at least **PAGESIZE**, no larger than **SEGSIZE**, and a multiple of **PAGESIZE**. The key should be set to **IO_MEM_MAP**. The flags field is not used.

iomp The bus memory to which the effective address described by the **eaddr** parameter should correspond. For real memory, the bus id should be set to **REALMEM_BID** and the bus address should be set to the real memory addres. The size field must be at least **PAGESIZE**, no larger than **SEGSIZE**, and a multiple of **PAGESIZE**. The key should be set to **IO_MEM_MAP**. The flags field is not used.

flags

The flags select page and segment attributes of the translation. Not all page attribute flags are compatible. See below for the valid combination of page attribute flags.

RMMAP_PAGE_W
PowerPC ”Write Through” page attribute. Valid with all other flags. If set, page operates write–through. If clear, operates write–back.

RMMAP_PAGE_W
PowerPC ”Write Through” page attribute. Write–through mode is not supported, and if this flag is set, **EINVAL** will be reported.

RMMAP_PAGE_I
PowerPC ”Cache Inhibited” page attribute. Valid with all other flags. If set, page operates cache inhibited. If clear, page is considered cacheable.

RMMAP_PAGE_I
PowerPC ”Cache Inhibited” page attribute. This flag is valid for I/O mappings, but is not allowed for real memory mappings.

RMMAP_PAGE_M
PowerPC ”Memory Coherency Required” page attribute. Valid with all other flags. If set, accesses to a location are serialized within the processor complex. Otherwise, there is no guaranteed ordering. The default operating mode of AIX for real memory pages has this bit set.

RMMAP_PAGE_M
PowerPC ”Memory Coherency Required” page attribute. This flag is optional for I/O mappings, however, it is required for memory mappings. The default operating mode of AIX for real memory pages has this bit set.

RMMAP_PAGE_G
PowerPC ”Guarded” page attribute. Valid with all other flags. When set, the processor will not make unnecessary (speculative) references to the page. This includes out of order read/write operations and branch fetching. When clear, normal PowerPC speculative execution rules apply. This bit does not exist on the 601 microprocessor and is ignored.

RMMAP_PAGE_G
PowerPC ”Guarded” page attribute. This flag is optional for I/O mappings, and must be 0 for real memory mappings. Note that although optional for I/O, it is strongly recommended that this be set for I/O mappings. When set, the processor will not make unnecessary (speculative) references to the page. This includes out of order read/write operations and branch fetching. When clear, normal PowerPC speculative execution rules apply. This bit does not exist on the 601 microprocessor and is ignored.

RMMAP_RDONLY
When set, the page protection bits used in the **HTAB** will not allow write operations regardless of the setting of the key bit in the associated segment register. Exactly one of: **RMMAP_RDONLY**, and **RMMAP_RDWR** must be specified.

RMMAP_RDWR
When set, the page protection bits used in the **HTAB** will allow read and write operations regardless of the setting of the key bit in the associated segment register. Exactly one of: **RMMAP_RDONLY**, and **RMMAP_RDWR** must be specified.

RMMAP_PRELOAD
When set, the protection attributes of this region will be entered immediately into the hardware page table. This is very slow initially, but prevents each referenced page in the region from faulting in separately. This is only advisory. The **rmmap_create64** reserves the right to preload regions which do not specify this flag and to ignore the flag on regions which do. This flag is not maintained as an attribute of the map region, it is used only during the current call.

RMMAP_INHERIT
When set, this specifies that the translation region created by this **rmmap_create64** invocation should be inherited on a **fork** operation, to the child process. This inheritance is achieved with copy–semantics. That is to say that the child has its own private mapping to the same I/O or real memory address range as the parent.

Description

The translation regions created with the **rmmap_create64** kernel service are maintained in I/O mapping segments. Any single such segment may translate up to 256 Megabytes of memory mapped I/O in a single region. The only granularity for which the **rmmap_remove64** service may be invoked is a single mapping created by a single call to **rmmap_create64**.

There are constraints on the size of the mapping and the flags parameter, described later, which will cause the call to fail regardless of whether adequate effective address space exists.

If the **rmmap_create64** kernel service is called with the effective address of zero (0), the function will attempt to find free space in the process address space. If successful, an I/O mapping segment is created and the effective address (which is passed by reference) is changed to the effective address that is mapped to the first page of the iomp memory.

If **rmmap_create64** kernel service is called with a non–zero effective address, it is taken as the desired effective address that should translate to the passed iomp memory. This function verifies that the requested range is free. If not, it fails and returns **EINVAL**. If the mapping at the effective address is not contained in a single segment, the function fails and returns **ENOSPC**. Otherwise, the region is allocated and the effective address is not modified. The effective address is mapped to the first page of iomp memory. References outside of the mapped regions but within the same segment are invalid.

The effective address (if provided) and the bus address (or real address for real memory mappings) must be a multiple of **PAGESIZE** or **EINVAL** is returned.

If the **rmmap_create64** kernel service is called with a length which is either not a multiple of **PAGESIZE**, is less than **PAGESIZE**, or is greater than **SEGSIZE**, **EINVAL** is returned. This return code takes precedence in cases where otherwise the segment would overflow and **ENOSPC** is returned.

I/O mapping segments are not inherited by child processes after a **fork** subroutine except when **RMMAP_INHERIT** is specified. These segments are deleted by **exec**,**exit**, or **rmmap_remove64** of the last range in a segment.

Only certain combinations of page flags are permitted, depending on the type of memory being mapped. For real memory mappings, **RMMAP_PAGE_M** is required while **RMMAP_PAGE_W**, **RMMAP_PAGE_I**, and **RMMAP_PAGE_G** are not allowed. For I/O mappings, it is valid to specify only **RMMAP_PAGE_M**, with no other page attribute flags. It is also valid to specify **RMMAP_PAGE_I** and optionally, either or both of the **RMMAP_PAGE_M**, and **RMMAP_PAGE_G**. **RMMAP_PAGE_W** is never allowed.

Execution Environment

The **rmmap_create64** kernel service can be called from the process environment only.

Return Values

On successful completion, the **rmmap_create64** kernel service returns zero and modifies the effective address to the value at which the newly created mapping region was attached to the process address space. Otherwise, it returns one of:

EINVAL	Some type of parameter error occured. These include, but are not limited to, size errors and mutually exclusive flag selections.
ENOMEM	The operating system could not allocate the necessary data structures to represent the mapping.
ENOSPC	Effective address space exhausted in the region indicated by **eaddr**.
EPERM	This hardware platform does not implement this service.

Implementation Specifics

The **rmmap_create64** kernel service is part of Base Operating System (BOS) Runtime.

This service only functions on PowerPC microprocessors.

The real address range described by the **iomp** parameter must be unique within this I/O mapping segment.

Related Information

The **rmmap_remove64** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

rmmap_getwimg Kernel Service

Purpose

Returns wimg information about a particular effective address range within an effective address to real address translation region.

Syntax

```
#include <sys/adspace.h>



int rmmap_getwimg(eaddr, npages, results)
unsigned long long eaddr;
unsigned int  npages;
char* results;
```

Parameters

eaddr	The process effective address of the start of the desired mapping region. This address should point somewhere inside the first page of the range. This address is interpreted as a 64–bit quantity if the current user address space is 64–bits, and is interpreted as a 32–bit (not remapped) quantity if the current user address space is 32–bits.
npages	The number of pages whose wimg information is returned, starting from the page indicated by **eaddr**.
results	This is an array of bytes, where the wimg information is returned. The address of this is passed in by the caller, and **rmmap_getwimg** stores the wimg information for each page in the range in each successive byte in this array. The size of this array is indicated by *npages* as specified by the caller. The caller is responsible for ensuring that the storage allocated for this array is large enough to hold *npage* bytes.

Description

The wimg information corresponding to the input effective address range is returned.

This routine only works for regions previously mapped with an I/O mapping segment as created by **rmmap_create64** or **rmmap_create**.

npages should not be such that the range crosses a segment boundary. If it does, EINVAL is returned.

The wimg information is returned in the **results** array. Each element of the **results** array is a character. Each character may be added with the following fields to examine wimg information: **RMMAP_PAGE_W**, **RMMAP_PAGE_I**, **RMMAP_PAGE_M** or **RMMAP_PAGE_G**. The array is valid if the return value is 0.

Execution Environment

The **rmmap_getwimg** kernel service is called from the process environment only.

Return Values

0	Successful completion. Indicates that the *results* array is valid and should be examined.
EINVAL	An error occurred. Most likely the region was not mapped via **rmmap_create64** or **rmmap_create** previously.
EINVAL	Input range crosses a certain boundary.
EINVAL	The hardware platform does not implement this service.

Implementation Specifics

The **rmmap_getwimg** kernel service is part of the Base Operating System (BOS) Runtime. This service only functions on PowerPC microprocessors.

Related Information

The **rmmap_create64** kernel service, the **rmmap_remove64** kernel service, the **rmmap_create** kernel service, the **rmmap_remove** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in AIX Version 4 Kernel Extensions and Device Support Programming Concepts.

rmmap_remove Kernel Service

Purpose

Destroys an effective address to real address translation region.

Syntax

```
#include <sys/adspace.h>
int rmmap_remove (eaddrp);
void **eaddrp;
```

Parameters

eaddrp	Pointer to the process effective address of the desired mapping region.

Description

Destroys an effective address to real address translation region. If **rmmap_remove** kernel service is called with the effective address within the region of a previously created I/O mapping segment, the region is destroyed.

Execution Environment

The **rmmap_remove** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
EINVAL	The provided *eaddr* does not correspond to a valid I/O mapping segment.
EINVAL	This hardware platform does not implement this service.

Implementation Specifics

This service must be called from the process level.

The **rmmap_remove** kernel service is part of Base Operating System (BOS) Runtime. This service only functions on PowerPC microprocessors.

This service only functions on PowerPC microprocessors.

Related Information

The **rmmap_create** Kernel Service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

rmmap_remove64 Kernel Service

Purpose

Destroys an effective address to real address translation region.

Syntax

#include <sys/adspace.h>

int rmmap_remove64 (*eaddr***);**
unsigned long long *eaddr***;**

Parameter

eaddr	The process effective address of the desired mapping region. This address is interpreted as a 64–bit quantity if the current user address space is 64–bits, and is interpreted as a 32–bit (not remapped) quantity if the current user address space is 32–bits.

Description

If **rmmap_remove64** is called with the effective address within the region of a previously created I/O mapping segment, the region is destroyed.

Execution Environment

The **rmmap_remove64** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
EINVAL	The provided *eaddr* does not correspond to a valid I/O mapping segment.
EINVAL	This hardware platform does not implement this service.

Implementation Specifics

The **rmmap_remove64** kernel service is part of Base Operating System (BOS) Runtime.

This service only functions on PowerPC microprocessors.

Related Information

The **rmmap_create64** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

rtalloc Kernel Service

Purpose

Allocates a route.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <net/route.h>

void rtalloc (ro)
register struct route *ro;
```

Parameter

ro	Specifies the route.

Description

The **rtalloc** kernel service allocates a route, which consists of a destination address and a reference to a routing entry.

Execution Environment

The **rtalloc** kernel service can be called from either the process or interrupt environment.

Return Values

The **rtalloc** service has no return values.

Example

To allocate a route, invoke the **rtalloc** kernel service as follows:

```
rtalloc(ro);
```

Implementation Specifics

The **rtalloc** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

rtalloc_gr Kernel Service

Purpose

Allocates a route.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <net/route.h>

void rtalloc_gr (ro, gidlist)
register struct route *ro;
struct gidstruct *gidlist;
```

Parameter

ro	Specifies the route.
gidlist	Points to the group list.

Description

The **rtalloc_gr** kernel service allocates a route, which consists of a destination address and a reference to a routing entry.

A route can be allocated only if its group id restrictions specify that it can be used by a user with the *gidlist* that is passed in.

Execution Environment

The **rtalloc_gr** kernel service can be called from either the process or interrupt environment.

Return Values

The **rtalloc_gr** service has no return values.

Example

To allocate a route, invoke the **rtalloc_gr** kernel service as follows:

```
rtalloc_gr (ro, gidlist);
```

Implementation Specifics

The **rtalloc_gr** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

The **rtalloc** kernel service.

rtfree Kernel Service

Purpose

Frees the routing table entry.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <net/route.h>

int rtfree (rt)
register struct rtentry *rt;
```

Parameter

rt	Specifies the routing table entry.

Description

The **rtfree** kernel service frees the entry it is passed from the routing table. If the route does not exist, the **panic** service is called. Otherwise, the **rtfree** service frees the **mbuf** structure that contains the route and decrements the routing reference counters.

Execution Environment

The **rtfree** kernel service can be called from either the process or interrupt environment.

Return Values

The **rtfree** kernel service has no return values.

Example

To free a routing table entry, invoke the **rtfree** kernel service as follows:

```
rtfree(rt);
```

Implementation Specifics

The **rtfree** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **panic** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

rtinit Kernel Service

Purpose

Sets up a routing table entry typically for a network interface.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/socket.h>
#include <net/route.h>

int rtinit (ifa, cmd, flags)
struct ifaddr *ifa;
int cmd, flags;
```

Parameters

ifa	Specifies the address of an **ifaddr** structure containing destination address, interface address, and netmask.
cmd	Specifies a request to add or delete route entry.
flags	Identifies routing flags, as defined in the /**usr**/**include**/**net**/**route.h** file.

Description

The **rtinit** kernel service creates a routing table entry for an interface. It builds an **rtentry** structure using the values in the *ifa* and *flags* parameters.

The **rtinit** service then calls the **rtrequest** kernel service and passes the *cmd* parameter and the **rtentry** structure to process the request. The *cmd* parameter contains either the value **RTM_ADD** (a request to add the route entry) or the value **RTM_DELETE** (delete the route entry). Valid routing flags to set are defined in the /**usr**/**include**/**route.h** file.

Execution Environment

The **rtinit** kernel service can be called from either the process or interrupt environment.

Return Values

The **rtinit** kernel service returns values from the **rtrequest** kernel service.

Example

To set up a routing table entry, invoke the **rtinit** kernel service as follows:

```
rtinit(ifa, RMT_ADD, flags ( RTF_DYNAMIC);
```

Implementation Specifics

The **rtinit** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **rtrequest** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

rtredirect Kernel Service

Purpose

Forces a routing table entry with the specified destination to go through a given gateway.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/mbuf.h>
#include <net/route.h>

rtredirect (dst, gateway, netmask, flags, src, rtp)
struct sockaddr *dst, *gateway, *netmask, *src;
int flags;
struct rtentry **rtp;
```

Parameters

dst	Specifies the destination address.
gateway	Specifies the gateway address.
netmask	Specifies the network mask for the route.
flags	Indicates routing flags as defined in the **/usr/include/net/route.h** file.
src	Identifies the source of the redirect request.
rtp	Indicates the address of a pointer to a **rtentry** structure. Used to return a constructed route.

Description

The **rtredirect** kernel service forces a routing table entry for a specified destination to go through the given gateway. Typically, the **rtredirect** service is called as a result of a routing redirect message from the network layer. The *dst*, *gateway*, and *flags* parameters are passed to the **rtrequest** kernel service to process the request.

Execution Environment

The **rtredirect** kernel service can be called from either the process or interrupt environment.

Return Values

0	Indicates a successful operation.

If a bad redirect request is received, the routing statistics counter for bad redirects is incremented.

Example

To force a routing table entry with the specified destination to go through the given gateway, invoke the **rtredirect** kernel service:

```
rtredirect(dst, gateway, netmask, flags, src, rtp);
```

Implementation Specifics

The **rtredirect** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **rtinit** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

rtrequest Kernel Service

Purpose

Carries out a request to change the routing table.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/mbuf.h>
#include <net/if.h>
#include <net/af.h>
#include <net/route.h>

int rtrequest (req, dst, gateway, netmask, flags, ret_nrt)
int req;
struct sockaddr *dst, *gateway, *netmask;
int flags;
struct rtentry **ret_nrt;
```

Parameters

req	Specifies a request to add or delete a route.
dst	Specifies the destination part of the route.
gateway	Specifies the gateway part of the route.
netmask	Specifies the network mask to apply to the route.
flags	Identifies routing flags, as defined in the **/usr/include/net/route.h** file.
ret_nrt	Specifies to return the resultant route.

Description

The **rtrequest** kernel service carries out a request to change the routing table. Interfaces call the **rtrequest** service at boot time to make their local routes known for routing table ioctl operations. Interfaces also call the **rtrequest** service as the result of routing redirects. The request is either to add (if the *req* parameter has a value of **RMT_ADD**) or delete (the *req* parameter is a value of **RMT_DELETE**) the route.

Execution Environment

The **rtrequest** kernel service can be called from either the process or interrupt environment.

Return Values

0	Indicates a successful operation.
ESRCH	Indicates that the route was not there to delete.
EEXIST	Indicates that the entry the **rtrequest** service tried to add already exists.
ENETUNREACH	Indicates that the **rtrequest** service cannot find the interface for the route.
ENOBUFS	Indicates that the **rtrequest** service cannot get an **mbuf** structure to add an entry.

Example

To carry out a request to change the routing table, invoke the **rtrequest** kernel service as follows:

```
rtrequest(RTM_ADD, dst, gateway, netmask, flags, &rtp);
```

Implementation Specifics

The **rtrequest** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **rtinit** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

rtrequest_gr Kernel Service

Purpose

Carries out a request to change the routing table.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/mbuf.h>
#include <net/if.h>
#include <net/af.h>
#include <net/route.h>

int rtrequest_gr (req, dst, gateway, gidlist, netmask, flags,
ret_nrt)
int req;
struct sockaddr *dst, *gateway, *netmask;
int flags;
struct rtentry **ret_nrt;
struct gidstruct *gidlist;
```

Parameters

req	Specifies a request to add or delete a route.
dst	Specifies the destination part of the route.
gateway	Specifies the gateway part of the route.
gidlist	Points to the group list.
netmask	Specifies the network mask to apply to the route.
flags	Identifies routing flags, as defined in the /**usr**/**include**/**net**/**route.h** file.
ret_nrt	Specifies to return the resultant route.

Description

The **rtrequest_gr** kernel service carries out a request to change the routing table. Interfaces call the **rtrequest_gr** service at boot time to make their local routes known for routing table ioctl operations. Interfaces also call the **rtrequest_gr** service as the result of routing redirects. The request is either to add (if the *req* parameter has a value of **RMT_ADD**) or delete (the *req* parameter is a value of **RMT_DELETE**) the route.

The *gidlist* parameter specifies a list of group id restrictions. A route can be allocated only if its group id restrictions specify that it can be used by the user on whose behalf the allocation is done. A route with a NULL *gidlist* can be used by any user.

Execution Environment

The **rtrequest_gr** kernel service can be called from either the process or interrupt environment.

Return Values

0	Indicates a successful operation.
ESRCH	Indicates that the route was not there to delete.
EEXIST	Indicates that the entry the **rtrequest_gr** service tried to add already exists.

ENETUNREACH	Indicates that the **rtrequest_gr** service cannot find the interface for the route.
ENOBUFS	Indicates that the **rtrequest_gr** service cannot get an **mbuf** structure to add an entry.

Example

To carry out a request to change the routing table, invoke the **rtrequest_gr** kernel service as follows:

```
rtrequest_gr(RTM_ADD, dst, gateway, netmask, flags, &rtp);
```

Implementation Specifics

The **rtrequest_gr** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **rtinit** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

The **rtrequest** kernel service.

rusage_incr Kernel Service

Purpose

Increments a field of the **rusage** structure.

Syntax

```
#include <sys/encap.h>

void rusage_incr (field, amount)
int field;
int amount;
```

Parameters

field Specifies the field to increment. It must have one of the following values:

RUSAGE_INBLOCK
Denotes the `ru_inblock` field. This field specifies the number of times the file system performed input.

RUSAGE_OUTBLOCK
Denotes the `ru_outblock` field. This field specifies the number of times the file system performed output.

RUSAGE_MSGRCV
Denotes the `ru_msgrcv` field. This field specifies the number of IPC messages received.

RUSAGE_MSGSENT
Denotes the `ru_msgsnd` field. This field specifies the number of IPC messages sent.

amount Specifies the amount to increment to the field.

Description

The **rusage_incr** kernel service increments the field specified by the *field* parameter of the calling process' **rusage** structure by the amount *amount*.

Execution Environment

The **rusage_incr** kernel service can be called from the process environment only.

Return Values

The **rusage_incr** kernel service has no return values.

Implementation Specifics

The **rusage_incr** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **getrusage** subroutine.

schednetisr Kernel Service

Purpose

Schedules or invokes a network software interrupt service routine.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <net/netisr.h>

int schednetisr (anisr)
int anisr;
```

Parameter

anisr	Specifies the software interrupt number to issue.

Description

The **schednetisr** kernel service schedules or calls a network interrupt service routine. The **add_netisr** kernel service establishes interrupt service routines. If the service was added with a service level of **NET_OFF_LEVEL**, the **schednetisr** kernel service directly calls the interrupt service routine. If the service level was **NET_KPROC**, a network kernel dispatcher is notified to call the interrupt service routine.

Execution Environment

The **schednetisr** kernel service can be called from either the process or interrupt environment.

Return Values

EFAULT	Indicates that a network interrupt service routine does not exist for the specified interrupt number.
EINVAL	Indicates that the *anisr* parameter is out of range.

Implementation Specifics

The **schednetisr** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **add_netisr** kernel service, **del_netisr** kernel service.

Network Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

selnotify Kernel Service

Purpose

Wakes up processes waiting in a **poll** or **select** subroutine or in the **fp_poll** kernel service.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

void selnotify (id, subid, rtnevents)
int id;
int subid;
ushort rtnevents;
```

Parameters

id	Indicates a primary resource identification value. This value along with the subidentifier (specified by the *subid* parameter) is used by the kernel to notify the appropriate processes of the occurrence of the indicated events. If the resource on which the event has occurred is a device driver, this parameter must be the device major/minor number (that is, a **dev_t** structure that has been cast to an **int**). The kernel has reserved values for the *id* parameter that do not conflict with possible device major or minor numbers for sockets, message queues, and named pipes.
subid	Helps identify the resource on which the event has occurred for the kernel. For a multiplexed device driver, this is the number of the channel on which the requested events occurred. If the device driver is nonmultiplexed, the *subid* parameter must be set to 0.
rtnevents	Consists of a set of bits indicating the requested events that have occurred on the specified device or channel. These flags have the same definition as the event flags that were provided by the *events* parameter on the unsatisfied call to the object's select routine.

Description

The **selnotify** kernel service should be used by device drivers that support select or poll operations. It is also used by the kernel to support select or poll requests to sockets, named pipes, and message queues.

The **selnotify** kernel service wakes up processes waiting on a **select** or **poll** subroutine. The processes to be awakened are those specifying the given device and one or more of the events that have occurred on the specified device. The **select** and **poll** subroutines allow a process to request information about one or more events on a particular device. If none of the requested events have yet happened, the process is put to sleep and re–awakened later when the events actually happen.

The **selnotify** service should be called whenever a previous call to the device driver's **ddselect** entry point returns and both of the following conditions apply:

- The status of all requested events is false.
- Asynchronous notification of the events is requested.

The **selnotify** service can be called for other than these conditions but performs no operation.

Sequence of Events for Asynchronous Notification

The device driver must store information about the events requested while in the driver's **ddselect** routine under the following conditions:

- None of the requested events are true (at the time of the call).
- The **POLLSYNC** flag is not set in the *events* parameter.

The **POLLSYNC** flag, when not set, indicates that asynchronous notification is desired. In this case, the **selnotify** service should be called when one or more of the requested events later becomes true for that device and channel.

When the device driver finds that it can satisfy a **select** request, (perhaps due to new input data) and an unsatisfied request for that event is still pending, the **selnotify** service is called with the following items:

- Device major and minor number specified by the *id* parameter
- Channel number specified by the *subid* parameter
- Occurred events specified by the *rtnevents* parameter

These parameters describe the device instance and requested events that have occurred on that device. The notifying device driver then resets its requested–events flags for the events that have occurred for that device and channel. The reset flags thus indicate that those events are no longer requested.

If the *rtnevents* parameter indicated by the call to the **selnotify** service is no longer being waited on, no processes are awakened.

Execution Environment

The **selnotify** kernel service can be called from either the process or interrupt environment.

Return Values

The **selnotify** service has no return values.

Implementation Specifics

The **selnotify** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **ddselect** device driver entry point.

The **fp_poll** kernel service, **fp_select** kernel service, **selreg** kernel service.

The **poll** subroutine, **select** subroutine.

Kernel Extension and Device Driver Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

selreg Kernel Service

Purpose

Registers an asynchronous poll or select request with the kernel.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/poll.h>

int selreg (corl, dev_id, unique_id, reqevents, notify)
int corl;
int dev_id;
int unique_id;
ushort reqevents;
void (*notify) ( );
```

Parameters

corl	The correlator for the poll or select request. The *corl* parameter is used by the **poll** and **select** subroutines to correlate the returned events in a specific select control block with a process' file descriptor or message queue.
dev_id	Primary resource identification value. Along with the *unique_id* parameter, the *dev_id* parameter is used to record in the select control block the resource on which the requested poll or select events are expected to occur.
unique_id	Unique resource identification value. Along with the *dev_id* parameter, the *unique_id* parameter denotes the resource on which the requested events are expected to occur. For a multiplexed device driver, this parameter specifies the number of the channel on which the requested events are expected to occur. For a nonmultiplexed device driver, this parameter must be set to 0.
reqevents	Requested events parameter. The *reqevents* parameter consists of a set of bit flags denoting the events for which notification is being requested. These flags have the same definitions as the event flags provided by the *events* parameter on the unsatisfied call to the object's **select** subroutine (see the **sys**/**poll.h** file for the definitions). **Note:** The **POLLSYNC** bit flag should not be set in this parameter.
notify	Notification routine entry point. This parameter points to a notification routine used for nested poll and select calls.

Description

The **selreg** kernel service is used by **select** file operations in the top half of the kernel to register an unsatisfied asynchronous poll or select event request with the kernel. This registration enables later calls to the **selnotify** kernel service from resources in the bottom half of the kernel to correctly identify processes awaiting events on those resources.

The event requests may originate from calls to the **poll** or **select** subroutine, from processes, or from calls to the **fp_poll** or **fp_select** kernel service. A **select** file operation calls the **selreg** kernel service under the following circumstances:

- The poll or select request is asynchronous (the **POLLSYNC** flag is not set for the requested event's bit flags).

- The poll or select request determines (by calling the underlying resource's **ddselect** entry point) that the requested events have not yet occurred.

A registered event request takes the form of a select control block. The select control block is a structure containing the following:

- Requested event bit flags
- Returned event bit flags
- Primary resource identifier
- Unique resource identifier
- Pointer to a **proc** table entry
- File descriptor correlator
- Pointer to a notification routine that is non–null only for nested calls to the **poll** and **select** subroutines

The **selreg** kernel service allocates and initializes a select control block each time it is called.

When an event occurs on a resource that supports the **select** file operation, the resource calls the **selnotify** kernel service. The **selnotify** kernel service locates all select control blocks whose primary and unique identifiers match those of the resource, and whose requested event flags match the occurred events on the resource. Then, for each of the matching control blocks, the **selnotify** kernel service takes one of two courses of action, depending upon whether the control block's notification routine pointer is non–null (nested) or null (non–nested):

- In nested calls to the **select** or **poll** subroutines, the notification routine is called with the primary and unique resource identifiers, the returned event bit flags, and the process identifiers.
- In non–nested calls to the **select** or **poll** subroutine (the usual case), the SSEL bit of the process identified in the block is cleared, the returned event bit flags in the block are updated, and the process is awakened. A process awakened in this manner completes the **poll** or **select** call in which it was sleeping. The **poll** or **select** subroutine then collects the returned event bit flags in its processes' select control blocks for return to the user mode process, deallocates the control blocks, and returns tallys of the numbers of requested events that occurred to the user process.

Execution Environment

The **selreg** kernel service can be called from the process environment only.

Returns Values

0	Indicates successful completion.
EAGAIN	Indicates the **selreg** kernel service was unable to allocate a select control block.

Implementation Specifics

The **selreg** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **ddselect** device driver entry point.

The **fp_poll** kernel service, **fp_select** kernel service, **selnotify** kernel service.

The **poll** subroutine, **select** subroutine.

Select and Poll Support and Kernel Extension and Device Driver Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

setjmpx Kernel Service

Purpose

Allows saving the current execution state or context.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int setjmpx (jump_buffer)
label_t *jump_buffer;
```

Parameter

jump_buffer	Specifies the address of the caller–supplied jump buffer that was specified on the call to the **setjmpx** service.

Description

The **setjmpx** kernel service saves the current execution state, or context, so that a subsequent **longjmpx** call can cause an immediate return from the **setjmpx** service. The **setjmpx** service saves the context with the necessary state information including:

- The current interrupt priority.
- Whether the process currently owns the kernel mode lock.

Other state variables include the nonvolatile general purpose registers, the current program's table of contents and stack pointers, and the return address.

Calls to the **setjmpx** service can be nested. Each call to the **setjmpx** service causes the context at this point to be pushed to the top of the stack of saved contexts.

Execution Environment

The **setjmpx** kernel service can be called from either the process or interrupt environment.

Return Values

Nonzero value	Indicates that a **longjmpx** call caused the **setjmpx** service to return.
0	Indicates any other circumstances.

Implementation Specifics

The **setjmpx** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **clrjmpx** kernel service, **longjmpx** kernel service.

Handling Signals While in a System Call, Exception Processing, Implementing Kernel Exception Handlers, Process and Exception Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

setpinit Kernel Service

Purpose

Sets the parent of the current kernel process to the initialization process.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/device.h>

int setpinit()
```

Description

The **setpinit** kernel service can be called by a kernel process to set its parent process to the **init** process. This is done to redirect the death of child signal for the termination of the kernel process. As a result, the init process is allowed to perform its default zombie process cleanup.

The **setpinit** service is used by a kernel process that can terminate, but does not want the user–mode process under which it was created to receive a death of child process notification.

Execution Environment

The **setpinit** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
EINVAL	Indicates that the current process is not a kernel process.

Implementation Specifics

The **setpinit** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

Using Kernel Processes and Process and Exception Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

setuerror Kernel Service

Purpose

Allows kernel extensions to set the `ut_error` field in the **u** area.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int setuerror (errno)
int errno;
```

Parameter

errno	Contains a value found in the **/usr/include/sys/errno.h** file that is to be copied to the current thread `u t_error` field.

Description

The **setuerror** kernel service allows a kernel extension in a process environment to set the `u t _error` field in current thread's **uthread** structure. Kernel extensions providing system calls available to user–mode applications typically use this service. For system calls, the value of the `u t_error` field in the per thread **uthread** structure is copied to the **errno** global variable by the system call handler before returning to the caller.

Execution Environment

The **setuerror** kernel service can be called from the process environment only.

Return Codes

The **setuerror** kernel service returns the *errno* parameter.

Implementation Specifics

The **setuerror** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **getuerror** kernel service.

Kernel Extension and Device Driver Management Kernel Services and Understanding System Call Execution in *AIX Kernel Extensions and Device Support Programming Concepts*.

sig_chk Kernel Service

Purpose

Provides a kernel process the ability to poll for receipt of signals.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/signal.h>

int sig_chk ()
```

Description

Attention: A system crash will occur if the **sig_chk** service is not called by a kernel process.

The **sig_chk** kernel service can be called by a kernel thread in kernel mode to determine if any unmasked signals have been received. Signals do not preempt threads because serialization of critical data areas would be lost. Instead, threads must poll for signals, either periodically or after a long sleep has been interrupted by a signal.

The **sig_chk** service checks for any pending signal that has a specified *signal catch* or *default* action. If one is found, the service returns the signal number as its return value. It also removes the signal from the pending signal mask. If no signal is found, this service returns a value of 0. The **sig_chk** service does not return signals that are blocked or ignored. It is the responsibility of the kernel process to handle the signal appropriately.

For kernel–only threads, the **sig_chk** kernel service clears the returned signal from the list of pending signals. For other kernel threads, the signal is not cleared, but left pending. It will be delivered to the kernel thread as soon as it returns to the user mode.

Understanding Kernel Threads in *AIX Kernel Extensions and Device Support Programming Concepts* provides more information about kernel–only thread signal handling.

Execution Environment

The **sig_chk** kernel service can be called from the process environment only.

Return Values

Upon completion, the **sig_chk** service returns a value of 0 if no pending unmasked signal is found. Otherwise, it returns a nonzero signal value indicating the number of the highest priority signal that is pending. Signal values are defined in the **/usr/include/sys/signal.h** file.

Implementation Specifics

The **sig_chk** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

Introduction to Kernel Processes, Process and Exception Management Kernel Services, and Kernel Process Signal and Exception Handling in *AIX Kernel Extensions and Device Support Programming Concepts*.

simple_lock or simple_lock_try Kernel Service

Purpose

Locks a simple lock.

Syntax

```
#include <sys/lock_def.h>

void simple_lock (lock_addr)
simple_lock_t lock_addr;

boolean_t simple_lock_try (lock_addr)
simple_lock_t lock_addr;
```

Parameter

lock_addr	Specifies the address of the lock word to lock.

Description

The **simple_lock** kernel service locks the specified lock; it blocks if the lock is busy. The lock must have been previously initialized with the **simple_lock_init** kernel service. The **simple_lock** kernel service has no return values.

The **simple_lock_try** kernel service tries to lock the specified lock; it returns immediately without blocking if the lock is busy. If the lock is free, the **simple_lock_try** kernel service locks it. The lock must have been previously initialized with the **simple_lock_init** kernel service.

Note: When using simple locks to protect thread–interrupt critical sections, it is recommended that you use the **disable_lock** kernel service instead of calling the **simple_lock** kernel service directly.

Execution Environment

The **simple_lock** and **simple_lock_try** kernel services can be called from the process environment only.

Return Values

The **simple_lock_try** kernel service has the following return values:

TRUE	Indicates that the simple lock has been successfully acquired.
FALSE	Indicates that the simple lock is busy, and has not been acquired.

Implementation Specifics

The **simple_lock** and **simple_lock_try** kernel services are part of the Base Operating System (BOS) Runtime.

Related Information

The **disable_lock** kernel service, **lock_mine** kernel service, **simple_lock_init** kernel service, **simple_unlock** kernel service.

Understanding Locking and Locking Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*

simple_lock_init Kernel Service

Purpose

Initializes a simple lock.

Syntax

```
#include <sys/lock_def.h>

void simple_lock_init (lock_addr)
simple_lock_t lock_addr;
```

Parameter

lock_addr	Specifies the address of the lock word.

Description

The **simple_lock_init** kernel service initializes a simple lock. This kernel service must be called before the simple lock is used. The simple lock must previously have been allocated with the **lock_alloc** kernel service.

Execution Environment

The **simple_lock_init** kernel service can be called from the process environment only.

The **simple_lock_init** kernel service may be called either the process or interrupt environments.

Return Values

The **simple_lock_init** kernel service has no return values.

Implementation Specifics

The **simple_lock_init** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **lock_alloc** kernel service, **lock_free** kernel service, **simple_lock** kernel service, **simple_lock_try** kernel service, **simple_unlock** kernel service.

Understanding Locking and Locking Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*

simple_unlock Kernel Service

Purpose

Unlocks a simple lock.

Syntax

```
#include <sys/lock_def.h>

void simple_unlock (lock_addr)
simple_lock_t lock_addr;
```

Parameter

lock_addr Specifies the address of the lock word to unlock.

Description

The **simple_unlock** kernel service unlocks the specified simple lock. The lock must be held by the thread which calls the **simple_unlock** kernel service. Once the simple lock is unlocked, the highest priority thread (if any) which is waiting for it is made runnable, and may compete for the lock again. If at least one kernel thread was waiting for the lock, the priority of the calling kernel thread is recomputed.

Note: When using simple locks to protect thread–interrupt critical sections, it is recommended that you use the **unlock_enable** kernel service instead of calling the **simple_unlock** kernel service directly.

Execution Environment

The **simple_unlock** kernel service can be called from the process environment only.

Return Values

The **simple_unlock** kernel service has no return values.

Implementation Specifics

The **simple_unlock** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **lock_mine** kernel service, **simple_lock_init** kernel service, **simple_lock** kernel service, **simple_lock_try** kernel service, **unlock_enable** kernel service.

Understanding Locking and Locking Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*

sleep Kernel Service

Purpose

Forces the calling kernel thread to wait on a specified channel.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/pri.h>
#include <sys/proc.h>

int sleep (chan, priflags)
void *chan;
int priflags;
```

Parameters

chan	Specifies the channel number. For the **sleep** service, this parameter identifies the channel to wait for (sleep on).
priflags	Specifies two conditions: • The priority at which the kernel thread is to run when it is reactivated. • Flags indicating how a signal is to be handled by the **sleep** kernel service.

The valid flags and priority values are defined in the /**usr**/**include**/**sys**/**pri.h** file.

Description

The **sleep** kernel service is provided for compatibility only and should not be invoked by new code. The **e_sleep_thread** or **et_wait** kernel service should be used when writing new code.

The **sleep** service puts the calling kernel thread to sleep, causing it to wait for a wakeup to be issued for the channel specified by the *chan* parameter. When the process is woken up again, it runs with the priority specified in the *priflags* parameter. The new priority is effective until the process returns to user mode.

All processes that are waiting on the channel are restarted at once, causing a race condition to occur between the activated threads. Thus, after returning from the **sleep** service, each thread should check whether it needs to sleep again.

The channel specified by the *chan* parameter is simply an address that by convention identifies some event to wait for. When the kernel or kernel extension detects such an event, the **wakeup** service is called with the corresponding value in the *chan* parameter to start up all the threads waiting on that channel. The channel identifier must be unique systemwide. The address of an external kernel variable (which can be defined in a device driver) is generally used for this value.

If the **SWAKEONSIG** flag is not set in the *priflags* parameter, signals do not terminate the sleep. If the **SWAKEONSIG** flag is set and the **PCATCH** flag is not set, the kernel calls the **longjmpx** kernel service to resume the context saved by the last **setjmpx** call if a signal interrupts the sleep. Therefore, any system call (such as those calling device driver **ddopen**, **ddread**, and **ddwrite** routines) or kernel process that does an interruptible sleep without the **PCATCH** flag set must have set up a context using the **setjmpx** kernel service. This allows the sleep to resume in case a signal is sent to the sleeping process.

Attention: The caller of the **sleep** service must own the kernel–mode lock specified by the *kernel_lock* parameter. The **sleep** service does not provide a compatible level of serialization if the kernel lock is not owned by the caller of the **sleep** service.

Execution Environment

The **sleep** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
1	Indicates that a signal has interrupted a sleep with both the **PCATCH** and **SWAKEONSIG** flags set in the *priflags* parameter.

Implementation Specifics

The **sleep** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

Locking Strategy in Kernel Mode in *AIX Kernel Extensions and Device Support Programming Concepts*.

Process and Exception Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

subyte Kernel Service

Purpose

Stores a byte of data in user memory.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int subyte (uaddr, c)
uchar *uaddr;
uchar c;
```

Parameters

uaddr	Specifies the address of user data.
c	Specifies the character to store.

Description

The **subyte** kernel service stores a byte of data at the specified address in user memory. It is provided so that system calls and device heads can safely access user data. The **subyte** service ensures that the user has the appropriate authority to:

- Access the data.
- Protect the operating system from paging I/O errors on user data.

The **subyte** service should only be called while executing in kernel mode in the user process.

Execution Environment

The **subyte** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
–1	Indicates a *uaddr* parameter that is not valid for one of the following reasons: • The user does not have sufficient authority to access the data. • The address is not valid. • An I/O error occurs when the user data is referenced.

Implementation Specifics

The **subyte** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **fubyte** kernel service, **fuword** kernel service, **suword** kernel service.

Accessing User–Mode Data While in Kernel Mode and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

subyte64 Kernel Service

Purpose

Stores a byte of data in user memory.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/uio.h>

int subyte64 ( uaddr64, c )
unsigned long long uaddr64;
 char c;
```

Parameter

uaddr64	Specifies the address of user data.
c	Specifies the character to store.

Description

The **subyte64** kernel service stores a byte of data at the specified address in user memory. It is provided so that system calls and device heads can safely access user data. The **subyte64** service ensures that the user has the appropriate authority to:

- Access the data.
- Protect the operating system from paging I/O errors on user data.

This service will operate correctly for both 32–bit and 64–bit user address spaces. The *uaddr64* parameter is interpreted as being a non–remapped 32–bit address for the case where the current user address space is 32– bits. If the current user address space is 64–bits, then **uaddr64** is treated as a 64–bit address.

The **subyte64** service should be called only while executing in kernel mode in the user process.

Execution Environment

The **subyte64** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
–1	Indicates a uaddr64 parameter that is not valid because: The user does not have sufficient authority to access the data, or The address is not valid, or An I/O error occurs while referencing the user data.

Implementation Specifics

The **subyte64** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **fubyte64** kernel service, **fuword64** kernel service, and **suword64** kernel service.

Accessing User–Mode Data While in Kernel Mode and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

suser Kernel Service

Purpose

Determines the privilege state of a process.

Syntax

```
#include <sys/types.h>
#include  <sys/errno.h>

int suser (ep)
char *ep;
```

Parameter

ep	Points to a character variable where the **EPERM** value is stored on failure.

Description

The **suser** kernel service checks whether a process has any effective privilege (that is, whether the process's `uid` field equals 0).

Execution Environment

The **suser** kernel service can be called from the process environment only.

Return Values

0	Indicates failure. The character pointed to by the *ep* parameter is set to the value of **EPERM**. This indicates that the calling process does not have any effective privilege.
Nonzero value	Indicates success (the process has the specified privilege).

Implementation Specifics

This kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Security Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

suword Kernel Service

Purpose

Stores a word of data in user memory.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int suword (uaddr, w)
int *uaddr;
int w;
```

Parameters

uaddr	Specifies the address of user data.
w	Specifies the word to store.

Description

The **suword** kernel service stores a word of data at the specified address in user memory. It is provided so that system calls and device heads can safely access user data. The **suword** service ensures that the user had the appropriate authority to:

- Access the data.
- Protect the operating system from paging I/O errors on user data.

The **suword** service should only be called while executing in kernel mode in the user process.

Execution Environment

The **suword** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
–1	Indicates a *uaddr* parameter that is not valid for one of these reasons: • The user does not have sufficient authority to access the data. • The address is not valid. • An I/O error occurs when the user data is referenced.

Implementation Specifics

The **suword** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **fubyte** kernel service, **fuword** kernel service, **subyte** kernel service.

Memory Kernel Services and Accessing User–Mode Data While in Kernel Mode in *AIX Kernel Extensions and Device Support Programming Concepts*.

suword64 Kernel Service

Purpose

Stores a word of data in user memory.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/uio.h>

int suword64 (uaddr64, w)
unsigned long long uaddr64;
 int w;
```

Parameter

uaddr64	Specifies the address of user data.
w	Specifies the word to store.

Description

The **suword64** kernel service stores a word of data at the specified address in user memory. It is provided so that system calls and device heads can safely access user data. The **suword64** service ensures that the user has the appropriate authority to:

- Access the data.
- Protect the operating system from paging I/O errors on user data.

This service will operate correctly for both 32–bit and 64–bit user address spaces. The *uaddr64* parameter is interpreted as being a non–remapped 32–bit address for the case where the current user address space is 32–bits. If the current user address space is 64–bits, then **uaddr64** is treated as a 64–bit address.

The **suword64** service should be called only while executing in kernel mode in the user process.

Execution Environment

The **suword64** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
–1	Indicates a *uaddr64* parameter that is not valid because: The user does not have sufficient authority to access the data, or The address is not valid, or An I/O error occurs while referencing the user data.

Implementation Specifics

The **suword64** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **fubyte64** kernel service, **fuword64** kernel service, and **subyte64** kernel service.

Accessing User–Mode Data While in Kernel Mode and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

talloc Kernel Service

Purpose

Allocates a timer request block before starting a timer request.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/timer.h>

struct trb *talloc()
```

Description

The **talloc** kernel service allocates a timer request block. The user must call it before starting a timer request with the **tstart** kernel service. If successful, the **talloc** service returns a pointer to a pinned timer request block.

Execution Environment

The **talloc** kernel service can be called from the process environment only.

Return Values

The **talloc** service returns a pointer to a timer request block upon successful allocation of a **trb** structure. Upon failure, a null value is returned.

Implementation Specifics

The **talloc** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **tfree** kernel service, **tstart** kernel service, **tstop** kernel service.

Timer and Time–of–Day Kernel Services and Using Fine Granularity Timer Services and Structures in *AIX Kernel Extensions and Device Support Programming Concepts*.

tfree Kernel Service

Purpose

Deallocates a timer request block.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/timer.h>

void tfree (t)
struct trb *t;
```

Parameter

t	Points to the timer request structure to be freed.

Description

The **tfree** kernel service deallocates a timer request block that was previously allocated with a call to the **talloc** kernel service. The caller of the **tfree** service must first cancel any pending timer request associated with the timer request block being freed before attempting to free the request block. Canceling the timer request block can be done using the **tstop** kernel service.

Execution Environment

The **tfree** kernel service can be called from either the process or interrupt environment.

Return Values

The **tfree** service has no return values.

Implementation Specifics

The **tfree** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **talloc** kernel service, **tstart** kernel service, **tstop** kernel service.

Timer and Time–of–Day Kernel Services and Using Fine Granularity Timer Services and Structures in *AIX Kernel Extensions and Device Support Programming Concepts*.

thread_create Kernel Service

Purpose

Creates a new kernel thread in the calling process.

Syntax

```
#include <sys/thread.h>

tid_t thread_create ()
```

Description

The **thread_create** kernel service creates a new kernel–only thread in the calling process. The thread's ID is returned; it is unique system wide.

The new thread does not begin running immediately; its state is set to **TSIDL**. The execution will start after a call to the **kthread_start** kernel service. If the process is exited prior to the thread being made runnable, the thread's resources are released immediately. The thread's signal mask is inherited from the calling thread; the set of pending signals is cleared. Signals sent to the thread are marked pending while the thread is in the **TSIDL** state.

If the calling thread is bound to a specific processor, the new thread will also be bound to the processor.

Execution Environment

The **thread_create** kernel service can be called from the process environment only.

Return Values

Upon successful completion, the new thread's ID is returned. Otherwise, –1 is returned, and the error code can be checked by calling the **getuerror** kernel service.

Error Codes

EAGAIN	The total number of kernel threads executing system wide or the maximum number of kernel threads per process would be exceeded.
ENOMEM	There is not sufficient memory to create the kernel thread.

Implementation Specifics

The **thread_create** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **kthread_start** kernel service.

Process and Exception Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

thread_self Kernel Service

Purpose

Returns the caller's kernel thread ID.

Syntax

```
#include <sys/thread.h>
tid_t thread_self ()
```

Description

The **thread_self** kernel service returns the thread process ID of the calling process.

The **thread_self** service can also be used to check the environment that the routine is being executed in. If the caller is executing in the interrupt environment, the **thread_self** service returns a process ID of –1. If a routine is executing in a process environment, the **thread_self** service obtains the thread process ID.

Execution Environment

The **thread_self** kernel service can be called from either the process or interrupt environment.

Return Values

–1 Indicates that the **thread_self** service was called from an interrupt environment.

The **thread_self** service returns the thread process ID of the current process if called from a process environment.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

Process and Exception Management Kernel Services and Understanding Execution Environments in *AIX Kernel Extensions and Device Support Programming Concepts*.

thread_setsched Kernel Service

Purpose

Sets kernel thread scheduling parameters.

Syntax

```
#include <sys/thread.h>
#include <sys/sched.h>

int thread_setsched (tid, priority, policy)
tid_t tid;
int priority;
int policy;
```

Parameters

tid	Specifies the kernel thread.
priority	Specifies the priority. It must be in the range from 0 to **PRI_LOW**; 0 is the most favored priority.
policy	Specifies the scheduling policy. It must have one of the following values:

SCHED_FIFO	Denotes fixed priority first–in first–out scheduling.
SCHED_RR	Denotes fixed priority round–robin scheduling.
SCHED_OTHER	Denotes the default AIX scheduling policy.

Description

The **thread_setsched** subroutine sets the scheduling parameters for a kernel thread. This includes both the priority and the scheduling policy, which are specified in the *priority* and *policy* parameters. The calling and the target thread must be in the same process.

When setting the scheduling policy to **SCHED_OTHER**, the system chooses the priority; the *priority* parameter is ignored. The only way to influence the priority of a thread using the default AIX scheduling policy is to chnage the process nice value.

The calling thread must belong to a process with root authority to change the scheduling policy of a thread to either **SCHED_FIFO** or **SCHED_RR**.

Execution Environment

The **thread_setsched** kernel service can be called from the process environment only.

Return Values

Upon successful completion, 0 is returned. Otherwise, –1 is returned, and the error code can be checked by calling the **getuerror** kernel service.

Error Codes

EINVAL	The *priority* or *policy* parameters are not valid.
EPERM	The calling kernel thread does not have sufficient privilege to perform the operation.
ESRCH	The kernel thread *tid* does not exist.

Implementation Specifics

The **thread_setsched** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **thread_create** kernel service.

Process and Exception Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

thread_terminate Kernel Service

Purpose

Terminates the calling kernel thread.

Syntax

```
#include <sys/thread.h>

void thread_terminate ()
```

Description

The **thread_terminate** kernel service terminates the calling kernel thread and cleans up its structure and its kernel stack. If it is the last thread in the process, the process will exit.

The **thread_terminate** kernel service is automatically called when a thread returns from its entry point routine (defined in the call to the **kthread_start** kernel service).

Execution Environment

The **thread_terminate** kernel service can be called from the process environment only.

Return Values

The **thread_terminate** kernel service never returns.

Implementation Specifics

The **thread_terminate** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **kthread_start** kernel service.

Process and Exception Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

timeout Kernel Service

Attention: This service should not be used in AIX Version 4, because it is not multi–processor safe. The base kernel timer and watchdog services should be used instead. See talloc and w_init for more information.

Purpose

Schedules a function to be called after a specified interval.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

void timeout (func, arg, ticks)
void (*func)();
caddr_t *arg;
int ticks;
```

Parameters

func	Indicates the function to be called.
arg	Indicates the parameter to supply to the function specified by the *func* parameter.
ticks	Specifies the number of timer ticks that must occur before the function specified by the *func* parameter is called. Many timer ticks can occur per second. The HZ label found in the **/usr/include/sys/m_param.h** file can be used to determine the number of ticks per second.

Description

The **timeout** service is not part of the kernel. However, it is a compatibility service provided in the **libsys.a** library. To use the **timeout** service, a kernel extension must have been bound with the **libsys.a** library. The **timeout** service, like the associated kernel services **untimeout** and **timeoutcf**, can be bound and used only in the pinned part of a kernel extension or the bottom half of a device driver because these services use interrupt disable for serialization.

The **timeout** service schedules the function pointed to by the *func* parameter to be called with the *arg* parameter after the number of timer ticks specified by the *ticks* parameter. Use the **timeoutcf** routine to allocate enough callout elements for the maximum number of simultaneous active time outs that you expect.

Note: The **timeoutcf** routine must be called before calling the **timeout** service.

Calling the **timeout** service without allocating a sufficient number of callout table entries can result in a kernel panic because of a lack of pinned callout table elements. The value of a timer tick depends on the hardware's capability. You can use the **restimer** subroutine to determine the minimum granularity.

Multiple pending **timeout** requests with the same *func* and *arg* parameters are not allowed.

The func Parameter

The function specified by the *func* parameter should be declared as follows:

```
void func (arg)
void *arg;
```

Execution Environment

The **timeout** routine can be called from either the process or interrupt environment.

The function specified by the *func* parameter is called in the interrupt environment. Therefore, it must follow the conventions for interrupt handlers.

Return Values

The **timeout** service has no return values.

Implementation Specifics

The **timeout** routine is part of Base Operating System (BOS) Runtime.

Related Information

The **untimeout** kernel service.

The **timeoutcf** kernel subroutine.

The **restimer** subroutine.

Timer and Time–of–Day Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

timeoutcf Subroutine for Kernel Services

Attention: This service should not be used in AIX Version 4, because it is not multi–processor safe. The base kernel timer and watchdog services should be used instead. See talloc and w_init for more information.

Purpose

Allocates or deallocates callout table entries for use with the **timeout** kernel service.

Library

libsys.a (Kernel extension runtime routines)

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int timeoutcf (cocnt)
int cocnt;
```

Parameter

cocnt	Specifies the callout count. This value indicates the number of callout elements by which to increase or decrease the current allocation. If this number is positive, the number of callout entries for use with the **timeout** service is increased. If this number is negative, the number of elements is decreased by the amount specified.

Description

The **timeoutcf** subroutine is not part of the kernel. It is a compatibility service provided in the **libsys.a** library. To use the **timeoutcf** subroutine, a kernel extension must have been bound with the **libsys.a** library. The **timeoutcf** subroutine, like the associated kernel libsys services **untimeout** and **timeout**, can be bound and used only in the pinned part of a kernel extension or the bottom half of a device driver because these services use interrupt disable for serialization.

The **timeoutcf** subroutine registers an increase or decrease in the number of callout table entries available for the **timeout** subroutine to use. Before a subroutine can use the **timeout** kernel service, the **timeoutcf** subroutine must increase the number of callout table entries available to the **timeout** kernel service. It increases this number by the maximum number of outstanding time outs that the routine can have pending at one time.

The **timeoutcf** subroutine should be used to decrease the amount of callout table entries by the amount it was increased under the following conditions:

- The routine using the **timeout** subroutine has finished using it.
- The calling routine has no more outstanding time–out requests pending.

Typically the **timeoutcf** subroutine is called in a device driver's **open** and **close** routine. It is called to allocate and deallocate sufficient elements for the maximum expected use of the **timeout** kernel service for that instance of the open device.

Attention: A kernel panic results under either of these two circumstances:

- A request to decrease the callout table allocation is made that is greater than the number of unused callout table entries.
- The **timeoutcf** subroutine is called in an interrupt environment.

Execution Environment

The **timeoutcf** subroutine can be called from the process environment only.

Return Values

0 Indicates a successful allocation or deallocation of the requested callout table entries.

–1 Indicates an unsuccessful operation.

Implementation Specifics

The **timeoutcf** subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **timeout** kernel service.

Timer and Time–of–Day Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

trcgenk Kernel Service

Purpose

Records a trace event for a generic trace channel.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/trchkid.h>

void trcgenk (chan, hk_word, data_word, len, buf)
unsigned int chan, hk_word, data_word, len;
char *buf;
```

Parameters

chan Specifies the channel number for the trace session. This number is obtained from the **trcstart** subroutine.

hk_word An integer containing a hook ID and a hook type:

- **hk_id** A hook identifier is a 12–bit value. For user programs, the hook ID can be a value from 0x010 to 0x0FF.
- **hk_type** A 4–bit hook type. The **trcgenk** kernel service automatically records this information.

data_word Specifies a word of user–defined data.

len Specifies the length in bytes of the buffer specified by the *buf* parameter.

buf Points to a buffer of trace data. The maximum amount of trace data is 4096 bytes.

Description

The **trcgenk** kernel service records a trace event if a trace session is active for the specified trace channel. If a trace session is not active, the **trcgenk** kernel service simply returns. The **trcgenk** kernel service is located in pinned kernel memory.

The **trcgenk** kernel service is used to record a trace entry consisting of an *hk_word* entry, a *data_word* entry, and a variable number of bytes of trace data.

Execution Environment

The **trcgenk** kernel service can be called from either the process or interrupt environment.

Return Values

The **trcgenk** kernel service has no return values.

Implementation Specifics

The **trcgenk** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **trace** daemon.

The **trcgenkt** kernel service.

The **trcgen** subroutine, **trcgent** subroutine, **trchook** subroutine, **trcoff** subroutine, **trcon** subroutine, **trcstart** subroutine, **trcstop** subroutine.

RAS Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

trcgenkt Kernel Service

Purpose

Records a trace event, including a time stamp, for a generic trace channel.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/trchkid.h>

void trcgenkt (chan, hk_word, data_word, len, buf)
unsigned int chan, hk_word, data_word, len;
char *buf;
```

Parameters

chan	Specifies the channel number for the trace session. This number is obtained from the **trcstart** subroutine.
hk_word	An integer containing a hook ID and a hook type: **hk_id** A hook identifier is a 12–bit value. For user programs, the hook ID can be a value from 0x010 to 0x0FF. **hk_type** A 4–bit hook type. The **trcgenkt** service automatically records this information.
data_word	Specifies a word of user–defined data.
len	Specifies the length, in bytes, of the buffer identified by the *buf* parameter.
buf	Points to a buffer of trace data. The maximum amount of trace data is 4096 bytes.

Description

The **trcgenkt** kernel service records a trace event if a trace session is active for the specified trace channel. If a trace session is not active, the **trcgenkt** service simply returns. The **trcgenkt** kernel service is located in pinned kernel memory.

The **trcgenkt** service records a trace entry consisting of an *hk_word* entry, a *data_word* entry, a variable number of bytes of trace data, and a time stamp.

Execution Environment

The **trcgenkt** kernel service can be called from either the process or interrupt environment.

Return Values

The **trcgenkt** service has no return values.

Implementation Specifics

The **trcgenkt** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **trace** daemon.

The **trcgenk** kernel service.

The **trcgen** subroutine, **trcgent** subroutine, **trchook** subroutine, **trcoff** subroutine, **trcon** subroutine, **trcstart** subroutine, **trcstop** subroutine.

RAS Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

trcgenkt Kernel Service for Data Link Control (DLC) Devices

Purpose

Records a trace event, including a time stamp, for a DLC trace channel.

Syntax

```
#include <sys/trchkid.h>

void trcgenkt (chan, hk_word, data_word, len, buf)
unsigned int chan, hk_word, data_word, len;
char *buf;
```

Parameters

chan Specifies the channel number for the trace session. This number is obtained from the **trcstart** subroutine.

hk_word Contains the trace hook identifier defined in the /**usr**/**include**/**sys**/**trchkid.h** file. The types of link trace entries registered using the hook ID include:

HKWD_SYSX_DLC_START
Start link station completions

HKWD_SYSX_DLC_TIMER
Time–out completions

HKWD_SYSX_DLC_XMIT
Transmit completions

HKWD_SYSX_DLC_RECV
Receive completions

HKWD_SYSX_DLC_HALT
Halt link station completions

data_word Specifies trace data format field. This field varies depending on the hook ID. Each of these definitions are in the /**usr**/**include**/**sys**/**gdlextcb.h** file:

- The first half–word always contains the data link protocol field including one of these definitions:

DLC_DL_SDLC	SDLC
DLC_DL_HDLC	HDLC
DLC_DL_BSC	BISYNC
DLC_DL_ASC	ASYNC
DLC_DL_PCNET	PC Network
DLC_DL_ETHER	Standard Ethernet

DLC_DL_802_3	IEEE 802.3
DLC_DL_TOKEN	Token–Ring

- On start or halt link station completion, the second half–word contains the physical link protocol in use:

DLC_PL_EIA232	EIA–232D Telecommunications
DLC_PL_EIA366	EIA–366 Auto Dial
DLC_PL_X21	CCITT X.21 Data Network
DLC_PL_PCNET	PC Network Broadband
DLC_PL_ETHER	Standard Baseband Ethernet
DLC_PL_SMART	Smart Modem Auto Dial
DLC_PL_802_3	IEEE 802.3 Baseband Ethernet
DLC_PL_TBUS	IEEE 802.4 Token Bus
DLC_PL_TRING	IEEE 802.5 Token–Ring
DLC_PL_EIA422	EIA–422 Telecommunications
DLC_PL_V35	CCITT V.35 Telecommunications
DLC_PL_V25BIS	CCITT V.25 bis Autodial for Telecommunications

- On timeout completion, the second half–word contains the type of timeout occurrence:

DLC_TO_SLOW_POLL	Slow station poll
DLC_TO_IDLE_POLL	Idle station poll
DLC_TO_ABORT	Link station aborted
DLC_TO_INACT	Link station receive inactivity
DLC_TO_FAILSAFE	Command failsafe
DLC_TO_REPOLL_T1	Command repoll
DLC_TO_ACK_T2	I–frame acknowledgment

- On transmit completion, the second half–word is set to the data link control bytes being sent. Some transmit packets only have a single control byte; in that case, the second control byte is not displayed.
- On receive completion, the second half–word is set to the data link control bytes that were received. Some receive packets only have a single control byte; in that case, the second control byte is not displayed.

len Specifies the length in bytes of the entry specific data specified by the *buf* parameter.

buf Specifies the pointer to the entry specific data that consists of:

Start Link Station Completions
Link station diagnostic tag and the remote station's name and address.

Time–out Completions
No specific data is recorded.

Transmit Completions
Either the first 80 bytes or all the transmitted data, depending on the short/long trace option.

Receive Completions
Either the first 80 bytes or all the received data, depending on the short/long trace option.

Halt Link Station Completions
Link station diagnostic tag, the remote station's name and address, and the result code.

Description

The **trcgenkt** kernel service records a trace event if a trace session is active for the specified trace channel. If a trace session is not active, the **trcgenkt** kernel service simply returns. The **trcgenkt** kernel service is located in pinned kernel memory.

The **trcgenkt** kernel service is used to record a trace entry consisting of an *hk_word* entry, a *data_word* entry, a variable number of bytes of trace data, and a time stamp.

Execution Environment

The **trcgenkt** kernel service can be called from either the process or interrupt environment.

Return Values

The **trcgenkt** kernel service has no return values.

Implementation Specifics

This kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **trcgenk** kernel service, **trcgenkt** kernel service.

The **trace** daemon.

Generic Data Link Control (GDLC) Environment Overview and RAS Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

tstart Kernel Service

Purpose

Submits a timer request.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/timer.h>

void tstart (t)
struct trb *t;
```

Parameter

t	Points to a timer request structure.

Description

The **tstart** kernel service submits a timer request with the timer request block specified by the *t* parameter as input. The caller of the **tstart** kernel service must first call the **talloc** kernel service to allocate the timer request structure. The caller must then initialize the structure's fields before calling the **tstart** kernel service.

Once the request has been submitted, the kernel calls the `t->func` timer function when the amount of time specified by the `t->timeout.it` value has elapsed. The `t->func` timer function is called on an interrupt level. Therefore, code for this routine must follow conventions for interrupt handlers.

The **tstart** kernel service examines the `t->flags` field to determine if the timer request being submitted represents an absolute request or an incremental one. An absolute request is a request for a time out at the time represented in the **it_value** structure. An incremental request is a request for a time out at the time represented by now, plus the time in the **it_value** structure.

The caller should place time information for both absolute and incremental timers in the **itimerstruc_t t.it** value substructure. The **T_ABSOLUTE** absolute request flag is defined in the **/usr/include/sys/timer.h** file and should be ORed into the `t->flag` field if an absolute timer request is desired.

Modifications to the system time are added to incremental timer requests, but not to absolute ones. Consider the user who has submitted an absolute timer request for noon on 12/25/88. If a privileged user then modifies the system time by adding four hours to it, then the timer request submitted by the user still occurs at noon on 12/25/88.

By contrast, suppose it is presently 12 noon and a user submits an incremental timer request for 6 hours from now (to occur at 6 p.m.). If, before the timer expires, the privileged user modifies the system time by adding four hours to it, the user's timer request will then expire at 2200 (10 p.m.).

Execution Environment

The **tstart** kernel service can be called from either the process or interrupt environment.

Return Values

The **tstart** service has no return values.

Implementation Specifics

The **tstart** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **talloc** kernel service, **tfree** kernel service, **tstop** kernel service.

Timer and Time–of–Day Kernel Services and Using Fine Granularity Timer Services and Structures in *AIX Kernel Extensions and Device Support Programming Concepts*.

tstop Kernel Service

Purpose

Cancels a pending timer request.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/timer.h>

int tstop (t)
struct trb *t;
```

Parameter

t	Specifies the pending timer request to cancel.

Description

The **tstop** kernel service cancels a pending timer request. The **tstop** kernel service must be called before a timer request block can be freed with the **tfree** kernel service.

In a multiprocessor environment, the timer function associated with a timer request block may be active on another processor when the **tstop** kernel service is called. In this case, the timer request cannot be canceled. A multiprocessor–safe driver must therefore check the return code and take appropriate action if the cancel request failed.

In a uniprocessor environment, the call always succeeds. This is untrue in a multiprocessor environment, where the call will fail if the timer is being handled by another processor. Therefore, the function now has a return value, which is set to 0 if successful, or –1 otherwise. Funnelled device drivers do not need to check the return value since they run in a logical uniprocessor environment. Multiprocessor–safe and multiprocessor–efficient device drivers need to check the return value in a loop. In addition, if a driver uses locking, it must release and reacquire its lock within this loop, as shown below:

```
while (tstop(&trb))
  release_then_reacquire_dd_lock;
                /* null statement if locks not used */
```

Execution Environment

The **tstop** kernel service can be called from either the process or interrupt environment.

Return Values

0	Indicates that the request was successfully canceled.
–1	Indicates that the request could not be canceled.

Implementation Specifics

The **tstop** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **talloc** kernel service, **tfree** kernel service, **tstart** kernel service.

Timer and Time–of–Day Kernel Services, Using Fine Granularity Timer Services and Structures, Using Multiprocessor–Safe Timer Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

uexadd Kernel Service

Purpose

Adds a systemwide exception handler for catching user–mode process exceptions.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/except.h>

void uexadd (exp)
struct uexcepth *exp;
```

Parameter

exp Points to an exception handler structure. This structure must be pinned and is used for registering user–mode process exception handlers. The **uexcepth** structure is defined in the **/usr/include/sys/except.h** file.

Description

The **uexadd** kernel service is typically used to install a systemwide exception handler to catch exceptions occurring during execution of a process in user mode. The **uexadd** kernel service adds the exception handler structure specified by the *exp* parameter, to the chain of exception handlers to be called if an exception occurs while a process is executing in user mode. The last exception handler registered is the first exception handler called for a user–mode exception.

The **uexcepth** structure has:

- A chain element used by the kernel to chain the registered user exception handlers.
- A function pointer defining the entry point of the exception handler being added.

Additional exception handler–dependent information can be added to the end of the structure, but must be pinned.

Attention: The **uexcepth** structure must be pinned when the **uexadd** kernel service is called. It must remain pinned and unmodified until after the call to the **uexdel** kernel service to delete the specified exception handler. Otherwise, the system may crash.

Execution Environment

The **uexadd** kernel service can be called from the process environment only.

Return Values

The **uexadd** kernel service has no return values.

Implementation Specifics

The **uexadd** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **uexdel** kernel service and User–Mode Exception Handler for the **uexadd** Kernel Service.

User–Mode Exception Handling and Kernel Extension and Device Driver Management Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

User–Mode Exception Handler for the uexadd Kernel Service

Purpose

Handles exceptions that occur while a kernel thread is executing in user mode.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/except.h>

int func (exp, type, tid, mst)
struct excepth *exp;
int type;
tid_t tid;
struct mstsave *mst;
```

Parameters

exp	Points to the **excepth** structure used to register this exception handler.
mst	Points to the current **mstsave** area for the process. This pointer can be used to access the **mstsave** area to obtain additional information about the exception.
tid	Specifies the thread ID of the kernel thread that was executing at the time of the exception.
type	Denotes the type of exception that has occurred. This type value is platform–specific. Specific values are defined in the /**usr**/**include**/**sys**/**except.h** file.

Description

The user–mode exception handler (*exp*–>**func**) is called for synchronous exceptions that are detected while a kernel thread is executing in user mode. The kernel exception handler saves exception information in the **mstsave** area of the structure. For user–mode exceptions, it calls the first exception handler found on the user exception handler list. The exception handler executes in an interrupt environment at the priority level of either **INTPAGER** or **INTIODONE**.

If the registered exception handler returns a return code indicating that the exception was handled, the kernel exits from the exception handler without calling additional exception handlers from the list. If the exception handler returns a return code indicating that the exception was not handled, the kernel invokes the next exception handler on the list. The last exception handler in the list is the default handler. This is typically signalling the thread.

The kernel exception handler must not page fault. It should also register an exception handler using the **setjmpx** kernel service if any exception–handling activity can result in an exception. This is important particularly if the exception handler is handling the I/O. If the exception handler did not handle the exception, the return code should be set to the **EXCEPT_NOT_HANDLED** value for user–mode exception handling.

Execution Environment

The user–mode exception handler for the **uexadd** kernel service is called in the interrupt environment at the **INTPAGER** or **INTIODONE** priority level.

Return Values

EXCEPT_HANDLED	Indicates that the exception was successfully handled.
EXCEPT_NOT_HANDLED	Indicates that the exception was not handled.

Implementation Specifics

This routine is part of Base Operating System (BOS) Runtime.

Related Information

The **uexadd** kernel service.

User–Mode Exception Handling and Kernel Extension and Device Driver Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

uexblock Kernel Service

Purpose

Makes the currently active kernel thread nonrunnable when called from a user–mode exception handler.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/except.h>

void uexblock (tid)
tid_t *tid;
```

Parameter

tid Specifies the thread ID of the currently active kernel thread to be put into a wait state.

Description

The **uexblock** kernel service puts the currently active kernel thread specified by the *tid* parameter into a wait state until the **uexclear** kernel service is used to make the thread runnable again. If the **uexblock** kernel service is called from the process environment, the *tid* parameter must specify the current active thread; otherwise the system will crash with a kernel panic.

The **uexblock** kernel service can be used to lazily control user–mode threads access to a shared serially usable resource. Multiple threads can use a serially used resource, but only one process at a time. When a thread attempts to but cannot access the resource, a user–mode exception can be set up to occur. This gives control to an exception handler registered by the **uexadd** kernel service. This exception handler can then block the thread using the **uexblock** kernel service until the resource is made available. At this time, the **uexclear** kernel service can be used to make the blocked thread runnable.

Execution Environment

The **uexblock** kernel service can be called from either the process or interrupt environment.

Return Values

The **uexblock** service has no return values.

Implementation Specifics

The **uexblock** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **uexclear** kernel service.

User–Mode Exception Handling and Kernel Extension and Device Driver Management Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

uexclear Kernel Service

Purpose

Makes a kernel thread blocked by the **uexblock** service runnable again.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/except.h>

void uexclear (tid)
tid_t *tid;
```

Parameter

tid Specifies the thread ID of the previously blocked kernel thread to be put into a run state.

Description

The **uexclear** kernel service puts a kernel thread specified by the *tid* parameter back into a runnable state after it was made nonrunnable by the **uexblock** kernel service. A thread that has been sent a **SIGSTOP** stop signal is made runnable again when it receives the **SIGCONT** continuation signal.

The **uexclear** kernel service can be used to lazily control user–mode thread access to a shared serially usable resource. A serially used resource is usable by more than one thread, but only by one at a time. When a thread attempts to access the resource but does not have access, a user–mode exception can be setup to occur.

This setup gives control to an exception handler registered by the **uexadd** kernel service. Using the **uexblock** kernel service, this exception handler can then block the thread until the resource is later made available. At that time, the **uexclear** service can be used to make the blocked thread runnable.

Execution Environment

The **uexclear** kernel service can be called from either the process or interrupt environment.

Return Values

The **uexclear** service has no return values.

Implementation Specifics

The **uexclear** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **uexblock** kernel service.

User–Mode Exception Handling and Kernel Extension and Device Driver Management Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

uexdel Kernel Service

Purpose

Deletes a previously added systemwide user–mode exception handler.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/except.h>

void uexdel (exp)
struct uexcepth *exp;
```

Parameter

exp Points to the exception handler structure used to add the exception handler with the **uexadd** kernel service.

Description

The **uexdel** kernel service removes a user–mode exception handler from the systemwide list of exception handlers maintained by the kernel's exception handler.

The **uexdel** kernel service removes the exception handler structure specified by the *exp* parameter from the chain of exception handlers to be called if an exception occurs while a process is executing in user mode. Once the **uexdel** kernel service has completed, the specified exception handler is no longer called. In addition, the **uexcepth** structure can be modified, freed, or unpinned.

Execution Environment

The **uexdel** kernel service can be called from the process environment only.

Return Values

The **uexdel** kernel service has no return values.

Implementation Specifics

The **uexdel** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **uexadd** kernel service.

User–Mode Exception Handling and Kernel Extension and Device Driver Management Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

ufdcreate Kernel Service

Purpose

Allocates and initializes a file descriptor.

Syntax

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/file.h>

int ufdcreate (flags, ops, datap, type, fdp, cnp)

int flags;
struct fileops *ops;
void *datap;
short type;
int *fdp;
struct ucred *crp;
```

Parameters

flags Specifies the flags to save in a **file** structure. The **file** structure is defined in the **sys/file.h** file. If a **read** or **write** subroutine is called with the file descriptor returened by this routine, the **FREAD** and **FWRITE** flags must be set appropriately. Valid flags are defined in the **fcntl.h** file.

ops Points to the list of subsystem–supplied routines to call for the file system operations: read/write, ioctl, select, fstat, and close. The **fileops** structure is defined in the **sys/file.h** file. See ”File Operations”, on page 1-459 for more information.

datap Points to type–dependent structures. The system saves this pointer in the **file** structure. As a result, the pointer is available to the file operations when they are called.

type Specifies the unique type value for the **file** structure. Valid types are listed in the **sys/file.h** file.

fdp Points to an integer field where the file descriptor is stored on successful return.

crp Points to a credentials structure. This pointer is saved in the file struct for use in subsequent operations. It must be a valid **ucred** struct. The **crref()** kernel service can be used to obtain a **ucred** struct.

Description

The **ufdcreate** kernel service provides a file interface to kernel extensions. Kernel extensions use this service to create a file descriptor and file structure pair. Also, this service allows kernel extensions to provide their own file descriptor–based system calls, enabling read/write, ioctl, select, fstat, and close operations on objects outside the file system. The **ufdcreate** kernel services does not require the extension to understand or conform to the synchronization requirements of the logical file system (LFS).

The **ufdcreate** kernel service provides a file descriptor to the caller and creates the underlying file structure. The caller must include pointers to subsystem–supplied routines for the read/write, ioctl, select, fstat, and close operations. If any of the operations are not needed by the calling subsystem, then the caller must provide a pointer to an appropriate **errno** value. Typically, the **EOPNOTSUPP** value is used for this purpose. See ”File Operations”, on page 1-459 for information about the requirements for the subsystem–supplied routines.

Removing a File Descriptor

There is no corresponding operation to remove a file descriptor (and the attendant structures) created by the **ufdcreate** kernel service. To remove a file descriptor, use a call to the **close** subroutine. The **close** subroutine can be called from a routine or from within the kernel or kernel extension. If the close is not called, the file is closed when the process exits.

Once a call is made to the **ufdcreate** kernel service, the file descriptor is considered open before the call to the service returns. When a **close** or **exit** subroutine is called, the close file operation specified on the call to the **ufdcreate** interface is called.

File Operations

The **ufdcreate** kernel service allows kernel extensions to provide their own file descriptor–based system calls, enabling read/write, ioctl, select, fstat, and close operations on objects outside the file system. The **fileops** structure defined in the **sys/file.h** file provides interfaces for these routines.

read/write Requirements

The read/write operation manages input and output to the object specified by the *fp* parameter. The actions taken by this operation are dependent on the object type. The syntax for the operation is as follows:

```
#include <sys/types.h>
#include <sys/uio.h>

int (*fo_rw) (fp, rw, uiop, ext)

struct file *fp;
enum uio_rw rw;
struct uio *uiop;
int ext;
```

The parameters have the following values:

fp	Points to the **file** structure. This structure corresponds to the file descriptor used on the **read** or **write** subroutine.
rw	Contains a **UIO_READ** value for a read operation or **UIO_WRITE** value for a write operation.
uiop	Points to a **uio** structure. This structure describes the location and size information for the input and output requested. The **uio** structure is defined in the **uio.h** file.
ext	Specifies subsystem–dependent information. If the **readx** or **writex** subroutine is used, the value passed by the operation is passed through to this subroutine. Otherwise, the value is 0.

If successful, the **fo_rw** operation returns a value of 0. A nonzero return value should be programmed to indicate an error. See the **sys/errno.h** file for a list of possible values.

Note: On successful return, the `uiop->uio_resid` field must be updated to include the number of bytes of data actually transferred.

ioctl Requirements

The ioctl operation provides object–dependent special command processing. The **ioctl** subroutine performs a variety of control operations on the object associated with the specified open **file** structure. This subroutine is typically used with character or block special files and returns an error for ordinary files.

The control operation provided by the ioctl operation is specific to the object being addressed, as are the data type and contents of the *arg* parameter.

The syntax for the ioctl operation is as follows:

```
#include <sys/types.h>
#include <sys/ioctl.h>

int (*fo_ioctl) (fp, cmd, arg, ext, kflag)

struct file *fp;
int cmd, ext, kflag;
caddr_t arg;
```

The parameters have the following values:

fp	Points to the **file** structure. This structure corresponds to the file descriptor used by the **ioctl** subroutine.
cmd	Defines the specific request to be acted upon by this routine.
arg	Contains data that is dependent on the *cmd* parameter.
ext	Specifies subsystem–specific information. If the **ioctlx** subroutine is used, the value passed by the application is passed through to this subroutine. Otherwise, the value is 0.
kflag	Determines where the call is made from. The *kflag* parameter has the value **FKERNEL** (from the **fcntl.h** file) if this routine is called through the **fp_ioctl** interface. Otherwise, its value is 0.

If successful, the **fo_ioctl** operation returns a value of 0. For errors, the **fo_ioctl** operation should return a nonzero return value to indicate an error. Refer to the **sys/errno.h** file for the list of possible values.

select Requirements

The select operation performs a select operation on the object specified by the *fp* parameter. The syntax for this operation is as follows:

```
#include <sys/types.h>

int (*fo_select) (fp, corl, reqevents, rtneventsp, notify)

struct file *fp;
int corl;
ushort reqevents, *rtneventsp;
void (notify) ();
```

The parameters have the following values:

fp	Points to the **file** structure. This structure corresponds to the file descriptor used by the **select** subroutine.
corl	Specifies the ID used for correlation in the **selnotify** kernel service.
reqevents	Identifies the events to check. The poll and select functions define three standard event flags and one informational flag. The **sys/poll.h** file details the event bit definition. See the **fp_select** kernel service for information about the possible flags.
rtneventsp	Indicates the returned events pointer. This parameter, passed by reference, indicates the events that are true at the current time. The returned event bits include the request events and an error event indicator.
notify	Points to a routine to call when the specified object invokes the **selnotify** kernel service for an outstanding asynchronous select or poll event request. If no routine is to be called, this parameter must be null.

If successful, the **fo_select** operation returns a value of 0. This operation should return a nonzero return value to indicate an error. Refer to the **sys/errno.h** file for the list of possible values.

fstat Requirements

The fstat operation fills in an **attribute** structure. Depending on the object type specified by the *fp* parameter, many fields in the structure may not be applicable. The value passed back from this operation is dependent upon both the object type and what any routine that understands the type is expecting. The syntax for this operation is as follows:

```
#include <sys/types.h>

int (*fo_fstat) (fp, sbp)

struct file *fp;
struct stat *sbp;
```

The parameters have the following values:

fp Points to the **file** structure. This structure corresponds to the file descriptor used by the **stat** subroutine.

sbp Points to the **stat** structure to be filled in by this operation. The address supplied is in kernel space.

If successful, the **fo_fstat** operation returns a value of 0. A nonzero return value should be programmed to indicate an error. Refer to the **sys/errno.h** file for the list of possible values.

close Requirements

The close operation invalidates routine access to objects specified by the *fp* parameter and releases any data associated with that access. This operation is called from the **close** subroutine code when the **file** structure use count is decremented to 0. For example, if there are multiple accesses to an object (created by the **dup**, **fork**, or other subsystem–specific operation), the **close** subroutine calls the close operation when it determines that there is no remaining access through the **file** structure being closed.

A file descriptor is considered open once a file descriptor and **file** structure have been set up by the LFS. The close file operation is called whenever a close or exit is specified. As a result, the close operation must be able to close an object that is not fully open, depending on what the caller did before the **file** structure was initialized.

The syntax for the close operation is as follows:

```
#include <sys/file.h>

int (*fo_close) (fp)
struct file *fp;
```

The parameter is:

fp Points to the **file** structure. This structure corresponds to the file descriptor used by the **close** subroutine.

If successful, the **fo_close** operation returns a value of 0. This operation should return a nonzero return value to indicate an error. Refer to the **sys/errno.h** file for the list of possible values.

Execution Environment

The **ufdcreate** kernel service can be called from the process environment only.

Return Values

If the **ufdcreate** kernel service succeeds, it returns a value of 0. If the kernel service fails, it returns a nonzero value and sets the **errno** global variable.

Error Codes

The **ufdcreate** kernel service fails if one or more of the following errors occur:

EINVAL	The *ops* parameter is null, or the **fileops** structure does not have entries for for every operation.
EMFILE	All file descriptors for the process have already been allocated.
ENFILE	The system file table is full.

Implementation Specifics

The **ufdcreate** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **selnotify** kernel service.

The **close** subroutine, **exit**, **atexit**, or **_exit** subroutine, **ioctl** subroutine, **open** subroutine, **read** subroutine, **select** subroutine, **write** subroutine, **fp_select** subroutine.

Logical File System Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

ufdgetf Kernel Service

Purpose

Returns a pointer to a file structure associated with a file descriptor.

Syntax

```
#include <sys/file.h>

int ufdgetf(fd, fpp)
int fd;
struct file **fpp;
```

Parameters

fd	Identifies the file descriptor. The descriptor must be for an open file.
fpp	Points to a location to store the file pointer.

Description

The **ufdgetf** kernel service returns a pointer to a file structure associated with a file descriptor. The calling routine must have a use count on the file descriptor. To obtain a use count on the file descriptor, the caller must first call the **ufdhold** kernel service.

Execution Environment

The **ufdget** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
EBADF	Indicates that the *fd* parameter is not a file descriptor for an open file.

Implementation Specifics

This kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **ufdhold** kernel service.

ufdhold and ufdrele Kernel Service

Purpose

Increment or decrement a file descriptor reference count.

Syntax

```
int ufdhold(fd)
int fd;

int ufdrele(fd)
int fd;
```

Parameter

fd	Identifies the file descriptor.

Description

Attention: It is extremely important that the calls to **ufdhold** and **ufdrele** kernel service are balanced. If a file descriptor is held more times than it is released, the **close** subroutine on the descriptor never completes. The process hangs and cannot be killed. If the descriptor is released more times than it is held, the system panics.

The **ufdhold** and **ufdrele** kernel services increment and decrement a file–descriptor reference count. Together, these kernel services maintain the file descriptor reference count. The **ufdhold** kernel service increments the count. The **ufdrele** kernel service decrements the count.

These subroutines are supported for kernel extensions that provide their own file–descriptor–based system calls. This support is required for synchronization with the **close** subroutine.

When a thread is executing a file–descriptor–based system call, it is necessary that the logical file system (LFS) be aware of it. The LFS uses the count in the file descriptor to monitor the number of system calls currently using any particular file descriptor. To keep the count accurately, any thread using the file descriptor must increment the count before performing any operation and decrement the count when all activity using the file descriptor is completed for that system call.

Execution Environment

These kernel services can be called from the process environment only.

Return Values

0	Indicates successful completion.
EBADF	Indicates that the *fd* parameter is not a file descriptor for an open file.

Implementation Specifics

This kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **ufdgetf** kernel service.

The **close** subroutine.

uiomove Kernel Service

Purpose

Moves a block of data between kernel space and a space defined by a **uio** structure.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/uio.h>

int uiomove (cp, n, rw, uiop)
caddr_t cp;
int n;
uio_rw rw;
struct uio *uiop;
```

Parameters

cp Specifies the address in kernel memory to or from which data is moved.

n Specifies the number of bytes to move.

rw Indicates the direction of the move:

UIO_READ Copies data from kernel space to space described by the **uio** structure.

UIO_WRITE Copies data from space described by the **uio** structure to kernel space.

uiop Points to a **uio** structure describing the buffer used in the data transfer.

Description

The **uiomove** kernel service moves the specified number of bytes of data between kernel space and a space described by a **uio** structure. Device driver top halves, especially character device drivers, frequently use the **uiomove** service to transfer data into or out of a user area. The `uio_resid` and `uio_iovcnt` fields in the **uio** structure describing the data area must be greater than 0 or an error is returned.

The **uiomove** service moves the number of bytes of data specified by either the *n* or *uio_resid* parameter, whichever is less. If either the *n* or *uio_resid* parameter is 0, no data is moved. The `uio_segflg` field in the **uio** structure is used to indicate if the move is accessing a user– or kernel–data area, or if the caller requires cross–memory operations and has provided the required cross–memory descriptors. If a cross–memory operation is indicated, there must be a cross–memory descriptor in the **uio_xmem** array for each `iovec` element.

If the move is successful, the following fields in the **uio** structure are updated:

`uio_iov` Specifies the address of current `iovec` element to use.

`uio_xmem` Specifies the address of the current `xmem` element to use.

`uio_iovcnt` Specifies the number of remaining `iovec` elements.

`uio_iovdcnt` Specifies the number of already processed `iovec` elements.

`uio_offset` Specifies the character offset on the device performing the I/O.

`uio_resid`	Specifies the total number of characters remaining in the data area described by the **uio** structure.
`iov_base`	Specifies the address of the data area described by the current `iovec` element.
`iov_len`	Specifies the length of remaining data area in the buffer described by the current `iovec` element.

Execution Environment

The **uiomove** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
–1	Indicates that an error occurred for one of the following conditions:

ENOMEM	Indicates there was no room in the buffer.
EIO	Indicates a permanent I/O error file space.
ENOSPC	Out of file–space blocks.
EFAULT	Indicates a user location that is not valid.

Implementation Specifics

The **uiomove** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **uphysio** kernel service, **ureadc** kernel service, **uwritec** kernel service.

unlock_enable Kernel Service

Purpose

Unlocks a simple lock if necessary, and restores the interrupt priority.

Syntax

```
#include <sys/lock_def.h>

void unlock_enable (int_pri, lock_addr)
int int_pri;
simple_lock_t lock_addr;
```

Parameters

int_pri	Specifies the interrupt priority to restore. This must be set to the value returned by the corresponding call to the **disable_lock** kernel service.
lock_addr	Specifies the address of the lock word to unlock.

Description

The **unlock_enable** kernel service unlocks a simple lock if necessary, and restores the interrupt priority, in order to provide optimized thread–interrupt critical section protection for the system on which it is executing. On a multiprocessor system, calling the **unlock_enable** kernel service is equivalent to calling the **simple_unlock** and **i_enable** kernel services. On a uniprocessor system, the call to the **simple_unlock** service is not necessary, and is omitted. However, you should still pass the valid lock address which was used with the corresponding call to the **disable_lock** kernel service. Never pass a **NULL** lock address.

Execution Environment

The **unlock_enable** kernel service can be called from either the process or interrupt environment.

Return Values

The **unlock_enable** kernel service has no return values.

Implementation Specifics

The **unlock_enable** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **disable_lock** kernel service, **i_enable** kernel service, **simple_unlock** kernel service.

Understanding Locking, Locking Kernel Services, Understanding Interrupts, I/O Kernel Services, Interrupt Environment in *AIX Kernel Extensions and Device Support Programming Concepts*.

unlockl Kernel Service

Purpose

Unlocks a conventional process lock.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

void unlockl (lock_word)
lock_t *lock_word;
```

Parameter

lock_word Specifies the address of the lock word.

Description

Note: The **unlockl** kernel service is provided for compatibility only and should not be used in new code, which should instead use simple locks or complex locks.

The **unlockl** kernel service unlocks a conventional lock. Only the owner of a lock can unlock it. Once a lock is unlocked, the highest priority thread (if any) which is waiting for the lock is made runnable and may compete again for the lock. If there was at least one process waiting for the lock, the priority of the caller is recomputed. Preempting a System Call discusses how system calls can use locking kernel services when accessing global data.

The **lockl** and **unlockl** services do not maintain a nesting level count. A single call to the **unlockl** service unlocks the lock for the caller. The return code from the **lockl** service should be used to determine when to unlock the lock.

Note: The **unlockl** kernel service can be called with interrupts disabled, only if the event or lock word is pinned.

Execution Environment

The **unlockl** kernel service can be called from the process environment only.

Return Values

The **unlockl** service has no return values.

Example

A call to the **unlockl** service can be coded as follows:

```
int lock_ret;           /* return code from lockl() */
extern int lock_word;   /* lock word that is external
                           and was initialized to
                           LOCK_AVAIL */
...
/* get lock prior to using resource */
lock_ret = lockl(lock_word, LOCK_SHORT)
/* use resource for which lock was obtained */
...
/* release lock if this was not a nested use */
if ( lock_ret != LOCK_NEST )
   unlockl(lock_word);
```

Implementation Specifics

The **unlockl** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **lockl** kernel service.

Understanding Locking in *AIX Kernel Extensions and Device Support Programming Concepts*.

Locking Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*

Preempting a System Call in *AIX Kernel Extensions and Device Support Programming Concepts*.

Interrupt Environment in *AIX Kernel Extensions and Device Support Programming Concepts*.

unpin Kernel Service

Purpose

Unpins the address range in system (kernel) address space.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/pin.h>

int unpin (addr, length)
caddr addr;
int length;
```

Parameters

addr	Specifies the address of the first byte to unpin in the system (kernel) address space.
length	Specifies the number of bytes to unpin.

Description

The **unpin** kernel service decreases the pin count of each page in the address range. When the pin count is 0, the page is not pinned and can be paged out of real memory. Upon finding an unpinned page, the **unpin** service returns the **EINVAL** error code and leaves any remaining pinned pages still pinned.

The **unpin** service can only be called with addresses in the system (kernel) address space. The **unpinu** service should be used where the address space might be in either user or kernel space.

Execution Environment

The **unpin** kernel service can be called from either the process or interrupt environment.

Return Values

0	Indicates successful completion.
EINVAL	Indicates that the value of the *length* parameter is negative or 0. Otherwise, the area of memory beginning at the byte specified by the *base* parameter and extending for the number of bytes specified by the *len* parameter is not defined. If neither cause is responsible, an unpinned page was specified.

Implementation Specifics

The **unpin** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **pin** kernel service, **pinu** kernel service, **unpinu** kernel service.

Understanding Execution Environments and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

unpincode Kernel Service

Purpose

Unpins the code and data associated with a loaded object module.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/pin.h>

int unpincode (func)
int (*func) ( );
```

Parameter

func	Specifies an address used to determine the object module to be unpinned. The address is typically that of a function that is exported by this object module.

Description

The **unpincode** kernel service uses the **ltunpin** kernel service to decrement the pin count for the pages associated with the following items:

- Code associated with the object module
- Data area of the object module that contains the function specified by the *func* parameter

The loader entry for the module is used to determine the size of both the code and the data area.

Execution Environment

The **unpincode** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
EINVAL	Indicates that the *func* parameter is not a valid pointer to the function.
EFAULT	Indicates that the calling process does not have access to the area of memory that is associated with the module.

Implementation Specifics

The **unpincode** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **unpin** kernel service.

Understanding Execution Environments and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

unpinu Kernel Service

Purpose

Unpins the specified address range in user or system memory.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/uio.h>

int unpinu (base, len, segflg)
caddr_t base;
int len;
short segflg;
```

Parameters

base	Specifies the address of the first byte to unpin.
len	Indicates the number of bytes to unpin.
segflg	Specifies whether the data to unpin is in user space or system space. The values for this flag are defined in the **/usr/include/sys/uio.h** file. This value can be one of the following:

UIO_SYSSPACE	The region is mapped into the kernel address space.
UIO_USERSPACE	The region is mapped into the user address space.

Description

The **unpinu** service unpins a region of memory previously pinned by the **pinu** kernel service. When the pin count is 0, the page is not pinned and can be paged out of real memory. Upon finding an unpinned page, the **unpinu** service returns the **EINVAL** error code and leaves any remaining pinned pages still pinned.

The **unpinu** service should be used where the address space might be in either user or kernel space.

If the caller has a valid cross–memory descriptor for the address range, the **xmempin** and **xmemunpin** kernel services can be used instead of **pinu** and **unpinu**, and result in less pathlength.

Execution Environment

The **unpinu** service can be called in the process environment when unpinning data that is in either user space or system space. It can be called in the interrupt environment only when unpinning data that is in system space.

Return Values

0	Indicates successful completion.
EFAULT	Indicates that the memory region as specified by the *base* and *len* parameters is not within the address specified by the *segflg* parameter.

EINVAL	Indicates that the value of the *length* parameter is negative or 0. Otherwise, the area of memory beginning at the byte specified by the *base* parameter and extending for the number of bytes specified by the *len* parameter is not defined. If neither cause is responsible, an unpinned page was specified.

Implementation Specifics

The **unpinu** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **pin** kernel service, **unpin** kernel service, **xmempin** kernel service, **xmemunpin** kernel service.

Understanding Execution Environments and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

untimeout Kernel Service

Attention: This service should not be used in AIX Version 4, because it is not multi–processor safe. The base kernel timer and watchdog services should be used instead. See talloc and w_init for more information.

Purpose

Cancels a pending timer request.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

void untimeout (func, arg)
void (*func)();
caddr_t *arg;
```

Parameters

func	Specifies the function associated with the timer to be canceled.
arg	Specifies the function argument associated with the timer to be canceled.

Description

The **untimeout** kernel service is not part of the kernel. However, it is a compatibility service provided in the **libsys.a** library. To use the **untimeout** service, a kernel extension must have been bound with the **libsys.a** library. The **untimeout** service, like the associated kernel libsys services **timeoutcf** and **timeout**, can be bound and used only in the pinned part of a kernel extension or the bottom half of a device driver because these services use interrupt disable for serialization.

The **untimeout** kernel service cancels a specific request made with the **timeout** service. The *func* and *arg* parameters must match those used in the **timeout** kernel service request that is to be canceled.

Upon return, the specified timer request is canceled, if found. If no timer request matching *func* and *arg* is found, no operation is performed.

Execution Environment

The **untimeout** kernel service can be called from either the process or interrupt environment.

Return Values

The **untimeout** kernel service has no return values.

Implementation Specifics

The **untimeout** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **timeout** kernel service.

Timer and Time–of–Day Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

uphysio Kernel Service

Purpose

Performs character I/O for a block device using a **uio** structure.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/buf.h>
#include <sys/uio.h>

int uphysio (uiop, rw, buf_cnt, devno, strat, mincnt, minparms)
struct uio *uiop;
int rw;
uint buf_cnt;
dev_t devno;
int (*strat)( );
int (*mincnt)( );
void *minparms;
```

Parameters

uiop	Points to the **uio** structure describing the buffer of data to transfer using character–to–block I/O.
rw	Indicates either a read or write operation. A value of **B_READ** for this flag indicates a read operation. A value of **B_WRITE** for this flag indicates a write operation.
buf_cnt	Specifies the maximum number of **buf** structures to use when calling the strategy routine specified by the *strat* parameter. This parameter is used to indicate the maximum amount of concurrency the device can support and minimize the I/O redrive time. The value of the *buf_cnt* parameter can range from 1 to 64.
devno	Specifies the major and minor device numbers. With the **uphysio** service, this parameter specifies the device number to be placed in the **buf** structure before calling the strategy routine specified by the *strat* parameter.
strat	Represents the function pointer to the **ddstrategy** routine for the device.
mincnt	Represents the function pointer to a routine used to reduce the data transfer size specified in the **buf** structure, as required by the device before the strategy routine is started. The routine can also be used to update extended parameter information in the **buf** structure before the information is passed to the strategy routine.
minparms	Points to parameters to be used by the *mincnt* parameter.

Description

The **uphysio** kernel service performs character I/O for a block device. The **uphysio** service attempts to send to the specified strategy routine the number of **buf** headers specified by the *buf_cnt* parameter. These **buf** structures are constructed with data from the **uio** structure specified by the *uiop* parameter.

The **uphysio** service initially transfers data area descriptions from each `iovec` element found in the **uio** structure into individual **buf** headers. These headers are later sent to the strategy routine. The **uphysio** kernel service tries to process as many data areas as the

number of **buf** headers permits. It then invokes the strategy routine with the list of **buf** headers.

Preparing Individual buf Headers

The routine specified by the *mincnt* parameter is called before the **buf** header, built from an `iovec` element, is added to the list of **buf** headers to be sent to the strategy routine. The *mincnt* parameter is passed a pointer to the **buf** header along with the *minparms* pointer. This arrangement allows the *mincnt* parameter to tailor the length of the data transfer described by the **buf** header as required by the device performing the I/O. The *mincnt* parameter can also optionally modify certain device–dependent fields in the **buf** header.

When the *mincnt* parameter returns with no error, an attempt is made to pin the data buffer described by the **buf** header. If the pin operation fails due to insufficient memory, the data area described by the **buf** header is reduced by half. The **buf** header is again passed to the *mincnt* parameter for modification before trying to pin the reduced data area.

This process of downsizing the transfer specified by the **buf** header is repeated until one of the three following conditions occurs:

- The pin operation succeeds.
- The *mincnt* parameter indicates an error.
- The data area size is reduced to 0.

When insufficient memory indicates a failed pin operation, the number of **buf** headers used for the remainder of the operation is reduced to 1. This is because trying to pin multiple data areas simultaneously under these conditions is not desirable.

If the user has not already obtained cross–memory descriptors, further processing is required. (The `uio_segflg` field in the **uio** structure indicates whether the user has already initialized the cross–memory descriptors. The **usr/include/sys/uio.h** file contains information on possible values for this flag.)

When the data area described by the **buf** header has been successfully pinned, the **uphysio** service verifies user access authority for the data area. It also obtains a cross–memory descriptor to allow the device driver interrupt handler limited access to the data area.

Calling the Strategy Routine

After the **uphysio** kernel service obtains a cross–memory descriptor to allow the device driver interrupt handler limited access to the data area, the **buf** header is then put on a list of **buf** headers to be sent to the strategy routine specified by the *strat* parameter.

The strategy routine specified by the *strat* parameter is called with the list of **buf** headers when:

- The list reaches the number of **buf** structures specified by the *buf_cnt* parameter.
- The data area described by the **uio** structure has been completely described by **buf** headers.

The **buf** headers in the list are chained together using the `av_back` and `av_forw` fields before they are sent to the strategy routine.

Waiting for buf Header Completion

When all available **buf** headers have been sent to the strategy routine, the **uphysio** service waits for one or more of the **buf** headers to be marked complete. The **IODONE** handler is used to wake up the **uphysio** service when it is waiting for completed **buf** headers from the strategy routine.

When the **uphysio** service is notified of a completed **buf** header, the associated data buffer is unpinned and the cross–memory descriptor is freed. (However, the cross–memory descriptor is freed only if the user had not already obtained it.) An error is detected on the data transfer under the following conditions:

- The completed **buf** header has a nonzero `b_resid` field.
- The `b_flags` field has the **B_ERROR** flag set.

When an error is detected by the **uphysio** service, no new **buf** headers are sent to the strategy routine.

The **uphysio** service waits for any **buf** headers already sent to the strategy routine to be completed and then returns an error code to the caller. If no errors are detected, the **buf** header and any other completed **buf** headers are again used to send more data transfer requests to the strategy routine as they become available. This process continues until all data described in the **uio** structure has been transferred or until an error has been detected.

The **uphysio** service returns to the caller when:

- All **buf** headers have been marked complete by the strategy routine.
- All data specified by the **uio** structure has been transferred.

The **uphysio** service also returns an error code to the caller if an error is detected.

Error Detection by the uphysio Kernel Service

When it detects an error, the **uphysio** kernel service reports the error that was detected closest to the start of the data area described by the **uio** structure. No additional **buf** headers are sent to the strategy routine. The **uphysio** kernel service waits for all **buf** headers sent to the strategy routine to be marked complete.

However, additional **buf** headers may have been sent to the strategy routine between these two events:

- After the strategy routine detects the error.
- Before the **uphysio** service is notified of the error condition in the completed **buf** header.

When errors occur, various fields in the returned **uio** structure may or may not reflect the error. The `uio_iov` and `uio_iovcnt` fields are not updated and contain their original values.

The `uio_resid` and `uio_offset` fields in the returned **uio** structure indicate the number of bytes transferred by the strategy routine according to the sum of all (the `b_bcount` field minus the `b_resid` fields) fields in the **buf** headers processed by the strategy routine. These headers include the **buf** header indicating the error nearest the start of the data area described by the original **uio** structure. Any data counts in **buf** headers completed after the detection of the error are not reflected in the returned **uio** structure.

Execution Environment

The **uphysio** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
ENOMEM	Indicates that no memory is available for the required **buf** headers.
EAGAIN	Indicates that the operation fails due to a temporary insufficient resource condition.
EFAULT	Indicates that the `uio_segflg` field indicated user space and that the user does not have authority to access the buffer.

EIO or the b_error field in a buf header	Indicates an I/O error in a **buf** header processed by the strategy routine.
Return code from the mincnt parameter	Indicates that the return code from the *mincnt* parameter if the routine returned with a nonzero return code.

Implementation Specifics

The **uphysio** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **ddstrategy** device driver entry point.

The **geterror** kernel service, **iodone** kernel service.

The **mincnt** routine.

The **buf** structure, **uio** structure.

uphysio Kernel Service mincnt Routine

Purpose

Tailors a **buf** data transfer request to device–dependent requirements.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/buf.h>

int mincnt (bp, minparms)
struct buf *bp;
void *minparms;
```

Parameters

bp	Points to the **buf** structure to be tailored.
minparms	Points to parameters.

Description

Only the following fields in the **buf** header sent to the routine specified by the **uphysio** kernel service *mincnt* parameter can be modified by that routine:

- `b_bcount`
- `b_work`
- `b_options`

The *mincnt* parameter cannot modify any other fields without the risk of error. If the *mincnt* parameter determines that the **buf** header cannot be supported by the target device, the routine should return a nonzero return code. This stops the **buf** header and any additional **buf** headers from being sent to the **ddstrategy** routine.

The **uphysio** kernel service waits for all **buf** headers already sent to the strategy routine to complete and then returns with the return code from the *mincnt* parameter.

Implementation Specifics

The **uphysio** kernel service *mincnt* parameter is part of Base Operating System (BOS) Runtime.

Related Information

The **uphysio** kernel service.

uprintf Kernel Service

Purpose

Submits a request to print a message to the controlling terminal of a process.

Syntax

```
#include <sys/uprintf.h>

int uprintf (Format [,Value, ...])
char *Format;
```

Parameters

Format Specifies a character string containing either or both of two types of objects:

- Plain characters, which are copied to the message output stream.
- Conversion specifications, each of which causes 0 or more items to be retrieved from the *Value* parameter list. Each conversion specification consists of a % (percent sign) followed by a character that indicates the type of conversion to be applied:

  % Performs no conversion. Prints %.

  d, i Accepts an integer *Value* and converts it to signed decimal notation.

  u Accepts an integer *Value* and converts it to unsigned decimal notation.

  o Accepts an integer *Value* and converts it to unsigned octal notation.

  x Accepts an integer *Value* and converts it to unsigned hexadecimal notation.

  s Accepts a *Value* as a string (character pointer), and characters from the string are printed until a \ 0 (null character) is encountered. *Value* must be non–null and the maximum length of the string is limited to **UP_MAXSTR** characters.

  Field width or precision conversion specifications are not supported.

  The following constants are defined in the /**usr**/**include**/**sys**/**uprintf.h** file:

  – **UP_MAXSTR**

  – **UP_MAXARGS**

  – **UP_MAXCAT**

  – **UP_MAXMSG**

  The *Format* string may contain from 0 to the number of conversion specifications specified by the **UP_MAXARGS** constant. The maximum length of the *Format* string is the number of characters specified by the **UP_MAXSTR** constant. *Format* must be non–null.

  The maximum length of the constructed kernel message is limited to the number of characters specified by the **UP_MAXMSG** constant. Messages larger then the number of characters specified by the **UP_MAXMSG** constant are discarded.

Value Specifies, as an array, the value to be converted. The number, type, and order of items in the *Value* parameter list should match the conversion specifications within the *Format* string.

Description

The **uprintf** kernel service submits a kernel message request. Once the request has been successfully submitted, the **uprintfd** daemon constructs the message based on the *Format* and *Value* parameters of the request. The **uprintfd** daemon then writes the message to the process' controlling terminal.

Execution Environment

The **uprintf** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
ENOMEM	Indicates that memory is not available to buffer the request.
ENODEV	Indicates that a controlling terminal does not exist for the process.
ESRCH	Indicates that the **uprintfd** daemon is not active. No requests may be submitted.
EINVAL	Indicates that a string *Value* string pointer is null or the string *Value* parameter is greater than the number of characters specified by the **UP_MAXSTR** constant.
EINVAL	Indicates one of the following: • *Format* string pointer is null. • Number of characters in the *Format* string is greater than the number specified by the **UP_MAXSTR** constant. • Number of conversion specifications contained within the *Format* string is greater than the number specified by the **UP_MAXARGS** constant.

Implementation Specifies

The **uprintf** kernel service is part of the Base Operating System (BOS) Runtime.

Related Information

The **NLuprintf** kernel service.

The **uprintfd** daemon.

Process and Exception Management Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

ureadc Kernel Service

Purpose

Writes a character to a buffer described by a **uio** structure.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/uio.h>

int ureadc (c, uiop)
int c;
struct uio *uiop;
```

Parameters

c	Specifies a character to be written to the buffer.
uiop	Points to a **uio** structure describing the buffer in which to place a character.

Description

The **ureadc** kernel service writes a character to a buffer described by a **uio** structure. Device driver top half routines, especially character device drivers, frequently use the **ureadc** kernel service to transfer data into a user area.

The `uio_resid` and `uio_iovcnt` fields in the **uio** structure describing the data area must be greater than 0. If these fields are not greater than 0, an error is returned. The `uio_segflg` field in the **uio** structure is used to indicate whether the data is being written to a user– or kernel–data area. It is also used to indicate if the caller requires cross–memory operations and has provided the required cross–memory descriptors. The values for the flag are defined in the /**usr**/**include**/**sys**/**uio.h** file.

If the data is successfully written, the following fields in the **uio** structure are updated:

`uio_iov`	Specifies the address of current `iovec` element to use.
`uio_xmem`	Specifies the address of current `xmem` element to use (used for cross–memory copy).
`uio_iovcnt`	Specifies the number of remaining `iovec` elements.
`uio_iovdcnt`	Specifies the number of `iovec` elements already processed.
`uio_offset`	Specifies the character offset on the device from which data is read.
`uio_resid`	Specifies the total number of characters remaining in the data area described by the `uio` structure.
`iov_base`	Specifies the address of the next available character in the data area described by the current `iovec` element.
`iov_len`	Specifies the length of remaining data area in the buffer described by the current `iovec` element.

Execution Environment

The **ureadc** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
ENOMEM	Indicates that there is no room in the buffer.
EFAULT	Indicates that the user location is not valid for one of these reasons: • The `uio_segflg` field indicates user space and the base address (`iov_base` field) points to a location outside of the user address space. • The user does not have sufficient authority to access the location. • An I/O error occurs while accessing the location.

Implementation Specifics

The **ureadc** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **uiomove** kernel service, **uphysio** kernel service, **uwritec** kernel service.

The **uio** structure.

Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

uwritec Kernel Service

Purpose

Retrieves a character from a buffer described by a **uio** structure.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/uio.h>

int uwritec (uiop)
struct uio *uiop;
```

Parameter

uiop Points to a **uio** structure describing the buffer from which to read a character.

Description

The **uwritec** kernel service reads a character from a buffer described by a **uio** structure. Device driver top half routines, especially character device drivers, frequently use the **uwritec** kernel service to transfer data out of a user area. The `uio_resid` and `uio_iovcnt` fields in the **uio** structure must be greater than 0 or an error is returned.

The `uio_segflg` field in the **uio** structure indicates whether the data is being read out of a user– or kernel–data area. This field also indicates whether the caller requires cross–memory operations and has provided the required cross–memory descriptors. The values for this flag are defined in the /**usr/include/sys/uio.h** file.

If the data is successfully read, the following fields in the **uio** structure are updated:

`uio_iov`	Specifies the address of the current `iovec` element to use.
`uio_xmem`	Specifies the address of the current `xmem` element to use (used for cross–memory copy).
`uio_iovcnt`	Specifies the number of remaining `iovec` elements.
`uio_iovdcnt`	Specifies the number of `iovec` elements already processed.
`uio_offset`	Specifies the character offset on the device to which data is written.
`uio_resid`	Specifies the total number of characters remaining in the data area described by the **uio** structure.
`iov_base`	Specifies the address of the next available character in the data area described by the current `iovec` element.
`iov_len`	Specifies the length of the remaining data in the buffer described by the current `iovec` element.

Execution Environment

The **uwritec** kernel service can be called from the process environment only.

Return Values

Upon successful completion, the **uwritec** service returns the character it was sent to retrieve.

–1	Indicates that the buffer is empty or the user location is not valid for one of these three reasons: • The `uio_segflg` field indicates user space and the base address (`iov_base` field) points to a location outside of the user address space. • The user does not have sufficient authority to access the location. • An I/O error occurred while the location was being accessed.

Implementation Specifics

The **uwritec** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **uiomove** kernel service, **uphysio** kernel service, **ureadc** kernel service.

vec_clear Kernel Service

Purpose

Removes a virtual interrupt handler.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

void vec_clear (levsublev)
int levsublev;
```

Parameter

levsublev	Represents the value returned by **vec_init** kernel service when the virtual interrupt handler was defined.

Description

The **vec_clear** kernel service is not part of the base kernel but is provided by the device queue management kernel extension. This queue management kernel extension must be loaded into the kernel before loading any kernel extensions referencing these services.

The **vec_clear** kernel service removes the association between a virtual interrupt handler and the virtual interrupt level and sublevel that was assigned by the **vec_init** kernel service. The virtual interrupt handler at the sublevel specified by the *levsublev* parameter no longer registers upon return from this routine.

Execution Environment

The **vec_clear** kernel service can be called from the process environment only.

Return Values

The **vec_clear** kernel service has no return values. If no virtual interrupt handler is registered at the specified sublevel, no operation is performed.

Implementation Specifics

The **vec_clear** kernel service is part of the Device Queue Management kernel extension.

Related Information

The **vec_init** kernel service.

vec_init Kernel Service

Purpose

Defines a virtual interrupt handler.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int vec_init (level, routine, arg)
int level;
void (*routine) ();
int arg;
```

Parameters

level	Specifies the virtual interrupt level. This level value is not used by the **vec_init** kernel service and implies no relative priority. However, it is returned with the sublevel assigned for the registered virtual interrupt handler.
routine	Identifies the routine to call when a virtual interrupt occurs on a given interrupt sublevel.
arg	Specifies a value that is passed to the virtual interrupt handler.

Description

The **vec_init** kernel service is not part of the base kernel but provided by the device queue management kernel extension. This queue management kernel extension must be loaded into the kernel before loading any kernel extensions referencing these services.

The **vec_init** kernel service associates a virtual interrupt handler with a level and sublevel. This service searches the available sublevels to find the first unused one. The *routine* and *arg* parameters are used to initialize the open sublevel. The **vec_init** kernel service then returns the level and assigned sublevel.

There is a maximum number of available sublevels. If this number is exceeded, the **vec_init** service halts the system. This service should be called to initialize a virtual interrupt before any device queues using the virtual interrupt are created.

The *level* parameter is not used by the **vec_init** service. It is provided for compatibility reasons only. However, its value is passed back intact with the sublevel.

Execution Environment

The **vec_init** kernel service can be called from the process environment only.

Return Values

The **vec_init** kernel service returns a value that identifies the virtual interrupt level and assigned sublevel. The low–order 8 bits of this value specify the sublevel, and the high–order 8 bits specify the level. The **attchq** kernel service uses the same format. This level value is the same value as that supplied by the *level* parameter.

Implementation Specifics

The **vec_init** kernel service is part of the Device Queue Management kernel extension.

vfsrele Kernel Service

Purpose

Releases all resources associated with a virtual file system.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int vfsrele (vfsp)
struct vfs *vfsp;
```

Parameter

vfsp	Points to a virtual file system structure.

Description

The **vfsrele** kernel service releases all resources associated with a virtual file system.

When a file system is unmounted, the **VFS_UNMOUNTED** flag is set in the **vfs** structure, indicating that it is no longer valid to do path name–related operations within the file system. When this flag is set and a **VN_RELE** v–node operation releases the last active v–node within the file system, the **VN_RELE** v–node implementation must call the **vfsrele** kernel service to complete the deallocation of the **vfs** structure.

Execution Environment

The **vfsrele** kernel service can be called from the process environment only.

Return Values

The **vfsrele** kernel service always returns a value of 0.

Implementation Specifics

The **vfsrele** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Virtual File System Overview, Virtual File System (VFS) Kernel Services, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vm_att Kernel Service

Purpose

Maps a specified virtual memory object to a region in the current address space.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>

caddr_t vm_att (vmhandle, offset)
vmhandle_t  vmhandle;
caddr_t offset;
```

Parameters

vmhandle	Specifies the handle for the virtual memory object to be mapped.
offset	Specifies the offset in the virtual memory object and region.

Description

The **vm_att** kernel service performs the following tasks:

- Selects an unallocated region in the current address space and allocates it.
- Maps the virtual memory object specified by the *vmhandle* parameter with the access permission specified in the handle.
- Constructs the address in the current address space corresponding to the offset in the virtual memory object and region.

The **vm_att** kernel service assumes an address space model of fixed–size virtual memory objects and address space regions.

Attention: If there are no more free regions, this call cannot complete and calls the **panic** kernel service.

Execution Environment

The **vm_att** kernel service can be called from either the process or interrupt environment.

Return Values

The **vm_att** kernel service returns the address that corresponds to the *offset* parameter in the address space.

Implementation Specifics

The **vm_att** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **as_geth** kernel service, **as_getsrva**l kernel service, **as_puth** kernel service, **vm_det** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

vm_cflush Kernel Service

Purpose

Flushes the processor's cache for a specified address range.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>

void vm_cflush (eaddr, nbytes)
caddr_t eaddr;
int nbytes;
```

Parameters

eaddr	Specifies the starting address of the specified range.
nbytes	Specifies the number of bytes in the address range. If this parameter is negative or 0, no lines are invalidated.

Description

The **vm_cflush** kernel service writes to memory all modified cache lines that intersect the address range (*eaddr*, *eaddr* + *nbytes* –1). The *eaddr* parameter can have any alignment in a page.

The **vm_cflush** kernel service can only be called with addresses in the system (kernel) address space.

Execution Environment

The **vm_cflush** kernel service can be called from both the interrupt and the process environment.

Return Values

The **vm_cflush** kernel service has no return values.

Implementation Specifics

The **vm_cflush** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

vm_det Kernel Service

Purpose

Unmaps and deallocates the region in the current address space that contains a given address.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>

void vm_det (eaddr)
caddr_t  eaddr;
```

Parameter

eaddr	Specifies the effective address in the current address space. The region containing this address is to be unmapped and deallocated.

Description

The **vm_det** kernel service unmaps the region containing the *eaddr* parameter and deallocates the region, adding it to the free list for the current address space.

The **vm_det** kernel service assumes an address space model of fixed–size virtual memory objects and address space regions.

Attention: If the region is not mapped, or a system region is referenced, the system will halt.

Execution Environment

The **vm_det** kernel service can be called from either the process or interrupt environment.

Implementation Specifics

The **vm_det** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **vm_att** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

vm_handle Kernel Service

Purpose

Constructs a virtual memory handle for mapping a virtual memory object with a specified access level.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>

vmhandle_t vm_handle (vmid, key)
vmid_t  vmid;
int key;
```

Parameters

vmid	Specifies a virtual memory object identifier, as returned by the **vms_create** kernel service.
key	Specifies an access key. This parameter has a 1 value for limited access and a 0 value for unlimited access, respectively.

Description

The **vm_handle** kernel service constructs a virtual memory handle for use by the **vm_att** kernel service. The handle identifies the virtual memory object specified by the *vmid* parameter and contains the access key specified by the *key* parameter.

A virtual memory handle is used with the **vm_att** kernel service to map a virtual memory object into the current address space.

The **vm_handle** kernel service assumes an address space model of fixed–size virtual memory objects and address space regions.

Execution Environment

The **vm_handle** kernel service can be called from the process environment only.

Return Values

The **vm_handle** kernel service returns a virtual memory handle type.

Implementation Specifics

The **vm_handle** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **vm_att** kernel service, **vms_create** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

vm_makep Kernel Service

Purpose

Makes a page in client storage.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>

int vm_makep (vmid, pno)
vmid_t  vmid;
int pno;
```

Parameters

vmid	Specifies the ID of the virtual memory object.
pno	Specifies the page number in the virtual memory object.

Description

The **vm_makep** kernel service makes the page specified by the *pno* parameter addressable in the virtual memory object without requiring a page–in operation. The **vm_makep** kernel service is restricted to client storage.

The page is not initialized to any particular value. It is assumed that the page is completely overwritten. If the page is already in memory, a value of 0, indicating a successful operation, is returned.

Execution Environment

The **vm_makep** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
EINVAL	Indicates a virtual memory object type or page number that is not valid.
EFBIG	Indicates that the page number exceeds the file–size limit.

Implementation Specifics

The **vm_makep** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

vm_mount Kernel Service

Purpose

Adds a file system to the paging device table.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include  <sys/vmuser.h>

int vm_mount (type, ptr, nbufstr)
int type;
int (*ptr)();
int nbufstr;
```

Parameters

type	Specifies the type of device. The *type* parameter must have a value of **D_REMOTE**.
ptr	Points to the file system's strategy routine.
nbufstr	Specifies the number of **buf** structures to use.

Description

The **vm_mount** kernel service allocates an entry in the paging device table for the file system. This service also allocates the number of **buf** structures specified by the *nbufstr* parameter for the calls to the strategy routine.

Execution Environment

The **vm_mount** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
ENOMEM	Indicates that there is no memory for the **buf** structures.
EINVAL	Indicates that the file system strategy pointer is already in the paging device table.

Implementation Specifics

The **vm_mount** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **vm_umount** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

vm_move Kernel Service

Purpose

Moves data between a virtual memory object and a buffer specified in the **uio** structure.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
#include <sys/uio.h>

int vm_move (vmid, offset, limit, rw, uio)
vmid_t vmid;
caddr_t offset;
int limit;
enum uio_rw rw;
struct uio *uio;
```

Parameters

vmid	Specifies the virtual memory object ID.
offset	Specifies the offset in the virtual memory object.
limit	Indicates the limit on the transfer length. If this parameter is negative or 0, no bytes are transferred.
rw	Specifies a read/write flag that gives the direction of the move. The possible values for this parameter (**UIO_READ**, **UIO_WRITE**) are defined in the **/usr/include/sys/uio.h** file.
uio	Points to the **uio** structure.

Description

The **vm_move** kernel service moves data between a virtual memory object and the buffer specified in a **uio** structure.

This service determines the virtual addressing required for the data movement according to the offset in the object.

The **vm_move** kernel service is similar to the **uiomove** kernel service, but the address for the trusted buffer is specified by the *vmid* and *offset* parameters instead of as a **caddr_t** address. The offset size is also limited to the size of a **caddr_t** address since virtual memory objects must be smaller than this size.

Note: The **vm_move** kernel service does not support use of cross–memory descriptors.

I/O errors for paging space and a lack of paging space are reported as signals.

Execution Environment

The **vm_move** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
EFAULT	Indicates a bad address.
ENOMEM	Indicates insufficient memory.
ENOSPC	Indicates insufficient disk space.
EIO	Indicates an I/O error.

Other file system–specific **errno** global variables are returned by the virtual file system involved in the move function.

Implementation Specifics

The **vm_move** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **uiomove** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

vm_protectp Kernel Service

Purpose

Sets the page protection key for a page range.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include  <sys/vmuser.h>

int vm_protectp (vmid, pfirst, npages, key)
vmid_t  vmid;
int pfirst;
int npages;
int key;
```

Parameters

vmid	Specifies the identifier for the virtual memory object for which the page protection key is to be set.
pfirst	Specifies the first page number in the designated page range.
npages	Specifies the number of pages in the designated page range.
key	Specifies the value to be used in setting the page protection key for the designated page range.

Description

The **vm_protectp** kernel service is called to set the storage protect key for a given page range. The *key* parameter specifies the value to which the page protection key is set. The protection key is set for all pages touched by the specified page range that are resident in memory. The **vm_protectp** kernel service applies only to client storage.

If a page is not in memory, no state information is saved from a particular call to the **vm_protectp** service. If the page is later paged–in, it receives the default page protection key.

Execution Environment

The **vm_protectp** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
EINVAL	Indicates one of the following errors:

- Invalid virtual memory object ID.
- The starting page in the designated page range is negative.
- The number of pages in the page range is negative.
- The designated page range exceeds the size of virtual memory object.
- The target page range does not exist.

Implementation Specifics

The **vm_protectp** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

vm_qmodify Kernel Service

Purpose

Determines whether a mapped file has been changed.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>

int vm_qmodify (vmid)
vmid_t vmid;
```

Parameter

vmid	Specifies the ID of the virtual memory object to check.

Description

The **vm_qmodify** kernel service performs two tests to determine if a mapped file has been changed:

- The **vm_qmodify** kernel service first checks the virtual memory object modified bit, which is set whenever a page is written out.
- If the modified bit is 0, the list of page frames holding pages for this virtual memory object are examined to see if any page frame has been modified.

If both tests are false, the **vm_qmodify** kernel service returns a value of False. Otherwise, this service returns a value of True.

If the virtual memory object modified bit was set, it is reset to 0. The page frame modified bits are not changed.

Execution Environment

The **vm_qmodify** kernel service can be called from the process environment only.

Return Values

FALSE	Indicates that the virtual memory object has not been modified.
TRUE	Indicates that the virtual memory object has been modified.

Implementation Specifics

The **vm_qmodify** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

vm_release Kernel Service

Purpose

Releases virtual memory resources for the specified address range.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include  <sys/vmuser.h>

int  vm_release (vaddr, nbytes)
caddr_t vaddr;
int nbytes;
```

Parameters

vaddr	Specifies the address of the first byte in the address range to be released.
nbytes	Specifies the number of bytes to be released.

Description

The **vm_release** kernel service releases pages that intersect the specified address range from the *vaddr* parameter to the *vaddr* parameter plus the number of bytes specified by the *nbytes* parameter. The value in the *nbytes* parameter must be nonnegative and the caller must have write access to the pages specified by the address range.

Each page that intersects the byte range is logically reset to 0, and any page frame is discarded. A page frame in I/O state is marked for discard at I/O completion. That is, the page frame is placed on the free list when the I/O operation completes.

Note: All of the pages to be released must be in the same virtual memory object.

Execution Environment

The **vm_release** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
EACCES	Indicates that the caller does not have write access to the specified pages.
EINVAL	Indicates one of the following errors: • The specified region is not mapped. • The specified region is an I/O region. • The length specified in the *nbytes* parameter is negative. • The specified address range crosses a virtual memory object boundary.

Implementation Specifics

The **vm_release** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **vm_releasep** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

vm_releasep Kernel Service

Purpose

Releases virtual memory resources for the specified page range.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include  <sys/vmuser.h>

int  vm_releasep (vmid, pfirst, npages)
vmid_t  vmid;
int pfirst;
int npages;
```

Parameters

vmid	Specifies the virtual memory object identifier.
pfirst	Specifies the first page number in the specified page range.
npages	Specifies the number of pages in the specified page range.

Description

The **vm_releasep** kernel service releases pages for the specified page range in the virtual memory object. The values in the *pfirst* and *npages* parameters must be nonnegative.

Each page of the virtual memory object that intersects the page range (*pfirst*, *pfirst* + *npages* –1) is logically reset to 0, and any page frame is discarded. A page frame in the I/O state is marked for discard at I/O completion.

For working storage, paging–space disk blocks are freed and the storage–protect key is reset to the default value.

Note: All of the pages to be released must be in the same virtual memory object.

Execution Environment

The **vm_releasep** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
EINVAL	Indicates one of the following errors: • An invalid virtual memory object ID. • The starting page is negative. • Number of pages is negative. • Page range crosses a virtual memory object boundary.

Implementation Specifics

The **vm_releasep** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **vm_release** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

vms_create Kernel Service

Purpose

Creates a virtual memory object of the specified type, size, and limits.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>


int vms_create (vmid, type, gn, size, uplim, downlim)
vmid_t *vmid;
int  type;
struct gnode *gn;
int  size;
int  uplim;
int  downlim;
```

Parameters

vmid	Points to the variable in which the virtual memory object identifier is to be stored.
type	Specifies the virtual memory object type and options as an OR of bits. The *type* parameter must have the value of **V_CLIENT**. The **V_INTRSEG** flag specifies if the process can be interrupted from a page wait on this object.
gn	Specifies the address of the g–node for client storage.
size	Specifies the current size of the file (in bytes). This can be any valid file size. If the V_LARGE is specified, it is interpreted as number of pages.
uplim	Ignored. The enforcement of file size limits is done by comparing with the **u_limit** value in the **u** block.
downlim	Ignored.

Description

The **vms_create** kernel service creates a virtual memory object. The resulting virtual memory object identifier is passed back by reference in the *vmid* parameter.

The *size* parameter is used to determine the size in units of bytes of the virtual memory object to be created. This parameter sets an internal variable that determines the virtual memory range to be processed when the virtual memory object is deleted.

An entry for the file system is required in the paging device table when the **vms_create** kernel service is called.

Execution Environment

The **vms_create** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
ENOMEM	Indicates that no space is available for the virtual memory object.
ENODEV	Indicates no entry for the file system in the paging device table.
EINVAL	Indicates incompatible or bad parameters.

Implementation Specifics

The **vms_create** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **vms_delete** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

vms_delete Kernel Service

Purpose

Deletes a virtual memory object.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>

int vms_delete (vmid)
vmid_t  vmid;
```

Parameter

vmid	Specifies the ID of the virtual memory object to be deleted.

Description

The **vms_delete** kernel service deallocates the temporary resources held by the virtual memory object specified by the *vmid* parameter and then frees the control block. This delete operation can complete asynchronously, but the caller receives a synchronous return code indicating success or failure.

Releasing Resources

The completion of the delete operation can be delayed if paging I/O is still occurring for pages attached to the object. All page frames not in the I/O state are released.

If there are page frames in the I/O state, they are marked for discard at I/O completion and the virtual memory object is placed in the iodelete state. When an I/O completion occurs for the last page attached to a virtual memory object in the iodelete state, the virtual memory object is placed on the free list.

Execution Environment

The **vms_delete** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
EINVAL	Indicates that the *vmid* parameter is not valid.

Implementation Specifics

The **vms_delete** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **vms_create** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

vms_iowait Kernel Service

Purpose

Waits for the completion of all page–out operations for pages in the virtual memory object.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>

int vms_iowait (vmid)
vmid_t vmid;
```

Parameter

vmid	Identifies the virtual memory object for which to wait.

Description

The **vms_iowait** kernel service performs two tasks. First, it determines the I/O level at which all currently scheduled page–outs are complete for the virtual memory object specified by the *vmid* parameter. Then, the **vms_iowait** service places the current process in a wait state until this I/O level has been reached.

The I/O level value is a count of page–out operations kept for each virtual memory object.

The I/O level accounts for out–of–order processing by not incrementing the I/O level for new page–out requests until all previous requests are complete. Because of this, processes waiting on different I/O levels can be awakened after a single page–out operation completes.

If the caller holds the kernel lock, the **vms_ iowait** service releases the kernel lock before waiting and reacquires it afterwards.

Execution Environment

The **vms_iowait** kernel service can be called from the process environment only.

Return Values

0	Indicates that the page–out operations completed.
EIO	Indicates that an error occurred while performing I/O.

Implementation Specifics

The **vms_iowait** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

vm_uiomove Kernel Service

Purpose

Moves data between a virtual memory object and a buffer specified in the uio structure.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
#include <sys/uio.h>

int vm_uiomove (vmid, limit, rw, uio)
vmid_t vmid;
int limit;
enum uio_rw rw;
struct uio *uio;
```

Parameters

vmid	Specifies the virtual memory object ID.
limit	Indicates the limit on the transfer length. If this parameter is negative or 0, no bytes are transferred.
rw	Specifies a read/write flag that gives the direction of the move. The possible values for this parameter (**UIO_READ**, **UIO_WRITE**) are defined in the /**usr**/**include**/**sys**/**uio.h** file.
uio	Points to the **uio** structure.

Description

The **vm_uiomove** kernel service moves data between a virtual memory object and the buffer specified in a uio structure.

This service determines the virtual addressing required for the data movement according to the offset in the object.

The **vm_uiomove** kernel service is similar to the **uiomove** kernel service, but the address for the trusted buffer is specified by the *vmid* parameter and the uio_offset field of *offset* parameters instead of as a **caddr_t** address. The offset size is a 64 bit offset_t, which allows file offsets in client segments which are greater than 2 gigabytes. **vm_uiomove** must be used instead of **vm_move** if the client filesystem supports files which are greater than 2 gigabytes.

Note: The **vm_uiomove** kernel service does not support use of cross–memory descriptors.

I/O errors for paging space and a lack of paging space are reported as signals.

Execution Environment

The **vm_uiomove** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful operation.
EFAULT	Indicates a bad address.
ENOMEM	Indicates insufficient memory.
ENOSPC	Indicates insufficient disk space.
EIO	Indicates an I/O error.

Other file system–specific **errno** global variables are returned by the virtual file system involved in the move function.

Implementation Specifics

The **vm_uiomove** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **uiomove** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

vm_umount Kernel Service

Purpose

Removes a file system from the paging device table.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>

int vm_umount (type, ptr)
int type;
int (*ptr)();
```

Parameters

type	Specifies the type of device. The *type* parameter must have a value of **D_REMOTE**.
ptr	Points to the strategy routine.

Description

The **vm_umount** kernel service waits for all I/O for the device scheduled by the pager to finish. This service then frees the entry in the paging device table. The associated **buf** structures are also freed.

Execution Environment

The **vm_umount** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
EINVAL	Indicates that a file system with the strategy routine designated by the *ptr* parameter is not in the paging device table.

Implementation Specifics

The **vm_umount** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **vm_mount** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

vm_write Kernel Service

Purpose

Initiates page–out for a page range in the address space.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include  <sys/vmuser.h>

int vm_write (vaddr, nbytes, force)
int vaddr;
int nbytes;
int force;
```

Parameters

vaddr	Specifies the address of the first byte of the page range for which a page–out is desired.
nbytes	Specifies the number of bytes starting at the byte specified by the *vaddr* parameter. This parameter must be nonnegative. All of the bytes must be in the same virtual memory object.
force	Specifies a flag indicating that a modified page is to be written regardless of when it was last written.

Description

The **vm_write** kernel service initiates page–out for pages that intersect the address range *(vaddr*, *vaddr + nbytes*).

If the *force* parameter is nonzero, modified pages are written to disk regardless of how recently they have been written.

Page–out is initiated for each modified page. An unchanged page is left in memory with its reference bit set to 0. This makes the unchanged page a candidate for the page replacement algorithm.

The caller must have write access to the specified pages.

The initiated I/O is asynchronous. The **vms_iowait** kernel service can be called to wait for I/O completion.

Execution Environment

The **vm_write** kernel service can be called from the process environment only.

Return Values

0	Indicates a successful completion.
EINVAL	Indicates one of these four errors: • A region is not defined. • A region is an I/O region. • The length specified by the *nbytes* parameter is negative. • The address range crosses a virtual memory object boundary.
EACCES	Indicates that access does not permit writing.
EIO	Indicates a permanent I/O error.

Implementation Specifics

The **vm_write** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **vm_writep** kernel service, **vms_iowait** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

vm_writep Kernel Service

Purpose

Initiates page–out for a page range in a virtual memory object.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>

int vm_writep (vmid, pfirst, npages)
vmid_t  vmid;
int pfirst;
int npages;
```

Parameters

vmid	Specifies the identifier for the virtual memory object.
pfirst	Specifies the first page number at which page–out is to begin.
npages	Specifies the number of pages for which the page–out operation is to be performed.

Description

The **vm_writep** kernel service initiates page–out for the specified page range in the virtual memory object. I/O is initiated for modified pages only. Unchanged pages are left in memory, but their reference bits are set to 0.

The caller can wait for the completion of I/O initiated by this and prior calls by calling the **vms_iowait** kernel service.

Execution Environment

The **vm_writep** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
EINVAL	Indicates any one of the following errors: • The virtual memory object ID is not valid. • The starting page is negative. • The number of pages is negative. • The page range exceeds the size of virtual memory object.

Implementation Specifics

The **vm_writep** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **vm_write** kernel service, **vms_iowait** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_free Kernel Service

Purpose

Frees a v–node previously allocated by the **vn_get** kernel service.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int vn_free (vp)
struct vnode *vp;
```

Parameter

vp	Points to the v–node to be deallocated.

Description

The **vn_free** kernel service provides a mechanism for deallocating v–node objects used within the virtual file system. The v–node specified by the *vp* parameter is returned to the pool of available v–nodes to be used again.

Execution Environment

The **vn_free** kernel service can be called from the process environment only.

Return Values

The **vn_free** service always returns 0.

Implementation Specifics

The **vn_free** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **vn_get** kernel service.

Virtual File System Overview and Virtual File System (VFS) Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_get Kernel Service

Purpose

Allocates a virtual node.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int vn_get (vfsp, gnp, vpp)
struct vfs *vfsp;
struct gnode *gnp;
struct vnode **vpp;
```

Parameters

vfsp	Points to a **vfs** structure describing the virtual file system that is to contain the v–node. Any returned v–node belongs to this virtual file system.
gnp	Points to the g–node for the object. This pointer is stored in the returned v–node. The new v–node is added to the list of v–nodes in the g–node.
vpp	Points to the place in which to return the v–node pointer. This is set by the **vn_get** kernel service to point to the newly allocated v–node.

Description

The **vn_get** kernel service provides a mechanism for allocating v–node objects for use within the virtual file system environment. A v–node is first allocated from an effectively infinite pool of available v–nodes.

Upon successful return from the **vn_get** kernel service, the pointer to the v–node pointer provided (specified by the *vpp* parameter) has been set to the address of the newly allocated v–node.

The fields in this v–node have been initialized as follows:

`v_count`	Set to 1.
`v_vfsp`	Set to the value in the *vfsp* parameter.
`v_gnode`	Set to the value in the *gnp* parameter.
`v_next`	Set to list of others v–nodes with the same g–node.

All other fields in the v–node are zeroed.

Execution Environment

The **vn_get** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
ENOMEM	Indicates that the **vn_get** kernel service could not allocate memory for the v–node. (This is a highly unlikely occurrence.)

Implementation Specifics

The **vn_get** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **vn_free** kernel service.

Virtual File System Overview and Virtual File System (VFS) Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

waitcfree Kernel Service

Purpose

Checks the availability of a free character buffer.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/cblock.h>
#include <sys/sleep.h>

int waitcfree ( )
```

Description

The **waitcfree** kernel service is used to wait for a buffer which was allocated by a previous call to the **pincf** kernel service. If one is not available, the **waitcfree** kernel service waits until either a character buffer becomes available or a signal is received.

The **waitcfree** kernel service has no parameters.

Execution Environment

The **waitfree** kernel service can be called from the process environment only.

Return Values

EVENT_SUCC	Indicates a successful operation.
EVENT_SIG	Indicates that the wait was terminated by a signal.

Implementation Specifics

The **waitcfree** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **pincf** kernel service, **putc** kernel service, **putcb** kernel service, **putcbp** kernel service, **putcf** kernel service, **putcfl** kernel service, **putcx** kernel service.

I/O Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

waitq Kernel Service

Purpose

Waits for a queue element to be placed on a device queue.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/deviceq.h>

struct req_qe *waitq (queue_id)
cba_id queue_id;
```

Parameter

queue_id	Specifies the device queue identifier.

Description

The **waitq** kernel service is not part of the base kernel but is provided by the device queue management kernel extension. This queue management kernel extension must be loaded into the kernel before loading any kernel extensions referencing these services.

The **waitq** kernel service waits for a queue element to be placed on the device queue specified by the *queue_id* parameter. This service performs these two actions:

- Waits on the event mask associated with the device queue.
- Calls the **readq** kernel service to make the most favored queue element the active one.

Processes can only use the **waitq** kernel service to wait for a single device queue. Use the **et_wait** service to wait on the occurrence of more than one event, such as multiple device queues.

The **waitq** kernel service uses the **EVENT_SHORT** form of the **et_wait** kernel service. Therefore, a signal does not terminate the wait. Use the **et_wait** kernel service if you want a signal to terminate the wait.

The **readq** kernel service can be used to read the active queue element from a queue. It does not wait for a queue element if there are none in the queue.

> **Attention:** The server must not alter any fields in the queue element or the system may halt.

Execution Environment

The **waitq** kernel service can be called from the process environment only.

Return Values

The **waitq** service returns the address of the active queue element in the device queue.

Implementation Specifics

The **waitq** kernel service is part of the Device Queue Management kernel extension.

Related Information

The **et_wait** kernel service.

w_clear Kernel Service

Purpose

Removes a watchdog timer from the list of watchdog timers known to the kernel.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/watchdog.h>

int w_clear (w)
struct watchdog *w;
```

Parameter

w	Specifies the watchdog timer structure.

Description

The watchdog timer services, including the **w_clear** kernel service, are typically used to verify that an I/O operation completes in a reasonable time.

When the **w_clear** kernel service removes the watchdog timer, the *w*–>**count** watchdog count is no longer decremented. In addition, the *w*–>**func** watchdog timer function is no longer called.

In a uniprocessor environment, the call always succeeds. This is untrue in a multiprocessor environment, where the call will fail if the watchdog timer is being handled by another processor. Therefore, the function now has a return value, which is set to 0 if successful, or –1 otherwise. Funnelled device drivers do not need to check the return value since they run in a logical uniprocessor environment. Multiprocessor–safe and multiprocessor–efficient device drivers need to check the return value in a loop. In addition, if a driver uses locking, it must release and reacquire its lock within this loop, as shown below:

```
while (w_clear(&watchdog))
  release_then_reacquire_dd_lock;
                    /* null statement if locks not used */
```

Execution Environment

The **w_clear** kernel service can be called from the process environment only.

Return Values

0	Indicates that the watchdog timer was successfully removed.
–1	Indicates that the watchdog timer could not be removed.

Implementation Specifics

The **w_clear** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **w_init** kernel service, **w_start** kernel service, **w_stop** kernel service.

Timer and Time–of–Day Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

w_init Kernel Service

Purpose

Registers a watchdog timer with the kernel.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/watchdog.h>

int w_init (w)
struct watchdog *w;
```

Parameter

w	Specifies the watchdog timer structure.

Description

Attention: The watchdog structure must be pinned when the **w_init** service is called. It must remain pinned until after the call to the **w_clear** service. During this time, the watchdog structure must not be altered except by the watchdog services.

The watchdog timer services, including the **w_init** kernel service, are typically used to verify that an I/O operation completes in a reasonable time. The watchdog timer is initialized to the stopped state and must be started using the **w_start** service.

In a uniprocessor environment, the call always succeeds. This is untrue in a multiprocessor environment, where the call will fail if the watchdog timer is being handled by another processor. Therefore, the function now has a return value, which is set to 0 if successful, or –1 otherwise. Funnelled device drivers do not need to check the return value since they run in a logical uniprocessor environment. Multiprocessor–safe and multiprocessor–efficient device drivers need to check the return value in a loop. In addition, if a driver uses locking, it must release and reacquire its lock within this loop, as shown below:

```
while (w_init(&watchdog))
  release_then_reacquire_dd_lock;
                  /* null statement if locks not used */
```

The calling parameters for the watchdog timer function are:

```
void func (w)
struct watchdog *w;
```

Execution Environment

The **w_init** kernel service can be called from the process environment only.

Return Values

0	Indictates that the watchdog structure was successfully initialized.
–1	Indicates that the watchdog structure could not be initialized.

Implementation Specifics

The **w_init** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **w_clear** kernel service, **w_start** kernel service, **w_stop** kernel service.

Timer and Time–of–Day Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

w_start Kernel Service

Purpose

Starts a watchdog timer.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/watchdog.h>

void w_start (w)
struct watchdog *w;
```

Parameter

w	Specifies the watchdog timer structure.

Description

The watchdog timers, including the **w_start** kernel service, are typically used to verify that an I/O operation completes in a reasonable time. The **w_start** and **w_stop** kernel services are designed to allow the timer to be started and stopped efficiently. The kernel decrements the *w*–>**count** watchdog count every second. The kernel calls the *w*–>**func** watchdog timer function when the *w*–>**count** watchdog count reaches 0. A watchdog timer is ignored when the *w*–>**count** watchdog count is less than or equal to 0.

The **w_start** kernel service sets the *w*–>**count** watchdog count to a value of *w*–>**restart**.

Attention: The watchdog structure must be pinned when the **w_start** kernel service is called. It must remain pinned until after the call to the **w_clear** kernel service. During this time, the watchdog structure must not be altered except by the watchdog services.

Execution Environment

The **w_start** kernel service can be called from the process and interrupt environments.

Return Values

The **w_start** kernel service has no return values.

Implementation Specifics

The **w_start** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **w_clear** kernel service, **w_init** kernel service, **w_stop** kernel service.

Timer and Time–of–Day Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

w_stop Kernel Service

Purpose

Stops a watchdog timer.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/watchdog.h>

void w_stop (w)
struct watchdog *w;
```

Parameter

w	Specifies the watchdog timer structure.

Description

The watchdog timer services, including the **w_stop** kernel service, are typically used to verify that an I/O operation completes in a reasonable time. The **w_start** and **w_stop** kernel services are designed to allow the timer to be started and stopped efficiently. The kernel decrements the *w*–>**count** watchdog count every second. The kernel calls the *w*–>**func** watchdog timer function when the *w*–>**count** watchdog count reaches 0. A watchdog timer is ignored when *w*–>**count** is less than or equal to 0.

Attention: The watchdog structure must be pinned when the **w_stop** kernel service is called. It must remain pinned until after the call to the **w_clear** kernel service. During this time, the watchdog structure must not be altered except by the watchdog services.

Execution Environment

The **w_stop** kernel service can be called from the process and interrupt environments.

Return Values

The **w_stop** kernel service has no return values.

Implementation Specifics

The **w_stop** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **w_clear** kernel service, **w_init** kernel service, **w_start** kernel service.

Timer and Time–of–Day Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

xmalloc Kernel Service

Purpose

Allocates memory.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/malloc.h>

caddr_t xmalloc (size, align, heap)
int size;
int align;
caddr_t heap;
```

Parameters

size	Specifies the number of bytes to allocate.
align	Specifies the alignment characteristics for the allocated memory.
heap	Specifies the address of the heap from which the memory is to be allocated.

Description

The **xmalloc** kernel service allocates an area of memory out of the heap specified by the *heap* parameter. This area is the number of bytes in length specified by the *size* parameter and is aligned on the byte boundary specified by the *align* parameter. The *align* parameter is actually the log base 2 of the desired address boundary. For example, an *align* value of 4 requests that the allocated area be aligned on a 2^4 (16) byte boundary.

Two heaps are provided in the kernel segment for use by kernel extensions. The kernel extensions should use the **kernel_heap** value when allocating memory that is not pinned. They should also use the **pinned_heap** value when allocating memory that is pinned. In particular, the **pinned_heap** value should be specified when allocating memory that is to be always pinned or pinned for long periods of time. The memory is pinned upon successful return from the **xmalloc** kernel service. When allocating memory that can be pageable (or only pinned for short periods of time), the **kernel_heap** value should be specified. The **pin** and **unpin** kernel services should be used to pin and unpin memory from the heap when required.

Kernel extensions can use these services to allocate memory out of the kernel heaps. For example, the **xmalloc** (**128**,**3**,**kernel_heap**) kernel service allocates a 128–byte double word aligned area out of the kernel heap.

A kernel extension must use the **xmfree** kernel service to free the allocated memory. If it does not, subsequent allocations eventually are unsuccessful.

The **xmalloc** kernel service has two compatibility interfaces: **malloc** and **palloc**.

Execution Environment

The **xmalloc** kernel service can be called from the process environment only.

Return Values

Upon successful completion, the **xmalloc** kernel service returns the address of the allocated area. A null pointer is returned under the following circumstances:

- The requested memory cannot be allocated.
- The heap has not been initialized for memory allocation.

Implementation Specifics

The **xmalloc** kernel service is part of Base Operating System (BOS) Runtime.

The following additional interfaces to the **xmalloc** kernel service are provided:

- **malloc** (*size*) is equivalent to **xmalloc** (*size*, **0**, **kernel_heap**).
- **palloc** (*size*, *align*) is equivalent to **xmalloc** (*size*, *align*, **kernel_heap**).

Related Information

The **xmfree** kernel service.

Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

xmattach Kernel Service

Purpose

Attaches to a user buffer for cross–memory operations.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/xmem.h>

int xmattach (addr, count, dp, segflag)
char *addr;
int count;
struct xmem *dp;
int segflag;
```

Parameters

addr	Specifies the address of the user buffer to be accessed in a cross–memory operation.
count	Indicates the size of the user buffer to be accessed in a cross–memory operation.
dp	Specifies a cross–memory descriptor. The *dp*–>**aspace_id** variable must be set to a value of **XMEM_INVAL**.
segflag	Specifies a segment flag. This flag is used to determine the address space of the memory that the cross–memory descriptor applies to. The valid values for this flag can be found in the /**usr**/**include**/**xmem.h** file.

Description

The **xmattach** kernel service prepares the user buffer so that a device driver can access it without executing under the process that requested the I/O operation. A device top–half routine calls the **xmattach** kernel service. The **xmattach** kernel service allows a kernel process or device bottom–half routine to access the user buffer with the **xmemin** or **xmemout** kernel services. The device driver must use the **xmdetach** kernel service to inform the kernel when it has finished accessing the user buffer.

The kernel remembers which segments are attached for cross–memory operations. Resources associated with these segments cannot be freed until all cross–memory descriptors have been detached. ”Cross Memory Kernel Services” in Memory Kernel Services in in *AIX Kernel Extensions and Device Support Programming Concepts* describes how the cross–memory kernel services use cross–memory descriptors.

Note: When the **xmattach** kernel service remaps user memory containing the cross–memory buffer, the effects are machine–dependent. Also, cross–memory descriptors are not inherited by a child process.

Execution Environment

The **xmattach** kernel service can be called from the process environment only.

Return Values

XMEM_SUCC Indicates a successful operation.

XMEM_FAIL Indicates one of the following errors:

- The buffer size indicated by the *count* parameter is less than or equal to 0.
- The cross–memory descriptor is in use (*dp*–>**aspace_id != XMEM_INVAL**).
- The area of memory indicated by the *addr* and *count* parameters is not defined.

Implementation Specifics

The **xmattach64** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **uphysio** kernel service, **xmdetach** kernel service, **xmattach64** kernel service, **xmemin** kernel service, and **xmemout** kernel service.

Cross Memory Kernel Services, and Memory Kernel Services.

xmattach64 Kernel Service

Purpose

Attaches to a user buffer for cross–memory operations.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/xmem.h>

int xmattach64 ( addr64, count, dp, segflag )
unsigned long long  addr64 ;
 int count;
struct xmem *dp;
int segflags;
```

Parameters

addr64	Specifies the address of the user buffer to be accessed in a cross–memory operation.
count	Indicates the size of the user buffer to be accessed in a cross–memory operation.
dp	Specifies a cross–memory descriptor. The dp–>aspace_id variable must be set to a value of **XMEM_INVAL**.
segflag	Specifies a segment flag. This flag is used to determine the address space of the memory that the cross–memory descriptor applies to. The valid values for this flag can be found in the /**usr**/**include**/**xmem.h** file.

Description

The **xmattach64** kernel service prepares the user buffer so that a device driver can access it without executing under the process that requested the I/O operation. A device top–half routine calls the **xmattach64** kernel service. The **xmattach64** kernel service allows a kernel process or device bottom–half routine to access the user buffer with the **xmemin** or **xmemout** kernel services. The device driver must use the **xmdetach** kernel service to inform the kernel when it has finished accessing the user buffer. The kernel remembers which segments are attached for cross–memory operations. Resources associated with these segments cannot be freed until all cross–memory descriptors have been detached. See ”Cross Memory Kernel Services” in Memory Kernel Services

The address of the buffer to attach to: addr64, is interpreted as being either a 64–bit unremapped address, or a 32–bit unremapped address, as a function of both whether the current user–address space is 64 or 32–bits, and the input *segflag* parameter.

The input addr64 is interpreted to be a 64–bit address (in user space), if and only if, all of the following conditions apply:

- Input segflag is **USER_ADSPACE** or **USERI_ADSPACE** (and)
- Current user process address space is 64–bits.

In all other cases, the input address (addr64), is treated as a 32–bit unremapped address.

Execution Environment

The **xmattach64** kernel service can be called from the process environment only.

Return Values

XMEM_SUCC	Indicates a successful operation.
XMEM_FAIL	Indicates one of the following errors:

1. The buffer size indicated by the *count* parameter is less than or equal to 0.
2. The cross–memory descriptor is in use (dp–>aspace_id != XMEM_INVAL).
3. The area of memory indicated by the *addr64* and *count* parameters is not defined.
4. The buffer crosses more than one segment boundary.

Implementation Specifics

The **xmattach64** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **uphysio** kernel service, **xmdetach** kernel service, **xmattach** kernel service, **xmemin** kernel service, and **xmemout** kernel service. Cross Memory Kernel Services and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

xmdetach Kernel Service

Purpose

Detaches from a user buffer used for cross–memory operations.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/xmem.h>

int xmdetach (dp)
struct xmem *dp;
```

Parameter

dp	Points to a cross–memory descriptor initialized by the **xmattach** kernel service.

Description

The **xmdetach** kernel service informs the kernel that a user buffer can no longer be accessed. This means that some previous caller, typically a device driver bottom half or a kernel process, is no longer permitted to do cross–memory operations on this buffer. Subsequent calls to either the **xmemin** or **xmemout** kernel service using this cross–memory descriptor result in an error return. The cross–memory descriptor is set to *dp*–>**aspace_id = XMEM_INVAL** so that the descriptor can be used again. ”Cross Memory Kernel Services” in Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts* describes how the cross–memory kernel services use cross–memory descriptors.

Execution Environment

The **xmdetach** kernel service can be called from either the process or interrupt environment.

Return Values

XMEM_SUCC	Indicates successful completion.
XMEM_FAIL	Indicates that the descriptor was not valid or the buffer was not defined.

Implementation Specifics

The **xmdetach** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **xmattach** kernel service, **xmemin** kernel service, **xmemout** kernel service.

Cross Memory Kernel Services and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

xmemdma Kernel Service

Purpose

Prepares a page for direct memory access (DMA) I/O or processes a page after DMA I/O is complete.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/xmem.h>

int xmemdma (xp, xaddr, flag)
struct xmem *xp;
caddr_t xaddr;
int flag;
```

Parameters

xp Specifies a cross–memory descriptor.

xaddr Identifies the address specifying the page for transfer.

flag Specifies whether to prepare a page for DMA I/O or process it after DMA I/O is complete. Possible values are:

XMEM_ACC_CHK
Performs access checking on the page. When this flag is set, the page protection attributes are verified.

XMEM_HIDE Prepares the page for DMA I/O. For cache–inconsistent platforms, this preparation includes hiding the page by making it inaccessible.

XMEM_UNHIDE
Processes the page after DMA I/O. Also, this flag reveals the page and makes it accessible for cache–inconsistent platforms.

XMEM_WRITE_ONLY
Marks the intended transfer as outbound only. This flag is used with **XMEM_ACC_CHK** to indicate that read–only access to the page is sufficient.

Description

The **xmemdma** kernel service operates on the page specified by the *xaddr* parameter in the region specified by the cross–memory descriptor. If the cross–memory descriptor is for the kernel, the *xaddr* parameter specifies a kernel address. Otherwise, the *xaddr* parameter specifies the offset in the region described in the cross–memory descriptor.

The **xmemdma** kernel service is provided for machines that have processor–memory caches, but that do not perform DMA I/O through the cache. Device handlers for Micro Channel DMA devices use the **d_master** service and **d_complete** kernel service instead of the **xmemdma** kernel service.

If the *flag* parameter indicates **XMEM_HIDE** (that is, **XMEM_UNHIDE** is not set) and this is the first hide for the page, the **xmemdma** kernel service prepares the page for DMA I/O by flushing the cache and making the page invalid. When the **XMEM_UNHIDE** bit is set and this is the last unhide for the page, the following events take place:

1. The page is made valid.

   If the page is not in pager I/O state:

2. Any processes waiting on the page are readied.
3. The modified bit for the page is set unless the page has a read–only storage key.

The page is made not valid during DMA operations so that it is not addressable with any virtual address. This prevents any process from reading or loading any part of the page into the cache during the DMA operation.

The page specified must be in memory and must be pinned.

If the **XMEM_ACC_CHK** bit is set, then the **xmemdma** kernel service also verifies access permissions to the page. If the page access is read–only, then the **XMEM_WRITE_ONLY** bit must be set in the *flag* parameter.

Notes:

1. On the PowerPC platform, which is cache–consistent, the **xmemdma** kernel service does not hide or reveal the page nor does it perform any cache flushing. Instead, on the PowerPC platform, the service's primary function is for real–address translation.
2. This service is not supported for large–memory systems with greater than 4GB of physical memory addresses. For such systems, **xmemdma64** should be used.

Execution Environment

The **xmemdma** kernel service can be called from either the process or interrupt environment.

Return Values

On successful completion, the **xmemdma** service returns the real address corresponding to the *xaddr* and *xp* parameters.

Error Codes

The **xmemdma** kernel service returns a value of **XMEM_FAIL** if one of the following are true:

- The descriptor was invalid.
- The page specified by the *xaddr* or *xp* parameter is invalid.
- Access is not allowed to the page.

Implementation Specifics

The **xmemdma** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Cross Memory Kernel Services and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

Understanding Direct Memory Access (DMA) Transfer.

xmemdma64 Kernel Service

Purpose

Prepares a page for direct memory access (DMA) I/O or processes a page after DMA I/O is complete.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/xmem.h>

unsigned long long xmemdma64 (
struct xmem *dp,
caddr_t xaddr,>
int flags)
```

Parameters

dp Specifies a cross–memory descriptor.

xaddr Identifies the address specifying the page for transfer.

flags Specifies whether to prepare a page for DMA I/O or process it after DMA I/O is complete. Possible values are:

XMEM_HIDE Prepares the page for DMA I/O. If cache–inconsistent, then the data cache is flushed, the memory page is hidden, and the real page address is returned. If cache–consistent, then the modified bit is set and the real address of the page is returned.

XMEM_UNHIDE Processes the page after DMA I/O. Also, this flag reveals the page, readies any processes waiting on the page, and sets the modified bit accordingly.

XMEM_ACC_CHK Performs access checking on the page. When this flag is set, the page protection attributes are verified.

XMEM_WRITE_ONLY Marks the intended transfer as outbound only. This flag is used with **XMEM_ACC_CHK** to indicate that read–only access to the page is sufficient.

Description

The **xmemdma64** kernel service operates on the page specified by the *xaddr* parameter in the region specified by the cross–memory descriptor. If the cross–memory descriptor is for the kernel, the *xaddr* parameter specifies a kernel address. Otherwise, the *xaddr* parameter specifies the offset in the region described in the cross–memory descriptor.

The **xmemdma64** kernel service is provided for machines that have processor–memory caches, but that do not perform DMA I/O through the cache. Device handlers for Micro Channel DMA devices use the **d_master** service and **d_complete** kernel service instead of the **xmemdma64** kernel service.

If the *flag* parameter indicates **XMEM_HIDE** (that is, **XMEM_UNHIDE** is not set) and this is the first hide for the page, the **xmemdma64** kernel service prepares the page for DMA I/O by flushing the cache and making the page invalid. When the **XMEM_UNHIDE** bit is set and this is the last unhide for the page, the following events take place:

1. The page is made valid.

   If the page is not in pager I/O state:

2. Any processes waiting on the page are readied.
3. The modified bit for the page is set unless the page has a read–only storage key.

The page is made not valid during DMA operations so that it is not addressable with any virtual address. This prevents any process from reading or loading any part of the page into the cache during the DMA operation.

The page specified must be in memory and must be pinned.

If the **XMEM_ACC_CHK** bit is set, then the **xmemdma64** kernel service also verifies access permissions to the page. If the page access is read–only, then the **XMEM_WRITE_ONLY** bit must be set in the *flag* parameter.

Note: On the PowerPC platform, which is cache–consistent, the **xmemdma64** kernel service does not hide or reveal the page nor does it perform any cache flushing. Instead, on the PowerPC platform, the service's primary function is for real–address translation.

Execution Environment

The **xmemdma64** kernel service can be called from either the process or interrupt environment.

Return Values

On successful completion, the **xmemdma64** service returns the real address corresponding to the *xaddr* and *xp* parameters.

Error Codes

The **xmemdma64** kernel service returns a value of **XMEM_FAIL** if one of the following are true:

- The descriptor was invalid.
- The page specified by the *xaddr* or *xp* parameter is invalid.
- Access is not allowed to the page.

Implementation Specifics

The **xmemdma64** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

Cross Memory Kernel Services and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

Understanding Direct Memory Access (DMA) Transfer.

xmempin Kernel Service

Purpose

Pins the specified address range in user or system memory.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/uio.h>

int xmempin(base, len, xd)
caddr_t base;
int len;
struct xmem *xd;
```

Parameters

base	Specifies the address of the first byte to pin.
len	Indicates the number of bytes to pin.
xd	Specifies the cross–memory descriptor.

Description

The **xmempin** kernel service is used to pin pages backing a specified memory region which is defined in either system or user address space. Pinning a memory region prohibits the pager from stealing pages from the pages backing the pinned memory region. Once a memory region is pinned, accessing that region does not result in a page fault until the region is subsequently unpinned.

The **pinu** kernel service will not work on a mapped file.

The cross–memory descriptor must have been filled in correctly prior to the **xmempin** call (for example, by calling the **xmattach** kernel service). If the caller does not have a valid cross–memory descriptor, the **pinu** and **unpinu** kernel services must be used. The **xmempin** and **xmemunpin** kernel services have shorter pathlength than the **pinu** and **unpinu** kernel services.

Execution Environment

The **xmempin** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
EFAULT	Indicates that the memory region as specified by the *base* and *len* parameters is not within the address space specified by the *xd* parameter.
EINVAL	Indicates that the value of the length parameter is negative or 0. Otherwise, the area of memory beginning at the byte specified by the *base* parameter and extending for the number of bytes specified by the *len* parameter is not defined.
ENOMEM	Indicates that the **xmempin** kernel service is unable to pin the region due to insufficient real memory or because it has exceeded the systemwide pin count.

Implementation Specifics

The **xmempin** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **pin** kernel service, **unpin** kernel service, **pinu** kernel service, **xmemunpin** kernel service.

Understanding Execution Environments and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

xmemunpin Kernel Service

Purpose

Unpins the specified address range in user or system memory.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/uio.h>

int xmemunpin (base, len, xd)
caddr_t base;
int len;
struct xmem *xd;
```

Parameters

base	Specifies the address of the first byte to unpin.
len	Indicates the number of bytes to unpin.
xd	Specifies the cross–memory descriptor.

Description

The **xmemunpin** kernel service unpins a region of memory previously pinned by the **pinu** kernel service. When the pin count is 0, the page is not pinned and can be paged out of real memory. Upon finding an unpinned page, the **xmemunpin** kernel service returns the **EINVAL** error code and leaves any remaining pinned pages still pinned.

The **xmemunpin** service should be used where the address space might be in either user or kernel space.

The cross–memory descriptor must have been filled in correctly prior to the **xmempin** call (for example, by calling the **xmattach** kernel service). If the caller does not have a valid cross–memory descriptor, the **pinu** and **unpinu** kernel services must be used. The **xmempin** and **xmemunpin** kernel services have shorter pathlength than the **pinu** and **unpinu** kernel services.

Execution Environment

The **xmemunpin** kernel service can be called in the process environment when unpinning data that is in either user space or system space. It can be called in the interrupt environment only when unpinning data that is in system space.

Return Values

0	Indicates successful completion.
EFAULT	Indicates that the memory region as specified by the *base* and *len* parameters is not within the address specified by the *xd* parameter.
EINVAL	Indicates that the value of the length parameter is negative or 0. Otherwise, the area of memory beginning at the byte specified by the *base* parameter and extending for the number of bytes specified by the *len* parameter is not defined. If neither cause is responsible, an unpinned page was specified.

Implementation Specifics

The **xmemunpin** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **pin** kernel service, **unpin** kernel service, **pinu** kernel service, **unpinu** kernel service, **xmempin** kernel service.

Understanding Execution Environments and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

xmemin Kernel Service

Purpose

Performs a cross–memory move by copying data from the specified address space to kernel global memory.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/xmem.h>

int xmemin (uaddr, kaddr, count, dp)
caddr_t *uaddr;
caddr_t *kaddr;
int count;
struct xmem *dp;
```

Parameters

uaddr	Specifies the address in memory specified by a cross–memory descriptor.
kaddr	Specifies the address in kernel memory.
count	Specifies the number of bytes to copy.
dp	Specifies the cross–memory descriptor.

Description

The **xmemin** kernel service performs a cross–memory move. A cross–memory move occurs when data is moved to or from an address space other than the address space that the program is executing in. The **xmemin** kernel service copies data from the specified address space to kernel global memory.

The **xmemin** kernel service is provided so that kernel processes and interrupt handlers can safely access a buffer within a user process. Calling the **xmattach** kernel service prepares the user buffer for the cross–memory move.

The **xmemin** kernel service differs from the **copyin** and **copyout** kernel services in that it is used to access a user buffer when not executing under the user process. In contrast, the **copyin** and **copyout** kernel services are used only to access a user buffer while executing under the user process.

Execution Environment

The **xmemin** kernel service can be called from either the process or interrupt environment.

Return Values

XMEM_SUCC Indicates successful completion.

XMEM_FAIL Indicates one of the following errors:

- The user does not have the appropriate access authority for the user buffer.
- The user buffer is located in an address range that is not valid.
- The segment containing the user buffer has been deleted.
- The cross–memory descriptor is not valid.
- A paging I/O error occurred while the user buffer was being accessed.

If the user buffer is not in memory, the **xmemin** kernel service also returns an **XMEM_FAIL** error when executing on an interrupt level.

Implementation Specifics

The **xmemin** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **xmattach** kernel service, **xmdetach** kernel service, **xmemout** kernel service.

Cross Memory Kernel Services and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

xmemout Kernel Service

Purpose

Performs a cross–memory move by copying data from kernel global memory to a specified address space.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/xmem.h>

int xmemout (kaddr, uaddr, count, dp)
caddr_t *kaddr;
caddr_t *uaddr;
int count;
struct xmem *dp;
```

Parameters

kaddr	Specifies the address in kernel memory.
uaddr	Specifies the address in memory specified by a cross–memory descriptor.
count	Specifies the number of bytes to copy.
dp	Specifies the cross–memory descriptor.

Description

The **xmemout** kernel service performs a cross–memory move. A cross–memory move occurs when data is moved to or from an address space other than the address space that the program is executing in. The **xmemout** kernel service copies data from kernel global memory to the specified address space.

The **xmemout** kernel service is provided so that kernel processes and interrupt handlers can safely access a buffer within a user process. Calling the **xmattach** kernel service prepares the user buffer for the cross–memory move.

The **xmemout** kernel service differs from the **copyin** and **copyout** kernel services in that it is used to access a user buffer when not executing under the user process. In contrast, the **copyin** and **copyout** kernel services are only used to access a user buffer while executing under the user process.

Execution Environment

The **xmemout** kernel service can be called from either the process or interrupt environment.

Return Values

XMEM_SUCC Indicates successful completion.

XMEM_FAIL Indicates one of the following errors:

- The user does not have the appropriate access authority for the user buffer.
- The user buffer is located in an address range that is not valid.
- The segment containing the user buffer has been deleted.
- The cross–memory descriptor is not valid.
- A paging I/O error occurred while the user buffer was being accessed.

If the user buffer is not in memory, the **xmemout** service also returns an **XMEM_FAIL** error when executing on an interrupt level.

Implementation Specifics

The **xmemout** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **xmattach** kernel service, **xmdetach** kernel service, **xmemin** kernel service.

Cross Memory Kernel Services and Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

xmfree Kernel Service

Purpose

Frees allocated memory.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/malloc.h>

int xmfree (ptr, heap)
caddr_t ptr;
caddr_t heap;
```

Parameters

ptr	Specifies the address of the area in memory to free.
heap	Specifies the address of the heap from which the memory was allocated.

Description

The **xmfree** kernel service frees the area of memory pointed to by the *ptr* parameter in the heap specified by the *heap* parameter. This area of memory must be allocated with the **xmalloc** kernel service. In addition, the *ptr* pointer must be the pointer returned from the corresponding **xmalloc** call.

For example, the **xmfree** (*ptr*, **kernel_heap**) kernel service frees the area in the kernel heap allocated by *ptr*=**xmalloc** (*size*, *align*, **kernel_heap**).

A kernel extension must explicitly free any memory it allocates. If it does not, eventually subsequent allocations are unsuccessful. Pinned memory must also be unpinned before it is freed if allocated from the **kernel_heap**. The kernel does not keep track of which kernel extension owns various allocated areas in the heap. Therefore, the kernel never automatically frees these allocated areas on process termination or device close.

An additional interface to the **xmfree** kernel service is provided. The **free** (*ptr*) is equivalent to **xmfree** (*ptr*, **kernel_heap**).

Execution Environment

The **xmfree** kernel service can be called from the process environment only.

Return Values

0	Indicates successful completion.
–1	Indicates one of the following errors: • The area to be freed was not allocated with the **xmalloc** kernel service. • The heap was not initialized for memory allocation.

Implementation Specifics

The **xmfree** kernel service is part of Base Operating System (BOS) Runtime.

Related Information

The **xmalloc** kernel service.

Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

Chapter 2. Device Driver Operations

Standard Parameters to Device Driver Entry Points

Description

There are three parameters passed to device driver entry points that always have the same meanings: the *devno* parameter, the *chan* parameter, and the *ext* parameter.

The devno Parameter

This value, defined to be of type **dev_t**, specifies the device or subdevice to which the operation is directed. For convenience and portability, the /**usr**/**include**/**sys**/**sysmacros.h** file defines the following macros for manipulating device numbers:

major(*devno***)**	Returns the major device number.
minor(*devno***)**	Returns the minor device number.
makedev(*maj***,** *min***)**.	Constructs a composite device number in the format of *devno* from the major and minor device numbers given.

The chan Parameter

This value, defined to be of type **chan_t**, is the channel ID for a multiplexed device driver. If the device driver is not multiplexed, *chan* has the value of 0. If the driver is multiplexed, then the *chan* parameter is the **chan_t** value returned from the device driver's **ddmpx** routine.

The ext Parameter

The *ext* parameter, or extension parameter, is defined to be of type **int**. It is meaningful only with calls to such extended subroutines as the **openx**, **readx**, **writex**, and **ioctlx** subroutines. These subroutines allow applications to pass an extra, device–specific parameter to the device driver. This parameter is then passed to the **ddopen**, **ddread**, **ddwrite**, and **ddioctl** device driver entry points as the *ext* parameter. If the application uses one of the non–extended subroutines (for example, the **read** instead of the **readx** subroutine), then the *ext* parameter has a value of 0.

Note: Using the *ext* parameter is highly discouraged because doing so makes an application program less portable to other operating systems.

Related Information

The **ddioctl** device driver entry point, **ddmpx** device driver entry point, **ddopen** device driver entry point, **ddread** device driver entry point, **ddwrite** device driver entry point.

The **close** subroutine, **ioctl** subroutine, **lseek** subroutine, **open** subroutine, **read** subroutine, **write** subroutine.

Device Driver Kernel Extension Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

Programming in the Kernel Environment Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

buf Structure

Introduction to Kernel Buffers

For block devices, kernel buffers are used to buffer data transfers between a program and the peripheral device. These buffers are allocated in blocks of 4096 bytes. At any given time, each memory block is a member of one of two linked lists that the device driver and the kernel maintain:

Available buffer queue (avlist)	A list of all buffers available for use. These buffers do not contain data waiting to be transferred to or from a device.
Busy buffer queue (blist)	A list of all buffers that contain data waiting to be transferred to or from a device.

Each buffer has an associated buffer header called the **buf** structure pointing to it. Each buffer header has several parts:

- Information about the block
- Flags to show status information
- Busy list forward and backward pointers
- Available list forward and backward pointers

The device driver maintains the `av_forw` and `av_back` pointers (for the available blocks), while the kernel maintains the `b_forw` and `b_back` pointers (for the busy blocks).

buf Structure Variables for Block I/O

The **buf** structure, which is defined in the **/usr/include/sys/buf.h** file, includes the following fields:

`b_flags` Flag bits. The value of this field is constructed by logically ORing 0 or more of the following values:

B_WRITE	This operation is a write operation.
B_READ	This operation is a read data operation, rather than write.
B_DONE	I/O on the buffer has been done, so the buffer information is more current than other versions.
B_ERROR	A transfer error has occurred and the transaction has aborted.
B_BUSY	The block is not on the free list.
B_INFLIGHT	This I/O request has been sent to the physical device driver for processing.
B_AGE	The data is not likely to be reused soon, so prefer this buffer for reuse. This flag suggests that the buffer goes at the head of the free list rather than at the end.
B_ASYNC	Asynchronous I/O is being performed on this block. When I/O is done, release the block.
B_DELWRI	The contents of this buffer still need to be written out before the buffer can be reused, even though this block may be on the free list. This is used by the **write** subroutine when the system expects another write to the same block to occur soon.
B_NOHIDE	Indicates that the data page should not be hidden during direct memory access (DMA) transfer.
B_STALE	The data conflicts with the data on disk because of an I/O error.
B_MORE_DONE	When set, indicates to the receiver of this **buf** structure that more structures are queued in the **IODONE** level. This permits device drivers to handle all completed requests before processing any new requests.
B_SPLIT	When set, indicates that the transfer can begin anywhere within the data buffer.

`b_forw` The forward busy block pointer.

`b_back` The backward busy block pointer.

`av_forw` The forward pointer for a driver request queue.

`av_back` The backward pointer for a driver request queue.

`b_iodone` Anyone calling the strategy routine must set this field to point to their I/O done routine. This routine is called on the **INTIODONE** interrupt level when I/O is complete.

`b_dev` The major and minor device number.

`b_bcount` The byte count for the data transfer.

`b_un.b_addr` The memory address of the data buffer.

`b_blkno` The block number on the device.

`b_resid` Amount of data not transferred after error.

`b_event` Anchor for event list.

`b_xmemd` Cross–memory descriptor.

Related Information

The **ddstrategy** device driver entry point.

The **write** subroutine.

Device Driver Kernel Extension Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

Programming in the Kernel Environment Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

Cross Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

Character Lists Structure

Character device drivers, and other character–oriented support that can perform character–at–a–time I/O, can be implemented by using a common set of services and data buffers to handle characters in the form of *character lists*. A *character list* is a list or queue of characters. Some routines put characters in a list, and others remove the characters from the list.

Character lists, known as **clists**, contain a **clist** header and a chain of one or more data buffers known as character blocks**.** Putting characters on a queue allocates space (character blocks) from the common pool and links the character block into the data structure defining the character queue. Obtaining characters from a queue returns the corresponding space back to the pool.

A character list can be used to communicate between a character device driver top and bottom half. The **clist** header and the character blocks that are used by these routines must be pinned in memory, since they are accessed in the interrupt environment.

Users of the character list services must register (typically in the device driver **ddopen** routine) the number of character blocks to be used at any one time. This allows the kernel to manage the number of pinned character blocks in the character block pool. Similarly, when usage terminates (for example, when the device driver is closed), the using routine should remove its registration of character blocks. The **pincf** kernel service provides registration for character block usage.

The kernel provides four services for obtaining characters or character blocks from a character list: the **getc**, **getcb**, **getcbp**, and **getcx** kernel services. There are also four services that add characters or character blocks to character lists: the **putc**, **putcb**, **putcbp**, and **putcx** kernel services. The **getcf** kernel services allocates a free character block while the **putcf** kernel service returns a character block to the free list. Additionally, the **putcfl** kernel service returns a list of character buffers to the free list. The **waitcfree** kernel service determines if any character blocks are on the free list, and waits for one if none are available.

Using a Character List

For each character list you use, you must allocate a **clist** header structure. This **clist** structure is defined in the **/usr/include/sys/cblock.h** file.

You do not need to be concerned with maintaining the fields in the **clist** header, as the character list services do this for you. However, you should initialize the `c_cc` count field to 0, and both character block pointers (`c_cf` and `c_cl`) to null before using the **clist** header for the first time. The **clist** structure defines these fields.

Each buffer in the character list is a **cblock** structure, which is also defined in the **/usr/include/sys/cblock.h** file.

A character block data area does not need to be completely filled with characters. The **c_first** and **c_last** fields are zero–based offsets within the **c_data** array, which actually contains the data.

Only a limited amount of memory is available for character buffers. All character drivers share this pool of buffers. Therefore, you must limit the number of characters in your character list to a few hundred. When the device is closed, the device driver should make certain all of its character lists are flushed so the buffers are returned to the list of free buffers.

Related Information

The **getc** kernel service, **getcb** kernel service, **getcbp** kernel service, **getcf** kernel service, **getcx** kernel service, **pincf** kernel service, **putc** kernel service, **putcb** kernel service, **putcbp** kernel service, **putcf** kernel service, **putcfl** kernel service, **putcx** kernel service, **waitcfree** kernel service.

Device Driver Kernel Extension Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

Programming in the Kernel Environment Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

uio Structure

Introduction

The user I/O or **uio** structure is a data structure describing a memory buffer to be used in a data transfer. The **uio** structure is most commonly used in the read and write interfaces to device drivers supporting character or raw I/O. It is also useful in other instances in which an input or output buffer can exist in different kinds of address spaces, and in which the buffer is not contiguous in virtual memory.

The **uio** structure is defined in the /**usr**/**include**/**sys**/**uio.h** file.

Description

The **uio** structure describes a buffer that is not contiguous in virtual memory. It also indicates the address space in which the buffer is defined. When used in the character device read and write interface, it also contains the device open–mode flags, along with the device read/write offset.

The kernel provides services that access data using a **uio** structure. The **ureadc**, **uwritec**, **uiomove**, and **uphysio** kernel services all perform data transfers into or out of a data buffer described by a **uio** structure. The **ureadc** kernel service writes a character into the buffer described by the **uio** structure. The **uwritec** kernel service reads a character from the buffer. These two services have names opposite from what you would expect, since they are named for the user action initiating the operation. A read on the part of the user thus results in a device driver writing to the buffer, while a write results in a driver reading from the buffer.

The **uiomove** kernel service copies data to or from a buffer described by a **uio** structure from or to a buffer in the system address space. The **uphysio** kernel service is used primarily by block device drivers providing raw I/O support. The **uphysio** kernel service converts the character read or write request into a block read or write request and sends it to the **ddstrategy** routine.

The buffer described by the **uio** structure can consist of multiple noncontiguous areas of virtual memory of different lengths. This is achieved by describing the data buffer with an array of elements, each of which consists of a virtual memory address and a byte length. Each element is defined as an `iovec` element. The **uio** structure also contains a field specifying the total number of bytes in the data buffer described by the structure.

Another field in the **uio** structure describes the address space of the data buffer, which can either be system space, user space, or cross–memory space. If the address space is defined as cross memory, an additional array of cross–memory descriptors is specified in the **uio** structure to match the array of `iovec` elements.

The **uio** structure also contains a byte offset (`uio_offset)`. This field is a 64 bit integer (`offset_t)`; it allows the file system to send I/O requests to a device driver's read & write entry points which have logical offsets beyond 2 gigabytes. Device drivers must use care not to cause a loss of significance by assigning the offset to a 32 bit variable or using it in calculations that overflow a 32 bit variable.

The called routine (device driver) is permitted to modify fields in the **uio** and **iovec** structures as the data transfer progresses. The final `uio_resid` count is in fact used to determine how much data was transferred. Therefore this count must be decremented, with each operation, by the number of bytes actually copied.

The **uio** structure contains the following fields:

`uio_iov`	A pointer to an array of **iovec** structures describing the user buffer for the data transfer.
`uio_xmem`	A pointer to an array of **xmem** structures containing the cross–memory descriptors for the **iovec** array.
`uio_iovcnt`	The number of yet–to–be–processed **iovec** structures in the array pointed to by the `uio_iov` pointer. The count must be at least 1. If the count is greater than 1, then a *scatter–gather* of the data is to be performed into or out of the areas described by the **iovec** structures.
`uio_iovdcnt`	The number of already processed **iovec** structures in the **iovec** array.
`uio_offset`	The file offset established by a previous **lseek**, **llseek** subroutine call. Most character devices ignore this variable, but some, such as the **/dev/mem** pseudo–device, use and maintain it.
`uio_segflg`	A flag indicating the type of buffer being described by the **uio** structure. This flag typically describes whether the data area is in user or kernel space or is in cross–memory. Refer to the **/usr/include/sys/uio.h** file for a description of the possible values of this flag and their meanings.
`uio_fmode`	The value of the file mode that was specified on opening the file or modified by the **fcntl** subroutine. This flag describes the file control parameters. The **/usr/include/sys/fcntl.h** file contains specific values for this flag.
`uio_resid`	The byte count for the data transfer. It must not exceed the sum of all the `iov_len` values in the array of **iovec** structures. Initially, this field contains the total byte count, and when the operation completes, the value must be decremented by the actual number of bytes transferred.

The **iovec** structure contains the starting address and length of a contiguous data area to be used in a data transfer. The **iovec** structure is the element type in an array pointed to by the `uio_iov` field in the **uio** structure. This array can contain any number of **iovec** structures, each of which describes a single unit of contiguous storage. Taken together, these units represent the total area into which, or from which, data is to be transferred. The `uio_iovcnt` field gives the number of **iovec** structures in the array.

The **iovec** structure contains the following fields:

`iov_base`	A variable in the **iovec** structure containing the base address of the contiguous data area in the address space specified by the `uio_segflag` field. The length of the contiguous data area is specified by the `iov_len` field.
`iov_len`	A variable in the **iovec** structure containing the byte length of the data area starting at the address given in the **iov_base** variable.

Related Information

The **ddread** device driver entry point, **ddwrite** device driver entry point.

The **uiomove** kernel service, **uphysio** kernel service, **ureadc** kernel service, **uwritec** kernel service.

The **fcntl** subroutine, **lseek** subroutine.

Device Driver Kernel Extension Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

Programming In the Kernel Environment Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

Cross Memory Kernel Services in *AIX Kernel Extensions and Device Support Programming Concepts*.

ddclose Device Driver Entry Point

Purpose

Closes a previously open device instance.

Syntax

```
#include <sys/device.h>
#include <sys/types.h>

int ddclose (devno, chan)
dev_t devno;
chan_t chan;
```

Parameters

devno	Specifies the major and minor device numbers of the device instance to close.
chan	Specifies the channel number.

Description

The **ddclose** entry point is called when a previously opened device instance is closed by the **close** subroutine or **fp_close** kernel service. The kernel calls the routine under different circumstances for non–multiplexed and multiplexed device drivers.

For non–multiplexed device drivers, the kernel calls the **ddclose** routine when the last process having the device instance open closes it. This causes the g–node reference count to be decremented to 0 and the g–node to be deallocated.

For multiplexed device drivers, the **ddclose** routine is called for each close associated with an explicit open. In other words, the device driver's **ddclose** routine is invoked once for each time its **ddopen** routine was invoked for the channel.

In some instances, data buffers should be written to the device before returning from the **ddclose** routine. These are buffers containing data to be written to the device that have been queued by the device driver but not yet written.

Non–multiplexed device drivers should reset the associated device to an idle state and change the device driver device state to closed. This can involve calling the **fp_close** kernel service to issue a close to an associated open device handler for the device. Returning the device to an idle state prevents the device from generating any more interrupt or direct memory access (DMA) requests. DMA channels and interrupt levels allocated for this device should be freed, until the device is re–opened, to release critical system resources that this device uses.

Multiplexed device drivers should provide the same device quiescing, but not in the **ddclose** routine. Returning the device to the idle state and freeing its resources should be delayed until the **ddmpx** routine is called to deallocate the last channel allocated on the device.

In all cases, the device instance is considered closed once the **ddclose** routine has returned to the caller, even if a nonzero return code is returned.

Execution Environment

The **ddclose** routine is executed only in the process environment. It should provide the required serialization of its data structures by using the locking kernel services in conjunction with a private lock word defined in the driver.

Return Values

The **ddclose** entry point can indicate an error condition to the user–mode application program by returning a nonzero return code. This causes the subroutine call to return a value of –1. It also makes the return code available to the user–mode application in the **errno** global variable. The return code used should be one of the values defined in the /**usr**/**include**/**sys**/**errno.h** file.

The device is always considered closed even if a nonzero return code is returned.

When applicable, the return values defined in the POSIX 1003.1 standard for the **close** subroutine should be used.

Related Information

The **ddopen** device driver entry point.

The **fp_close** kernel service, **i_clear** kernel service, **i_disable** kernel service.

The **close** subroutine, **open** subroutine.

Device Driver Kernel Extension Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

Programming in the Kernel Environment Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

ddconfig Device Driver Entry Point

Purpose

Performs configuration functions for a device driver.

Syntax

```
#include <sys/device.h>
#include <sys/types.h>

int ddconfig (devno, cmd, uiop)
dev_t devno;
int cmd;
struct uio *uiop;
```

Parameters

devno	Specifies the major and minor device numbers.
cmd	Specifies the function to be performed by the **ddconfig** routine.
uiop	Points to a **uio** structure describing the relevant data area for configuration information.

Description

The **ddconfig** entry point is used to configure a device driver. It can be called to do the following tasks:

- Initialize the device driver.
- Terminate the device driver.
- Request configuration data for the supported device.
- Perform other device–specific configuration functions.

The **ddconfig** routine is called by the device's Configure, Unconfigure, or Change method. Typically, it is called once for each device number (major and minor) to be supported. This is, however, device–dependent. The specific device method and **ddconfig** routine determines the number of times it is called.

The **ddconfig** routine can also provide additional device–specific functions relating to configuration, such as returning device vital product data (VPD). The **ddconfig** routine is usually invoked through the **sysconfig** subroutine by the device–specific Configure method.

Device drivers and their methods typically support these values for the *cmd* parameter:

CFG_INIT Initializes the device driver and internal data areas. This typically involves the minor number specified by the *devno* parameter, for validity. The device driver's **ddconfig** routine also installs the device driver's entry points in the device switch table, if this was the first time called (for the specified major number). This can be accomplished by using the **devswadd** kernel service along with a **devsw** structure to add the device driver's entry points to the device switch table for the major device number supplied in the *devno* parameter.

The **CFG_INIT** command parameter should also copy the device–dependent information (found in the device–dependent structure provided by the caller) into a static or dynamically allocated save area for the specified device. This information should be used when the **ddopen** routine is later called.

The device–dependent structure's address and length are described in the **uio** structure pointed to by the *uiop* parameter. The **uiomove** kernel service can be used to copy the device–dependent structure into the device driver's data area.

When the **ddopen** routine is called, the device driver passes device–dependent information to the routines or other device drivers providing the device handler role in order to initialize the device. The delay in initializing the device until the **ddopen** call is received is useful in order to delay the use of valuable system resources (such as DMA channels and interrupt levels) until the device is actually needed.

CFG_TERM Terminates the device driver associated with the specified device number, as represented by the *devno* parameter.The **ddconfig** routine determines if any opens are outstanding on the specified *devno* parameter. If none are, the **CFG_TERM** command processing marks the device as terminated, disallowing any subsequent opens to the device. All dynamically allocated data areas associated with the specified device number should be freed.

If this termination removes the last minor number supported by the device driver from use, the **devswdel** kernel service should be called to remove the device driver's entry points from the device switch table for the specified *devno* parameter.

If opens are outstanding on the specified device, the terminate operation is rejected with an appropriate error code returned. The Unconfigure method can subsequently unload the device driver if all uses of it have been terminated.

To determine if all the uses of the device driver have been terminated, a device method can make a **sysconfig** subroutine call. By using the **sysconfig SYS_QDVSW** operation, the device method can learn whether or not the device driver has removed itself from the device switch table.

CFG_QVPD Queries device–specific vital product data (VPD).

For this function, the calling routine sets up a **uio** structure pointed at by the *uiop* parameter to the **ddconfig** routine. This **uio** structure defines an area in the caller's storage in which the **ddconfig** routine is to write the VPD. The **uiomove** kernel service can be used to provide the data copy operation.

The data area pointed at by the *uiop* parameter has two different purposes, depending on the *cmd* function. If the **CFG_INIT** command has been requested, the **uiop** structure describes the location and length of the device–dependent data structure (DDS) from which to read the information. If the **CFG_QVPD** command has been requested, the **uiop** structure describes the area in which to write vital product data information. The content and

format of this information is established by the specific device methods in conjunction with the device driver.

The **uiomove** kernel service can be used to facilitate copying information into or out of this data area. The format of the **uio** structure is defined in the /**usr**/**include**/**sys**/**uio.h** file and described further in the **uio** structure.

Execution Environment

The **ddconfig** routine and its operations are called in the process environment only.

Return Values

The **ddconfig** routine sets the return code to 0 if no errors are detected for the operation specified. If an error is to be returned to the caller, a nonzero return code should be provided. The return code used should be one of the values defined in the /**usr**/**include**/**sys**/**errno.h** file.

If this routine was invoked by a **sysconfig** subroutine call, the return code is passed to its caller (typically a device method). It is passed by presenting the error code in the **errno** global variable and providing a –1 return code to the subroutine.

Related Information

The **sysconfig** subroutine.

The **ddopen** device driver entry point.

The **devswadd** kernel service, **devswdel** kernel service, **uiomove** kernel service.

The **uio** structure.

Device Driver Kernel Extension Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

Programming in the Kernel Environment Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

dddump Device Driver Entry Point

Purpose

Writes system dump data to a device.

Syntax

```
#include <sys/device.h>

int dddump (devno, uiop, cmd, arg, chan, ext)
dev_t devno;
struct uio *uiop;
int cmd, arg;
chan_t chan;
int ext;
```

Parameters

devno	Specifies the major and minor device numbers.
uiop	Points to the **uio** structure describing the data area or areas to be dumped.
cmd	The parameter from the kernel dump function that specifies the operation to be performed.
arg	The parameter from the caller that specifies the address of a parameter block associated with the kernel dump command.
chan	Specifies the channel number.
ext	Specifies the extension parameter.

Description

The kernel dump routine calls the **dddump** entry point to set up and send dump requests to the device. The **dddump** routine is optional for a device driver. It is required only when the device driver supports a device as a target for a possible kernel dump.

If this is the case, it is important that the system state change as little as possible when performing the dump. As a result, the **dddump** routine should use the minimal amount of services in writing the dump data to the device.

The *cmd* parameter can specify any of the following dump commands:

DUMPINIT Initialization a device in preparation for supporting a system dump. The specified device instance must have previously been opened. The *arg* parameter points to a **dumpio_stat** structure, defined in /**usr**/**include**/**sys**/**dump.h**. This is used for returning device–specific status in case of an error.

The **dddump** routine should pin all code and data that the device driver uses to support dump writing. This is required to prevent a page fault when actually performing a write of the dump data. (Pinned code should include the **dddump** routine.) The **pin** or **pincode** kernel service can be used for this purpose.

DUMPQUERY Determines the maximum and minimum number of bytes that can be transferred to the device in one **DUMPWRITE** command. For network dumps, the address of the write routine used in transferring dump data to the network dump device is also sent. The *uiop* parameter is not used and is null for this command. The *arg* parameter is a pointer to a **dmp_query** structure, as defined in the /**usr**/**include**/**sys**/**dump.h** file. The **dmp_query** structure contains the following fields:

`min_tsize` Minimum transfer size (in bytes).

`max_tsize` Maximum transfer size (in bytes).

`dumpwrite` Address of the write routine.

Note: Communications device drivers providing remote dump support must supply the address of the write routine used in transferring dump data to the device. The kernel dump function uses logical link control (LLC) to transfer the dump data to the device using the `dumpwrite` field.

The **DUMPQUERY** command returns the data transfer size information in the **dmp_query** structure pointed to by the *arg* parameter. The kernel dump function then uses a buffer between the minimum and maximum transfer sizes (inclusively) when writing dump data.

If the buffer is not the size found in the `max_tsize` field, then its size must be a multiple of the value in the `min_tsize` field. The `min_tsize` field and the `max_tsize` field can specify the same value.

DUMPSTART Suspends current device activity and provide whatever setup of the device is needed before receiving a **DUMPWRITE** command. The *arg* parameter points to a **dumpio_stat** structure, defined in /**usr**/**include**/**sys**/**dump.h**. This is used for returning device–specific status in case of an error.

DUMPWRITE Writes dump data to the target device. The **uio** structure pointed to by the *uiop* parameter specifies the data area or areas to be written to the device and the starting device offset. The *arg* parameter points to a **dumpio_stat** structure, defined in /**usr**/**include**/**sys**/**dump.h**. This is used for returning device–specific status in case of an error. Code for the **DUMPWRITE** command should minimize its reliance on system services, process dispatching, and such interrupt services as the **INTIODONE** interrupt priority or device hardware interrupts.

Note: The **DUMPWRITE** command must never cause a page fault. This is ensured on the part of the caller, since the data areas to be dumped have been determined to be in memory. The device driver must ensure that all of its code, data and stack accesses are to pinned memory during its **DUMPINIT** command processing.

DUMPEND Indicates that the kernel dump has been completed. Any cleanup of the device state should be done at this time.

DUMPTERM Indicates that the specified device is no longer a selected dump target device. If no other devices supported by this **dddump** routine have a **DUMPINIT** command outstanding, the **DUMPTERM** code should unpin any resources pinned when it received the **DUMPINIT** command. (The **unpin** kernel service is available for unpinning memory.) The **DUMPTERM** command is received before the device is closed.

DUMPREAD Receives the acknowledgment packet for previous **DUMPWRITE** operations to a communications device driver. If the device driver receives the acknowledgment within the specified time, it returns a 0 and the response data is returned to the kernel dump function in the *uiop* parameter. If the device driver does not receive the acknowledgment within the specified time, it returns a value of **ETIMEDOUT**.

The *arg* parameter contains a timeout value in milliseconds.

Execution Environment

The **DUMPINIT dddump** operation is called in the process environment only. The **DUMPQUERY**, **DUMPSTART**, **DUMPWRITE**, **DUMPEND**, and **DUMPTERM dddump** operations can be called in both the process environment and interrupt environment.

Return Values

The **dddump** entry point indicates an error condition to the caller by returning a nonzero return code.

Related Information

The **devdump** kernel service, **dmp_add** kernel service, **dmp_del** kernel service, **dmp_prinit** kernel service, **pin** kernel service, **pincode** kernel service, **unpin** kernel service.

The **dump** special file.

The **uio** structure.

Device Driver Kernel Extension Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

Programming in the Kernel Environment Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

ddioctl Device Driver Entry Point

Purpose

Performs the special I/O operations requested in an **ioctl** or **ioctlx** subroutine call.

Syntax

```
#include <sys/device.h>

int ddioctl (devno, cmd, arg, devflag, chan, ext)
dev_t devno;
int cmd, arg;
ulong devflag;
chan_t chan;
int ext;
```

Parameters

devno	Specifies the major and minor device numbers.
cmd	The parameter from the **ioctl** subroutine call that specifies the operation to be performed.
arg	The parameter from the **ioctl** subroutine call that specifies an additional argument for the *cmd* operation.
devflag	Specifies the device open or file control flags.
chan	Specifies the channel number.
ext	Specifies the extension parameter.

Description

When a program issues an **ioctl** or **ioctlx** subroutine call, the kernel calls the **ddioctl** routine of the specified device driver. The **ddioctl** routine is responsible for performing whatever functions are requested. In addition, it must return whatever control information has been specified by the original caller of the **ioctl** subroutine. The *cmd* parameter contains the name of the operation to be performed.

Most ioctl operations depend on the specific device involved. However, all ioctl routines must respond to the following command:

IOCINFO	Returns a **devinfo** structure (defined in the /**usr**/**include**/**sys**/**devinfo.h** file) that describes the device. (Refer to the description of the special file for a particular device in the Application Programming Interface.) Only the first two fields of the data structure need to be returned if the remaining fields of the structure do not apply to the device.

The *devflag* parameter indicates one of several types of information. It can give conditions in which the device was opened. (These conditions can subsequently be changed by the **fcntl** subroutine call.) Alternatively, it can tell which of two ways the entry point was invoked:

- By the file system on behalf of a using application
- Directly by a kernel routine using the **fp_ioctl** kernel service

Thus flags in the *devflag* parameter have the following definitions, as defined in the /**usr**/**include**/**sys**/**device.h** file:

DKERNEL	Entry point called by kernel routine using the **fp_ioctl** service.
DREAD	Open for reading.
DWRITE	Open for writing.

DAPPEND	Open for appending.
DNDELAY	Device open in nonblocking mode.

Execution Environment

The **ddioctl** routine is executed only in the process environment. It should provide the required serialization of its data structures by using the locking kernel services in conjunction with a private lock word defined in the driver.

Return Values

The **ddioctl** entry point can indicate an error condition to the user–mode application program by returning a nonzero return code. This causes the **ioctl** subroutine to return a value of –1 and makes the return code available to the user–mode application in the **errno** global variable. The error code used should be one of the values defined in the /**usr**/**include**/**sys**/**errno.h** file.

When applicable, the return values defined in the POSIX 1003.1 standard for the **ioctl** subroutine should be used.

Related Information

The **fp_ioctl** kernel service.

The **fcntl** subroutine, **ioctl** or **ioctlx** subroutine, **open** subroutine.

Device Driver Kernel Extension Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

Virtual File System Kernel Extensions Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

Special Files Overview in *AIX Files Reference*.

Programming in the Kernel Environment Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

ddmpx Device Driver Entry Point

Purpose

Allocates or deallocates a channel for a multiplexed device driver.

Syntax

```
#include <sys/device.h>
#include <sys/types.h>

int ddmpx (devno, chanp, channame)
dev_t devno;
chan_t *chanp;
char *channame;
```

Parameters

devno	Specifies the major and minor device numbers.
chanp	Specifies the channel ID, passed by reference.
channame	Points to the path name extension for the channel to be allocated.

Description

Only multiplexed character class device drivers can provide the **ddmpx** routine, and *every* multiplexed driver must do so. The **ddmpx** routine cannot be provided by block device drivers even when providing *raw* read/write access.

A multiplexed device driver is a character class device driver that supports the assignment of channels to provide finer access control to a device or virtual subdevice. This type of device driver has the capability to decode special channel–related information appended to the end of the path name of the device's special file. This path name extension is used to identify a logical or virtual subdevice or channel.

When an **open** or **creat** subroutine call is issued to a device instance supported by a multiplexed device driver, the kernel calls the device driver's **ddmpx** routine to allocate a channel.

The kernel calls the **ddmpx** routine when a channel is to be allocated or deallocated. Upon allocation, the kernel dynamically creates g–nodes (in–core i–nodes) for channels on a multiplexed device to allow the protection attributes to differ for various channels.

To allocate a channel, the **ddmpx** routine is called with a *channame* pointer to the path name extension. The path name extension starts after the first / (slash) character that follows the special file name in the path name. The **ddmpx** routine should perform the following actions:

- Parse this path name extension.
- Allocate the corresponding channel.
- Return the channel ID through the *chanp* parameter.

If no path name extension exists, the *channame* pointer points to a null character string. In this case, an available channel should be allocated and its channel ID returned through the *chanp* parameter.

If no error is returned from the **ddmpx** routine, the returned channel ID is used to determine if the channel was already allocated. If already allocated, the g–node for the associated channel has its reference count incremented. If the channel was not already allocated, a new g–node is created for the channel. In either case, the device driver's **ddopen** routine is called with the channel number assigned by the **ddmpx** routine. If a nonzero return code is

returned by the **ddmpx** routine, the channel is assumed not to have been allocated, and the device driver's **ddopen** routine is not called.

If a close of a channel is requested so that the channel is no longer used (as determined by the channel's g–node reference count going to 0), the kernel calls the **ddmpx** routine. The **ddmpx** routine deallocates the channel after the **ddclose** routine was called to close the last use of the channel. If a nonzero return code is returned by the **ddclose** routine, the **ddmpx** routine is still called to deallocate the channel. The **ddclose** routine's return code is saved, to be returned to the caller. If the **ddclose** routine returned no error, but a nonzero return code was returned by the **ddmpx** routine, the channel is assumed to be deallocated, although the return code is returned to the caller.

To deallocate a channel, the **ddmpx** routine is called with a null *channame* pointer and the channel ID passed by reference in the *chanp* parameter. If the channel g–node reference count has gone to 0, the kernel calls the **ddmpx** routine to deallocate the channel after invoking the **ddclose** routine to close it. The **ddclose** routine should not itself deallocate the channel.

Execution Environment

The **ddmpx** routine is called in the process environment only.

Return Values

If the allocation or deallocation of a channel is successful, the **ddmpx** routine should return a return code of 0. If an error occurs on allocation or deallocation, this routine returns a nonzero value.

The return code should conform to the return codes described for the **open** and **close** subroutines in the POSIX 1003.1 standard, where applicable. Otherwise, the return code should be one defined in the **/usr/include/sys/errno.h** file.

Related Information

The **ddclose** device driver entry point, **ddopen** device driver entry point.

The **close** subroutine, **open** or **creat** subroutine.

Device Driver Kernel Extension Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

Programming in the Kernel Environment Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

ddopen Device Driver Entry Point

Purpose

Prepares a device for reading, writing, or control functions.

Syntax

```
#include <sys/device.h>

int ddopen (devno, devflag, chan, ext)
dev_t devno;
ulong devflag;
chan_t chan;
int ext;
```

Parameters

devno	Indicates major and minor device numbers.
devflag	Specifies open file control flags.
chan	Specifies the channel number.
ext	Specifies the extension parameter.

Description

The kernel calls the **ddopen** routine of a device driver when a program issues an **open** or **creat** subroutine call. It can also be called when a system call, kernel process, or other device driver uses the **fp_opendev** or **fp_open** kernel service to use the device.

The **ddopen** routine must first ensure exclusive access to the device, if necessary. Many character devices, such as printers and plotters, should be opened by only one process at a time. The **ddopen** routine can enforce this by maintaining a static flag variable, which is set to 1 if the device is open and 0 if not.

Each time the **ddopen** routine is called, it checks the value of the flag. If the value is other than 0, the **ddopen** routine returns with a return code of **EBUSY** to indicate that the device is already open. Otherwise, the **ddopen** routine sets the flag and returns normally. The **ddclose** entry point later clears the flag when the device is closed.

Since most block devices can be used by several processes at once, a block driver should not try to enforce opening by a single user.

The **ddopen** routine must initialize the device if this is the first open that has occurred. Initialization involves the following steps:

1. The **ddopen** routine should allocate the required system resources to the device (such as DMA channels, interrupt levels, and priorities). It should, if necessary, register its device interrupt handler for the interrupt level required to support the target device. (The **i_init** and **d_init** kernel services are available for initializing these resources.)
2. If this device driver is providing the head role for a device and another device driver is providing the handler role, the **ddopen** routine should use the **fp_opendev** kernel service to open the device handler.

   Note: The **fp_opendev** kernel service requires a *devno* parameter to identify which device handler to open. This *devno* value, taken from the appropriate device dependent structure (DDS), should have been stored in a special save area when this device driver's **ddconfig** routine was called.

Flags Defined for the devflag Parameter

The *devflag* parameter has the following flags, as defined in the **/usr/include/sys/device.h** file:

DKERNEL	Entry point called by kernel routine using the **fp_opendev** or **fp_open** kernel service.
DREAD	Open for reading.
DWRITE	Open for writing.
DAPPEND	Open for appending.
DNDELAY	Device open in nonblocking mode.

Execution Environment

The **ddopen** routine is executed only in the process environment. It should provide the required serialization of its data structures by using the locking kernel services in conjunction with a private lock word defined in the driver.

Return Values

The **ddopen** entry point can indicate an error condition to the user–mode application program by returning a nonzero return code. Returning a nonzero return code causes the **open** or **creat** subroutines to return a value of –1 and makes the return code available to the user–mode application in the **errno** global variable. The return code used should be one of the values defined in the **/usr/include/errno.h** file.

If a nonzero return code is returned by the **ddopen** routine, the open request is considered to have failed. No access to the device instance is available to the caller as a result. In addition, for nonmultiplexed drivers, if the failed open was the first open of the device instance, the kernel calls the driver's **ddclose** entry point to allow resources and device driver state to be cleaned up. If the driver was multiplexed, the kernel does not call the **ddclose** entry point on an open failure.

When applicable, the return values defined in the POSIX 1003.1 standard for the **open** subroutine should be used.

Related Information

The **ddclose** device driver entry point, **ddconfig** device driver entry point.

The **d_init** kernel service, **fp_open** kernel service, **fp_opendev** kernel service, **i_enable** kernel service, **i_init** kernel service.

The **close** subroutine, **creat** subroutine, **open** subroutine.

Device Driver Kernel Extension Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

Programming in the Kernel Environment Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

ddread Device Driver Entry Point

Purpose

Reads in data from a character device.

Syntax

```
#include <sys/device.h>
#include <sys/types.h>

int ddread (devno, uiop, chan, ext)
dev_t devno;
struct uio *uiop;
chan_t chan;
int ext;
```

Parameters

devno	Specifies the major and minor device numbers.
uiop	Points to a **uio** structure describing the data area or areas in which to be written.
chan	Specifies the channel number.
ext	Specifies the extension parameter.

Description

When a program issues a **read** or **readx** subroutine call or when the **fp_rwuio** kernel service is used, the kernel calls the **ddread** entry point.

This entry point receives a pointer to a **uio** structure that provides variables used to specify the data transfer operation.

Character device drivers can use the **ureadc** and **uiomove** kernel services to transfer data into and out of the user buffer area during a **read** subroutine call. These services receive a pointer to the **uio** structure and update the fields in the structure by the number of bytes transferred. The only fields in the **uio** structure that cannot be modified by the data transfer are the `uio_fmode` and `uio_segflg` fields.

For most devices, the **ddread** routine sends the request to the device handler and then waits for it to finish. The waiting can be accomplished by calling the **e_sleep** kernel service. This service suspends the driver and the process that called it and permits other processes to run until a specified event occurs.

When the I/O operation completes, the device usually issues an interrupt, causing the device driver's interrupt handler to be called. The interrupt handler then calls the **e_wakeup** kernel service specifying the awaited event, thus allowing the **ddread** routine to resume.

The `uio_resid` field initially contains the total number of bytes to read from the device. If the device driver supports it, the `uio_offset` field indicates the byte offset on the device from which the read should start.

The `uio_offset` field is a 64 bit integer (`offset_t`); this allows the file system to send I/O requests to a device driver's read & write entry points which have logical offsets beyond 2 gigabytes. Device drivers must use care not to cause a loss of significance by assigning the offset to a 32 bit variable or using it in calculations that overflow a 32 bit variable.

If no error occurs, the `uio_resid` field should be 0 on return from the **ddread** routine to indicate that all requested bytes were read. If an error occurs, this field should contain the number of bytes remaining to be read when the error occurred.

If a read request starts at a valid device offset but extends past the end of the device's capabilities, no error should be returned. However, the `uio_resid` field should indicate the number of bytes not transferred. If the read starts at the end of the device's capabilities, no error should be returned. However, the `uio_resid` field should not be modified, indicating that no bytes were transferred. If the read starts past the end of the device's capabilities, an **ENXIO** return code should be returned, without modifying the `uio_resid` field.

When the **ddread** entry point is provided for raw I/O to a block device, this routine usually translates requests into block I/O requests using the **uphysio** kernel service.

Execution Environment

The **ddread** routine is executed only in the process environment. It should provide the required serialization of its data structures by using the locking kernel services in conjunction with a private lock word defined in the driver.

Return Values

The **ddread** entry point can indicate an error condition to the caller by returning a nonzero return code. This causes the subroutine call to return a value of –1. It also makes the return code available to the user–mode program in the **errno** global variable. The error code used should be one of the values defined in the /**usr**/**include**/**sys**/**errno.h** file.

When applicable, the return values defined in the POSIX 1003.1 standard for the **read** subroutine should be used.

Related Information

The **ddwrite** device driver entry point.

The **e_sleep** kernel service, **e_wakeup** kernel service, **fp_rwuio** kernel service, **uiomove** kernel service, **uphysio** kernel service, **ureadc** kernel service.

The **uio** structure.

The **read**, **readx** subroutines.

Select/Poll Logic for ddwrite and ddread Routines.

Device Driver Kernel Extension Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

Programming in the Kernel Environment Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

ddrevoke Device Driver Entry Point

Purpose

Ensures that a secure path to a terminal is provided.

Syntax

```
#include <sys/device.h>
#include <sys/types.h>

int ddrevoke (devno, chan, flag)
dev_t devno;
chan_t chan;
int flag;
```

Parameters

devno	Specifies the major and minor device numbers.
chan	Specifies the channel number. For a multiplexed device driver, a value of –1 in this parameter means access to all channels is to be revoked.
flag	Currently defined to have the value of 0. (Reserved for future extensions.)

Description

The **ddrevoke** entry point can be provided only by character class device drivers. It cannot be provided by block device drivers even when providing raw read/write access. A **ddrevoke** entry point is required only by device drivers supporting devices in the Trusted Computing Path to a terminal (for example, by the **/dev/ lft** and **/dev/tty** files for the low function terminal and teletype device drivers). The **ddrevoke** routine is called by the **frevoke** and **revoke** subroutines.

The **ddrevoke** routine revokes access to a specific device or channel (if the device driver is multiplexed). When called, the **ddrevoke** routine should terminate all processes waiting in the device driver while accessing the specified device or channel. It should terminate the processes by sending a SIGKILL signal to all processes currently waiting for a specified device or channel data transfer. The current process is not to be terminated.

If the device driver is multiplexed and the channel ID in the *chan* parameter has the value –1, all channels are to be revoked.

Execution Environment

The **ddrevoke** routine is called in the process environment only.

Return Values

The **ddrevoke** routine should return a value of 0 for successful completion, or a value from the **/usr/include/errno.h** file on error.

Files

/dev/lft	Specifies the path of the LFT special file.
/dev/tty	Specifies the path of the tty special file.

Related Information

The **frevoke** subroutine, **revoke** subroutine.

LFT Subsystem Component Structure Overview , Device Driver Kernel Extension Overview, Programming in the Kernel Environment Overview, in *AIX Kernel Extensions and Device Support Programming Concepts*.

The TTY Subsystem Overview in *AIX General Programming Concepts: Writing and Debugging Programs*.

ddselect Device Driver Entry Point

Purpose

Checks to see if one or more events has occurred on the device.

Syntax

```
#include <sys/device.h>
#include <sys/poll.h>

int ddselect (devno, events, reventp, chan)
dev_t devno;
ushort events;
ushort *reventp;
int chan;
```

Parameters

devno	Specifies the major and minor device numbers.
events	Specifies the events to be checked.
reventp	Returned events pointer. This parameter, passed by reference, is used by the **ddselect** routine to indicate which of the selected events are true at the time of the call. The returned events location pointed to by the *reventp* parameter is set to 0 before entering this routine.
chan	Specifies the channel number.

Description

The **ddselect** entry point is called when the **select** or **poll** subroutine is used, or when the **fp_select** kernel service is invoked. It determines whether a specified event or events have occurred on the device.

Only character class device drivers can provide the **ddselect** routine. It cannot be provided by block device drivers even when providing raw read/write access.

Requests for Information on Events

The *events* parameter represents possible events to check as flags (bits). There are three basic events defined for the **select** and **poll** subroutines, when applied to devices supporting select or poll operations:

POLLIN	Input is present on the device.
POLLOUT	The device is capable of output.
POLLPRI	An exceptional condition has occurred on the device.

A fourth event flag is used to indicate whether the **ddselect** routine should record this request for later notification of the event using the **selnotify** kernel service. This flag can be set in the *events* parameter if the device driver is not required to provide asynchronous notification of the requested events:

POLLSYNC	This request is a synchronous request only. The routine need not call the **selnotify** kernel service for this request even if the events later occur.

Additional event flags in the *events* parameter are left for device–specific events on the **poll** subroutine call.

Select Processing

If one or more events specified in the *events* parameter are true, the **ddselect** routine should indicate this by setting the corresponding bits in the *reventp* parameter. Note that the *reventp* returned events parameter is passed by reference.

If none of the requested events are true, then the **ddselect** routine sets the returned events parameter to 0. It is passed by reference through the *reventp* parameter. It also checks the **POLLSYNC** flag in the *events* parameter. If this flag is true, the **ddselect** routine should just return, since the event request was a synchronous request only.

However, if the **POLLSYNC** flag is false, the **ddselect** routine must notify the kernel when one or more of the specified events later happen. For this purpose, the routine should set separate internal flags for each event requested in the *events* parameter.

When any of these events become true, the device driver routine should use the **selnotify** service to notify the kernel. The corresponding internal flags should then be reset to prevent re–notification of the event.

Sometimes the device can be in a state in which a supported event or events can never be satisfied (such as when a communication line is not operational). In this case, the **ddselect** routine should simply set the corresponding *reventp* flags to 1. This prevents the **select** or **poll** subroutine from waiting indefinitely. As a result however, the caller will not in this case be able to distinguish between satisfied events and unsatisfiable ones. Only when a later request with an **NDELAY** option fails will the error be detected.

Note: Other device driver routines (such as the **ddread**, **ddwrite** routines) may require logic to support select or poll operations.

Execution Environment

The **ddselect** routine is executed only in the process environment. It should provide the required serialization of its data structures by using the locking kernel services in conjunction with a private lock word defined in the driver.

Return Values

The **ddselect** routine should return with a return code of 0 if the select or poll operation requested is valid for the resource specified. Requested operations are not valid, however, if either of the following is true:

- The device driver does not support a requested event.
- The device is in a state in which poll and select operations are not accepted.

In these cases, the **ddselect** routine should return with a nonzero return code (typically **EINVAL**), and without setting the relevant *reventp* flags to 1. This causes the **poll** subroutine to return to the caller with the **POLLERR** flag set in the returned events parameter associated with this resource. The **select** subroutine indicates to the caller that all requested events are true for this resource.

When applicable, the return values defined in the POSIX 1003.1 standard for the **select** subroutine should be used.

Related Information

The **ddread** device driver entry point, **ddwrite** device driver entry point.

The **fp_select** kernel service, **selnotify** kernel service.

The **poll** subroutine, **select** subroutine.

Programming in the Kernel Environment Overview and Device Driver Kernel Extension Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

ddstrategy Device Driver Entry Point

Purpose

Performs block–oriented I/O by scheduling a read or write to a block device.

Syntax

```
void ddstrategy (bp)
struct buf *bp;
```

Parameter

bp Points to a **buf** structure describing all information needed to perform the data transfer.

Description

When the kernel needs a block I/O transfer, it calls the **ddstrategy** strategy routine of the device driver for that device. The strategy routine schedules the I/O to the device. This typically requires the following actions:

- The request or requests must be added on the list of I/O requests that need to be processed by the device.
- If the request list was empty before the preceding additions, the device's start I/O routine must be called.

Required Processing

The **ddstrategy** routine can receive a single request with multiple **buf** structures. However, it is not required to process requests in any specific order.

The strategy routine can be passed a list of operations to perform. The `av_forw` field in the **buf** header describes this null–terminated list of **buf** headers. This list is not doubly linked: the `av_back` field is undefined.

Block device drivers must be able to perform multiple block transfers. If the device cannot do multiple block transfers, or can only do multiple block transfers under certain conditions, then the device driver must transfer the data with more than one device operation.

Kernel Buffers and Using the buf Structure

An area of memory is set aside within the kernel memory space for buffering data transfers between a program and the peripheral device. Each kernel buffer has a header, the **buf** structure, which contains all necessary information for performing the data transfer. The **ddstrategy** routine is responsible for updating fields in this header as part of the transfer.

The caller of the strategy routine should set the `b_iodone` field to point to the caller's I/O done routine. When an I/O operation is complete, the device driver calls the **iodone** kernel service, which then calls the I/O done routine specified in the `b_iodone` field. The **iodone** kernel service makes this call from the **INTIODONE** interrupt level.

The value of the `b_flags` field is constructed by logically ORing zero or more possible `b_flags` field flag values.

Attention: Do not modify any of the following fields of the **buf** structure passed to the **ddstrategy** entry point: the `b_forw`, `b_back`, `b_dev`, `b_un`, or `b_blkno` field. Modifying these fields can cause unpredictable and disastrous results.

Attention: Do not modify any of the following fields of a **buf** structure acquired with the **geteblk** service: the `b_flags`, `b_forw`, `b_back`, `b_dev`, `b_count`, or `b_un` field. Modifying any of these fields can cause unpredictable and disastrous results.

Execution Environment

The **ddstrategy** routine must be coded to execute in an interrupt handler execution environment (device driver bottom half). That is, the routine should neither touch user storage, nor page fault, nor sleep.

Return Values

The **ddstrategy** routine, unlike other device driver routines, does not return a return code. Any error information is returned in the appropriate fields within the **buf** structure pointed to by the *bp* parameter.

When applicable, the return values defined in the POSIX 1003.1 standard for the **read** and **write** subroutines should be used.

Related Information

The **geteblk** kernel service, **iodone** kernel service.

The **buf** structure.

The **read** subroutine, **write** subroutine.

Device Driver Kernel Extension Overview, Understanding Device Driver Structure and Understanding Device Driver Classes, Programming in the Kernel Environment Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

ddwrite Device Driver Entry Point

Purpose

Writes out data to a character device.

Syntax

```
#include <sys/device.h>
#include <sys/types.h>

int ddwrite (devno, uiop, chan, ext)
dev_t devno;
struct uio *uiop;
chan_t chan;
int ext;
```

Parameters

devno	Specifies the major and minor device numbers.
uiop	Points to a **uio** structure describing the data area or areas from which to be written.
chan	Specifies the channel number.
ext	Specifies the extension parameter.

Description

When a program issues a **write** or **writex** subroutine call or when the **fp_rwuio** kernel service is used, the kernel calls the **ddwrite** entry point.

This entry point receives a pointer to a **uio** structure, which provides variables used to specify the data transfer operation.

Character device drivers can use the **uwritec** and **uiomove** kernel services to transfer data into and out of the user buffer area during a **write** subroutine call. These services are passed a pointer to the **uio** structure. They update the fields in the structure by the number of bytes transferred. The only fields in the **uio** structure that are not potentially modified by the data transfer are the `uio_fmode` and `uio_segflg` fields.

For most devices, the **ddwrite** routine queues the request to the device handler and then waits for it to finish. The waiting is typically accomplished by calling the **e_sleep** kernel service to wait for an event. The **e_sleep** kernel service suspends the driver and the process that called it and permits other processes to run.

When the I/O operation is completed, the device usually causes an interrupt, causing the device driver's interrupt handler to be called. The interrupt handler then calls the **e_wakeup** kernel service specifying the awaited event, thus allowing the **ddwrite** routine to resume.

The `uio_resid` field initially contains the total number of bytes to write to the device. If the device driver supports it, the `uio_offset` field indicates the byte offset on the device from where the write should start.

The `uio_offset` field is a 64 bit integer (`offset_t`); this allows the file system to send I/O requests to a device driver's read & write entry points which have logical offsets beyond 2 gigabytes. Device drivers must use care not to cause a loss of significance by assigning the offset to a 32 bit variable or using it in calculations that overflow a 32 bit variable.

If no error occurs, the `uio_resid` field should be 0 on return from the **ddwrite** routine to indicate that all requested bytes were written. If an error occurs, this field should contain the number of bytes remaining to be written when the error occurred.

If a write request starts at a valid device offset but extends past the end of the device's capabilities, no error should be returned. However, the `uio_resid` field should indicate the number of bytes not transferred. If the write starts at or past the end of the device's capabilities, no data should be transferred. An error code of **ENXIO** should be returned, and the `uio_resid` field should not be modified.

When the **ddwrite** entry point is provided for raw I/O to a block device, this routine usually uses the **uphysio** kernel service to translate requests into block I/O requests.

Execution Environment

The **ddwrite** routine is executed only in the process environment. It should provide the required serialization of its data structures by using the locking kernel services in conjunction with a private lock word defined in the driver.

Return Values

The **ddwrite** entry point can indicate an error condition to the caller by returning a nonzero return value. This causes the subroutine to return a value of –1. It also makes the return code available to the user–mode program in the **errno** global variable. The error code used should be one of the values defined in the /**usr**/**include**/**sys**/**errno.h** file.

When applicable, the return values defined in the POSIX 1003.1 standard for the **write** subroutine should be used.

Related Information

The **ddread** device driver entry point.

The **CIO_GET_FASTWRT** ddioctl.

The **e_sleep** kernel service, **e_wakeup** kernel service, **fp_rwuio** kernel service, **uiomove** kernel service, **uphysio** kernel service, **uwritec** kernel service.

The **uio** structure.

The **write** and **writex** subroutines.

Device Driver Kernel Extension Overview, Understanding Device Driver Roles, Understanding Interrupts, Understanding Locking in *AIX Kernel Extensions and Device Support Programming Concepts*.

Select/Poll Logic for ddwrite and ddread Routines

Description

The **ddread** and **ddwrite** entry points require logic to support the **select** and **poll** operations. Depending on how the device driver is written, the interrupt routine may also need to include this logic as well.

The select/poll logic is required wherever code checks on the occurrence of desired events. At each point where one of the selection criteria is found to be true, the device driver should check whether a notification is due for that selection. If so, it should call the **selnotify** kernel service to notify the kernel of the event.

The *devno*, *chan*, and *revents* parameters are passed to the **selnotify** kernel service to indicate which device and which events have become true.

Related Information

The **ddread** device driver entry point, **ddselect** device driver entry point, **ddwrite** device driver entry point.

The **selnotify** kernel service.

The **poll** subroutine, **select** subroutine.

Device Driver Kernel Extension Overview and Programming in the Kernel Environment Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

Chapter 3. File System Operations

List of Virtual File System Operations

The following entry points are specified by the virtual file system interface for performing operations on **vfs** structures:

vfs_cntl	Issues control operations for a file system.
vfs_init	Initializes a virtual file system.
vfs_mount	Mounts a virtual file system.
vfs_root	Finds the root v–node of a virtual file system.
vfs_statfs	Obtains virtual file system statistics.
vfs_sync	Forces file system updates to permanent storage.
vfs_umount	Unmounts a virtual file system.
vfs_vget	Gets the v–node corresponding to a file identifier.

The following entry points are specified by the Virtual File System interface for performing operations on v–node structures:

vn_access	Tests a user's permission to access a file.
vn_close	Releases the resources associated with a v–node.
vn_create	Creates and opens a new file.
vn_fclear	Releases portions of a file (by zeroing bytes).
vn_fid	Builds a file identifier for a v–node.
vn_fsync	Flushes in–memory information and data to permanent storage.
vn_ftrunc	Decreases the size of a file.
vn_getacl	Gets information about access control, by retrieving the access control list.
vn_getattr	Gets the attributes of a file.
vn_hold	Assures that a v–node is not destroyed, by incrementing the v–node's use count.
vn_ioctl	Performs miscellaneous operations on devices.
vn_link	Creates a new directory entry for a file.
vn_lockctl	Sets, removes, and queries file locks.
vn_lookup	Finds an object by name in a directory.
vn_map	Associates a file with a memory segment.
vn_mkdir	Creates a directory.
vn_mknod	Creates a file of arbitrary type.
vn_open	Gets read and/or write access to a file.
vn_rdwr	Reads or writes a file.
vn_readdir	Reads directory entries in standard format.
vn_readlink	Reads the contents of a symbolic link.
vn_rele	Releases a reference to a virtual node (v–node).
vn_remove	Unlinks a file or directory.
vn_rename	Renames a file or directory.
vn_revoke	Revokes access to an object.
vn_rmdir	Removes a directory.

vn_select	Polls a v–node for pending I/O.
vn_setacl	Sets information about access control for a file.
vn_setattr	Sets attributes of a file.
vn_strategy	Reads or writes blocks of a file.
vn_symlink	Creates a symbolic link.
vn_unmap	Destroys a file or memory association.

Related Information

vfs_cntl Entry Point

Purpose

Implements control operations for a file system.

Syntax

```
int vfs_cntl (vfsp, cmd, arg, argsize, crp)
struct vfs *vfsp;
int cmd;
caddr_t arg;
unsigned long argsize;
struct ucred *crp;
```

Parameters

vfsp	Points to the file system for which the control operation is to be issued.
cmd	Specifies which control operation to perform.
arg	Identifies data specific to the control operation.
argsize	Identifies the length of the data specified by the *arg* parameter.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vfs_cntl** entry point is invoked by the logical file system to request various control operations on the underlying file system. A file system implementation can define file system–specific *cmd* parameter values and corresponding control functions. The *cmd* parameter for these functions should have a minimum value of 32768. These control operations can be issued with the **fscntl** subroutine.

Note: The only system–supported control operation is **FS_EXTENDFS**. This operation increases the file system size and accepts an *arg* parameter that specifies the new size. The **FS_EXTENDFS** operation ignores the *argsize* parameter.

Execution Environment

The **vfs_cntl** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Non–zero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure. Typical values include:

EINVAL	Indicates that the *cmd* parameter is not a supported control, or the *arg* parameter is not a valid argument for the command.
EACCES	Indicates that the *cmd* parameter requires a privilege that the current process does not have.

Related Information

The **fscntl** subroutine.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vfs_hold or vfs_unhold Kernel Service

Purpose

Holds or releases a **vfs** structure.

Syntax

```
#include <sys/vfs.h>

void vfs_hold(vfsp)
struct vfs *vfsp;

void vfs_unhold(vfsp)
struct vfs *vfsp;
```

Parameter

vfsp	Points to a **vfs** structure.

Description

The **vfs_hold** kernel service holds a **vfs** structure and the **vfs_unhold** kernel service releases it. These routines manage a use count for a virtual file system (VFS). A use count greater than 1 prevents the virtual file system from being unmounted.

Execution Environment

These kernel services can be called from the process environment only.

Return Values

None

Implementation Specifics

These kernel services are part of Base Operating System (BOS) Runtime.

Related Information

vfs_init Entry Point

Purpose

Initializes a virtual file system.

Syntax

```
int vfs_init (gfsp)
struct gfs *gfsp;
```

Parameter

gfsp	Points to a file system's attribute structure.

Description

The **vfs_init** entry point is invoked to initialize a file system. It is called when a file system implementation is loaded to perform file system–specific initialization.

The **vfs_init** entry point is not called through the virtual file system switch. Instead, it is called indirectly by the **gfsadd** kernel service when the **vfs_init** entry point address is stored in the **gfs** structure passed to the **gfsadd** kernel service as a parameter. (The **vfs_init** address is placed in the `gfs_init` field of the **gfs** structure.) The **gfs** structure is defined in the /**usr**/**include**/**sys**/**gfs.h** file.

Note: The return value for the **vfs_init** entry point is passed back as the return value from the **gfsadd** kernel service.

Execution Environment

The **vfs_init** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the /**usr**/**include**/**sys**/**errno.h** file to indicate failure.

Related Information

The **gfsadd** kernel service.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

vfs_mount Entry Point

Purpose

Mounts a virtual file system.

Syntax

```
int vfs_mount (vfsp)
struct vfs *vfsp;
struct ucred *crp;
```

Parameter

vfsp	Points to the newly created **vfs** structure.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vfs_mount** entry point is called by the logical file system to mount a new file system. This entry point is called after the **vfs** structure is allocated and initialized. Before this structure is passed to the **vfs_mount** entry point, the logical file system:

- Guarantees the syntax of the **vmount** or **mount** subroutines.
- Allocates the **vfs** structure.
- Resolves the stub to a virtual node (v–node). This is the `vfs_mntdover` field in the **vfs** structure.
- Initializes the following virtual file system fields:

`vfs_flags`	Initialized depending on the type of mount. This field takes the following values: **VFS_MOUNTOK** The user has write permission in the stub's parent directory and is the owner of the stub. **VFS_SUSER** The user has root user authority. **VFS_NOSUID** Execution of **setuid** and **setgid** programs from this mount are not allowed. **VFS_NODEV** Opens of devices from this mount are not allowed.
`vfs_type`	Initialized to the / (root) file system type when the **mount** subroutine is used. If the **vmount** subroutine is used, the **vfs_type** field is set to the *type* parameter supplied by the user. The logical file system verifies the existence of the *type* parameter.
`vfs_ops`	Initialized according to the `vfs_type` field.
`vfs_mntdover`	Identifies the v–node that refers to the stub path argument. This argument is supplied by the **mount** or **vmount** subroutine.
`vfs_date`	Holds the time stamp. The time stamp specifies the time to initialize the virtual file system.
`vfs_number`	Indicates the unique number sequence representing this virtual file system.
`vfs_mdata`	Initialized with the **vmount** structure supplied by the user. The virtual file system data is detailed in the **/usr/include/sys/vmount.h** file. All arguments indicated by this field are copied to kernel space.

Execution Environment

The **vfs_mount** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related Information

The **mount** subroutine, **vmount** subroutine.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

vfs_root Entry Point

Purpose

Returns the root v–node of a virtual file system (VFS).

Syntax

```
int vfs_root (vfsp, vpp, crp)
struct vfs *vfsp;
struct vnode **vpp;
struct ucred *crp;
```

Parameters

vfsp	Points to the **vfs** structure.
vpp	Points to the place to return the v–node pointer.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vfs_root** entry point is invoked by the logical file system to get a pointer to the root v–node of the file system. When successful, the *vpp* parameter points to the root virtual node (v–node) and the v–node hold count is incremented.

Execution Environment

The **vfs_root** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the /**usr**/**include**/**sys**/**errno.h** file to indicate failure.

Related Information

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Understanding Data Structures and Header Files for Virtual File Systems, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vfs_search Kernel Service

Purpose

Searches the vfs list.

Syntax

```
int vfs_search (vfs_srchfcn, srchargs)
(int (*vfs_srchfcn)(struct vfs *caddr_t);
caddr_t srchargs;
```

Parameters

vfs_srchfcn	Points to a search function. The search function is identified by the *vfs_srchfcn* parameter. This function is used to examine or modify an entry in the vfs list. The search function is called once for each currently active VFS. If the search function returns a value of 0, iteration through the vfs list continues to the next entry. If the return value is nonzero, **vfs_search** kernel service returns to its caller, passing back the return value from the search function. When the system invokes this function, the system passes it a pointer to a virtual file system (VFS) and the *srchargs* parameter.
srchargs	Points to data to be used by the serach function. This pointer is not used by the **vfs_search** kernel service but is passed to the search function.

Description

The **vfs_search** kernel service searches the vfs list. This kernel service allows a process outside the file system to search the vfs list. The **vfs_search** kernel service locks out all activity in the vfs list during a search. Then, the kernel service iterates through the vfs list and calls the search function on each entry.

The search function must not request locks that could result in deadlock. In particular, any attempt to do lock operations on the vfs list or on other VFS structures could produce deadlock.

The performance of the **vfs_search** kernel service may not be acceptable for functions requiring quick response. Iterating through the vfs list and making an indirect function call for each structure is inherently slow.

Execution Environment

The **vfs_search** kernel service can be called from the process environment only.

Return Values

This kernel service returns the value returned by the last call to the search function.

Implementation Specifics

This kernel service is part of Base Operating System (BOS) Runtime.

Related Information

vfs_statfs Entry Point

Purpose

Returns virtual file system statistics.

Syntax

```
int vfs_stafs (vfsp, stafsp, crp)
struct vfs *vfsp;
struct statfs *stafsp;
struct ucred *crp;
```

Parameters

vfsp	Points to the **vfs** structure being queried. This structure is defined in the /**usr**/**include**/**sys**/**vfs.h** file.
stafsp	Points to a **statfs** structure. This structure is defined in the /**usr**/**include**/**sys**/**statfs.h** file.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vfs_stafs** entry point is called by the logical file system to obtain file system characteristics. Upon return, the **vfs_statfs** entry point has filled in the following fields of the **statfs** structure:

`f_blocks`	Specifies the number of blocks.
`f_files`	Specifies the total number of file system objects.
`f_bsize`	Specifies the file system block size.
`f_bfree`	Specifies the number of free blocks.
`f_ffree`	Specifies the number of free file system objects.
`f_fname`	Specifies a 32–byte string indicating the file system name.
`f_fpack`	Specifies a 32–byte string indicating a pack ID.
`f_name_max`	Specifies the maximum length of an object name.

Fields for which a **vfs** structure has no values are set to 0.

Execution Environment

The **vfs_statfs** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the /**usr**/**include**/**sys**/**errno.h** file to indicate failure.

Related Information

The **statfs** subroutine.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Understanding Data Structures and Header Files for Virtual File Systems, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vfs_sync Entry Point

Purpose

Requests that file system changes be written to permanent storage.

Syntax

```
int vfs_sync (*gfsp)
struct gfs *gfsp;
```

Parameter

gfsp Points to a **gfs** structure. The **gfs** structure describes the file system type. This structure is defined in the /**usr**/**include**/**sys**/**gfs.h** file.

Description

The **vfs_sync** entry point is used by the logical file system to force all data associated with a particular virtual file system type to be written to its storage. This entry point is used to establish a known consistent state of the data.

Note: The **vfs_sync** entry point is called once per file system type rather than once per virtual file system.

Execution Environment

The **vfs_sync** entry point can be called from the process environment only.

Return Values

The **vfs_sync** entry point is advisory. It has no return values.

Related Information

The **sync** subroutine.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vfs_umount Entry Point

Purpose

Unmounts a virtual file system.

Syntax

```
int vfs_umount (vfsp, crp)
struct vfs *vfsp;
struct ucred *crp;
```

Parameters

vfsp	Points to the **vfs** structure being unmounted. This structure is defined in the **/usr/include/sys/vfs.h** file.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vfs_umount** entry point is called to unmount a virtual file system. The logical file system performs services independent of the virtual file system that initiate the unmounting. The logical file system services:

- Guarantee the syntax of the **uvmount** subroutine.
- Perform permission checks:
  - If the *vfsp* parameter refers to a device mount, then the user must have root user authority to perform the operation.
  - If the *vfsp* parameter does not refer to a device mount, then the user must have root user authority or write permission in the parent directory of the mounted–over virtual node (v–node), as well as write permission to the file represented by the mounted–over v–node.
- Ensure that the virtual file system being unmounted contains no mount points for other virtual file systems.
- Ensure that the root v–node is not in use except for the mount. The root v–node is also referred to as the mounted v–node.
- Clear the `v_mvfsp` field in the stub v–node. This prevents lookup operations already in progress from traversing the soon–to–be unmounted mount point.

The logical file system assumes that, if necessary, successful **vfs_umount** entry point calls free the root v–node. An error return from the **vfs_umount** entry point causes the mount point to be re–established. A 0 (zero) returned from the **vfs_umount** entry point indicates the routine was successful and that the **vfs** structure was released.

Execution Environment

The **vfs_umount** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related Information

The **umount** subroutine, **uvmount** subroutine, **vmount** subroutine.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Virtual File System Kernel Extensions Overview, Understanding Data Structures and Header Files for Virtual File Systems, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vfs_vget Entry Point

Purpose

Converts a file identifier into a virtual node (v–node).

Syntax

```
int vfs_vget (vfsp, vpp, fidp, crp)
struct vfs *vfsp;
struct vnode **vpp;
struct fileid *fidp;
struct ucred *crp;
```

Parameters

vfsp Points to the virtual file system that is to contain the v–node. Any returned v–node should belong to this virtual file system.

vpp Points to the place to return the v–node pointer. This is set to point to the new v–node. The fields in this v–node should be set as follows:

- `v_vntype` The type of v–node dependent on private data.
- `v_count` Set to at least 1 (one).
- `v_pdata` If a new file, set to the private data for this file system.

fidp Points to a file identifier. This is a file system–specific file identifier that must conform to the **fileid** structure.

Note: If the *fidp* parameter is invalid, the *vpp* parameter should be set to a null value by the **vfs_vget** entry point.

crp Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vfs_vget** entry point is called to convert a file identifier into a v–node. This entry point uses information in the *vfsp* and *fidp* parameters to create a v–node or attach to an existing v–node. This v–node represents, logically, the same file system object as the file identified by the *fidp* parameter.

If the v–node already exists, successful operation of this entry point increments the v–node use count and returns a pointer to the v–node. If the v–node does not exist, the **vfs_vget** entry point creates it using the **vn_get** kernel service and returns a pointer to the new v–node.

Execution Environment

The **vfs_vget** entry point can be called from the process environment only.

Return Values

0 Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure. A typical value includes:

EINVAL Indicates that the remote virtual file system specified by the *vfsp* parameter does not support chained mounts.

Related Information

The **vn_get** kernel service.

The **access** subroutine.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_access Entry Point

Purpose

Requests validation of user access to a virtual node (v–node).

Syntax

```
int vn_access (vp, mode, who, crp)
struct vnode *vp;
int mode;
int who;
struct ucred *crp;
```

Parameters

vp Points to the v–node.

mode Identifies the access mode.

who Specifies the IDs for which to check access. This parameter should be one of the following values, which are defined in the /**usr**/**include**/**sys**/**access.h** file:

- **ACC_SELF** Determines if access is permitted for the current process. The effective user and group IDs and the supplementary group ID of the current process are used for the calculation.
- **ACC_ANY** Determines if the specified access is permitted for any user, including the object owner. The *mode* parameter must contain only one of the valid modes.
- **ACC_OTHERS** Determines if the specified access is permitted for any user, excluding the owner. The *mode* parameter must contain only one of the valid modes.
- **ACC_ALL** Determines if the specified access is permitted for all users. (This is a useful check to make when files are to be written blindly across networks.) The *mode* parameter must contain only one of the valid modes.

crp Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_access** entry point is used by the logical volume file system to validate access to a v–node. This entry point is used to implement the **access** subroutine. The v–node is held for the duration of the **vn_access** entry point. The v–node count is unchanged by this entry point.

In addition, the **vn_access** entry point is used for permissions checks from within the file system implementation. The valid types of access are listed in the /**usr**/**include**/**sys**/**access.h** file. Current modes are read, write, execute, and existence check.

Note: The **vn_access** entry point must ensure that write access is not requested on a read–only file system.

Execution Environment

The **vn_access** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the /**usr**/**include**/**sys**/**errno.h** file to indicate failure. A typical value includes:

EACCESS	Indicates no access is allowed.

Related Information

The **access** subroutine.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_close Entry Point

Purpose

Closes a file associated with a v–node (virtual node).

Syntax

```
int vn_close (vp, flag, vinfo, crp)
struct vnode *vp;
int flag;
caddr_t vinfo;
struct ucred *crp;
```

Parameters

vp	Points to the v–node.
flag	Identifies the flag word from the file pointer.
vinfo	This parameter is not used.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_close** entry point is used by the logical file system to announce that the file associated with a given v–node is now closed. The v–node continues to remain active but will no longer receive read or write requests through the **vn_rdwr** entry point.

A **vn_close** entry point is called only when the use count of an associated file structure entry goes to 0 (zero).

Note: The v–node is held over the duration of the **vn_close** entry point.

Execution Environment

The **vn_close** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Note: The **vn_close** entry point may fail and an error will be returned to the application. However, the v–node is considered closed.

Related Information

The **close** subroutine.

The **vn_open** entry point, **vn_rele** entry point.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_create Entry Point

Purpose

Creates a new file.

Syntax

```
int vn_create (dp, vpp, flag, pname, mode, vinfop, crp)
struct vnode *dp;
struct vnode **vpp;
int flag;
char *pname;
int mode;
caddr_t *vinfop;
struct ucred *crp;
```

Parameters

dp	Points to the virtual node (v–node) of the parent directory.
vpp	Points to the place in which the pointer to a v–node for the newly created file is returned.
flag	Specifies an integer flag word. The **vn_create** entry point uses this parameter to open the file.
pname	Points to the name of the new file.
mode	Specifies the mode for the new file.
vinfop	This parameter is unused.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_create** entry point is invoked by the logical file system to create a regular (v–node type **VREG**) file in the directory specified by the *dp* parameter. (Other v–node operations create directories and special files.) Virtual node types are defined in the /**usr**/**include**/**sys**/**vnode.h** file. The v–node of the parent directory is held during the processing of the **vn_create** entry point.

To create a file, the **vn_create** entry point does the following:

- Opens the newly created file.
- Checks that the file system associated with the directory is not read–only.

Note: The logical file system calls the **vn_lookup** entry point before calling the **vn_create** entry point.

Execution Environment

The **vn_create** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the /**usr**/**include**/**sys**/**errno.h** file to indicate failure.

Related Information

The **vn_lookup** entry point.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_create_attr Entry Point

Purpose

Creates a new file.

Syntax

int vn_create_attr *(dvp, vpp, flags, name, vap, vcf, finfop, crp)*
struct vnode **dvp*;
struct vnode **vpp*;
int flags;
char **name*;
struct vattr **vap*;
int *vcf*;
caddr_t *finfop*;
struct ucred **crp*;

Parameters

dvp	Points to the directory vnode.
vpp	Points to the newly created vnode pointer.
flags	Specifies file creation flags.
name	Specifies the name of the file to create.
vattr	Points to the initial attributes.
vcf	Specifies create flags.
finfop	Specifies address of finfo field.
crp	Specifies user's credentials.

Description

The **vn_create_attr** entry point is used to create a new file. This operation is similar to the vn_create entry point except that the initial file attributes are passed in a vattr structure.

The va_mask field in the vattr structure identifies which attributes are to be applied. For example, if the AT_SIZE bit is set, then the file system should use va_size for the initial file size. For all vn_create_attr calls, at least AT_TYPE and AT_MODE must be set.

The vcf parameter controls how the new vnode is to be activated. If vcf is set to VC_OPEN, then the new object should be opened. If vcf is VC_LOOKUP, then the new object should be created, but not opened. If vcf is VC_DEFAULT, then the new object should be created, but the vnode for the object is not activated.

File systems that do not define GFS_VERSION421 in their gfs flags do not need to supply a vn_create_attr entry point. The logical file system will funnel all creation requests through the old vn_create entry point.

Execution Environment

The **vn_create_attr** entry point can be called from the process environment only.

Return Values

Zero	Indicates a successful operation; **vpp* contains a pointer to the new vnode.
Nonzero	Indicates that the operation failed; return values should be chosen from the **/usr/include/sys/errno.h** file.

Related Information

The **open** subroutine, **mknod** subroutine.

Virtual File System Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes), and Virtual File System Kernel Extensions Overview.

List of Virtual File System Operations.

vn_fclear Entry Point

Purpose

Releases portions of a file.

Syntax

```
int vn_fclear (vp, flags, offset, len, vinfo, crp)
struct vnode *vp;
int flags;
offset_t offset;
offset_t len;
caddr_t vinfo;
struct ucred *crp;
```

Parameters

vp	Points to the virtual node (v–node) of the file.
flags	Identifies the flags from the open file structure.
offset	Indicates where to start clearing in the file.
len	Specifies the length of the area to be cleared.
vinfo	This parameter is unused.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_fclear** entry point is called from the logical file system to clear bytes in a file, returning whole free blocks to the underlying file system. This entry point performs the clear regardless of whether the file is mapped.

Upon completion of the **vn_fclear** entry point, the logical file system updates the file offset to reflect the number of bytes cleared.

Execution Environment

The **vn_fclear** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related Information

The **fclear** subroutine.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_fid Entry Point

Purpose

Builds a file identifier for a virtual node (v–node).

Syntax

```
int vn_fid (vp, fidp, crp)
struct vnode *vp;
struct fileid *fidp;
struct ucred *crp;
```

Parameters

vp	Points to the v–node that requires the file identifier.
fidp	Points to where to return the file identifier.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_fid** entry point is invoked to build a file identifier for the given v–node. This file identifier must contain sufficient information to find a v–node that represents the same file when it is presented to the **vfs_get** entry point.

Execution Environment

The **vn_fid** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the /**usr**/**include**/**sys**/**errno.h** file to indicate failure.

Related Information

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_finfo Entry Point

Purpose

Returns information about a file.

Syntax

```
int
vn_finfo (vp, cmd, bufp, length, crp)
struct vnode *vp;
int cmd;
void *bufp;
int  length;
struct ucred  *crp;
```

Parameters

vp	Points to the vnode to be queried.
cmd	Specifies the command parameter.
bufp	Points to the buffer for the information.
length	Specifies the length of the buffer.
crp	Specifies user's credentials.

Description

The **vn_finfo** entry point is used to query a file system. It is used primarily to implement the **pathconf** and **fpathonf** subroutines. The **command** parameter defines what type of query is being done. The query commands and the associated data structures are defined in **<sys/finfo.h>**. If the file system does not support the particular query, it should return ENOSYS.

File systems that do not define GFS_VERSION421 in their gfs flags do not need to supply a **vn_finfo** entry point. If the command is FI_PATHCONF, then the logical file system returns generic pathconf information. If the query is other than FI_PATHCONF, then the request fails with EINVAL.

Execution Environment

The **vn_finfo** entry point can be called from the process environment only.

Return Values

Zero	Indicates a successful operation.
Nonzero	Indicates that the operation failed; return values should be chosen from the **/usr/include/sys/errno.h** file.

Related Information

The **pathconf**, **fpathconf** subroutine.

Virtual File System Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*, and Virtual File System Kernel Extensions Overview.

vn_fsync Entry Point

Purpose

Flushes information in memory and data to disk.

Syntax

```
int vn_fsync (vp, flags, crp)
struct vnode *vp;
int flags;
struct ucred *crp;
```

Parameters

vp	Points to the virtual node (v–node) of the file.
flags	Identifies flags from the open file.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_fsync** entry point is called by the logical file system to request that all modifications associated with a given v–node be flushed out to permanent storage. This must be synchronously so that the caller can be assured that all I/O has completed successfully.

Execution Environment

The **vn_fsync** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero values are returned from the /**usr**/**include**/**sys**/**errno.h** file to indicate failure.

Related Information

The **fsync** subroutine.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_fsync_range Entry Point

Purpose

Flushes file data to disk.

Syntax

```
int
vn_fsync_range (vp, flags, fd, offset, length, crp)
struct vnode *vp;
int flags;
int fd;
offset_t  offset;
offset_t  length;
struct ucred  *crp;
```

Parameters

vp	Points to the vnode.
flags	Specifies the File flags.
fd	Specifies the File descriptor.
length	Specifies the length of the flush request.
crp	Specifies user's credentials.

Description

The **vn_fsync_range** entry point is used to flush file data and meta–data to disk. The *offset* and *length* parameters define the range that needs to be flushed. If length is given as zero, then the entire file past offset should be flushed.

The *flags* parameter controls how the flushing should be done. If the O_SYNC flag is set, then the flush should be done according to the synchronized file I/O integrity completion rules. If O_DSYNC is set, then the flush should be done according to the synchronized data I/O integrity completion rules.

File systems that do not define GFS_VERSION421 in their gfs flags do not need to supply a **vn_fsync_range** entry point. The logical file system will funnel all fsync requests through the old vn_fsync entry point.

Execution Environment

The **vn_fsync_range** entry points can be called from the process environment only.

Return Values

Zero	Indicates a successful operation.
Nonzero	Indicates that the operation failed; return values should be chosen from the **/usr/include/sys/errno.h** file.

Related Information

The **fsync**, **fdatasync**, **fsync_range** subroutines.

Virtual File System Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*, and Virtual File System Kernel Extensions Overview.

vn_ftrunc Entry Point

Purpose

Truncates a file.

Syntax

```
int vn_ftrunc (vp, flags, length, vinfo, crp)
struct vnode *vp;
int flags;
offset_t length;
caddr_t vinfo;
struct ucred *crp;
```

Parameters

vp	Points to the virtual node (v–node) of the file.
flags	Identifies flags from the open file structure.
length	Specifies the length to which the file should be truncated.
vinfo	This parameter is unused.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_ftrunc** entry point is invoked by the logical file system to decrease the length of a file by truncating it. This operation is unsuccessful if any process other than the caller has locked a portion of the file past the specified offset.

Execution Environment

The **vn_ftrunc** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the /**usr**/**include**/**sys**/**errno.h** file to indicate failure.

Related Information

The **ftruncate** subroutine.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_getacl Entry Point

Purpose

Retrieves the access control list (ACL) for a file.

Syntax

```
#include <sys/acl.h>

int vn_getacl (vp, uiop, crp)
struct vnode *vp;
struct uio *uiop;
struct ucred *crp;
```

Description

The **vn_getacl** entry point is used by the logical file system to retrieve the access control list (ACL) for a file to implement the **getacl** subroutine.

Parameters

vp	Specifies the virtual node (v–node) of the file system object.
uiop	Specifies the **uio** structure that defines the storage for the ACL.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Execution Environment

The **vn_getacl** entry point can be called from the process environment only.

Return Values

0	Indicates a successful operation.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure. A valid value includes:

ENOSPC	Indicates that the buffer size specified in the *uiop* parameter was not large enough to hold the ACL. If this is the case, the first word of the user buffer (data in the **uio** structure specified by the *uiop* parameter) is set to the appropriate size.

Related Information

The **chacl** subroutine, **chmod** subroutine, **chown** subroutine, **statacl** subroutine.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_getattr Entry Point

Purpose

Gets the attributes of a file.

Syntax

```
int vn_getattr (vp, vap, crp)
struct vnode *vp;
struct vattr *vap;
struct ucred *crp;
```

Parameters

vp	Specifies the virtual node (v–node) of the file system object.
vap	Points to a **vattr** structure.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_getattr** entry point is called by the logical file system to retrieve information about a file. The **vattr** structure indicated by the *vap* parameter contains all the relevant attributes of the file. The **vattr** structure is defined in the /**usr**/**include**/**sys**/**vattr.h** file. This entry point is used to implement the **stat**, **fstat**, and **lstat** subroutines.

Note: The indicated v–node is held for the duration of the **vn_getattr** subroutine.

Execution Environment

The **vn_getattr** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the /**usr**/**include**/**sys**/**errno.h** file to indicate failure.

Related Information

The **statx** subroutine.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_hold Entry Point

Purpose

Assures that a virtual node (v–node) is not destroyed.

Syntax

```
int vn_hold (vp)
struct vnode *vp;
```

Parameter

vp	Points to the v–node.

Description

The **vn_hold** entry point increments the `v_count` field, the hold count on the v–node, and the v–node's underlying g–node (generic node). This incrementation assures that the v–node is not deallocated.

Execution Environment

The **vn_hold** entry point can be called from the process environment only.

Return Values

The **vn_hold** entry point cannot fail and therefore has no return values.

Related Information

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes), Understanding Generic I–nodes (G–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_ioctl Entry Point

Purpose

Requests I/O control operations on special files.

Syntax

```
int vn_ioctl (vp, cmd, arg, flags, ext, crp)
struct vnode *vp;
int cmd;
caddr_t arg;
int flags, ext;
struct ucred *crp;
```

Parameters

vp	Points to the virtual node (v–node) on which to perform the operation.
cmd	Identifies the specific command. Common operations for the **ioctl** subroutine are defined in the **/usr/include/sys/ioctl.h** file. The file system implementation can define other **ioctl** operations.
arg	Defines a command–specific argument. This parameter can be a single word or a pointer to an argument (or result structure).
flags	Identifies flags from the open file structure.
ext	Specifies the extended parameter passed by the **ioctl** subroutine. The **ioctl** subroutine always sets the *ext* parameter to 0.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_ioctl** entry point is used by the logical file system to perform miscellaneous operations on special files. If the file system supports special files, the information is passed down to the **ddioctl** entry point of the device driver associated with the given v–node.

Execution Environment

The **vn_ioctl** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure. A valid value includes:

EINVAL	Indicates the file system does not support the entry point.

Related Information

The **ioctl** subroutine.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_link Entry Point

Purpose

Requests a hard link to a file.

Syntax

```
int vn_link (vp, dp, name, crp)
struct vnode *vp;
struct vnode *dp;
caddr_t *name;
struct ucred *crp;
```

Parameters

vp	Points to the virtual node (v–node) to link to. This v–node is held for the duration of the linking process.
dp	Points to the v–node for the directory in which the link is created. This v–node is held for the duration of the linking process.
name	Identifies the new name of the entry.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_link** entry point is invoked to create a new hard link to an existing file as part of the link subroutine. The logical file system ensures that the *dp* and *vp* parameters reside in the same virtual file system, which is not read–only.

Execution Environment

The **vn_link** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related Information

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_lockctl Entry Point

Purpose

Sets, checks, and queries record locks.

Syntax

```
int vn_lockctl (vp, offset, lckdat, cmd, retry_fn, retry_id, crp)
struct vnode *vp;
offset_t offset;
struct eflock *lckdat;
int cmd;
int (*retry_fn)();
caddr_t retry_id;
struct ucred *crp;
```

Parameters

vp Points to the file's virtual node (v–node).

offset Indicates the file offset from the open file structure. This parameter is used to establish where the lock region begins.

lckdat Points to the **elock** structure. This structure describes the lock operation to perform.

cmd

Identifies the type of lock operation the **vn_lockctl** entry point is to perform. It is a bit mask that takes the following lock–control values:

SETFLCK If set, performs a lock set or clear. If clear, returns the lock information. The `l_type` field in the **eflock** structure indicates whether a lock is set or cleared.

SLPFLCK If the lock is unavailable immediately, wait for it. This is only valid when the **SETFLCK** flag is set.

retry_fn Points to a subroutine that is called when a lock is retried. This subroutine is not used if the lock is granted immediately.

Note: If the *retry_fn* parameter is not a null value, the **vn_lockctl** entry point will not sleep, regardless of the **SLPFLCK** flag.

retry_id Points to the location where a value can be stored. This value can be used to correlate a retry operation with a specific lock or set of locks. The retry value is only used in conjunction with the *retry_fn* parameter.

Note: This value is an opaque value and should not be used by the caller for any purpose other than a lock correlation. (This value should not be used as a pointer.)

crp Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_lockctl** entry point is used to request record locking. This entry point uses the information in the **eflock** structure to implement record locking.

If a requested lock is blocked by an existing lock, the **vn_lockctl** entry point should establish a sleeping lock with the retry subroutine address (specified by the *retry_fn* parameter) stored in the entry point. The **vn_lockctl** entry point then returns a correlating ID value to the caller (in the *retry_id* parameter), along with an exit value of **EAGAIN**. When the

sleeping lock is later awakened, the retry subroutine is called with the *retry_id* parameter as its argument.

eflock Structure

The **eflock** structure is defined in the **/usr/include/sys/flock.h** file and includes the following fields:

`l_type` Specifies type of lock. This field takes the following values:

- **F_RDLCK** Indicates read lock.
- **F_WRLCK** Indicates write lock.
- **F_UNLCK** Indicates unlock this record. A value of **F_UNLCK** starting at 0 until 0 for a length of 0 means unlock all locks on this file. Unlocking is done automatically when a file is closed.

`l_whence` Specifies location that the `l_start` field offsets.

`l_start` Specifies offset from the `l_whence` field.

`l_len` Specifies length of record. If this field is 0, the remainder of the file is specified.

`l_vfs` Specifies virtual file system that contains the file.

`l_sysid` Specifies value that uniquely identifies the host for a given virtual file system. This field must be filled in before the call to the **vn_lockctl** entry point.

`l_pid` Specifies process ID (PID) of the lock owner. This field must be filled in before the call to the **vn_lockctl** entry point.

Execution Environment

The **vn_lockctl** entry point can be called from the process environment only.

Return Values

0 Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure. Valid values include:

EAGAIN Indicates a blocking lock exists and the caller did not use the **SLPFLCK** flag to request that the operation sleep.

ERRNO Returns an error number from the **/usr/include/sys/errno.h** file on failure.

Related Information

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_lookup Entry Point

Purpose

Returns a v–node for a given name in a directory.

Syntax

```
int vn_lookup (dvp, vpp, name, vattrp , crp)
struct vnode *dvp;
struct vnode **vpp;
char *name;
struct vattr *vattrp;
struct ucred *crp;
```

Parameters

dvp	Points to the virtual node (v–node) of the directory to be searched. The logical file system verifies that this v–node is of a VDIR type.
name	Points to a null–terminated character string containing the file name to look up.
vattrp	Points to a **vattr** structure. If this pointer is NULL, no action is required of the file system implementation. If it is not NULL, the attributes of the file specified by the *name* parameter are returned at the address passed in the *vattrp* parameter.
vpp	Points to the place to which to return the v–node pointer, if the pointer is found. Otherwise, a null character should be placed in this memory location.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_lookup** entry point is invoked by the logical file system to find a v–node. It is used by the kernel to convert application–given path names to the v–nodes that represent them.

The use count in the v–node specified by the *dvp* parameter is incremented for this operation, and it is not decremented by the file system implementation.

If the name is found, a pointer to the desired v–node is placed in the memory location specified by the *vpp* parameter, and the v–node hold count is incremented. (In this case, this entry point returns 0.) If the file name is not found, a null character is placed in the *vpp* parameter, and the function returns a **ENOENT** value. Errors are reported with a return code from the **/usr/include/sys/errno.h** file. Possible errors are usually specific to the particular virtual file system involved.

Execution Environment

The **vn_lookup** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related Information

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_map Entry Point

Purpose

Validates file mapping requests.

Syntax

```
int vn_map (vp, addr, length, offset, flags, crp)
struct vnode *vp;
caddr_t addr;
uint length;
uint offset;
uint flags;
struct ucred *crp;
```

Parameters

Note: The *addr*, *offset*, and *length* parameters are unused in the current implementation. The file system is expected to store the segment ID with the file in the `gn_seg` field of the g–node for the file.

vp Points to the virtual node (v–node) of the file.

addr Identifies the location within the process address space where the mapping is to begin.

length Specifies the maximum size to be mapped.

offset Specifies the location within the file where the mapping is to begin.

flags Identifies what type of mapping to perform. This value is composed of bit values defined in the **/usr/include/sys/shm.h** file. The following values are of particular interest to file system implementations:

SHM_RDONLY The virtual memory object is read–only.

SHM_COPY The virtual memory object is copy–on–write. If this value is set, updates to the segment are deferred until an **fsync** operation is performed on the file. If the file is closed without an **fsync** operation, the modifications are discarded. The application that called the **vn_map** entry point is also responsible for calling the **vn_fsync** entry point.

Note: Mapped segments do not reflect modifications made to a copy–on–write segment.

crp Points to the **cred** structure. This structure contains data that applications can use to validate access permission.

Description

The **vn_map** entry point is called by the logical file system to validate mapping requests resulting from the **mmap** or **shmat** subroutines. The logical file system creates the virtual memory object (if it does not already exist) and increments the object's use count.

Execution Environment

The **vn_map** entry point can be called from the process environment only.

Return Values

0 Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related Information

The **shmat** subroutine, **vn_fsync** entry point.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_map_lloff Entry Point

Purpose

Announces intention to map a file.

Syntax

```
int
vn_map_lloff (vp, addr, offset, length, mflags, fflags, crp)
struct vnode *vp;
caddr_t  addr;
offset_t  offset;
offset_t  length;
int mflags;
int fflags;
struct ucred  *crp;
```

Parameters

vp	Points to the vnode to be queried.
addr	Unused.
offset	Specifies the starting offset for the map request.
length	Specifies the length of the mapping request.
mflags	Specifies the mapping flags.
fflags	Specifies the file flags.
crp	Specifies user's credentials.

Description

The **vn_map_lloff** entry point is used to tell the file system that the file is going to be accessed by memory mapped loads and stores. The file system should fail the request if it does not support memory mapping. This interface allows applications to specify starting offsets that are larger than 2 gigabytes.

File systems that do not define GFS_VERSION421 in their gfs flags do not need to supply a **vn_map_lloff** entry point.

Execution Environment

The **vn_map_lloff** entry point can be called from the process environment only.

Return Values

Zero	Indicates a successful operation.
Nonzero	Indicates that the operation failed; return values should be chosen from the **/usr/include/sys/errno.h** file.

Related Information

The **shmat** and **mmap** subroutines.

Virtual File System Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*, and Virtual File System Kernel Extensions Overview.

vn_mkdir Entry Point

Purpose

Creates a directory.

Syntax

```
int vn_mkdir (dp, name, mode, crp)
struct vnode *dp;
caddr_t name;
int mode;
struct ucred *crp;
```

Parameters

dp	Points to the virtual node (v–node) of the parent directory of a new directory. This v–node is held for the duration of the entry point.
name	Specifies the name of a new directory.
mode	Specifies the permission modes of a new directory.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_mkdir** entry point is invoked by the logical file system as the result of the **mkdir** subroutine. The **vn_mkdir** entry point is expected to create the named directory in the parent directory associated with the *dp* parameter. The logical file system ensures that the *dp* parameter does not reside on a read–only file system.

Execution Environment

The **vn_mkdir** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the /**usr**/**include**/**sys**/**errno.h** file to indicate failure.

Related Information

The **mkdir** subroutine.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_mknod Entry Point

Purpose

Creates a special file.

Syntax

```
int vn_mknod (dvp, name, mode, dev, crp)
struct vnode *dvp;
caddr_t *name;
int mode;
dev_t dev;
struct ucred *crp;
```

Parameters

dvp	Points to the virtual node (v–node) for the directory to contain the new file. This v–node is held for the duration of the **vn_mknod** entry point.
name	Specifies the name of a new file.
mode	Identifies the integer mode that indicates the type of file and its permissions.
dev	Identifies an integer device number.
crp	Points to the **cred** structure. This structure contains data that applications can use to validate access permission.

Description

The **vn_mknod** entry point is invoked by the logical file system as the result of a **mknod** subroutine. The underlying file system is expected to create a new file in the given directory. The file type bits of the *mode* parameter indicate the type of file (regular, character special, or block special) to be created. If a special file is to be created, the *dev* parameter indicates the device number of the new special file.

The logical file system verifies that the *dvp* parameter does not reside in a read–only file system.

Execution Environment

The **vn_mknod** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related Information

The **mknod** subroutine.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_open Entry Point

Purpose

Requests that a file be opened for reading or writing.

Syntax

```
int vn_open (vp, flag, ext, vinfop, crp)
struct vnode *vp;
int flag;
caddr_t ext;
caddr_t vinfop;
struct ucred *crp;
```

Parameters

vp	Points to the virtual node (v–node) associated with the desired file. The v–node is held for the duration of the open process.
flag	Specifies the type of access. Access modes are defined in the **/usr/include/sys/fcntl.h** file. **Note:** The **vn_open** entry point does not use the FCREAT mode.
ext	Points to external data. This parameter is used if the subroutine is opening a device.
vinfop	This parameter is not currently used.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_open** entry point is called to initiate a process access to a v–node and its underlying file system object. The operation of the **vn_open** entry point varies between virtual file system (VFS) implementations. A successful **vn_open** entry point must leave a v–node count of at least 1.

The logical file system ensures that the process is not requesting write access (with the FWRITE or FTRUNC mode) to a read–only file system.

Execution Environment

The **vn_open** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related Information

The **open** subroutine.

The **vn_close** entry point.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_rdwr Entry Point

Purpose

Requests file I/O.

Syntax

```
int vn_rdwr (vp, op, flags, uiop, ext, vinfo, vattrp, crp)
struct vnode *vp;
enum uio_rw op;
int flags;
struct uio *uiop;
int ext;
caddr_t vinfo;
struct vattr *vattrp;
struct ucred *crp;
```

Parameters

vp Points to the virtual node (v–node) of the file.

op Specifies a number that indicates a read or write operation. This parameter has a value of either **UIO_READ** or **UIO_WRITE**. These values are found in the **/usr/include/sys/uio.h** file.

flags Identifies flags from the open file structure.

uiop Points to a **uio** structure. This structure describes the count, data buffer, and other I/O information.

ext Provides an extension for special purposes. Its use and meaning are specific to virtual file systems, and it is usually ignored except for devices.

vinfo This parameter is currently not used.

vattrp Points to a vattr structure. If this pointer is NULL, no action is required of the file system implementation. If it is not NULL, the attributes of the file specified by the *vp* parameter are returned at the address passed in the *vattrp* parameter.

crp Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_rdwr** entry point is used to request that data be read or written from an object represented by a v–node. The **vn_rdwr** entry point does the indicated data transfer and sets the number of bytes *not* transferred in the `uio_resid` field. This field is 0 (zero) on successful completion.

Execution Environment

The **vn_rdwr** entry point can be called from the process environment only.

Return Values

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure. The **vn_rdwr** entry point returns an error code if an operation did not transfer all the data requested. The only exception is if an end of file is reached on a read request. In this case, the operation still returns 0.

Related Information

The **vn_create** entry point, **vn_open** entry point.

The **read** subroutine, **write** subroutine.

Virtual File System Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes), and Virtual File System Kernel Extensions Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_rdwr_attr Entry Point

Purpose

Reads or writes data to or from a file.

Syntax

```
int vn_rdwr_attr (vp, rw, fflags, uiop, vinfo, prevap, postvap,
crp)
struct vnode *vp;
enum uio_rw  rw;
int fflags;
struct uio  *uiop;
int ext;
caddr_t  vinfo;
struct vattr*prevap;
struct vattr*postvap;
struct ucred  *crp;
```

Parameters

vp	Points to the vnode to be read or written.
rw	Specifies a flag indicating read or write.
fflags	Specifies the file flags.
uiop	Points to the uiop structure describing the operation.
ext	Specifies the extension parameter passed to readx or writex.
vinfo	Specifies the *vinfo* parameter from the file table entry.
prevap	Points to an attributes structure for pre–operation attributes.
postvap	Points to an attributes structure for post–operation attributes.
crp	Specifies user's credentials.

Description

The **vn_rdwr_attr** entry point is used to read and write files. The arguments are identical to the **vn_rdwr** entry point. The prevap and postvap pointers are used to return file attributes before and after the operation.

File systems that do not define GFS_VERSION421 in their gfs flags do not need to supply a **vn_rdwr_attr** entry point.

Execution Environment

The **vn_rdwr_attr** entry point can be called from the process environment only.

Return Values

Zero	Indicates a successful operation.
Nonzero	Indicates that the operation failed; return values should be chosen from the **/usr/include/sys/errno.h** file.

Related Information

Virtual File System Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*, and Virtual File System Kernel Extensions Overview.

vn_readdir Entry Point

Purpose

Reads directory entries in standard format.

Syntax

```
int vn_readdir (vp, uiop, crp)
struct vnode *vp;
struct uio *uiop;
struct ucred *crp;
```

Parameters

vp	Points to the virtual node (v–node) of the directory.
uiop	Points to the **uio** structure that describes the data area into which to put the block of **dirent** structures. The starting directory offset is found in the `uiop->uio_offset` field and the size of the buffer area is found in the `uiop->uio_resid` field.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_readdir** entry point is used to access directory entries in a standard way. These directories should be returned as an array of **dirent** structures. The /**usr**/**include**/**sys**/**dir.h** file contains the definition of a **dirent** structure.

The **vn_readdir** entry point does the following:

- Copies a block of directory entries into the buffer specified by the *uiop* parameter.
- Sets the `uiop->uio_resid` field to indicate the number of bytes read.

The End–of–file character should be indicated by not reading any bytes (not by a partial read). This provides directories with the ability to have some hidden information in each block.

The virtual file system–specific implementation is also responsible for setting the `uio_offset` field to the offset of the next whole block to be read.

Execution Environment

The **vn_readdir** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the /**usr**/**include**/**sys**/**errno.h** file to indicate failure.

Related Information

The **readdir** subroutine.

The **uio** structure.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, and Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_readdir_eofp Entry Point

Purpose

Returns directory entries.

Syntax

```
int
vn_readdirr_eofp (vp, uiop, eofp, crp)
struct vnode *vp;
struct uio  *uiop;
int *eofp;
struct ucred  *crp;
```

Parameters

vp	Points to the directory vnode to be processed.
uiop	Points to the uiop structure describing the user's buffer.
eofp	Points to a word that places the eop structure.
crp	Specifies user's credentials.

Description

The **vn_readdir_eofp** entry point is used to read directory entries. It is similar to **vn_readdir** except that it takes the additional parameter, *eofp*. The location pointed to by the *eofp* parameter should be set to 1 if the readdir request reached the end of the directory. Otherwise, it should be set to 0.

File systems that do not define GFS_VERSION421 in their gfs flags do not need to supply a **vn_readdir_eofp** entry point.

Execution Environment

The **vn_readdir_eofp** entry point can be called from the process environment only.

Return Values

Zero	Indicates a successful operation.
Nonzero	Indicates that the operation failed; return values should be chosen from the **/usr/include/sys/errno.h** file.

Related Information

The **readdir** subroutine.

Virtual File System Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*, and Virtual File System Kernel Extensions Overview.

vn_readlink Entry Point

Purpose

Reads the contents of a symbolic link.

Syntax

```
int vn_readlink (vp, uio, crp)
struct vnode *vp;
struct uio *uio;
struct ucred *crp;
```

Parameters

vp	Points to a virtual node (v–node) structure. The **vn_readlink** entry point holds this v–node for the duration of the routine.
uio	Points to a **uio** structure. This structure contains the information required to read the link. In addition, it contains the return buffer for the **vn_readlink** entry point.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_readlink** entry point is used by the logical file system to get the contents of a symbolic link, if the file system supports symbolic links. The logical file system finds the v–node (virtual node) for the symbolic link, so this routine simply reads the data blocks for the symbol link.

Execution Environment

The **vn_readlink** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the /**usr**/**include**/**sys**/**errno.h** file to indicate failure.

Related Information

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_rele Entry Point

Purpose

Releases a reference to a virtual node (v–node).

Syntax

```
int vn_rele (vp,)
struct vnode *vp;
```

Parameter

vp	Points to the v–node.

Description

The **vn_rele** entry point is used by the logical file system to release the object associated with a v–node. If the object was the last reference to the v–node, the **vn_rele** entry point then calls the **vn_free** kernel service to deallocate the v–node.

If the virtual file system (VFS) was unmounted while there were open files, the logical file system sets the **VFS_UNMOUNTING** flag in the **vfs** structure. If the flag is set and the v–node to be released is the last v–node on the chain of the **vfs** structure, then the virtual file system must be deallocated with the **vn_rele** entry point.

Execution Environment

The **vn_rele** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related Information

The **vn_free** kernel service.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_remove Entry Point

Purpose

Unlinks a file or directory.

Syntax

```
int vn_remove (vp, dvp, name, crp)
struct vnode *vp;
struct vnode *dvp;
char *name;
struct ucred *crp;
```

Parameters

vp	Points to a virtual node (v–node). The v–node indicates which file to remove and is held over the duration of the **vn_remove** entry point.
dvp	Points to the v–node of the parent directory. This directory contains the file to be removed. The directory's v–node is held for the duration of the **vn_remove** entry point.
name	Identifies the name of the file.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_remove** entry point is called by the logical file system to remove a directory entry (or link) as the result of a call to the **unlink** subroutine.

The logical file system assumes that the **vn_remove** entry point calls the **vn_rele** entry point. If the link is the last reference to the file in the file system, the disk resources that the file is using are released.

The logical file system ensures that the directory specified by the *dvp* parameter does not reside in a read–only file system.

Execution Environment

The **vn_remove** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the /**usr**/**include**/**sys**/**errno.h** file to indicate failure.

Related Information

The **unlink** subroutine.

The **vn_rele** entry point.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_rename Entry Point

Purpose

Renames a file or directory.

Syntax

```
int vn_rename (srcvp, srcdvp, oldname, destvp, destdvp, newname,
crp)
struct vnode *srcvp;
struct vnode *srcdvp;
char *oldname;
struct vnode *destvp;
struct vnode *destdvp;
char *newname;
struct ucred *crp;
```

Parameters

srcvp	Points to the virtual node (v–node) of the object to rename.
srcdvp	Points to the v–node of the directory where the *srcvp* parameter resides. The parent directory for the old and new object can be the same.
oldname	Identifies the old name of the object.
destvp	Points to the v–node of the new object. This pointer is used only if the new object exists. Otherwise, this parameter is the null character.
destdvp	Points to the parent directory of the new object. The parent directory for the new and old objects can be the same.
newname	Points to the new name of the object.
crp	Points to the **cred** structure. This structure contains data that applications can use to validate access permission.

Description

The **vn_rename** entry point is invoked by the logical file system to rename a file or directory. This entry point provides the following renaming actions:

- Renames an old object to a new object that exists in a different parent directory.
- Renames an old object to a new object that does not exist in a different parent directory.
- Renames an old object to a new object that exists in the same parent directory.
- Renames an old object to a new object that does not exist in the same parent directory.

To ensure that this entry point routine executes correctly, the logical file system guarantees the following:

- File names are not renamed across file systems.
- The old and new objects (if specified) are not the same.
- The old and new parent directories are of the same type of v–node.

Execution Environment

The **vn_rename** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related Information

The **rename** subroutine.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_revoke Entry Point

Purpose

Revokes all access to an object.

Syntax

```
int vn_revoke (vp, cmd, flag, vinfop, crp)
struct vnode *vp;
int cmd;
int flag;
caddr_t vinfop;
struct ucred *crp;
```

Parameters

vp Points to the virtual node (v–node) containing the object.

cmd Indicates whether the calling process holds the file open. This parameter takes the following values:

0	The process did not have the file open.
1	The process had the file open.
2	The process had the file open and the reference count in the file structure was greater than 1.

flag Identifies the flags from the **file** structure.

vinfop This parameter is currently unused.

crp Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_revoke** entry point is called to revoke further access to an object.

Execution Environment

The **vn_revoke** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related Information

The **frevoke** subroutine, **revoke** subroutine.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_rmdir Entry Point

Purpose

Removes a directory.

Syntax

```
int vn_rmdir (vp, dp, pname, crp)
struct vnode *vp;
struct vnode *dp;
char *pname;
struct ucred *crp;
```

Parameters

vp	Points to the virtual node (v–node) of the directory.
dp	Points to the parent of the directory to remove.
pname	Points to the name of the directory to remove.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_rmdir** entry point is invoked by the logical file system to remove a directory object. To remove a directory, the directory must be empty (except for the current and parent directories). Before removing the directory, the logical file system ensures the following:

- The *vp* parameter is a directory.
- The *vp* parameter is not the root of a virtual file system.
- The *vp* parameter is not the current directory.
- The *dp* parameter does not reside on a read–only file system.

Note: The *vp* and *dp* parameters' v–nodes (virtual nodes) are held for the duration of the routine.

Execution Environment

The **vn_rmdir** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the /**usr**/**include**/**sys**/**errno.h** file to indicate failure.

Related Information

The **rmdir** subroutine.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_seek Entry Point

Purpose

Validates file offsets.

Syntax

```
int vn_seek (vp, offsetp, crp)
struct vnode *vp;
offset_t *offp;
struct ucred *crp;
```

Parameters

vp	Points to the virtual node (v–node) of the file.
offp	Points to the location of the new offset to validate.
crp	Points to the user's credential.

Description

Note: The **vn_seek Entry Point** applies to Version 4.2 and later releases.

The **vn_seek** entry point is called by the logical file system to validate a new offset that has been computed by the **lseek**, **llseek**, and **lseek64** subroutines. The file system implementation should check the offset pointed to by *offp* and if it is acceptable for the file, return zero. If the offset is not acceptable, the routine should return a non–zero value. **EINVAL** is the suggested error value for invalid offsets.

File systems which do not wish to do offset validation can simply return 0. File systems which do not provide the **vn_seek** entry point will have a maximum offset of **OFF_MAX** (2 gigabytes minus 1) enforced by the logical file system.

Execution Environment

The **vn_seek** entry point is be called from the process environment only.

Return Values

0	Indicates success.
Nonzero	Return values are returned the /**usr**/**include**/**sys**/**errno.h** file to indicate failure.

Related Information

The **lseek, llseek,** and**, lseek64** subroutines.

The Large File Enabled Programming Environment Overview.

vn_select Entry Point

Purpose

Polls a virtual node (v–node) for immediate I/O.

Syntax

```
int vn_select (vp, correl, e, re, notify, vinfo, crp)
struct vnode *vp;
int correl;
int e;
int re;
int (*notify)();
caddr_t vinfo;
struct ucred *crp;
```

Parameters

vp	Points to the v–node to be polled.
correl	Specifies the ID used for correlation in the **selnotify** kernel service.
e	Identifies the requested event.
re	Returns an events list. If the v–node is ready for immediate I/O, this field should be set to indicate the requested event is ready.
notify	Specifies the subroutine to call when the event occurs. This parameter is for nested polls.
vinfo	Is currently unused.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_select** entry point is invoked by the logical file system to poll a v–node to determine if it is immediately ready for I/O. This entry point is used to implement the **select** and **poll** subroutines.

File system implementation can support constructs, such as devices or pipes, that support the select semantics. The **fp_select** kernel service provides more information about select and poll requests.

Execution Environment

The **vn_select** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related Information

The **poll** subroutine, **select** subroutine.

The **fp_select** kernel service, **selnotify** kernel service.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_setacl Entry Point

Purpose

Sets the access control list (ACL) for a file.

Syntax

```
#include <sys/acl.h>

int vn_setacl (vp, uiop, crp)
struct vnode *vp;
struct uio *uiop;
struct ucred *crp;
```

Parameters

vp	Specifies the virtual node (v–node) of the file system object.
uiop	Specifies the **uio** structure that defines the storage for the call arguments.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_setacl** entry point is used by the logical file system to set the access control list (ACL) on a file.

Execution Environment

The **vn_setacl** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure. Valid values include:

ENOSPC	Indicates that the space cannot be allocated to hold the new ACL information.
EPERM	Indicates that the effective user ID of the process is not the owner of the file and the process is not privileged.

Related Information

The **uio** structure.

The **chacl** subroutine, **chown** subroutine, **chmod** subroutine, **statacl** subroutine.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_setattr Entry Point

Purpose

Sets attributes of a file.

Syntax

```
int vn_setattr (vp, cmd, arg1, arg2, arg3, crp)
struct vnode *vp;
int cmd;
int arg1;
int arg2;
int arg3;
struct ucred *crp;
```

Parameters

vp Points to the virtual node (v–node) of the file.

cmd Defines the setting operation. This parameter takes the following values:

V_OWN Sets the user ID (UID) and group ID (GID) to the UID and GID values of the new file owner. The *flag* argument indicates which ID is affected.

V_UTIME Sets the access and modification time for the new file. If the *flag* parameter has the value of **T_SETTIME**, then the specific values have not been provided and the access and modification times of the object should be set to current system time. If the **T_SETTIME** value is not specified, the values are specified by the *atime* and *mtime* variables.

V_MODE Sets the file mode.

The /**usr**/**include**/**sys**/**vattr.h** file contains the definitions for the three command values.

arg1, arg2, arg3 Specify the command arguments. The values of the command arguments depend on which command calls the **vn_setattr** entry point.

crp Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_setattr** entry point is used by the logical file system to set the attributes of a file. This entry point is used to implement the **chmod**, **chownx**, and **utime** subroutines.

The values that the *arg* parameters take depend on the value of the *cmd* parameter. The **vn_setattr** entry point accepts the following *cmd* values and *arg* parameters:

Possible cmd Values for the vn_setattr Entry Point

Command	V_OWN	V_UTIME	V_MODE
arg1	**int** *flag*;	**int** *flag*;	**int** *mode*;
arg2	**int** *uid*;	**timestruc_t** **atime*;	Unused
arg3	**int** *gid*;	**timestruc_t** **mtime*;	Unused

Note: For **V_UTIME**, if arg2 or arg3 is NULL, then the corresponding time field, *atime* and *mtime*, of the file should be left unchanged.

Execution Environment

The **vn_setattr** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related Information

The **chmod** subroutine, **chownx** subroutine, **utime** subroutine.

Virtual File System Kernel Extensions Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_strategy Entry Point

Purpose

Accesses blocks of a file.

Syntax

```
int vn_strategy (vp, bp, crp)
struct vnode *vp;
struct buf *bp;
struct ucred *crp;
```

Parameters

vp	Points to the virtual node (v–node) of the file.
bp	Points to a **buf** structure that describes the buffer.
crp	Points to the **cred** structure. This structure contains data that applications can use to validate access permission.

Description

Note: The **vn_strategy** entry point is not implemented in Version 3.2 of the operating system.

The **vn_strategy** entry point accesses blocks of a file. This entry point is intended to provide a block–oriented interface for servers for efficiency in paging.

Return Values

0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related Information

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_symlink Entry Point

Purpose

Creates a symbolic link.

Syntax

```
int vn_symlink (vp, linkname, target, crp)
struct vnode *vp;
char *linkname;
char *target;
struct ucred *crp;
```

Parameters

vp	Points to the virtual node (v–node) of the parent directory where the link is created.
linkname	Points to the name of the new symbolic link. The logical file system guarantees that the new link does not already exit.
target	Points to the name of the object to which the symbolic link points. This name need not be a fully qualified path name or even an existing object.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_symlink** entry point is called by the logical file system to create a symbolic link. The path name specified by the *linkname* parameter is the name of the new symbolic link. This symbolic link points to the object named by the *target* parameter.

Execution Environment

The **vn_symlink** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related Information

The **symlink** subroutine.

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

vn_unmap Entry Point

Purpose

Unmaps a file.

Syntax

```
int vn_unmap (vp, flag, crp)
struct vnode *vp;
ulong flag;
struct ucred *crp;
```

Parameters

vp	Points to the v–node (virtual node) of the file.
flag	Indicates how the file was mapped. This flag takes the following values: **SHM_RDONLY** The virtual memory object is read–only. **SHM_COPY** The virtual memory object is copy–on–write.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vn_unmap** entry point is called by the logical file system to unmap a file. When this entry point routine completes successfully, the use count for the memory object should be decremented and (if the use count went to 0) the memory object should be destroyed. The file system implementation is required to perform only those operations that are unique to the file system. The logical file system handles virtual–memory management operations.

Execution Environment

The **vn_unmap** entry point can be called from the process environment only.

Return Values

0	Indicates success.

Nonzero return values are returned from the /**usr**/**include**/**sys**/**errno.h** file to indicate failure.

Related Information

Virtual File System Overview, Virtual File System Kernel Extensions Overview, Logical File System Overview, Understanding Virtual Nodes (V–nodes) in *AIX Kernel Extensions and Device Support Programming Concepts*.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Vos remarques sur ce document / Technical publication remark form

Titre / Title : Bull Technical Reference Kernel & Subsystems Volume 1/2

Nº Reférence / Reference Nº : 86 A2 85AP 05

Daté / Dated : February 1999

ERREURS DETECTEES / ERRORS IN PUBLICATION

AMELIORATIONS SUGGEREES / SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Vos remarques et suggestions seront examinées attentivement.
Si vous désirez une réponse écrite, veuillez indiquer ci-après votre adresse postale complète.

Your comments will be promptly investigated by qualified technical personnel and action will be taken as required.
If you require a written reply, please furnish your complete mailing address below.

NOM / NAME : _____ Date : _____

SOCIETE / COMPANY : _____

ADRESSE / ADDRESS : _____

_____

Remettez cet imprimé à un responsable BULL ou envoyez-le directement à :

Please give this technical publication remark form to your BULL representative or mail to:

BULL ELECTRONICS ANGERS
CEDOC
34 Rue du Nid de Pie – BP 428
49004 ANGERS CEDEX 01
FRANCE

Technical Publications Ordering Form

Bon de Commande de Documents Techniques

To order additional publications, please fill up a copy of this form and send it via mail to:
Pour commander des documents techniques, remplissez une copie de ce formulaire et envoyez-la à :

BULL ELECTRONICS ANGERS
CEDOC
ATTN / MME DUMOULIN
34 Rue du Nid de Pie – BP 428
49004 ANGERS CEDEX 01
FRANCE

Managers / Gestionnaires :
Mrs. / Mme : **C. DUMOULIN** +33 (0) 2 41 73 76 65
Mr. / M : **L. CHERUBIN** +33 (0) 2 41 73 63 96

FAX : +33 (0) 2 41 73 60 19
E–Mail / Courrier Electronique : srv.Cedoc@franp.bull.fr

Or visit our web site at: / Ou visitez notre site web à:
`http://www-frec.bull.com` (PUBLICATIONS, Technical Literature, Ordering Form)

CEDOC Reference # Nº Référence CEDOC	**Qty** Qté	**CEDOC Reference #** Nº Référence CEDOC	**Qty** Qté	**CEDOC Reference #** Nº Référence CEDOC	**Qty** Qté
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
[_ _] : **no revision number means latest revision** / pas de numéro de révision signifie révision la plus récente					

NOM / NAME : ______________________________ Date : __________

SOCIETE / COMPANY : ______________________________

ADRESSE / ADDRESS : ______________________________

____

PHONE / TELEPHONE : __________________ FAX : __________________

E–MAIL : ______________________________

For Bull Subsidiaries / Pour les Filiales Bull :
Identification: ______________________________

For Bull Affiliated Customers / Pour les Clients Affiliés Bull :
Customer Code / Code Client : ______________________________

For Bull Internal Customers / Pour les Clients Internes Bull :
Budgetary Section / Section Budgétaire : ______________________________

For Others / Pour les Autres :
Please ask your Bull representative. / Merci de demander à votre contact Bull.

PLACE BAR CODE IN LOWER LEFT CORNER

BULL ELECTRONICS ANGERS
CEDOC
34 Rue du Nid de Pie – BP 428
49004 ANGERS CEDEX 01
FRANCE

ORDER REFERENCE
86 A2 85AP 05

Bull

Utiliser les marques de découpe pour obtenir les étiquettes.
Use the cut marks to get the labels.

AIX
Technical
Reference
Kernel &
Subsystems
Volume 1/2
86 A2 85AP 05

AIX
Technical
Reference
Kernel &
Subsystems
Volume 1/2
86 A2 85AP 05

AIX
Technical
Reference
Kernel &
Subsystems
Volume 1/2
86 A2 85AP 05